**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>

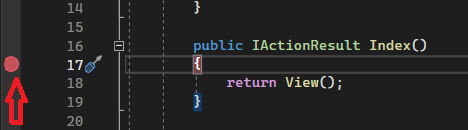
<p>Learn about <a href="https://docs.microsoft.com/aspnet/core">building Web apps with ASP.NET Core</a>.</p>

</div>

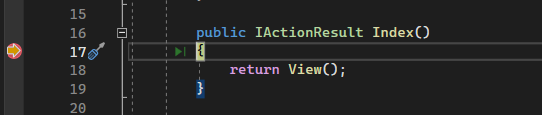
**Run our application** [**BulkBookWeb**] and we can see the route in action. In the **URL** right now, nothing is present “**https://localhost:7059**”. What should happen if there are no **Controllers** and **Actions** in the URL, we define that inside the **Program.cs**. We said of nothing is present,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

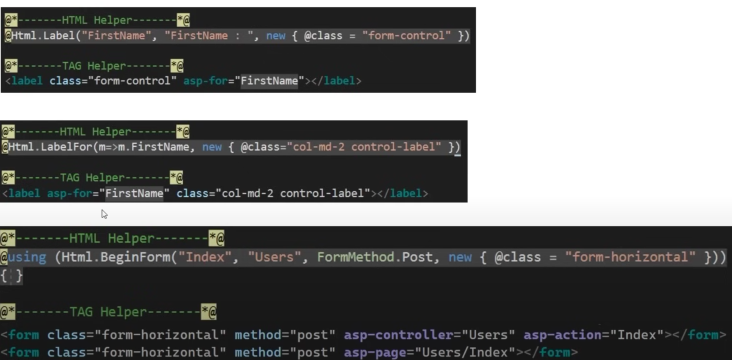
<p>Use this page to detail your site's privacy policy.</p>

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.

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;

}

}

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.

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

// Add services to the container.

builder.Services.AddControllersWithViews();

--Put Here--

var app = builder.Build();

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**.

**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

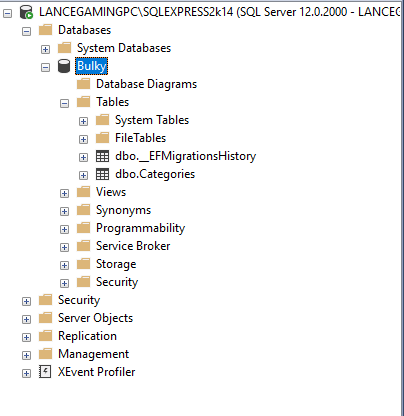
"AllowedHosts": "\*",

"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.cs** 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();

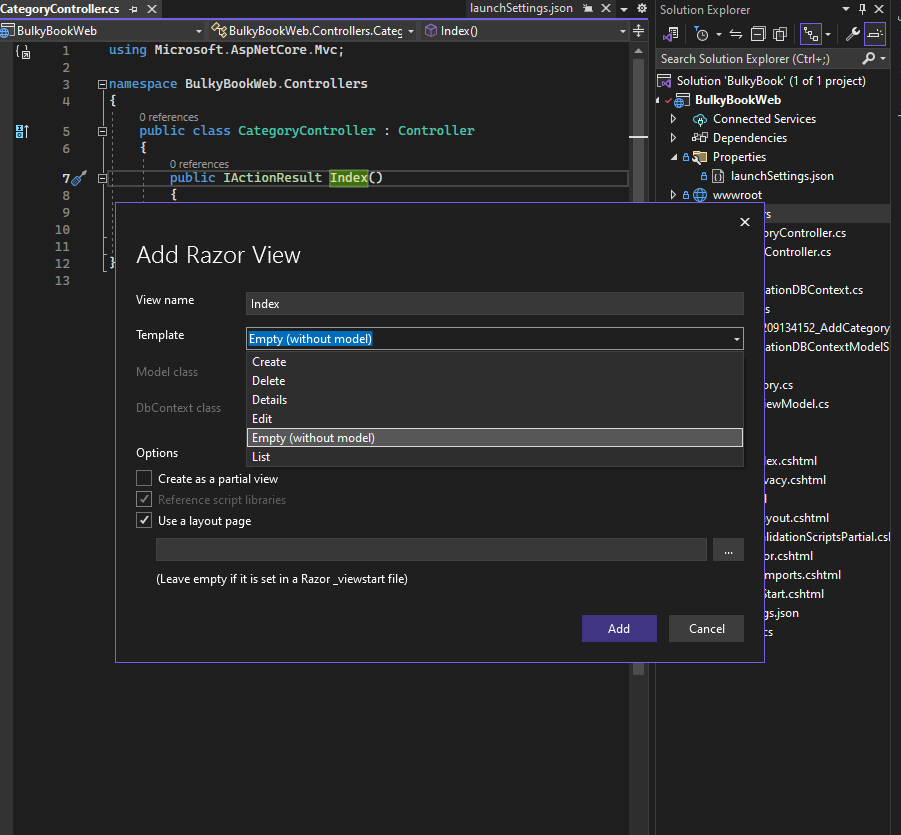
}

}

}

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 \_L**ayout 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>

**Displaying Categories list into the View Category.**

We want to display all the **categories** if there are any in the table format inside the **Index View**. For that, we will have to actually **retrieve all the category list** from our database inside the Index action method.

using Microsoft.AspNetCore.Mvc;

namespace BulkyBookWeb.Controllers

{

public class CategoryController : Controller

{

public IActionResult Index()

{

return View();

}

}

}

So that, we open up the **\_Layout.cshtml** page right here, where we have two options of **Home** and the **Privacy** page below.

<div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">

<ul class="navbar-nav flex-grow-1">

<li class="nav-item">

<**a** class="nav-link text-dark" **asp-area**="" **asp-controller**="Home" **asp-action**="Index">Home</**a**>

</li>

<li class="nav-item">

<**a** class="nav-link text-dark" **asp-area**="" **asp-controller**="Home" **asp-action**="Privacy">Privacy</**a**>

</li>

</ul>

</div>

Rather than **Privacy**, let’s change that to **Category** right here. To navigate to our Category index change the **asp-controller=”Category”** and **asp-action=”Index”** which is highlighted below: Optional only we can remove **asp-area=””** since we are not using it.

<div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">

<ul class="navbar-nav flex-grow-1">

<li class="nav-item">

<**a** class="nav-link text-dark" **asp-area**="" **asp-controller**="Home" **asp-action**="Index">Home</**a**>

</li>

<li class="nav-item">

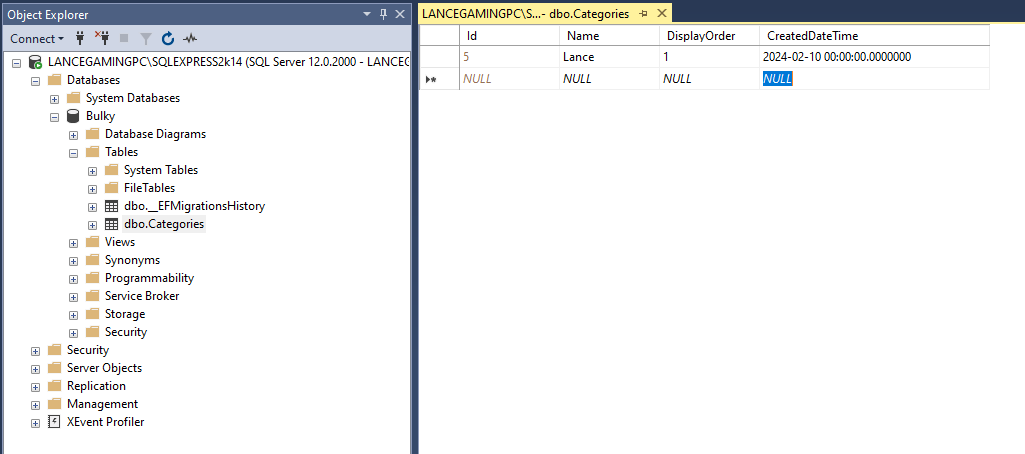
<**a** class="nav-link text-dark" **asp-area**="" **asp-controller**="Category" **asp-action**="Index">Category</**a**>

</li>

</ul>

</div>

Now we that configure we are enable to navigate to our category action method. Now we need to **retrieve all of the Categories from our database**. Switch back to our MSSQL **Bulky** Database, and just for purpose, **right click** the **Categories** table and select **Edit top 200 Rows,** inside we’ll add a dummy data.



Now let’s go back and we want to retrieve that inside the **CategoryController**. Now just like we’ve said before that we will be using **Entity Framework Core**, and the main file that we have is the **ApplicationDbContext.cs.** Using that we **can access our database object**, because inside that file, we have the **Categories** table. So how can we create an object of this **ApplicationDbContext**? And use that to call our database and table? Now that is the beauty of **Dependency Injection**, **we do not have to create an object of this class**, everything will be done for us, because we have configured that inside our container **Program.cs**, and we want to use this service.

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

builder.Configuration.GetConnectionString("DefaultConnection")

));

var app = builder.Build();

Because of **Dependency Injection**, we will **not have to create its object**, we can think that the object is already there, we just have to tell our application that **“**please send me the object of that **ApplicationDbContext”**. So how do we do that? Inside the controller **CategoryController**, we want the **ApplicationDbContext** to work with **Database**. So first create a **private readonly** field which will be **ApplicationDbContext** and call that “**\_db**”, we will add the using statement there “**using BulkyBookWeb.Data**” above. Now need to tell our application that we need an implementation of this **ApplicationDbContext** where the connection to database is already made, and we can recreate some records right away. So for that we will have to use **Constructor (ctor)**. And here whatever is registered inside the container we can access that, inside our container **(Program.cs)** , we have registered **ApplicationDbContext** which is highlighted above. So inside the **constructor**

“**public CategoryController()**”, we can get an implementation of that just like this:

“public CategoryController(**ApplicationDbContext db**)”, so this **db** will have all the implementation of connection string and tables that are needed to retrieve the data. So we will populate our local db object with this implementation, type this inside our constructor “**\_db = db**”. We can use the \_**db** to retrieve our **Categories**, the syntax of that is very simple

“**var objCategorList = \_db.Categories.ToList();**”

We will first access the **\_db**, and on there we will have all the **Dbset**. The **Dbset** that we want to work on right now is **Categories**, and we want to convert it to **.ToList()**; and retrieve that. So it will go to **database**, it will retrieve all of the **Categories**, it will convert that to a **list**, and it will **assign that inside the category list**. We can see how beautiful this is, **we do not have to write a SELECT statement** to retrieve all the categories from the table. There is **no SQL coding required**, we just have to write **\_db.(DbSetName)** and that will retrieve all the records, then we convert that to a list and we assign that to a variable ” **\_db.(DbSetName).ToList();**”.

using BulkyBookWeb.Data;

using Microsoft.AspNetCore.Mvc;

namespace BulkyBookWeb.Controllers

{

public class CategoryController : Controller

{

private readonly ApplicationDBContext \_db;

public CategoryController(ApplicationDBContext db)

{

\_db = db;

}

public IActionResult Index()

{

var objCategoryList = \_db.Categories.ToList();

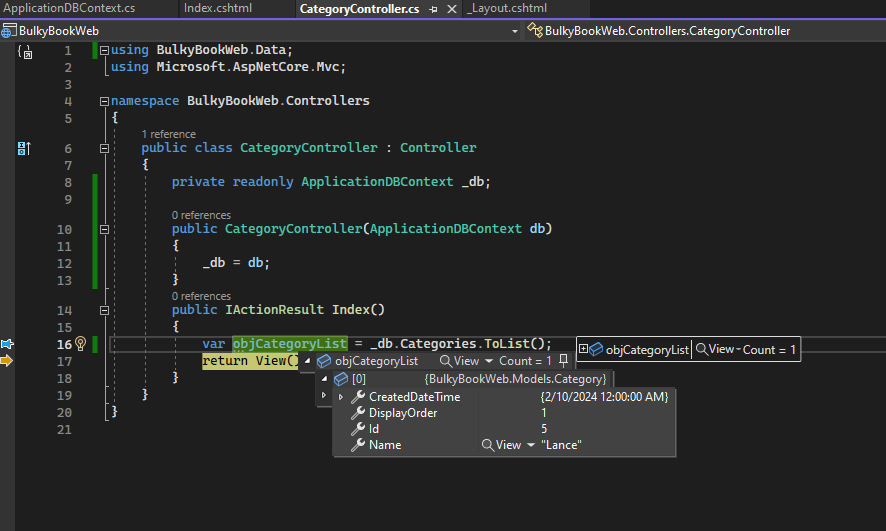
return View();

}

}

}

Now we test that this is working, we will add a **debugging point** by adding **red dot/ debugging point** to the “**return View();**”, if we run the application, our application will **hit the debugging point** when we load that page. So if we click on the **Category**, it will hit our **debugging point**. We can see that it is **yellow (the debugging point)**, and once we hover on that “**var objCategoryList”;** we can see there is **one count**. And if we **expand it more**, we can see it is the **same record that we created inside our database**. So it goes to the database, it fetches the **Categories** and it does that automatically.



Now we have a **list** that is already populated here, we just need to **pass** this list to our **View**, and we can run a **foreach loop on that list**, and we can display all the categories. Now that we have all the categories inside this object **objCategoryList**, we can copy that and we will pass that to our **View**. We can change that **var** and we can change that to strongly typed code, so this will be **IEnumerable<Category>** you can see the red squiggly line in **<Category>** to remove that just use the quick fix **CTRL + .** and we’ll add the using statement “**using BulkyBookWeb.Models**”.

using BulkyBookWeb.Data;

using BulkyBookWeb.Models;

using Microsoft.AspNetCore.Mvc;

namespace BulkyBookWeb.Controllers

{

public class CategoryController : Controller

{

private readonly ApplicationDBContext \_db;

public CategoryController(ApplicationDBContext db)

{

\_db = db;

}

public IActionResult Index()

{

IEnumerable<Category> objCategoryList = \_db.Categories;

return View(objCategoryList);

}

}

}

Now since we have an **IEnumerable** here and not a list, if we want, we can also remove the **.ToList();**. This is not a required thing. But if this was a list, then we would have used the **.ToList();** (when we use the **“var objCategoryList”**), so we are passing that to our View. Now that we are passing an IEnumerable of Category to our view, we need to **capture that inside our View as well**. So let’s open up that View “**Category/Index.cshtml**”. So right here, we first need to capture the model that we are passing from the controller, for that we will be using “**@**” sign, and we will write “**model**” in **all lowercase**. If we make this a capital model that won’t work, in order to fetch what is being passed from the **View**, we have to use all lowercase at the right model sign. So what exactly is being passed from the controller action method (**CategoryController.cs**), it is an **IEnumerable<Category>**, we can copy that, and we can paste that above **Category/Index.cshtml.**

@model IEnumerable<Category>

@{

ViewData["Title"] = "Index";

}

<h1>Index Category</h1>

In that way our **view** knows that whatever we **receive** here will be an **IEnumerable of Category**. Since it is **IEnumerable** we can **iterate** through that. Now if you have never worked with **razor pages**, we can use both **HTML** as well as some **C# code directly inside** the view. So first, let’s use the table attribute, give that some class of **<table class = “table table-bordered table-striped” style=”width:100%”>**. Complete code display here below:

@model IEnumerable<Category>

@{

ViewData["Title"] = "Index";

}

<h1>Index Category</h1>

<table class = "table table-bordered table-striped" style="width:100%">

<thead>

<tr>

<th>

Category Name

</th>

<th>

Display Order

</th>

</tr>

</thead>

<tbody>

@foreach (var obj in Model)// Model is our IEnumerable<Category> class above

{

<tr>

<td width="50%">

@obj.Name

</td>

<td width="30%">

@obj.DisplayOrder

</td>

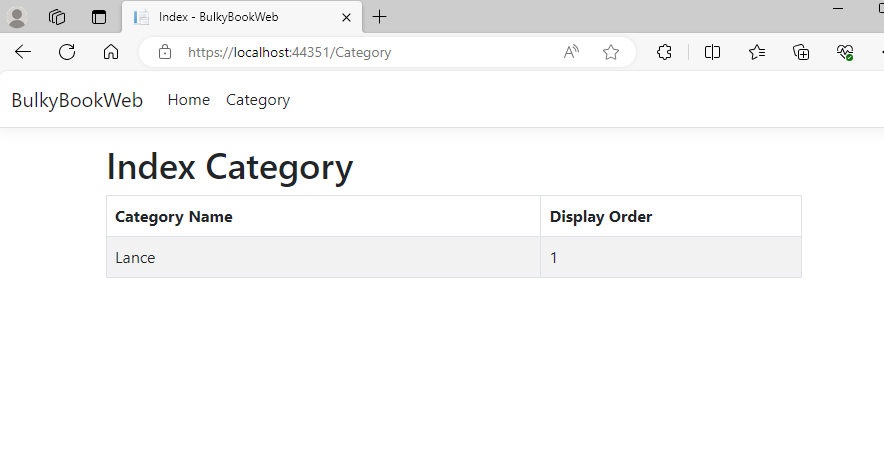
</tr>

}

</tbody>

</table>

Inside the table, we have **<thead></thead>** for **header** and **<tbody></tbody>** for the **body** of the table. The <**th**> tag is used to provide a header in a table cell, whereas the <**thead**> tag is used to provide a header for a **table group**.What will be all the columns that we have to display? The first one will be **Category Nam**e, and the next one will be **Display Order**. We can also display the date time and ID if we wanted to,but now we just want to display two columns, now that was the **header**. Now inside the body as we can see we putted an **foreach loop** to **iterate** through the **IEnumerable** and display the **actual values**. As we said before, we can use the C# code directly inside the Views using the “**@” directive.** So if we press @ we will see that we have a foreach right here. We can next enter (var obj in Model) the **Model** is from our **IEnumerable<Category>** inside our **CategoryController.cs** that we’ve entered earlier.That way it will **iterate** through all of the **properties inside this Model** in a **foreach loop**. Now for each of the **property**, we want to display a table row**<tr></tr>**. So we added above the **<tr></tr>** and also the **<td></td>** tag **inside table row** and first we want to display the **Name**, we have putted the **@obj.Name** inside our **<td></td>** with the **width=50%** and below we’ve also putted **@obj.DisplayOrder** also it is inside our **<td></td>** with the **width=30%**. The **<td> HTM**L element defines a **cell** of a table that contains data and may be used as a child of the **<tr> table row** element. Now if we run the application and we go to our **Category**, we will see our one **category** that we have inside the database. This is looking perfect to display all of the categories.



**Adding bootswatch theme into our application.**

Next, what we want is **Edit** and **Delete** buttons here, and we want to make this page a little bit pretty. Before that we will be using **bootswatch themes**. Now to design our application, just go to bootswatch.com for bootstrap, we can scroll down and select a particular theme that we want, but in this scenario we will select the Solar one, click **Download** drop down and select **bootstrap.css** to download the theme. Now open the file and copy all the elements inside this file, and past it into our directory “wwwroot/css” right click the css folder and select “Add/New item” and search **style sheet** and name it bootswatchTheme.css, then paste the code that we copied from our downloaded bootswatch theme and save it. Another change that we would do is inside site.css they have added a btn-primary so let’s remove that and the anchor tag (if it exist) if nothing, then its all good. Now to add this into our project we need to go to our \_Layout.cshtml and inside we should see our CSS at the very top.

<!DOCTYPE html>

<html lang="en">

<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 don’t want to use our default CSS, in that case we will use our newly created theme which is the bootswatchTheme.css, change the site.css highlighted above to bootswatchTheme.css which we created earlier.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8" />

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

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

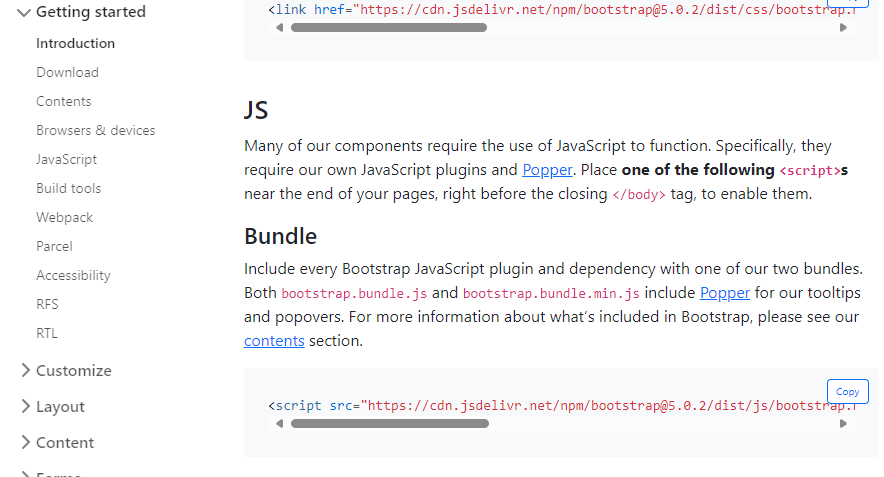
<**link** rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />

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

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

</head>

Now since we are using **Bootstrap five** with **bootswatch**, if we go to **bootstrap 5 form** in google search and click **Forms - Bootstrap v5.0** and at the left hand side click **Getting started/Introduction** below let’s **copy** the **JS Bundle.**

****

And let’s go back to our project inside the **\_Layout.cshtml** below at the very bottom lets replace the **bundle JS** that we currently had

</footer>

<**script** src="~/lib/jquery/dist/jquery.min.js"></**script**>

<**script** src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></**script**>

<**script** **src**="~/js/site.js" **asp-append-version**="true"></**script**>

@await RenderSectionAsync("Scripts", required: false)

</body>

</html>

And replace it with our new **bundle JS** that we’ve just copied from the website bootstrap.

</footer>

<**script** src="~/lib/jquery/dist/jquery.min.js"></**script**>

<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/js/bootstrap.bundle.min.js" integrity="sha384-MrcW6ZMFYlzcLA8Nl+NtUVF0sA7MsXsP1UyJoMp4YLEuNSfAP+JcXn/tWtIaxVXM" crossorigin="anonymous"></script>

<**script** **src**="~/js/site.js" **asp-append-version**="true"></**script**>

@await RenderSectionAsync("Scripts", required: false)

</body>

</html>

As of current the default version of **Bootstrap** is not five in **.NET 6**, this might change in the future, but that is the reason we’re switching to the latest version of Bootstrap. With that changed if we run our application we should see some changes it should look like the theme we’ve selected into the bootswatch.com website which is the Solar. Now lets go back to the website bootswatch and into our theme that we selected earlier we can click Preview, below we can see the **NavBar** and we’ll use that. Copy the code of the **NavBar** that we want to use and then paste it into our **\_Layout.cshtml** below **<header>** tag, above the current **<nav>** that we had.

<header>

--HERE---

<nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-shadow mb-3">

<div class="container-fluid">

<**a** class="navbar-brand" **asp-area**="" **asp-controller**="Home" **asp-action**="Index">BulkyBookWeb</**a**>

<button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target=".navbar-collapse" aria-controls="navbarSupportedContent"

aria-expanded="false" aria-label="Toggle navigation">

<span class="navbar-toggler-icon"></span>

</button>

<div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">

<ul class="navbar-nav flex-grow-1">

<li class="nav-item">

<**a** class="nav-link text-dark" **asp-controller**="Home" **asp-action**="Index">Home</**a**>

</li>

<li class="nav-item">

<**a** class="nav-link text-dark" **asp-controller**="Category" **asp-action**="Index">Category</**a**>

</li>

</ul>

</div>

</div>

</nav>

</header>

It should look like this now, highlighted below is the code of the NavBar we’ve just copied into bootswatch NavBar named Solar:

<header>

<nav class="navbar navbar-expand-lg bg-dark" data-bs-theme="dark">

<div class="container-fluid">

<a class="navbar-brand" href="#">Navbar</a>

<button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarColor02" aria-controls="navbarColor02" aria-expanded="false" aria-label="Toggle navigation">

<span class="navbar-toggler-icon"></span>

</button>

<div class="collapse navbar-collapse" id="navbarColor02">

<ul class="navbar-nav me-auto">

<li class="nav-item">

<a class="nav-link active" href="#">

Home

<span class="visually-hidden">(current)</span>

</a>

</li>

<li class="nav-item">

<a class="nav-link" href="#">Features</a>

</li>

<li class="nav-item">

<a class="nav-link" href="#">Pricing</a>

</li>

<li class="nav-item">

<a class="nav-link" href="#">About</a>

</li>

<li class="nav-item dropdown">

<a class="nav-link dropdown-toggle" data-bs-toggle="dropdown" href="#" role="button" aria-haspopup="true" aria-expanded="false">Dropdown</a>

<div class="dropdown-menu">

<a class="dropdown-item" href="#">Action</a>

<a class="dropdown-item" href="#">Another action</a>

<a class="dropdown-item" href="#">Something else here</a>

<div class="dropdown-divider"></div>

<a class="dropdown-item" href="#">Separated link</a>

</div>

</li>

</ul>

<**form** class="d-flex">

<input class="form-control me-sm-2" type="search" placeholder="Search">

<button class="btn btn-secondary my-2 my-sm-0" type="submit">Search</button>

</**form**>

</div>

</div>

</nav>

<nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-shadow mb-3">

<div class="container-fluid">

<**a** class="navbar-brand" **asp-area**="" **asp-controller**="Home" **asp-action**="Index">BulkyBookWeb</**a**>

<button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target=".navbar-collapse" aria-controls="navbarSupportedContent"

aria-expanded="false" aria-label="Toggle navigation">

<span class="navbar-toggler-icon"></span>

</button>

<div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">

<ul class="navbar-nav flex-grow-1">

<li class="nav-item">

<**a** class="nav-link text-dark" **asp-controller**="Home" **asp-action**="Index">Home</**a**>

</li>

<li class="nav-item">

<**a** class="nav-link text-dark" **asp-controller**="Category" **asp-action**="Index">Category</**a**>

</li>

</ul>

</div>

</div>

</nav>

</header>

Since we are making changes inside the view, if we remember, when we created the project, we enabled the razor runtime compilation, so what that should do is when we save here, and if we go back to our application and refresh it, it should load the changes because we are making changes directly inside the view but that does not seem to work as we can see. The reason behind that is we added the package of razor runtime compilation, Even though these are razor pages which we are using right now if we examine the **program.cs** file we are only adding controllers with views.

builder.Services.AddControllersWithViews();

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

builder.Configuration.GetConnectionString("DefaultConnection")

));

var app = builder.Build();

We will have to make one small change here on builder.Services, we will have to add razor pages and on that we will add razor runtime compilation.

builder.Services.AddControllersWithViews();

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

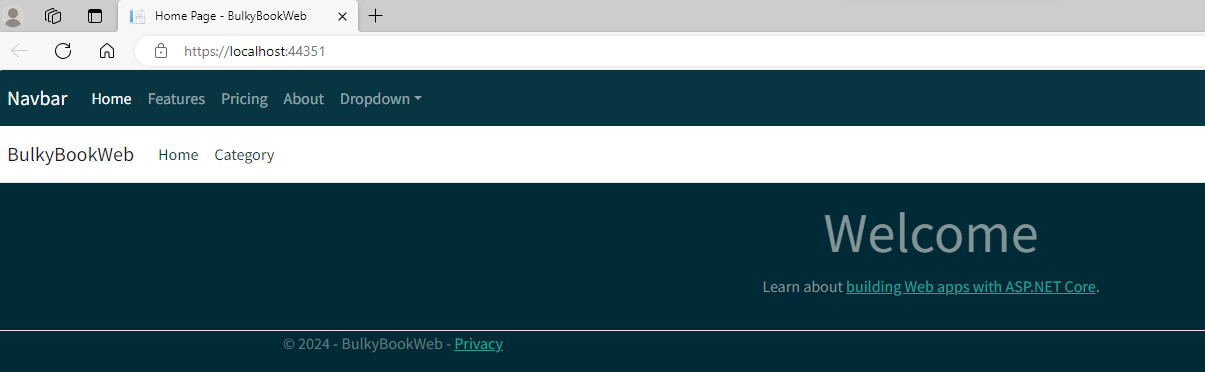
builder.Configuration.GetConnectionString("DefaultConnection")

));

builder.Services.AddRazorPages().AddRazorRuntimeCompilation();

var app = builder.Build();

So this one extra line that we have to add in an MVC project to get that runtime compilation. If we run the application we should see our new NavBar has been added.



Now, we don’t need all of the extra link above NavBar, we just want the Home and Category. This is the updated razor page that only displays Home and Category into our NavBar, we’ve added a small tweak to make our Index Category text color into white.

\_Layout.cshtml

<header>

<nav class="navbar navbar-expand-lg bg-dark" data-bs-theme="dark">

<div class="container-fluid">

<**a** class="navbar-brand" **asp-area**="" **asp-controller**="Home" **asp-action**="Index">BulkyBookWeb</**a**>

<button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarColor02" aria-controls="navbarColor02" aria-expanded="false" aria-label="Toggle navigation">

<span class="navbar-toggler-icon"></span>

</button>

<div class="collapse navbar-collapse" id="navbarColor02">

<ul class="navbar-nav me-auto">

<li class="nav-item">

<**a** class="nav-link" **asp-controller**="Home" **asp-action**="Index">Home</**a**>

</li>

<li class="nav-item">

<**a** class="nav-link" **asp-controller**="Category" **asp-action**="Index">Category</**a**>

</li>

</ul>

</div>

</div>

</nav>

</header>

Index.cshtml (Inside Views/Category) folder

@model IEnumerable<Category>

@{

ViewData["Title"] = "Index";

}

<h1>Index Category</h1>

<table class = "table table-bordered table-striped" style="width:100%">

<thead>

<tr>

<th>

<span style="color: white"> Category Name</span>

</th>

<th>

<span style="color: white"> Display Order</span>

</th>

</tr>

</thead>

<tbody>

@foreach (var obj in Model)// Model is our IEnumerable<Category> class above

{

<tr >

<td width="50%">

<span style="color: white">@obj.Name</span>

</td>

<td width="30%">

<span style="color: white">@obj.DisplayOrder</span>

</td>

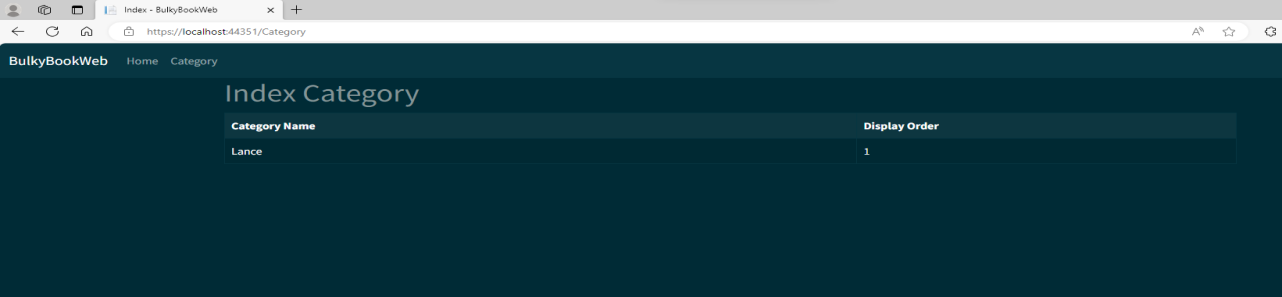
</tr>

}

</tbody>

</table>

Below we can see the updated design of our application.



So with this now we are using Bootstrap 5 in our project and we are using the bootswatch theme.

Inside our index view, what we can do is we can **add a button** to **create a category**. So if we go back to our **index inside our views category Index** change our header **<h1>Index Category</h1>**, so that it will be ready once we add a button above our tables. This code below can create right at the top of our tables.

<div class="container p-3">

<div class="row pt-4">

test button

</div>

</div>

we want to divide to divide this row into two parts. By default bootstrap divides each row into 12 parts if we want. But if we just want to divide it into two parts, then we will have **six** in one and **six** in the other.

|  |  |
| --- | --- |
| 6 here | 6 here |

Our table for data.

So here, we will add a div, give it a class of column six, that will combine the first six in one div, and we will add another div, give it a class of column six. So that will divide the screen into two parts. The first one we want to display is heading, so h2 and give it a class of text primary, and we want to display Category List above, it should create a text above named “**Category List**”. So we’ll close that **div** at the **end** put the **</div>** at the very end of our **Index.cshtml** to arrange our **Category List** above. Next we want to add button, right here want to add a link inside, then we want to go to a new action method that we will create inside the CategoryController.cs that will be for creating New Category. So above here, we will say that should go to the same controller, which is the category controller, and it should go to an action method, which will be Create. The create action method we don’t have it for the meantime, but we will create that next. We can also add couple of bootstrap classes button **btn-primary** , and above we want to name it **Create New Category** now we have our button. Now we’ll align this button on the right hand side, in bootstrap five we have **text-end** we will save that and that will bring it to the right side. Now to add spacing below our table add <br /><br /> which is line break to add space below our table rows. Code below.

@model IEnumerable<Category>

@{

ViewData["Title"] = "Index";

}

<div class="container p-3">

<div class="row pt-4">

<div class ="col-6">

<h2 class="text-primary">Category List</h2>

</div>

<div class="col-6 text-end">

<**a** **asp-controller**="Category" **asp-action**="Create" class="btn btn-primary">

Create New Category

</**a**>

</div>

</div>

<br /><br />

<table class = "table table-bordered table-striped" style="width:100%">

<thead>

<tr>

<th>

<span style="color: white"> Category Name</span>

</th>

<th>

<span style="color: white"> Display Order</span>

</th>

</tr>

</thead>

<tbody>

@foreach (var obj in Model)// Model is our IEnumerable<Category> class above

{

<tr >

<td width="50%">

<span style="color: white">@obj.Name</span>

</td>

<td width="30%">

<span style="color: white">@obj.DisplayOrder</span>

</td>

</tr>

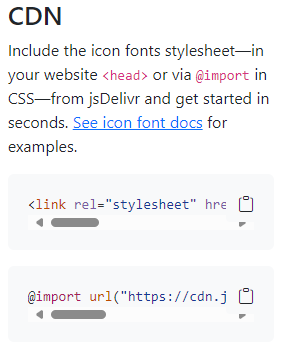
}

</tbody>

</table>

</div>

Now we also want to use some icons in our button Create New Category like font awesome. But with bootstrap five, they have given their own sets of icons. So why don’t we use that? If we go to get **getbootstrap.com**, we have the **Icons** tab, getting started with all of these icons is super simple we use like font awesome, but we had to register and do lots of thing. This is available quickly, we just click the link inside Install , and we have the CDN we will copy that



and go to **\_Layout.cshtml** above we will paste the CDN.

<!DOCTYPE html>

<html lang="en">

<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/bootswatchTheme.css" **asp-append-version**="true" />

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

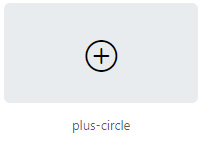
<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap-icons@1.11.3/font/bootstrap-icons.min.css">

</head>

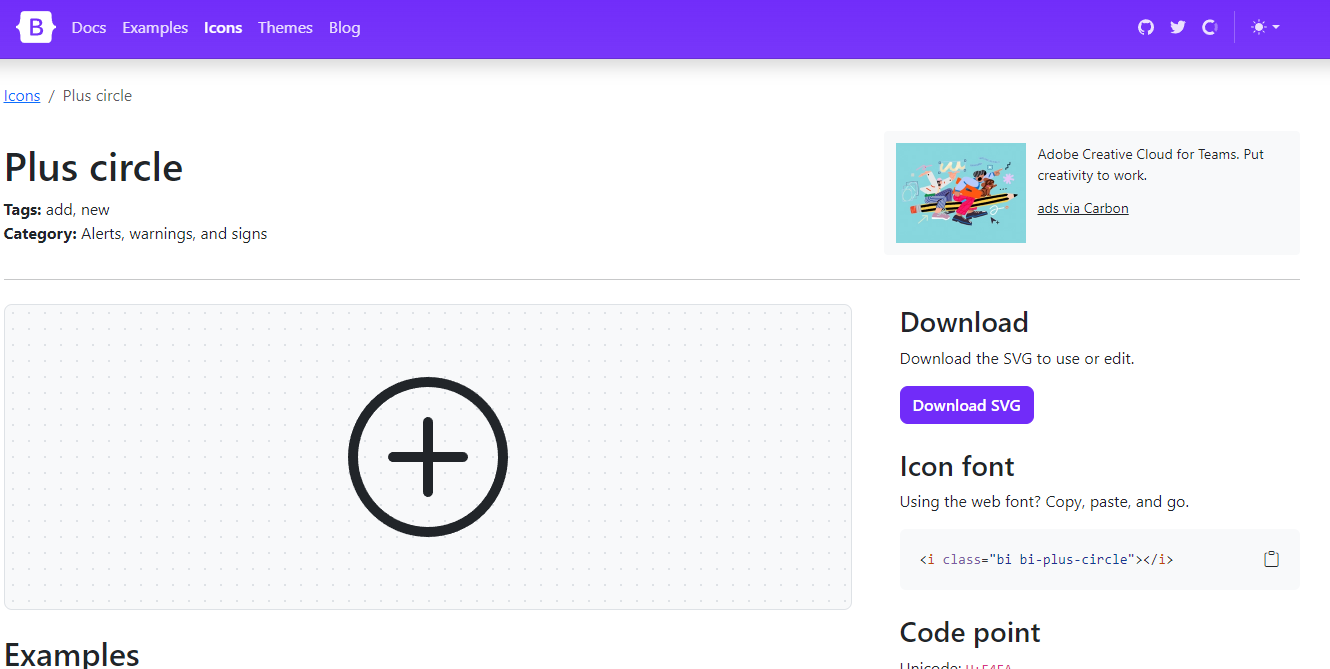
With that the **icons** are ready and available to us. Let’s go back to the bootstrap, and right on the search tab we can search icon for add, example: “**+**”.



Let’s search for plus, if we scroll down we have this plus circle.



Now click that and it will redirect us to the link of that **plus circle**.



We can copy that link and we will paste before **Create New Category** lets just add a non-breaking space by typing **&nbsp**; and save that, now we should see the icon is working perfectly.

@model IEnumerable<Category>

@{

ViewData["Title"] = "Index";

}

<div class="container p-3">

<div class="row pt-4">

<div class ="col-6">

<h2 class="text-primary">Category List</h2>

</div>

<div class="col-6 text-end">

<**a** **asp-controller**="Category" **asp-action**="Create" class="btn btn-primary">

<i class="bi bi-plus-circle"></i> &nbsp; Create New Category

</**a**>

</div>

</div>

<br /><br />

<table class = "table table-bordered table-striped" style="width:100%">

<thead>

<tr>

<th>

<span style="color: white"> Category Name</span>

</th>

<th>

<span style="color: white"> Display Order</span>

</th>

</tr>

</thead>

<tbody>

@foreach (var obj in Model)// Model is our IEnumerable<Category> class above

{

<tr >

<td width="50%">

<span style="color: white">@obj.Name</span>

</td>

<td width="30%">

<span style="color: white">@obj.DisplayOrder</span>

</td>

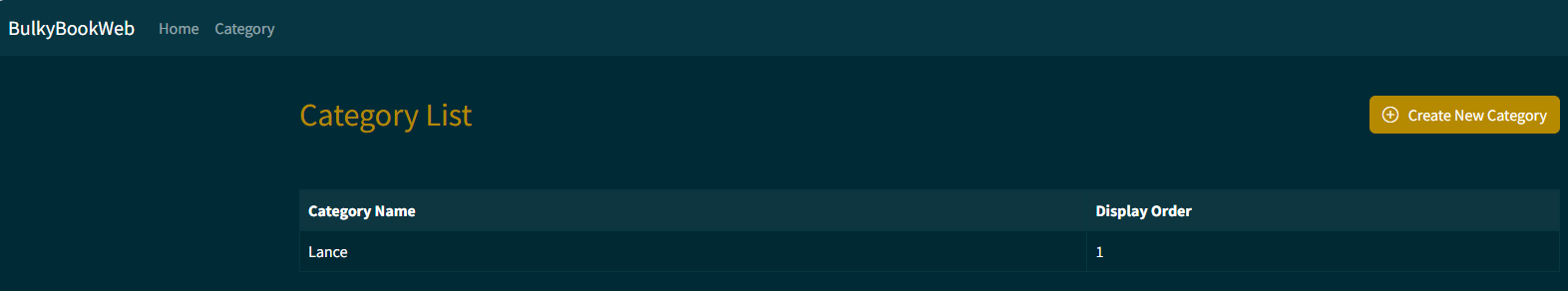
</tr>

}

</tbody>

</table>

</div>



With that we **have added icons**, but if we click the **Create New Category** we will see the **Page Not Found error**. that is because it is going inside our CategoryController action method. But inside category, there is no action method with the name of create. So let’s add that next. Inside our controller name **CategoryController.cs**, we can copy our action method Index and past it below one more time, then change the name here to **Create** and this will be a **GET** action method.

**Creating the Create function and Create View to save data into database.**

Now when someone hits the create button, we will give them the option to enter their name and display order and create a category. So when the View is being loaded you do not have to pass any model. We can keep that blank and then we can create a model directly inside the View. What do I mean by that? Let’s create a View here. By right clicking **Create**

using BulkyBookWeb.Data;

using BulkyBookWeb.Models;

using Microsoft.AspNetCore.Mvc;

namespace BulkyBookWeb.Controllers

{

public class CategoryController : Controller

{

private readonly ApplicationDBContext \_db;

public CategoryController(ApplicationDBContext db)

{

\_db = db;

}

public IActionResult Index()

{

IEnumerable<Category> objCategoryList = \_db.Categories.ToList();

return View(objCategoryList);

}

public IActionResult Create()

{

return View();

}

}

}

Then next, select **Add View/Razor View** well keep that empty, then **Add,** Now previously when we were working with Index, we said that this is the model that will passed from controller

@model IEnumerable<Category> this is found in our **Views/Category/Index.cshtml**. That is true, but it is not always the case. In our Create action method, we have not passing anything from our controller, but we still want to work on the **Category Model** and we want to **fetch its properties** when we **submit the form**. So for that also, we will have a **@model** for this View, that model will be the category model “**@model Category**”. So that model that we write inside **View** is not always the model that is passed from the **Controller**, but if we are passing a **Model** from the **Controller**, it must match what we have inside the **model to the View**. But if we are not passing anything inside the **controller**, like in **Create** above, then we can find our **View** with a model based on the data that we are collecting on the page. Here we are collecting the **Name** and **DisplayOrder**, we will use **Category** so that we can use **Tag Helpers** to find everything for us. Now, we might be thinking, what is this new term **Tag Helpers**? But before that, we’ll **create a form inside Create.cshtml** and give it a **method** of **post**, because we will be **posting our data**, because we want to **create** our **category**. In there we will create a **<div>**, and give it a **class of border padding of three** and **margin top of four**, in there, we will add a **<div>** and give it a **class of row** and **padding bottom of 2**, and we will add a **heading**, which will have a class of **text-primary** for the **yellow color**. And we will **display** the **heading** which is **Create Category**. Let’s add an **<hr />** , save this and **run the project**. The **<hr/>** tag defines a **thematic break** in an **HTML** page. This code is inside our **Views/Category/Create.cshtml**

@model Category

<**form** method="post">

<div class="border p-3 mt-4">

<div class="row pb-2">

<h2 class="text-primary">Create Category</h2>

<hr/>

</div>

</div>

</**form**>

So if we go to our Category now, and if we hit the Create Category Button, it will take us to the Create Category View that we created. Inside this Create action method we want to display a label and a text box. So outside of this tab we will add another <div> give it a class of margin bottom 3 in there, we want to display a label, you can just do label and give it a name like this Name that will work, But what we want to do is we want to bind everything on this page with our Category Model. Because of that, the .NET core team has provided Tag Helpers, which starts with “asp”, so here we have “asp-for”. When we add that in a label, we can use any of the **properties** from our **Model**. So we can see it is already displaying **Created Date Time**, **Display Order**, **Name**, and **ID**, we will select **Name** here, and we do not have to populate anything else. If we go back and if we refresh our application the output will be seen. Because what it does here is it **displays** the name of the **property** inside our **Category Model**. So we have it as **Name**, so that is what it will be displayed in the **UI** as well. Let’s go back to the **Create View**, and now we want an **input field**, so we will say input, and we will just say “**asp-for**” **Tag Helper**, this input is for name. That way it will **do all the binding**, and when this form is posted, it will **post an object of this category class** (**@model Category**) with the name populated, we can give it a bootstrap class of form-control. Once we refresh we can already see that **text box**. Now we add a text box from another field that will display **Display Order**, we can just copy the input type above for Name and modify it to **Display Order**. Now let’s add a **button** to actually **submit the form** and we’ll add a link to go back to our index page of the **category controller**. So we will **create a button** with the type **submit**, we will also give it some bootstrap classes of button btn-primary and that will give it a width of 150, now we’ll name the button as **Create**. Next well add a link below, this link we want to go back to the **Index Action** method inside the **category controller** to load all of the **categories**. It is more like a **back button link**. So first, what is the **controller** name, that is the **category controlle**r and what is the action there that is the **index action**, and we will give it some bootstrap class button btn-secondary, and we will also give it a style with the width of 150 pixels and name it **Back to Lis**t. When we click the **back to list** we can see that it is functional.

@model Category

<**form** method="post">

<div class="border p-3 mt-4">

<div class="row pb-2">

<h2 class="text-primary">Create Category</h2>

<hr/>

</div>

<div class="mb-3">

<**label** **asp-for**="Name"></**label**>

<**input** **asp-for**="Name" class="form-control" />

</div>

<div class="mb-3">

<**label** **asp-for**="DisplayOrder"></**label**>

<**input** **asp-for**="DisplayOrder" class="form-control" />

</div>

<button type="submit" class="btn btn-primary" style="width:150px">Create</button>

<**a** **asp-controller**="Category" **asp-action**="Index" class="btn btn-secondary" style="width:150px">

Back to List

</**a**>

</div>

</**form**>

Now, we want to **create** the **category** when we populate the details and hit the **Create button**. So when we hit the **create button**, we will have to create a **post** action method inside the **category controller**. And in that **post action method**, we will already be **fetching the category object** that is populated because we used **Tag Helpers**. So inside the **category Controller**, we can copy this **Create()** and paste **that below one more time**, just stop the application first. Inside the parameters will be receiving a **category object**, let us call that obj “**Category obj**” this will be **post action** method. Now if an **action method** is **post**, we have to write the attribute **[HttpPost]**, on the top of that. And then we also have to validate anti forgery token **[ValidateAntiForgeryToken]**, the **anti forgery token** is there to help and **prevent the cross site request forgery attack**. What it basically does is inside any forms that we have inside the application, it will automatically **inject a key** there, and that key will be validated at this step **[ValidateAntiForgeryToken]** this **validate that token** on all of the **post requests** . That **key must be valid** to **prevent cross site request forgery**. This is **not required**, but will **highly recommended** that to avoid the cross site request forgery. Now once we have the **category object** that is populated with the **Name** and **Display Order**, we want to **create that record inside the database**. So to do that we will use our **DbContext** and on there we have the **Categories**, then in order to **retrieve categories**, we did not have to write anything else. But when we have to add something to the table, we have to add method there, and we can see it expect a **category entity**. We already have that inside the “**obj”** that is what the user populated. So we will add that to our database. Now once we add it to the database, it is **not pushed to the database right now**. It will be pushed to the **database**, when we run the command **\_db.SaveChanges();**, at that point it goes to the database and **saves all the changes**. Once the changes are **save**, we have **return View();** here that will take us back to the same **Category View**. Let’s see when we are done, let’s go back to the Index so that we can see the **New Category** that was created. So rather than **return view**, we want to **redirect to an action**, we want to **redirect** to the **index action**, it will look for this **index** inside the same controller. If we had to go to an action method in some other controller, we can define the controller name right here

“**return RedirectToAction(“Index”,””)**;. But since we are in the same controller, we can just mention Index and that will work. Now try to run our application and try the “**Create**” button and it should work. Code in triggering the create button below:

using BulkyBookWeb.Data;

using BulkyBookWeb.Models;

using Microsoft.AspNetCore.Mvc;

namespace BulkyBookWeb.Controllers

{

public class CategoryController : Controller

{

private readonly ApplicationDBContext \_db;

public CategoryController(ApplicationDBContext db)

{

\_db = db;

}

public IActionResult Index()

{

IEnumerable<Category> objCategoryList = \_db.Categories.ToList();

return View(objCategoryList);

}

//GET

public IActionResult Create()

{

return View();

}

//POST

[HttpPost]

[ValidateAntiForgeryToken]

public IActionResult Create(Category obj)

{

\_db.Categories.Add(obj);

\_db.SaveChanges();

return RedirectToAction("Index");

}

}

}

This is working as expected and we are **able to create a new Category now**.

**Creating Server and Client side validations.**

But our create category is not perfect, because when we click create with an **empty text boxes** we will **run into an exception**, it says the exception **cannot Insert Null** means **empty**. We do not have any **validations**, but inside database, we said that this was not a **NULLABLE** column, so when the **Entity Framework** code tries to save, it gives the exception. We have to be much smarter to handle the validations. Now we should handle our validations inside the **server side**, which is **controller**, as well as inside **client side**, which is inside **Create View**. So how do we handle validations? Now let’s first do that at the **server side**. So when we receive a **Model** here, we can check whether this **Model is valid or not**, and what defines a valid model is that validations that we have right here. Name should be a required property.

This is inside Model folder named Category.cs Model

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;

}

}

Now lets go back to our **CategoryController.cs** and make sure that our model is valid. In order to check that with .NET Core, we have **model state is valid** that will determine if the model is valid or not. If that is valid, we want to Create and redirect to index, if that is not valid we just return back to the View with the object.

using BulkyBookWeb.Data;

using BulkyBookWeb.Models;

using Microsoft.AspNetCore.Mvc;

namespace BulkyBookWeb.Controllers

{

public class CategoryController : Controller

{

private readonly ApplicationDBContext \_db;

public CategoryController(ApplicationDBContext db)

{

\_db = db;

}

public IActionResult Index() // Displaying all the data in our database form Model Category

{

IEnumerable<Category> objCategoryList = \_db.Categories.ToList();

return View(objCategoryList);

}

//GET

public IActionResult Create() // For getting the view with input textboxes and getting Create and Back to list buttons.

{

return View();

}

//POST

[HttpPost]

[ValidateAntiForgeryToken]

public IActionResult Create(Category obj) //For Inserting data into database

{

if (ModelState.IsValid)

{

\_db.Categories.Add(obj);

\_db.SaveChanges();

return RedirectToAction("Index");

}

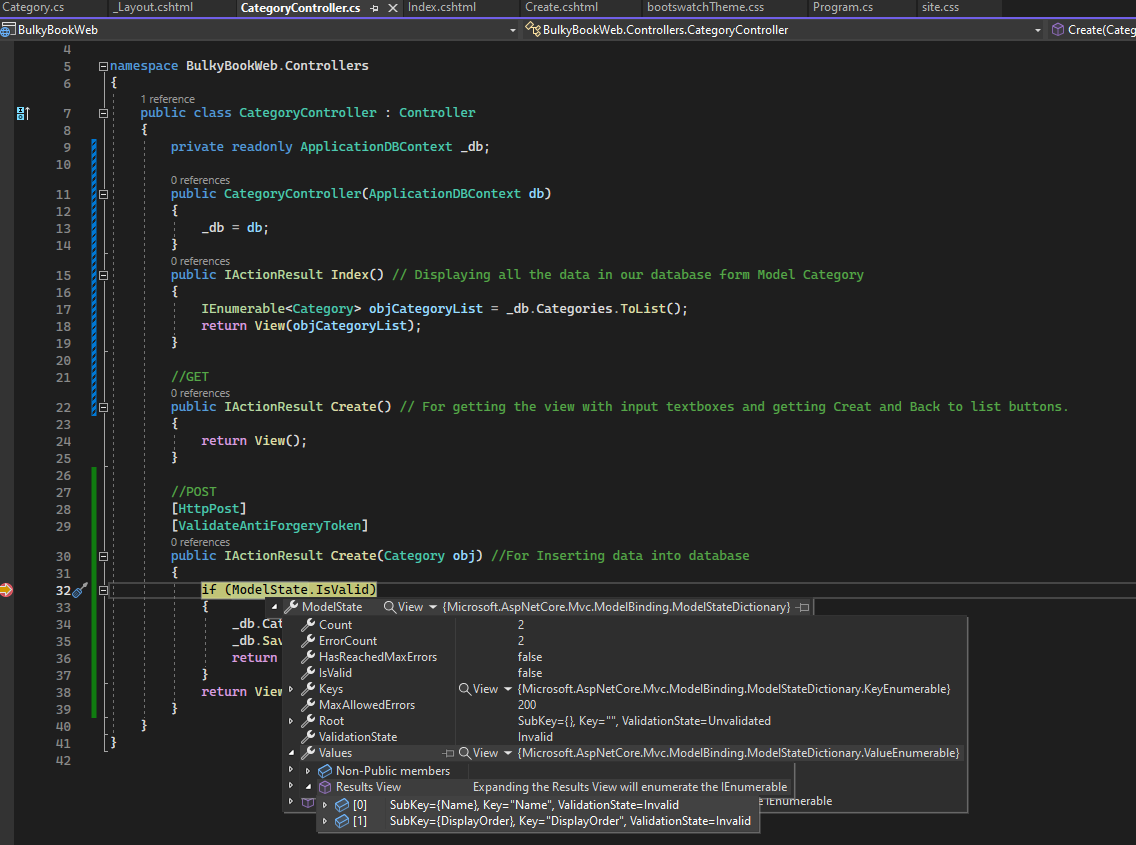
return View(obj);

}

}

}

Let’s run this configuration and create a break point in “**if (ModelState.IsValid)**”, now we will try to create a **category** with **empty data** inside text box, now it should hit our **break point**. When we examine the **is valid flag is false.** Means our **model stat is not valid**. But now which property in our Model state is not valid? It is important to know that because right now we just have two properties, it could be possible we have over 50 properties in a Model. So in that case, when we hover on **ModelState,** we expand that inside the error counts, and we can see we have two errors.



If we hover on the **Values** here, **Result View**, we will see which **properties are not valid**, we can see **Display order** is not valid as-well as the **Name**. Now let’s continue here, and this time let us populate Display Order and hit create button one more time, ModelState will still be not valid. But this time if we examine the **error count** is only one because we already fixed one of the errors by providing **Display Order**. So now we can see **Display Order is valid**, but name is still invalid because it is a required field. The server side validations are working. But with that it would be helpful if we are **displaying an error message right into our application**, so that user will know why it is failing. Doing that is super simple. We will **remove the break point** that we’ve just putted here, and back to the **Views/Category/Create.cshtml**. Just like we have “**asp**” for **Tag Helper**, we also have a **Tag Helper** for validation. So inside name, we can add a **span tag**, and we have the **Tag Helper** “**asp-validation**”, for this we have to define the **Model**. So we want to check validations **Name** here, and if there are any errors, we want to display them in **red color**. So we can give it a **bootstrap of class** of **text danger**, now copy this span we just create and we will paste that for **Display Order** as well. With that in place,make sure to **save this** and we will go back and refresh our application page. Now it should display an default error message once we leave the text box blank and click **Create** button.

@model Category

<**form** method="post">

<div class="border p-3 mt-4">

<div class="row pb-2">

<h2 class="text-primary">Create Category</h2>

<hr/>

</div>

<div class="mb-3">

<**label** **asp-for**="Name"></**label**>

<**input** **asp-for**="Name" class="form-control" />

<**span** **asp-validation-for**="Name" class="text-danger"></**span**>

</div>

<div class="mb-3">

<**label** **asp-for**="DisplayOrder"></**label**>

<**input** **asp-for**="DisplayOrder" class="form-control" />

<**span** **asp-validation-for**="DisplayOrder" class="text-danger"></**span**>

</div>

<button type="submit" class="btn btn-primary" style="width:150px">Create</button>

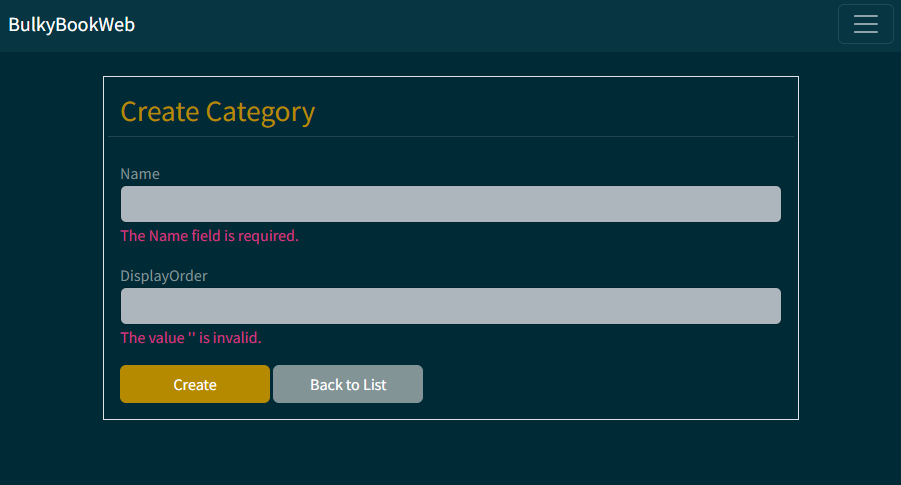
<**a** **asp-controller**="Category" **asp-action**="Index" class="btn btn-secondary" style="width:150px">

Back to List

</**a**>

</div>

</**form**>



So with that we have our **server side** validations in which it check at the controller level if the model state is valid or not. If it is not valid it populates the model state with **error message**. It returns back the **obj** and it will automatically displays the validations that we have added. We don’t need to write JavaScript or any other complex code. It is doing all of that, with the help of **Tag Helpers**. With server side validations, what if we want to display a summary at the top with all the error messages? If we want to do that .NET Core has a solution for that, we will need a <div> this time and not a <span> that we we’re using with asp-validation-for and inside <div></div> we can see we have a Tag Helper “asp-validation-summary”. This will be all in this case, it will display all the validations at the top in a summary format.

@model Category

<**form** method="post">

<div class="border p-3 mt-4">

<div class="row pb-2">

<h2 class="text-primary">Create Category</h2>

<hr/>

</div>

<**div** **asp-validation-summary**="All"></**div**>

<div class="mb-3">

<**label** **asp-for**="Name"></**label**>

<**input** **asp-for**="Name" class="form-control" />

<**span** **asp-validation-for**="Name" class="text-danger"></**span**>

</div>

<div class="mb-3">

<**label** **asp-for**="DisplayOrder"></**label**>

<**input** **asp-for**="DisplayOrder" class="form-control" />

<**span** **asp-validation-for**="DisplayOrder" class="text-danger"></**span**>

</div>

<button type="submit" class="btn btn-primary" style="width:150px">Create</button>

<**a** **asp-controller**="Category" **asp-action**="Index" class="btn btn-secondary" style="width:150px">

Back to List

</**a**>

</div>

</**form**>

Now run the application after we make this change, if we hit create with an empty values it will displays all the error message above it. When we select “All” it will display Model as well as properties. If we want to limit that we can change that to Model only and none, but if you are using Validation Summary, we recommend “All” and this is great. Now along with that, what if we want to add some custom validation? Like let’s say inside the **CategoryController.cs** here, we want to make sure that we do not add any Category which has the same Name and Display order. How can we add that to the Validation? So before we confirm if the ModelState is valid, we can check here if obj.Name is equal equal to obj.DisplayOrder, we will have to convert that to a string, if this is the same, we want to tell our ModelState that this is not valid. So for that on the ModelState itself, we can add custom model errors. Here we have a key and value pair, we can give any key name that we want, we will call this custom error. If we want we can also use string empty, but since this was a key, we have to make sure that we do not use the same key two times. Let us add a custom error that the Display Order cannot exactly match the name.

using BulkyBookWeb.Data;

using BulkyBookWeb.Models;

using Microsoft.AspNetCore.Mvc;

namespace BulkyBookWeb.Controllers

{

public class CategoryController : Controller

{

private readonly ApplicationDBContext \_db;

public CategoryController(ApplicationDBContext db)

{

\_db = db;

}

public IActionResult Index() // Displaying all the data in our database form Model Category

{

IEnumerable<Category> objCategoryList = \_db.Categories.ToList();

return View(objCategoryList);

}

//GET

public IActionResult Create() // For getting the view with input textboxes and getting Creat and Back to list buttons.

{

return View();

}

//POST

[HttpPost]

[ValidateAntiForgeryToken]

public IActionResult Create(Category obj) //For Inserting data into database

{

if(obj.Name == obj.DisplayOrder.ToString())

{

ModelState.AddModelError("CustomError","The DisplayOrder cannot exactly match the Name");

}

if (ModelState.IsValid)

{

\_db.Categories.Add(obj);

\_db.SaveChanges();

return RedirectToAction("Index");

}

return View(obj);

}

}

}

Now since we are making the change inside the **controller**, we will actually have to restart our application to see this changes. If we were working with the **View**, we did not need to restart the app. Now if we go to our Category and try to Create a Category with the same Name and Display Order. This time, we can see we have our custom validation that is being displayed in the Summary. Now if we did not have summary, then this would not have been displayed above. But what if we want to display the error message inside our name as well? We will stop the application, the key, we will just change that to a “**name**”, because inside the **Category.cs** Model, if we noticed we have Name. So inside the CategoryController, we are adding a new error inside the name property.

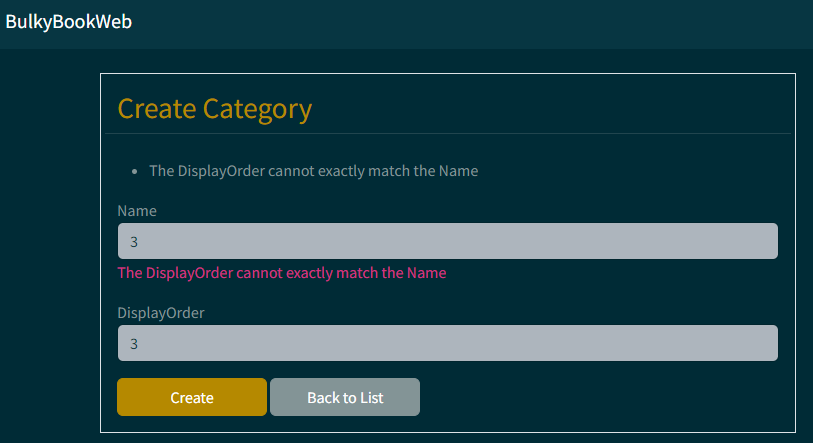
if(obj.Name == obj.DisplayOrder.ToString())

{

ModelState.AddModelError("name","The DisplayOrder cannot exactly match the Name");

}

Now let’s run this and give this a try.



Now we see it displays input cases. So with this we have seen how to add custom error message. But all of the validations that we have so far are done on the server side. Every time we hit the Create button, we will see that page will reload, so that means we are hitting server every time, even if the Name and DisplayOrder are empty. We go back and hit the server because page reloads. What if we want to make all of this validations on the client side? Now if we want to do the basic model validations like Name and value for the DisplayOrder on the client side it is pretty simple. If we examine inside the **Shared** folder, we have the **\_ValidationScriptsPartial.cshtml** that is using **jquery.validate**. It is inside the partial view. So if we want to do client side validations we just have to include this partial view into our **View/Category/Create.cshtml** . So that brings the big question, how do we use a partial view? We have not created one but we are using the one that is already available. It’s not a rule, but typically it’s good to name **Partial Views** with underscore. So even if in the future when we create a partial view, we will follow that convention. Now in order to include partial view it is pretty simple. We can directly say partial and then we have to write name of the View, so make sure to type the exact name, if we have a special mistake there, then this won’t be included. With that the errors go away. But inside the partial view, if we open up **\_ValidationScriptsPartial.cshtml** we have script files. So, we need to add that inside a script section of our View. So here we will have to create a section for Scripts, and inside there, we will have to add our Partial View, add that inside razor syntax.

@model Category

<**form** method="post">

<div class="border p-3 mt-4">

<div class="row pb-2">

<h2 class="text-primary">Create Category</h2>

<hr/>

</div>

<**div** **asp-validation-summary**="All"></**div**>

<div class="mb-3">

<**label** **asp-for**="Name"></**label**>

<**input** **asp-for**="Name" class="form-control" />

<**span** **asp-validation-for**="Name" class="text-danger"></**span**>

</div>

<div class="mb-3">

<**label** **asp-for**="DisplayOrder"></**label**>

<**input** **asp-for**="DisplayOrder" class="form-control" />

<**span** **asp-validation-for**="DisplayOrder" class="text-danger"></**span**>

</div>

<button type="submit" class="btn btn-primary" style="width:150px">Create</button>

<**a** **asp-controller**="Category" **asp-action**="Index" class="btn btn-secondary" style="width:150px">

Back to List

</**a**>

</div>

</**form**>

@section Scripts{

@{

<**partial** **name**="\_ValidationScriptsPartial"/>

}

}

A **partial view** should be used when you need to display the similar information on multiple views. Now once we test our configurations, we see, we get the validations, but we are not going to server this time (like every time we hit the Create button our page will reload). All the validations are done on client side, because of the script that we included. To double check, we can just add a debugging point here if(obj.Name == obj.DisplayOrder.ToString()) this is inside our **CategoryController.cs**, and if we type the same value in the **Name** and **Display Order** it should hit our debugging point, then the client side validations are valid. The validation that fails is the server side custom validation that we’ve added. Now that we’ve added the client side validations, one more thing, right now the default message that is being displayed, right now we have the default message that being displayed. On top of that we can see right now in Display Order, there is no space that is available, and it is the same with the validation. The reason behind this is if we go to our Category Model, we can see that’s the property name “DisplayOrder” this is the one that being displayed. So if we want to display something that is not the same as property name, we have **[DisplayName]** here which is inside component model. Inside this Data annotation we can just give name whatever name we want to display like this **[DisplayName(“Display Order”)].**

using System.ComponentModel;

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; }

[DisplayName("Display Order")]

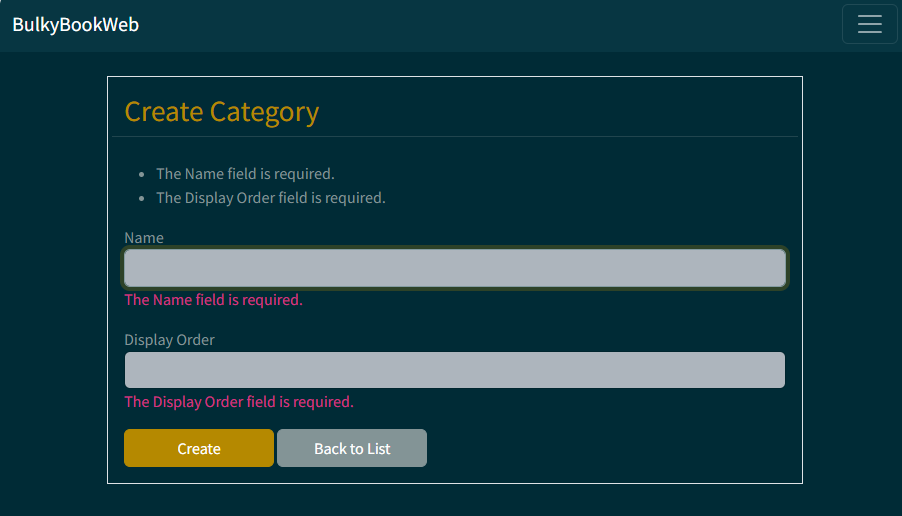
public int DisplayOrder { get; set; }

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

}

}

Now let’s give this a try by testing this. It is now giving us some space displayed in the error message.



We can check different Data Annotations documentations at **learn.microsoft.com** and search **System.ComponentMode.DataAnnotations Namespace**.Now let’s try the **range attribute** that specifies range for field. Inside our Category Model, Display Order, we will add a range attribute, and here we got minimum and maximum value so let’s say that the minimum is 1 and maximum of 100 **[Range(1, 100)]**. So now, if the display order is not in the range, it will give an error message. But the error message that we see on the screen with Validations are also customizable. So with range, if we want some other error message, we have that property right here, and we can give a custom error message that will have our custom error message as “display order must be between 1 and 100only!”

**[Range(1, 100,ErrorMessage = “Display Order must be between 1 and 100 only!”)].**

using System.ComponentModel;

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; }

[DisplayName("Display Order")]

[Range(1,100,ErrorMessage = "Display Order must be between 1 and 100 only!")]

public int DisplayOrder { get; set; }

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

}

}

Now, once we add into Display Order text box in more than 100 of value it will display an error message we’ve enter above. It directly displays our custom validation, and we saw how to add custom error messages, range validation and how to use display name. Now inside our Create View, we will comment the Validation Summary because, the inline validation is sufficient @\* <div asp-validation-summary="All"></div> \*@.

**Creating an Edit action method and Edit View.**

Now that we had our create functionality working as expected, it’s time to work on the added functionality. For Edit we will go to our **CategoryController.cs** and just like we have the **Create GET** and **Create Post**, we will copy that and paste it one more time. This time, we’ll call them for **Edit(),** change the action method names to Edit(). One thing that will be different for edit is when the page is loaded, it will display the existing functionality of the Category that was selected. Here we will retrieve an integer which will be **id**. Based on that **id** we have to retrieve that Category Details and display them. So, we can check here if **id** is NULL or if **id** is equal to zero, in those cases, we will return back not found. Because that is an invalid **id.** If that is not the case, we will retrieve the Category from database or **categoryFromDb**, and we will extract that using our **\_db.Categories**. Now this will retrieve complete list of all the tables. So on that we have few more ways of finding one of the entity. First is **SingleOrDefault**, which returns only one element every time, it will throw an exception if there is more than one element, id is a primary key, so there is no chance that there will be more than one element. Similarly we have **Single**, the difference between **Single** and **SingleOrDefault** is **Single** will throw an exception if no elements are found for that **id,** but **SingleOrDefault** will just return empty if there are no elements. So **SingleOrDefault** will not throw an exception if the element was not found. Similar to **SingleOrDefault** we also have **FirstOrDefault**, the difference between **SingleOrDefault, Single, and FirstOrDefault,** is that **SingleOrDefault** will throw and exception if there are more than one elements. **FirstOrDefault** will not throw an exception, it will return the first element of the list. So if more than one elements are found, **FirstOrDefault** will return one element singular default will throw an exception. Last but not the least, we have one method which is **Find,** find basically tries to find that based on the **primary key** of the table, we know that **id** is the primary key, so we can use **Find** here and pass the **id.** It will find the Category based on that **id** and assign that to our variable. Let us show also how we can use the **FirstOrDefault,** with **Find** you just had to pass the **id “\_db.Categories.Find(id);”,**  with **FirstOrDefault ,**we have to pass an expression here “**\_db.Categories.FirstOrDefault (u => u.Id==id);**” where we will have a generic object, let’s call that **u** and we will say **u goes to u.id** should be equal to the **id** that we have, if that matches in any of the **Categories** row, retrieve that and give me the first one out of them. There is also the “**\_db.Categories.SingleOrDefault (u => u.Id==id);”.** So these are three ways to retrieve a category from database based on the **id.** We will comment two of them out, but we want to test how that could be done using Entity Framework Core. Now once we retrieve the object, we will check here if category from **db** if that is NULL that means again, we will return not found. If we find that category,we will return that category to the View.

This is inside Controllers folder named CategoryController.cs Below our Create() action method.

//GET (UPDATING SECTION)

public IActionResult Edit(int? id) // For getting the view with input textboxes and getting Creat and Back to list buttons.

{

if(id == null || id == 0 )

{

return NotFound();

}

var categoryFromDb = \_db.Categories.Find(id);

//var categoryFromDbFirst = \_db.Categories.FirstOrDefault(u => u.Id == id);

//var categoryFromDbSingle = \_db.Categories.SingleOrDefault(u => u.Id == id);

if(categoryFromDb == null)

{

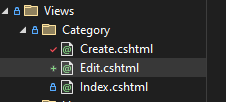
return NotFound();

}

return View(categoryFromDb);

}

So next what we will do is we will create this View which will have the category loaded, and we just need to display that. That View will look exactly same as the **Create View**. But the only thing different will be that it will be for **Edit Page.** So we can copy that View that we have for **Create.cshtml** Inside **Views/Category** Folder and paste that one more time and rename this to **Edit.cshtml** to give us a head start.



Let’s open that **Edit** View,our model will be the Category Model, but rather than Create Category, this will be **Edit Category.** Now when we edit the category, we want the validations. So that will stay the same. The button text will change that from Create to **Update,** and it is just submitting back. We do not have which action method it should submit to. If we do not provide any name here, it will by default submit to the same action method that is of the name of the GET. So since the GET is Edit() it will post on the same method which is Edit. But if we want to be explicit about the action, we can use that with the Tag Helper “asp-action” and we can write edit like that **asp-action=”Edit”**. So if we want we can be explicit, but that is not required, the backdoor list will stay the same.

This is inside Views/Category folder named Edit.cshtml.

@model Category

<**form** method="post" **asp-action**="Edit">

<div class="border p-3 mt-4">

<div class="row pb-2">

<h2 class="text-primary">Edit Category</h2>

<hr/>

</div>

@\* <div asp-validation-summary="All"></div> \*@

<div class="mb-3">

<**label** **asp-for**="Name"></**label**>

<**input** **asp-for**="Name" class="form-control" />

<**span** **asp-validation-for**="Name" class="text-danger"></**span**>

</div>

<div class="mb-3">

<**label** **asp-for**="DisplayOrder"></**label**>

<**input** **asp-for**="DisplayOrder" class="form-control" />

<**span** **asp-validation-for**="DisplayOrder" class="text-danger"></**span**>

</div>

<button type="submit" class="btn btn-primary" style="width:150px">Update</button>

<**a** **asp-controller**="Category" **asp-action**="Index" class="btn btn-secondary" style="width:150px">

Back to List

</**a**>

</div>

</**form**>

@section Scripts{

@{

<**partial** **name**="\_ValidationScriptsPartial"/>

}

}

Let’s go to the **Index.cshtml** View of this **Category** where we are displaying everything, we want to add a link to navigate to the **Edit Action method**. That will be three columns here, let us add another <th> tag, and inside the <td>, we will have one with the links. So inside this <td>, we can add a div with a class of width 75 and button group, we can also give that a role of group in there, let us add an anchor tag and we will use Tag Helpers inside this anchor tag. This will be for **Edit,** and if we go back to the site **icons.getbootstrap.com** and let’s try to find an icon for edit, we have this pencil square, copy the link at the right hand side and paste that here <a>**<here>**Edit</>. This Edit should take us to CategoryController and Edit action method, so if we go back to our Index.cshtml we want our “asp-controller” has to go to our Category controller and “asp-action” should go to Edit action method. But if we go take a look at our Edit action method, it also expects an **id** “**public IActionResult Edit(int? Id)”,** so we have to pass that **id** with our link. We can pass custom variable and say “**asp-route-**” with the name of the variable **id** and this will have to bind with our object, so we’ll write this code

**asp-route-id=”@obj.Id”.** We can see how dynamic we can get with the razor pages.

@model IEnumerable<Category>

@{

ViewData["Title"] = "Index";

}

<div class="container p-3">

<div class="row pt-4">

<div class ="col-6">

<h2 class="text-primary">Category List</h2>

</div>

<div class="col-6 text-end">

<**a** **asp-controller**="Category" **asp-action**="Create" class="btn btn-primary">

<i class="bi bi-plus-circle"></i> &nbsp; Create New Category

</**a**>

</div>

</div>

<br /><br />

<table class = "table table-bordered table-striped" style="width:100%">

<thead>

<tr>

<th>

<span style="color: white"> Category Name</span>

</th>

<th>

<span style="color: white"> Display Order</span>

</th>

<th></th>

</tr>

</thead>

<tbody>

@foreach (var obj in Model)// Model is our IEnumerable<Category> class above

{

<tr >

<td width="50%">

<span style="color: white">@obj.Name</span>

</td>

<td width="30%">

<span style="color: white">@obj.DisplayOrder</span>

</td>

<td>

<div class="w-75 btn-group" role="group">

<**a** **asp-controller**="Category" **asp-action**="Edit" **asp-route-id**="@obj.Id"

class="btn btn-primary mx-2"><i class="bi bi-pencil-square"></i>Edit</**a**>

</div>

</td>

</tr>

}

</tbody>

</table>

</div>

Let’s go back and refresh, then go to Category, we should see the Edit Button. We can see if we hove on Edit button, we can see the URL in lower left hand where the ID is changing. If we click Edit here, it will take us to Edit Category, and it automatically populates the Name and Display Order. Our Back to List is also working as expected. Now to make this button look pretty, we can just add two CSS classes button btn-primary and some margin, when we refresh our button will look better.



When the user hits the Update button, it will go to category controller, we have the POST action method for Edit. We can have the same validation that we had there. But if all the validations are valid, we do not have to call the **.Add(obj)**, and we want to call the **.Update(obj)** method on our Categories DbSet..**Update();** is a built in method with EF Core where based on the primary key, it will automatically update all of the properties, and we do not have to do any manual Update, it will take a look at the **obj** right here and it will find its primary key retrieve that from the database, it will check all the properties where the values have changed, and it will update all of those. So we can see it is handling all the complexity, and we just need an update statement to update any of the category based on that **id.** After that, make sure that we are saving the changes and we are redirecting back to the Index Action. If we examine here, we can see all the options that are available with our **DbSet**, we have **Add, Remove, AddAsync, AddRange** that will add multiple objects at the same time, we have **Update, and much more**. For this time, well select **Update**, and every time we made changes into our controller we need to restart our application. Now the update should work successfully.

This is inside Controllers folder named CategoryController.cs Below our Edit() GET action method.

//POST

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Edit(Category obj) //For Inserting data into database

{

if (obj.Name == obj.DisplayOrder.ToString())

{

ModelState.AddModelError("name", "The Display Order cannot exactly match the Name");

}

if (ModelState.IsValid)

{

\_db.Categories.Update(obj);

await \_db.SaveChangesAsync();

return RedirectToAction("Index");

}

return View(obj);

}

Our validations also working with Edit, because if we leave the text boxes empty it will show us an error message once we click the update button.

**Creating an Delete action method.**

Now in Delete Category we don’t want validations because we want to make sure that the text boxes are disabled.

The last functionality we want to work on is the Delete functionality inside category. So we will copy Edit as is, and we will paste that one more time below we still need the GET and the POST. Change the GET to **Delete()** same as below for the POST action method but this time we will name it **DeletePOST().** When we are deleting a category, we want to retrieve that category, and we want to display that to the **View** so it will stay the same with the **GET** action method what we have earlier in **Edit**. Now in the post action method we will pass the id

**public async Task<IActionResult> DeletePOST(int? id).** We don’t need inside any ModelState validation here because all we have to do is based on the **id** we have to retrieve the category, so we’ll say **var obj =** **\_db.Categories.Find(id);** based on the **id** we will find that if this is **NULL** we will

**return NotFound();.** Else on categories below we have remove and we will remove the object, next save the changes and redirect back to the Index page.

//Deleting Section

//GET(Once we click the Edit link it will populate the textboxes Name and DisplayOrder)

public async Task<IActionResult> Delete(int? id) // For getting the view with input textboxes and getting Create and Back to list button

{

if (id == null || id == 0)

{

return NotFound();

}

var categoryFromDb = await \_db.Categories.FindAsync(id);

//var categoryFromDbFirst = \_db.Categories.FirstOrDefault(u => u.Id == id);

//var categoryFromDbSingle = \_db.Categories.SingleOrDefault(u => u.Id == id);

if (categoryFromDb == null)

{

return NotFound();

}

return View(categoryFromDb);

}

//POST

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> DeletePOST(int? id) //For Inserting data into database

{

var obj = await \_db.Categories.FindAsync(id);

if (obj == null)

{

return NotFound();

}

\_db.Categories.Remove(obj);

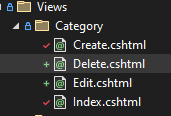
await \_db.SaveChangesAsync();

return RedirectToAction("Index");

}

Now the Delete View is similar and almost the same to the Edit View but without validations and the

text box are disabled code below:



@model Category

<**form** method="post" **asp-action**="DeletePOST">

<**input** **asp-for**="Id" hidden />

<div class="border p-3 mt-4">

<div class="row pb-2">

<h2 class="text-primary">Delete Category</h2>

<hr/>

</div>

@\* <div asp-validation-summary="All"></div> \*@

<div class="mb-3">

<**label** **asp-for**="Name"></**label**>

<**input** **asp-for**="Name" class="form-control" disabled/>

</div>

<div class="mb-3">

<**label** **asp-for**="DisplayOrder"></**label**>

<**input** **asp-for**="DisplayOrder" class="form-control" disabled />

</div>

<button type="submit" class="btn btn-danger" style="width:150px">Delete</button>

<**a** **asp-controller**="Category" **asp-action**="Index" class="btn btn-secondary" style="width:150px">

Back to List

</**a**>

</div>

</**form**>

@section Scripts{

@{

<**partial** **name**="\_ValidationScriptsPartial"/>

}

}

Since we are deleting the button must be **button-danger**. When running the delete function we will test it by adding a debugging point inside our controller, where we find that DeletePOST action Method. Now always remember that we need to put the Delete button displayed into our Index View.

@foreach (var obj in Model)// Model is our IEnumerable<Category> class above

{

<tr >

<td width="50%">

<span style="color: white">@obj.Name</span>

</td>

<td width="30%">

<span style="color: white">@obj.DisplayOrder</span>

</td>

<td>

<div class="w-90 btn-group" role="group">

<**a** **asp-controller**="Category" **asp-action**="Edit" **asp-route-id**="@obj.Id"

class="btn btn-primary mx-2"><i class="bi bi-pencil-square"></i>Edit</**a**>

<**a** **asp-controller**="Category" **asp-action**="Delete" **asp-route-id**="@obj.Id"

class="btn btn-danger mx-2"><i class="bi bi-trash"></i>Delete</**a**>

</div>

</td>

</tr>

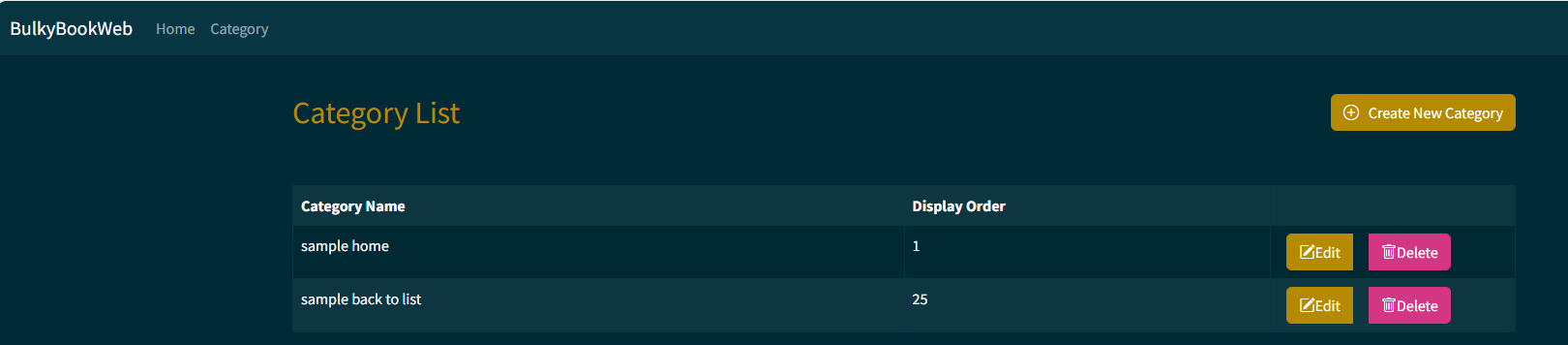
}

</tbody>

</table>

</div>

As we can notice above it is similar to our Edit button, but we set its button into **button-danger** which will change the color of our button to **red**.



Now once we hit that Delete button it will take us to our Delete Category View, and once we click the Delete button inside it will Delete the data and takes us back to the Index View. One last thing, right now inside our **Delete.cshtml** View we have the action above as DeletePOST

<**form** method="post" **asp-action**="DeletePOST">

If we change that to **Delete** it won’t work because inside our CategoryController.cs we have set it to DeletePOST in the POST action method.

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> DeletePOST(int? id) //For Inserting data into database

{

var obj = await \_db.Categories.FindAsync(id);

if (obj == null)

{

return NotFound();

}

\_db.Categories.Remove(obj);

await \_db.SaveChangesAsync();

return RedirectToAction("Index");

But if we want it to explicitly give that a different name, we have an ActionName attribute here and we can give it a delete. That way the controller will know that if a request comes in for Delete action method HTTP POST inside the CategoryController.cs, that is this particular action method.

This is inside Controllers folder named CategoryController.cs the POST action method named DeletePOST.

[HttpPost, ActionName(“Delete”)]

[ValidateAntiForgeryToken]

public async Task<IActionResult> DeletePOST(int? id) //For deleting into database.

{

var obj = await \_db.Categories.FindAsync(id);

if (obj == null)

{

return NotFound();

}

\_db.Categories.Remove(obj);

await \_db.SaveChangesAsync();

return RedirectToAction("Index");

This is inside our Delete View named Delete.cshtml

@model Category

<**form** method="post" **asp-action**="Delete"> //Change the DeletePOST into Delete and called into our CategoryController by expliciting it [HttpPost, ActionName(“Delete”)]

<**input** **asp-for**="Id" hidden />

<div class="border p-3 mt-4">

<div class="row pb-2">

<h2 class="text-primary">Delete Category</h2>

<hr/>

</div>

@\* <div asp-validation-summary="All"></div> \*@

<div class="mb-3">

<**label** **asp-for**="Name"></**label**>

<**input** **asp-for**="Name" class="form-control" disabled/>

</div>

<div class="mb-3">

<**label** **asp-for**="DisplayOrder"></**label**>

<**input** **asp-for**="DisplayOrder" class="form-control" disabled />

</div>

<button type="submit" class="btn btn-danger" style="width:150px">Delete</button>

<**a** **asp-controller**="Category" **asp-action**="Index" class="btn btn-secondary" style="width:150px">

Back to List

</**a**>

</div>

</**form**>

@section Scripts{

@{

<**partial** **name**="\_ValidationScriptsPartial"/>

}

}

As we seen above even our **Delete** View **asp-action=”Delete”** is **Delete** and inside our CategoryController.cs our POST action method for Delete is DeletePOST, it will still work because we explicitly added that to our **ActionName** above which is set to name **Delete**

[HttpPost, ActionName(“Delete”)].

Now our Delete functionality should now work.

**Displaying some alert if we Create, Edit and Delete some category section.**

We want to display some alerts when someone Deletes, Edits, or Create a category on the Category Index view how do we do that? For that we have something called as **TempData** in .NET Core. TempData has been provided with one single purpose, whatever we want to store in TempData stays there for only one request. After that, if we refresh the same page, that will be gone. That is perfect for displaying alerts of success or error message. Let’s take a look at how we’ll be doing that. Let’s say right now that when Create or Edit or Delete anything, we will add something to a TempData of success. So once the Create is successful into our CategoryController.cs Post action Method here, before we return back, we want to use TempData, so TempData and then we have to give any key inside there. Let’s call that **TempData[“success”] = “Category created successfully”;** on Create POST method. Now same when we are Editing we can copy this code and change the message into this **TempData[“success”] = “Category updated successfully”;** same in **Delete** post method **TempData[“success”] = “Category deleted successfully”;.** So we are storing some string inside TempData with the key of success.

//POST method

[HttpPost]

[ValidateAntiForgeryToken]

public async Task <IActionResult> Create(Category obj) //For Inserting data into database

{

if(obj.Name == obj.DisplayOrder.ToString())

{

ModelState.AddModelError("name","The Display Order cannot exactly match the Name");

}

if (ModelState.IsValid)

{

\_db.Categories.Add(obj);

await \_db.SaveChangesAsync();

TempData["success"] = "Category created successfully";

return RedirectToAction("Index");

}

return View(obj);

}

//Editing Section

//POST

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Edit(Category obj) //For Updating data into database

{

if (obj.Name == obj.DisplayOrder.ToString())

{

ModelState.AddModelError("name", "The Display Order cannot exactly match the Name");

}

if (ModelState.IsValid)

{

\_db.Categories.Update(obj);

await \_db.SaveChangesAsync();

TempData["success"] = "Category updated successfully";

return RedirectToAction("Index");

}

return View(obj);

}

//Deleting Section

//POST

[HttpPost, ActionName("Delete")] // Action name is delete because in our Delete View the action method that we named was "Delete"

[ValidateAntiForgeryToken]

public async Task<IActionResult> DeletePOST(int? id) //For Deleting data into database

{

var obj = await \_db.Categories.FindAsync(id);

if (obj == null)

{

return NotFound();

}

\_db.Categories.Remove(obj);

await \_db.SaveChangesAsync();

TempData["success"] = "Category deleted successfully";

return RedirectToAction("Index");

}

}

}

Now where do we want to extract this, we want to extract that inside the **Index** View. So at the top of **Index.cshtml** of our **Views/Category** we can check if the temp data of success is not null **“@if(TempData[“success”] != null)”.** The main important thing here is the key name of “success” must match what we entered into our CategoryController.cs action methods. If we use different key name, then this will not work. So if this does match, inside well display temp data and the value inside that. Below is inside our **Index.cshtml.**

@model IEnumerable<Category>

@{

ViewData["Title"] = "Index";

}

@if(TempData["success"] != null)

{

<h2>@TempData["success"]</h2>

}

<div class="container p-3">

<div class="row pt-4">

<div class ="col-6">

<h2 class="text-primary">Category List</h2>

</div>

<div class="col-6 text-end">

<**a** **asp-controller**="Category" **asp-action**="Create" class="btn btn-primary">

<i class="bi bi-plus-circle"></i> &nbsp; Create New Category

</**a**>

</div>

</div>

<br /><br />

<table class = "table table-bordered table-striped" style="width:100%">

<thead>

<tr>

<th>

<span style="color: white"> Category Name</span>

</th>

<th>

<span style="color: white"> Display Order</span>

</th>

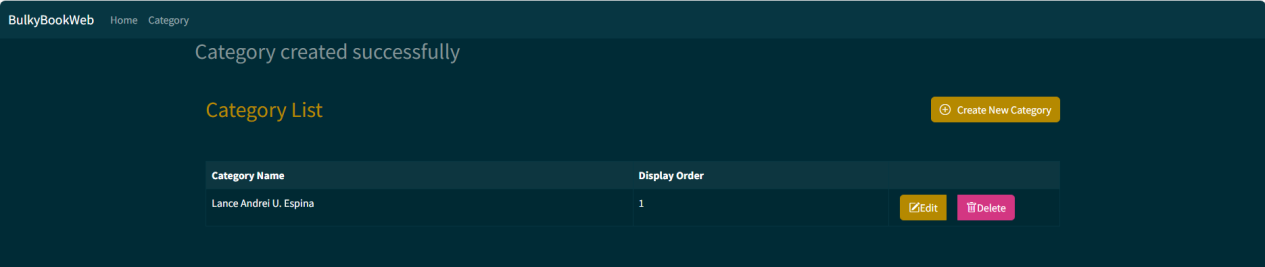
<th></th>

</tr>

</thead>

<tbody>

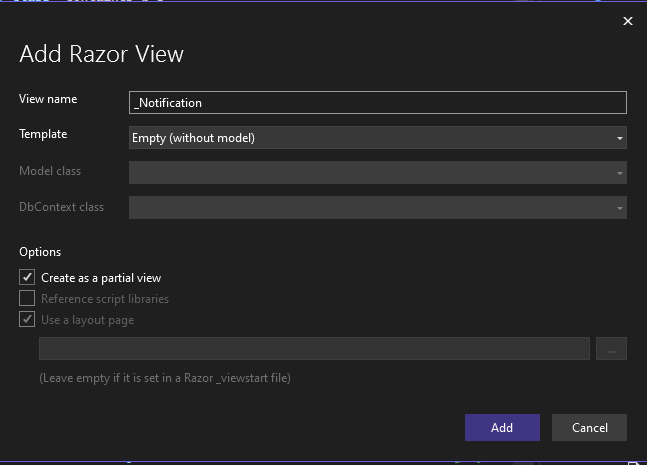
Now we can see this message display at the top when we once done creating, editing, and deleting some data. Once we reload the form it will automatically removed because that is the purpose of the **TempData**. It only stays into our memory for just one redirect, and after that it goes away.



Now this TempData can be used throughout the application and not just one page. So why not when we are checking this temp data success and displaying that do that at a global level. So we do not have to add this in all of our pages. For that Partial Views will be a perfect candidate. We can have some code inside the partial view and we can call that partial view everywhere we fought because right now it is four lines of code, but it is possible that this code will increase drastically, when we do something fancy or designing for notifications. So let’s move this code inside a Partial View and see how we can call that.

**Making our TempData global and move it to Partial View**

Now that we have added a code to display some of the alerts and notification, we want to make sure that this code is applicable on almost all the pages. Because in the future, if we add more pages, we don’t want to copy and paste the same code in other places. Now the best thing is inside the Shared folder, we will create the Partial Views. We will place all the Partial Views in this folder. Now in **Shared** folder right click and we’ll create a view, click **Add/View/Razor View** we will select the Razor View and not the empty one and once we proceeded we will check **Create as a partial view** and name it **\_Notification**. Now this should Creates us an Empty partial view.



Now what we want to do is Cut the TempData inside our **Index.cshtml** View and paste it inside our newly created partial view. Inside since we have an TempData for success we also want a TempData for Errors so we can display both success and errors notifications. This is from *\_Notification.cshtml* file

@if (TempData["success"] != null)

{

<h2>@TempData["success"]</h2>

}

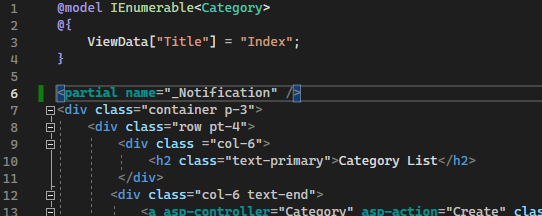
@if (TempData["error"] != null)

{

<h2>@TempData["error"]</h2>

}

Now inside our **Index.cshtml** View we can just simply type above **<partial name=”\_Notification” />**

****

Let’s say in the future, we want to add this to multiple pages, we can just have to add this partial tag and that is it. Now this configuration should work as we expect.

**Toastr Alerts for notification UI**

Search into our web browser the keyword Toastr, and select the one that is in GITHUB, we can test it by clicking the first link of Demo below. In order to use that, we will copy the **Minified** CSS link below, remember we can only copy the **CSS** link which is the second one and paste that to our **\_Layout.cshtml** which serves as our global UI. We can do that by typing this:

**<link rel=”stylesheet” href=”Paste the copied link here” />**

<!DOCTYPE html>

<html lang="en">

<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/bootswatchTheme.css" **asp-append-version**="true" />

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

<link rel="stylesheet" href="//cdnjs.cloudflare.com/ajax/libs/toastr.js/latest/toastr.min.css" />

<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap-icons@1.11.3/font/bootstrap-icons.min.css">

</head>

After we pasted this into our **\_Layout.cshtm**l file, next is we can copy that **Minified JS** link which is the first link above the CSS link that we’ve copied earlier. Once copied we can paste that inside our **\_Notification.cshtml** partial view. We can just first type

**<script src=”[cdnjs.cloudflare.com/ajax/libs/toastr.js/latest/toastr.min.js](https://cdnjs.cloudflare.com/ajax/libs/toastr.js/latest/toastr.min.js)”></script>**

Copy it and paste it into our error message below as well. Also, when we are using a **Toastr**, we will need the **jquery** **reference** and it can be found inside **wwwroot/lib/jquery/dist/jquery.min.js.** We can also type this above alongside with the Minified JS from toaster:

**<script src=”~/lib/jquery/dist/jquery.min.js”></script>**

Now our code inside **\_Notification.cshtml** should now look like this.

@if (TempData["success"] != null)

{

<**script** src="~/lib/jquery/dist/jquery.min.js"></**script**>

<script src="cdnjs.cloudflare.com/ajax/libs/toastr.js/latest/toastr.min.js"></script>

<h2>@TempData["success"]</h2>

}

@if (TempData["error"] != null)

{

<**script** src="~/lib/jquery/dist/jquery.min.js"></**script**>

<script src="cdnjs.cloudflare.com/ajax/libs/toastr.js/latest/toastr.min.js"></script>

<h2>@TempData["error"]</h2>

}

Next is we need the toastr alert. Back to **toastr GITHUB repository**,if we scroll below we should have a usage below which we can copy under **Other Options**. Now back to our **\_Notification.cshtml**, we will type a **<script> </script>** tag and we need that **java-script** code and construct it like this.

@if (TempData["success"] != null)

{

<**script** src="~/lib/jquery/dist/jquery.min.js"></**script**>

<script src="//cdnjs.cloudflare.com/ajax/libs/toastr.js/latest/toastr.min.js"></script>

<script type="text/javascript">

toastr.success('@TempData["success"]');

</script>

}

@if (TempData["error"] != null)

{

<**script** src="~/lib/jquery/dist/jquery.min.js"></**script**>

<script src="//cdnjs.cloudflare.com/ajax/libs/toastr.js/latest/toastr.min.js"></script>

<script type="text/javascript">

toastr.error('@TempData["error"]');

</script>

}

Now we’ve added the toastr js into our website application. Now let’s say that this notification is something we will have across all the pages. In that case, it does not make sense to add this one line in all the pages **<partial name=”\_Notification” /> ,** even though it’s just one line, it just does not make sense. So rather than adding this in all our pages , we can add this into our **\_Layout.cshtml** and that will be included automatically in all our pages.

<div class="container">

<main role="main" class="pb-3">

<**partial** **name**="\_Notification" />

@RenderBody()

</main>

</div>

So just before render body where we have all the body that we render for the page, we will paste our partial view, and that should include in all the pages that we created inside our application. So we’ll make sure that we include only things that are needed across all the pages, because this will be loaded every time our page loads. If we run an test it to our application the new toastr notification should now work.

**Scaffold CRUD**

Now that our application is up and running and everything is secured. Now we have some techniques with the Controller. Previously we’ve added the Controller manually from scratch, inside our **Controllers** folder we’ve added **CategoryController.cs** and we also created all the Views manually inside the **Views/Category** folder, which contains **Create, Delete, Edit and Index.cshtml**. Now there is one technique that we can do to instantly add this automatically inside our project. We can do that that by right clicking the **Controllers** folder select **Add/Controller**, next we can click the one name **MVC Controller with views using Entity Framework**. Now click Add, next and we can select our **Model Class** which is the **Category (BulkyBookWeb.Models),** next select DbContext class and select our **ApplicationDbContext (BulkyBookWeb.Data)** class file and set the **Controller name** into anything we want but don’t remove the Controller name. Example of that **CategoriesSampleController,** once all of that are filled we can click the **Add** button and it should **Scaffold everything** for us that creates a **Controller** named **CategoriesSampleController.cs** file inside **Controllers** folder and also it will create a new **Views** Inside **CategoriesSample** folder and contains file like **Create, Delete, Details, Edit and Index.cshtml.** We can run the application and manually type the controller and action method inside our browser to test what we’ve created. We can delete what we’ve created when we are not using it because that is only for sample, it should give us an idea how to create a scaffold CRUD once our base CRUD is established.

Fixing common errors with scaffolding when the type of error exist and gives a message like this **“there was an error running the selected code generator”.** To fix this error we can just open our Visual Studio Installer, and click Modify, then on the right hand side expand **ASP.NET and web development,** Then check “**.NET Framework project and item templates.**”. That should fix this problem. Second Solution is to uninstall into our NuGet package manager the **“Microsoft.EntityFrameworkCore”** and that’s it we can scaffold our controller.

**Deploying the application into Azure**