**EndSars Photo/Image Gallery Using C# Asp.Net Core 5 Mvc Cloudinary and Auth0 Api**

* Table of Content
* Introduction
* Required Tools We Need
* Setting Up
  + Cloudinary Account
  + Visual Studio
  + Auth0 Account
* List of Installed Packages

This project is aimed at providing a working tool for human right activist by automating image upload process as they embark on their work in difficult and hazard situations.

**Required Tools We Need**

1. Sign up with Cloudinary
2. Get Visual Studio 2019
3. Sign up with Auth0
4. Note pad
5. **Sql server**

**Setting Up**

**Cloudinary Account**

The first thing to do is to sign up on Cloudinary here <https://cloudinary.com/documentation/cloudinary_get_started> so that we can get access to the required Api for image upload.

Upon successful sign up, the following are automatically generated for you:

Cloud name:

API Key:

API Secret:

API Environment variable:

So click on the account settings section to see what your Api key, Cloud name and Api secrete are. Copy out this values and keep it in your notepad or any other word editor of your choice for later use with our application..

Since we intend to use Cloudinary within a DotNet application, we will rely heavily on this cloudinary documentation for Dotnet <https://cloudinary.com/documentation/dotnet_integration.>

At this moment, this would be all for Cloudinary. Next we set up our coding environment.

**Setting Up**

**Visual Studio**

The IDE for this project is visual studio 2019, so ope your browser and go to <https://visualstudio.microsoft.com/downloads/> click on Visual studio 2019 and download the community version.

Installation and set up procedure for the visual studio 2019 can be found here <https://www.tektutorialshub.com/visual-studio/how-to-download-and-install-visual-studio-2019/.>

Upon successful download and and installation, launch visual studio by clicking on the icon on your computer.

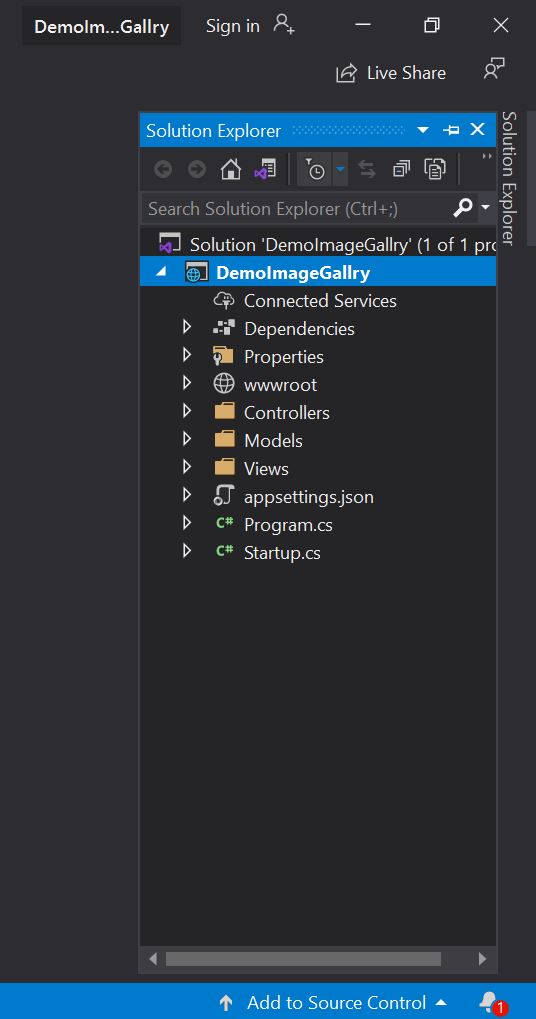
Click on create a new project then go to the search box and type Asp.Net Core Web Application. select Net

Give the project a name and then click on create. Lets call the name of this project ImageGallery.

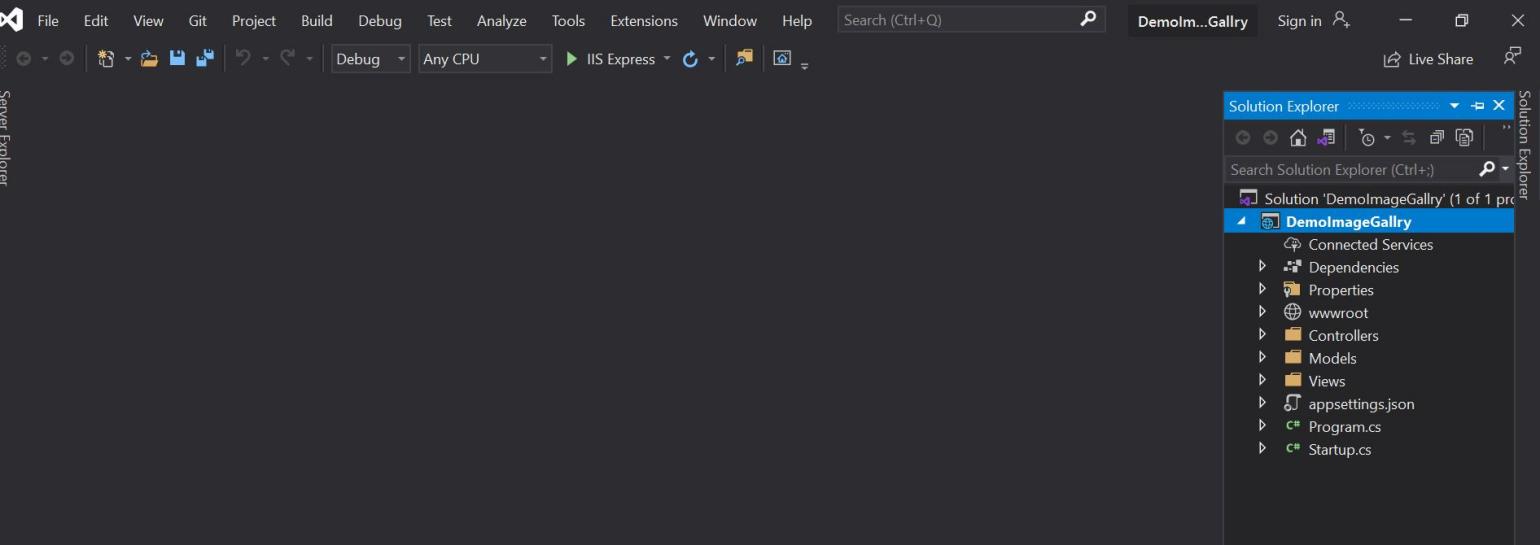
Then choose the version you want, in our case we will select the Net 5

Visual studio will prompt you to select which project template you prefer. In our case we will use the Asp.Net Core Web App ( Model-View-Controller). Leave the settings the way it is just click on create.

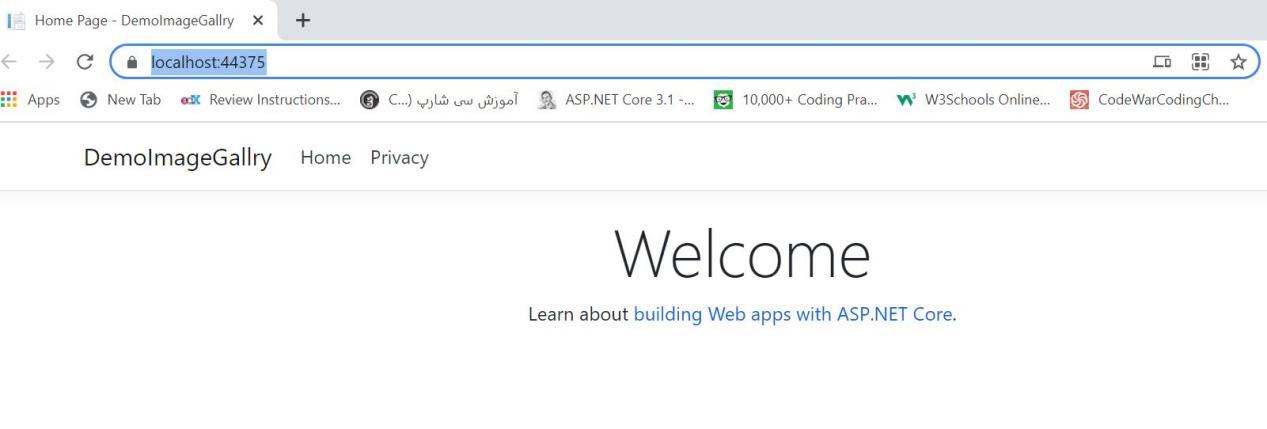
If everything was done right, you should see the Models,Views and Controllers folders respectively and any other folder in the project scaffold template.



Next, we need to test if this project was properly created before moving on to other things. To do this, run the application by clicking on control and hit F12 or just click on the little green play button located on the top menu bar.

****

**You should see a web browser fired up automatically similar to this shown below:**

****

Go to the url section of this page above and now copy the text written there. This should be similar to something like this one <https://localhost:44375/> . This value will be our call URL which we will use much later in the project. For now just save it inside your notepad

Callback URL : <https://localhost:44375/>

That will be all for now. Next we need to set up the Auth0 account.

**Setting Up**

**Auth0 Account**

If you don’t already have an account with Auth0, the first you have to do us to sign up for free by using this link <https://auth0.com/> click the sign up button to sign up.

As part of sign up process you to choose either the personal or company account type. Select what best suits you, in our case we will select the personal.

Upon successful sign up, a tenant domain name simply put a domain name is generated for you by Auth0.

Choose a Region where you want the app to be hosted then click the Create account button.

Then click on the link written “Create Application” it is similar to the figure below.

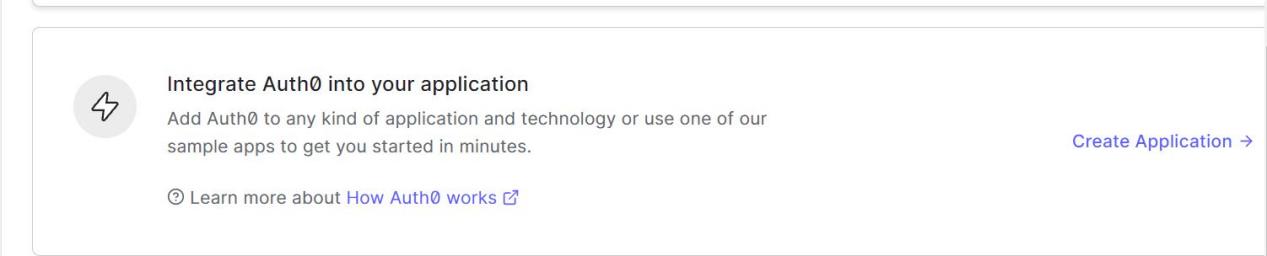
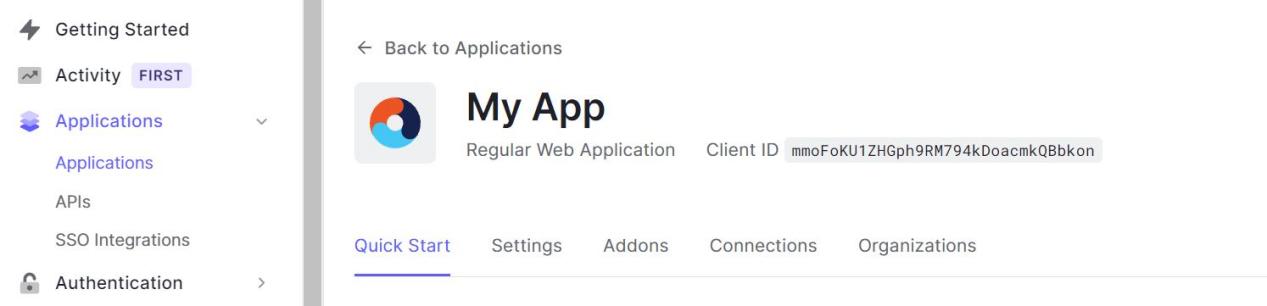


Fig XXX

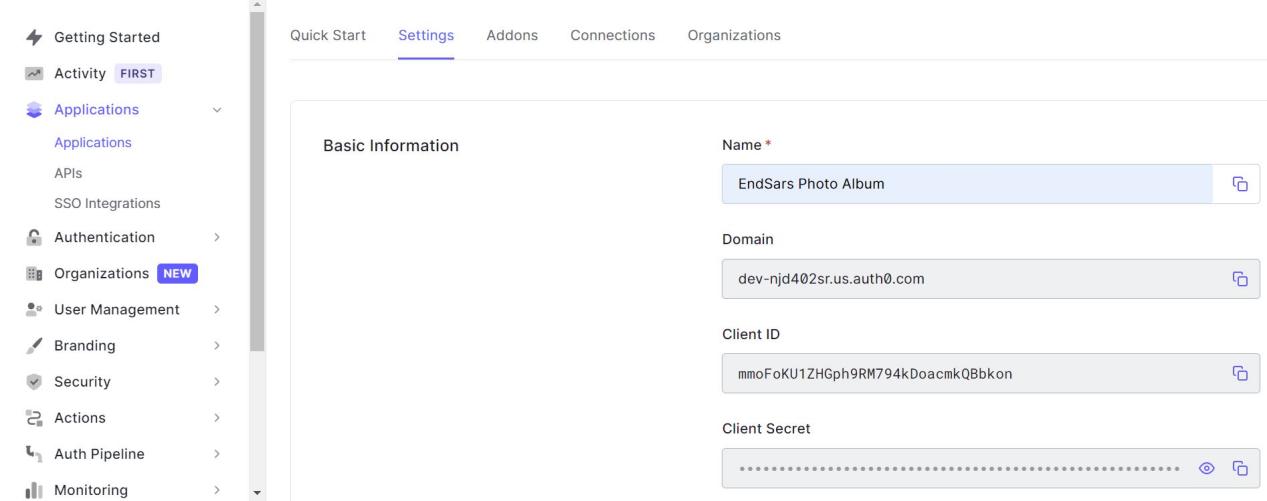
We now have to tell Auth0 few things about our DotNet app so both of them can be aware of each other. The Auth0 service will ask that you supply the following values such as :

* **Name**: This is asking you to fill in the name of your Dotnet application
* **Application Type**: Select the type of application you want to build
* **What Technology are using to build your project** :Select Asp.Net Core v3

When you have finished answering the all the questions above, you will be presented with your dash board similar to the figure XX below. The most important thing to note here is that a client Id number will be automatically generated for you copy this and paste it inside the notepad.



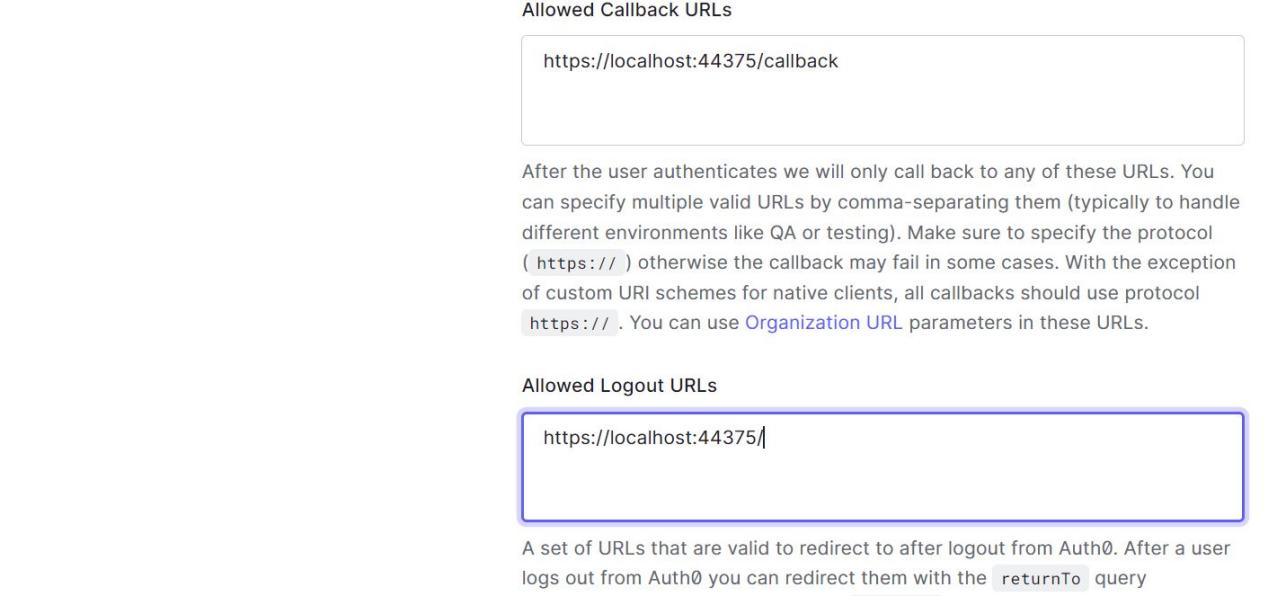
Next you have to make Auth0 to be aware of your Asp.net app, by clicking the Settings button in the figure above.



Copy and paste the following parameters Name, Domain, Client Id, Client secret which were automatically generated for you into the notepad document above.

Scroll down to the Application Url section of the settings, copy the call back Url from the notepad and paste this into the “Allowed Call Back Url” box and then attach /callback at the end of the text.

Next we also copy and past the call back Url into the “Allowed Logout Url”

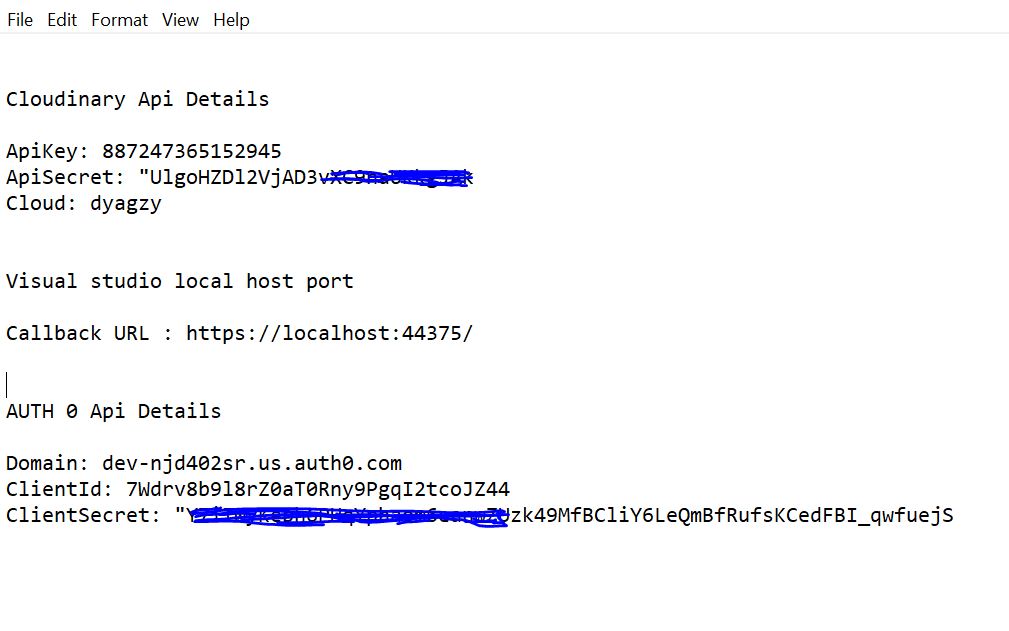


Lastly, click on save changes and that will be all for setting up.

We will consult the Auth0 Asp.Net documentation guide **https://auth0.com/docs/quickstart/webapp/aspnet-core/01-login#configure-your-application-to-use-auth0**

Much later as we develop this project.

Let’s quickly have a snap shot of what we have in our notepad file



The Api secret key of the cloudinary service and the Client secret of the Auth0 services are very sensitive information and must not be disclosed to anyone that is why I have coloured out some portion.

If you have followed all the steps correctly to this point then you are supposed to have the above parameters in your notepad file.

**Coding Proper**

We will return now to the visual studio and start proper coding for the project.

We will obey the SOLID principle

Right click on the solution file and click add new project. This will take you to the same screen where we selected the type of Asp.Net core template that we chose initially.

Go to the search box and type class library and select Class Library (.Net Core), name this SimpleImageGallery.Data

We need one more class library so repeat the process a second time but this time around, we will call the name of this class library SimpleImageGallery.Services. You can call these two class libraries any name you want because we want to keep things tidier and easy to debug we have to go through this route.

If you have done the above correctly then you should have 3 items in your solution explorer layout has shown below

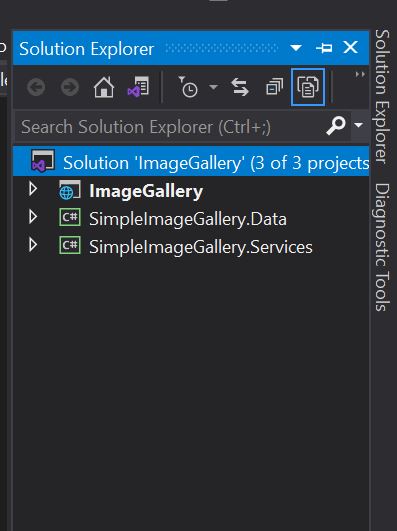


Fig XXX

We will refer to SimpleImageGallery.Data as the Data layer. This layer will handle all data access issues such as entity model classes, database class , Migration files etc.

We will refer to SimpleImageGallery.Services as the Service layer.This layer will handle repository service access to our application.

**PROJECT REFRECNCING**

Now we need to make the 3 separate projects above communicate with each other.

Starting from the ImageGallery project, click on dependencies, right click on projects then check the boxes for SimpleImageGallery.Data and SimpleImageGallery.Services respectively close the window.

Now move to the SimpleImageGallery.Services project click dependencies and right click project hen check the boxes for SimpleImageGallery.Data close the window.

Now right click the data layer and add a folder, name this folder Models. This models folder will house our entity classes which will be stored in our database.

Now right click the Models folder and add a class, name this class GalleryImage.

We need to populate this class with the properties below. This properties kind of mimics what an image gallery has such as date created, title etc

**ENTITY MODELS**

namespace SimpleImageGallery.Data.Models

{

public class GalleryImage

{

public int Id { get; set; }

public string Title { get; set; }

public DateTime Created { get; set; }

public string Url { get; set; }

public virtual IEnumerable<ImageTag> Tags { get; set; }

}

}

We add another entity class and populate it with the properties below which will represent an Image tag

namespace SimpleImageGallery.Data.Models

{

public class ImageTag

{

public int Id { get; set; }

public string Description { get; set; }

}

}

Right click on the web ImageGallery project and select ManageNugget package.

Click the browse button and type this Microsoft.EntityFrameworkCore.Design

when you find it click install button located at the top right and then on ok followed by I Accept to continue the installation.

See figure XXX for more examples

**DATABASE CONTEXT**

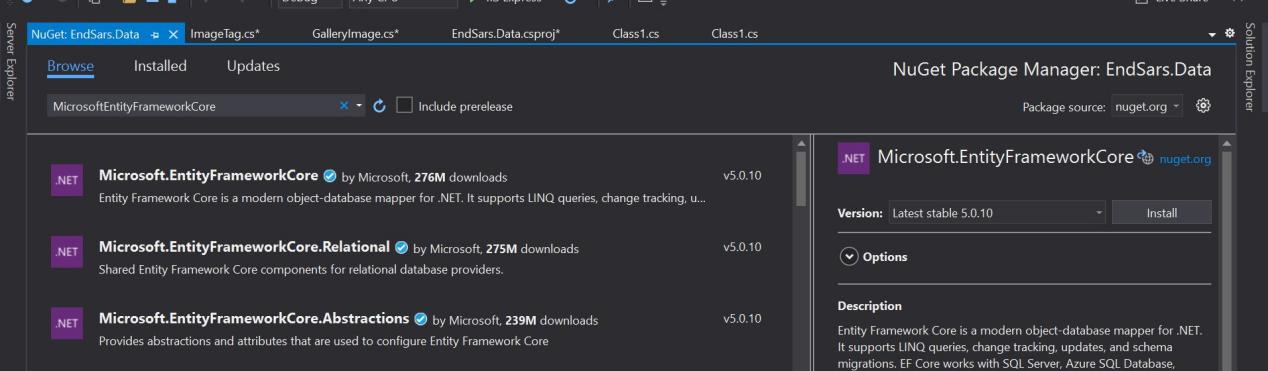
Next we need to add the Database context class to the data layer by right cliking the data layer and the select add a class, lets call this class SimpleImageGalleryDbContext.

The SimpleImageGalleryDbContext will represent our data access class I.e the class via which we can talk to the database.

We need to add some nugget packages which will aid data access to the database via the EntityFramework core.

So right click on the Data layer and select Manage Nugget Packages.

Click on browse and type the first nugget package into the box, when you find it click install button located at the top right and then on Ok followed by I Accept to continue the installation.



We will add three different packages below to the data access layer one after following the process above.

1. MicrosoftEntityFrameworkCore (5.0.10)
2. MicrosoftEntityFrameworkCore.SqlServer (5.01.0)
3. MicrosoftEntityFrameworkCore.Tools (5.01.0)

We need to tell the database the tables to create for us, so we need to add the DbSet properties for each of the entity classes in our model folder in the Data layer.

Return back to the SimpleImageGalleryDbContext class and make it inherit from the DbContext class present in the entityframeworkcore library. Add missing refrences by pressing control period I.e., “ctlr .”

Add a constructor of the SimpleImageGalleryDbContext and then pass in the (DbContextOptions options as a parameter and then call the base constructor of the Dbset and also pass in options as a parameter.

Your code should

public class SimpleImageGalleryDbContext: DbContext

{

public SimpleImageGalleryDbContext(DbContextOptions options) : base(options)

{

}

public DbSet <GalleryImage> GalleryImages { get; set; }

public DbSet<ImageTag> ImageTags { get; set; }

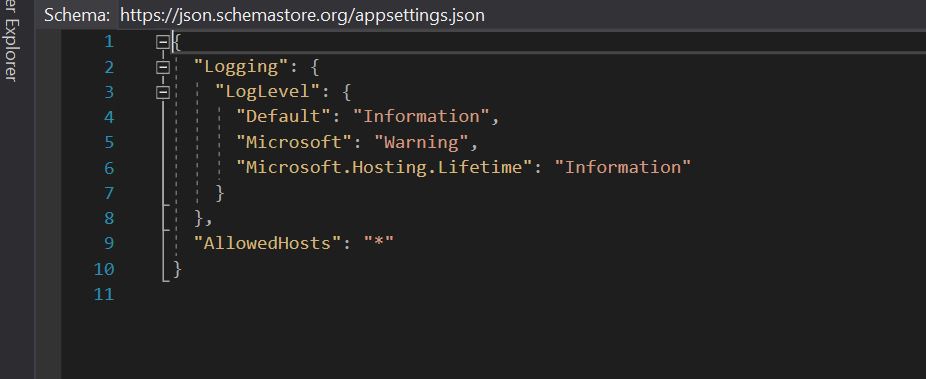
}

}

**MIGRATIONS**

Return to the ImageGallery project i.e, the web project and open up the appsettings.json file

Your appsettings.json file should look like the figure below



We need to add connection strings to the appsettings.json file

Just before the "AllowedHosts": "\*" add the your connection strings.



"ConnectionStrings": {

"DefaultConnection": "server =<put the name of your server here>;Database =ImageGalleryDb; Trusted\_Connection=True; MultipleActiveResultSets=true"

}

Now we have to register the Sql server in the StartUp class. So head to the configure method section of the StartUp class then add the following codes to it

services.AddDbContext<SimpleImageGalleryDbContext>(options =>

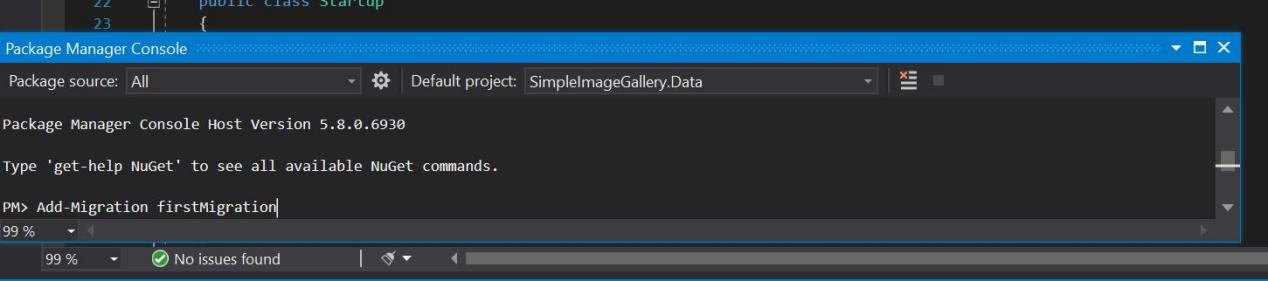
options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection"))

);

Add the necessary missing references

Now we are all set to run our migration

If you don’t already have your package manager console opened, go to view on the menu bar, select other windows and the select Package manager console and type the command below Add-Migration firstMigration press the enter key.



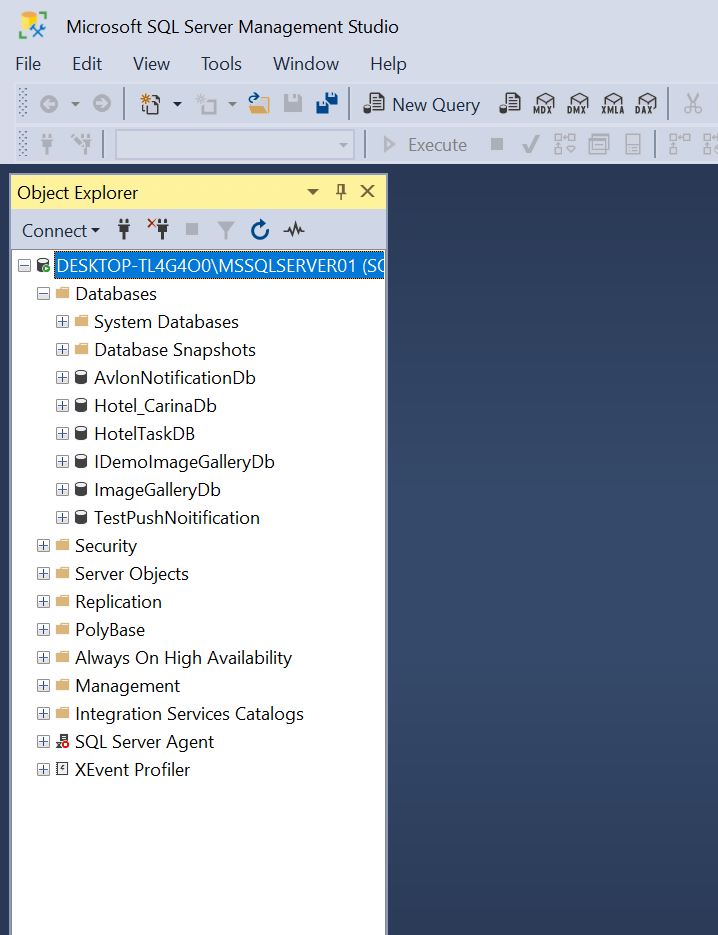
Next enter this second command Update-Database at the PM> prompt and press enter.

If everything works well to this and you have been following then you should see your new database inside the Sql server when it is opened.

So lets confirm this by launching the Sql server.

Now launch your Sql server and search for ImageGalleryDb.

Open it and open the Table to confirm that we have two tables i.e. ImageTags and the GalleryImages



Now lets go over to the Web app project (ImageGallery) open it and add a class to the models folder there. Lets call this class UploadImageModel

namespace ImageGallery.Models

{

public class UploadImageModel

{

public string Title { get; set; }

public IFormFile UploadImage { get; set; }

public string Tags { get; set; }

}

}

The UploadImage property is declared of type IFormFile because it is much easier for the Asp.net to identify this property with an image file or any other type of file.

Next we will add another class to the models folder and name it GalleryIndexModel. The GalleryIndexModel will have two properties as show below.

namespace ImageGallery.Models

{

public class GalleryIndexModel

{

public IEnumerable<GalleryImage> Images { get; set; }

public string SearchQuery { get; set; }

}

}

Next we will add another class to the models folder and name it GalleryDetailModel. The GalleryDetailModel will have five properties as shown below.

public class GalleryDetailModel

{

public int Id { get; set; }

public string Title { get; set; }

public DateTime Created { get; set; }

public string Url { get; set; }

public List<string> Tags { get; set; }

}

}

**INTERFACE DECLARATIONS**

At this point now we need to declare some basic interface service, which the user can use to get a single picture, get all pictures. Right click on the Data layer project and click add a class name it IImageService.

Copy the following codes into this interface class

public interface IImageService

{

IEnumerable<GalleryImage> GetAll();

IEnumerable<GalleryImage> GetWithTag(string tag);

GalleryImage GetById(int id);

}

This interface IEnumerable<GalleryImage> GetAll() helps us to retrieve all the pictures from the database.

This interface IEnumerable<GalleryImage> GetWithTag(string tag) helps us to retrieve all the pictures plus the tags from the database.

This interface IGalleryImage GetById(int id) helps us to retrieve a single picture from the database

We now need to implement the ImageService in the services layer. So head over to the Service layer, right click it and add the a class and name it ImageService.

This class will implement the ImageService interface class.

Create a constructor of the ImageService class and then inject an instance of the SimpleImageGalleryDbConetx class.



Copy the codes below as shown below

public IEnumerable<GalleryImage> GetAll()

{

return \_ctx.GalleryImages.Include(x => x.Tags);

}

public GalleryImage GetById(int id)

{

return GetAll().Where(img => img.Id == id).First();

//return \_ctx.GalleryImages.Find(id);

}

public IEnumerable<GalleryImage> GetWithTag(string tag)

{

return GetAll().Where(img => img.Tags.Any(t => t.Description == tag));

}

The GetAll method simply makes a db call to the database using the context \_ctx then assess the GalleryImages table where we will keep all the pictures or images. Then we use the Include extension method to load a secondary table

The GetById method is to get a single image from the database. Since the the GetAll method already returns a collection of all the images in the database, we can simply recall it and the filter it using the Where clause to narrow down the collection into what exactly we want. You have to specify the exact image you want by supplying the id of that particular image.

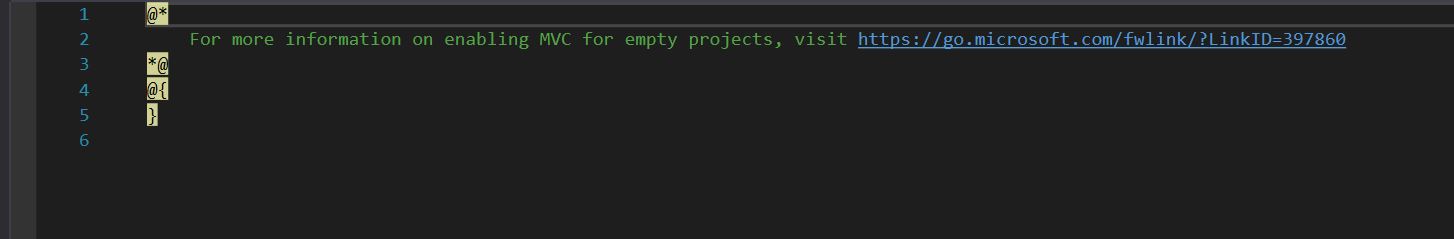
The GetWithTag is similar to the GetById method except that it returns the images alongside the tag description of each images.

**VIEWS**

Go back to the web project , right click on the Views folder and add another folder name it Gallery.

Right click on this Gallery folder and click add empty view.

Name it Index.cshtml



This comes with some little text as shown above. Delete all the text and lets make it ready to receive our view page codes to render the view page.

This view page usually contains a mix of html and C# which we use to render what the user will see on her browser page, underneath this is the razor engine which translate all the C# and html into client side language that the web understands.

Inject the model that contains the properties that we wish to render on the browser here by using the @model syntax to access where the model is located

@model ImageGallery.Models.GalleryIndexModel

**Copy and paste the snippet below**

<div class="container body-content" >

<div class=" row gallery-content">

@foreach (var image in Model.Images)

{

<div class="gallery-image-container drop-shadow">

<a **asp-controller**="Gallery" **asp-action**="Detail" **asp-route-id**="@image.Id">

<div class="gallery-image" style="background-image:url(@image.Url)"></div>

</a>

</div>

}

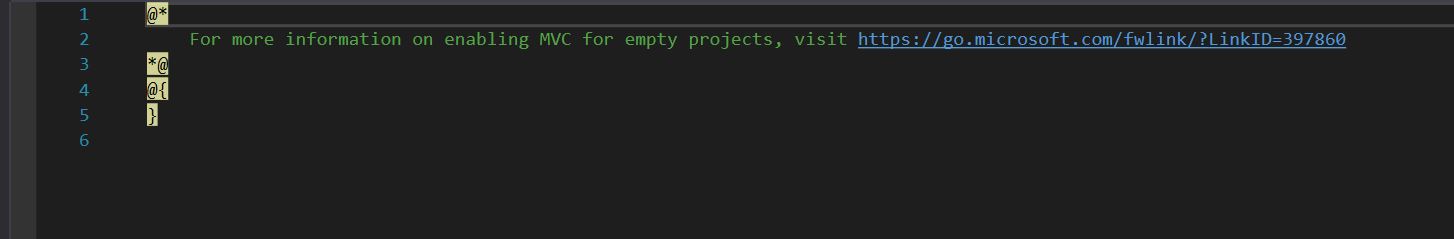
</div>

</div>

Also, right click on the Views folder and add another folder name it Image.

Right click on this Image folder and click add empty view.

Name it Upload.cshtml



@model ImageGallery.Models.UploadImageModel

Copy and paste this code below.

@model DemoImageGallry.Models.UploadImageModel

<div class="container body-content" >

<div class="row upload-container">

<div class="upload-form drop-shadow">

<form **asp-action**="UploadNewImage" method="post" enctype="multipart/form-data" id="upload-form">

<table border="0" cellpadding="2" cellspacing="2">

<tr>

<td>Title</td>

<td>

<input **asp-for**="Title" />

</td>

</tr>

<tr>

<td>Tags</td>

<td>

<input **asp-for**="Tags" />

</td>

</tr>

<tr>

<td>Upload Image</td>

<td>

<input **asp-for**="UploadImage" />

<input type="submit" class="btn btn-primary" />

</td>

</tr>

</table>

</form>

</div>

</div>

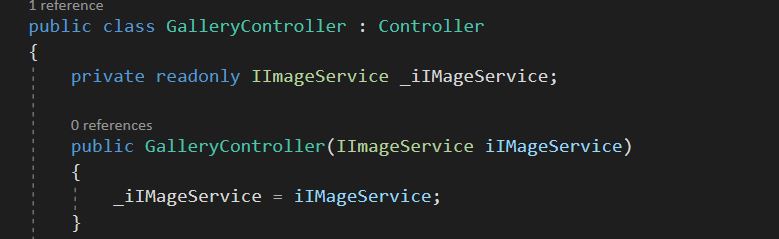
</div>

The code above basically helps to create a form with a minimalist styling for presenting an image on the browser.

**CONTROLLER**

Right on the controller folder and select add an empty controller. Name this controller GalleryController

Add a GalleryController constructor class to the GalleryController then inject the Image service as a constructor injection into this controller



Next we write the Http action method and name it Index

public IActionResult Index()

{

var imageList = \_iIMageService.GetAll();

var model = new GalleryIndexModel()

{

Images = imageList,

SearchQuery = ""

};

return View(model);

}

This method basically gets all the collection of images from the database, creates a new GalleryIndexModel and then passes these collections to it.

The model is then sent to the view for rendering.

**CSS STYLING**

Open the wwwroot folder in the web project, click the CSS folder and the double click on the site.css file to open it.

This file already contains the basic CSS styling that is used to render the default Asp.Net core Mvc page when the project is run.

We will only modify this file to accommodates the changes that we want to see whenever the Upload.cshtml file is rendered on the browser.

Add this code below to the site.css file

.gallery-content{

padding: 24px;

}

.gallery-image {

background-size: cover;

background-repeat: no-repeat;

background-position: center;

height: 280px;

width: 280px;

}

.gallery-image-container{

padding: 8px;

background: #fff;

border-radius: 2px;

display: inline-block;

height: 300px;

width: 300px;

margin: 12px;

position: relative;

}

.drop-shadow {

box-shadow: 0 10px 18px rgba(0,0,0,0.2), 0 5px 5px rgba(0,0,0,0.2);

}

.image-detail-container{

text-align: left;

}

.image-detail{

padding: 12px;

border-radius:2px;

}

.image-detail-meta{

padding:12px;

}.submit{

padding:2px;

}

.btn-file{

margin:6px;

}

#upload-form{

padding:60px;

margin:auto;

width:50%;

}

.upload-container{

padding:2%;

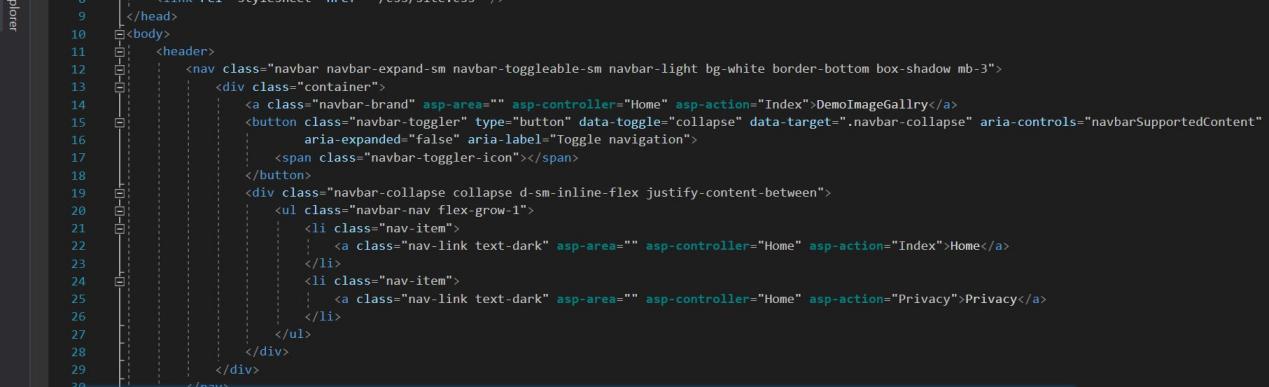
}

**ADD UPLOAD BUTTON**

Next we need to add an upload button to the navigation bar so that we can begin to upload images to the database

To do this, open the \_Layout.cshtml file located in the Shared folder

When this file is opened, it already contains some default codes used for rendering the layout.



Identify the portion of the \_Layout.cshtml that has this code and delete them

<a class="navbar-brand" **asp-area**="" **asp-controller**="Home" **asp-action**="Index">End Sars Photo Album</a>

<button class="navbar-toggler" type="button" data-toggle="collapse" data-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-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>

Paste this code below to replace what you just deleted above

<a class="navbar-brand" **asp-area**="" **asp-controller**="Gallery" **asp-action**="Index">EndSars Photo Album</a>

<div class="navbar-collapse collapse">

<ul class="nav navbar-nav">

<li><a **asp-controller**="Image" **asp-action**="Upload">Upload</a></li>

</ul>

</div>

Basically, we have simply add an Upload link button to the nav bar section of the page when the project is launched. Then we now instruction the button “Upload” that whenever it is clicked, the page should hit the **asp-controller**="Gallery" and then the a**sp-action**="Index" should hit the index action method.

Lastly, before we test our application, go to the Startup class and lets modify the mvc route.



Locate the app.UseEndPoints method and it should look like what is above.

Edit the Contoller=Home to Controller=Gallery as shown in the code snippet below.

app.UseEndpoints(endpoints =>

{

endpoints.MapControllerRoute(

name: "default",

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

});

**REGISTER INTERFACE SERVICE**

Now we need to register the IImage interface class in the StartUp class.

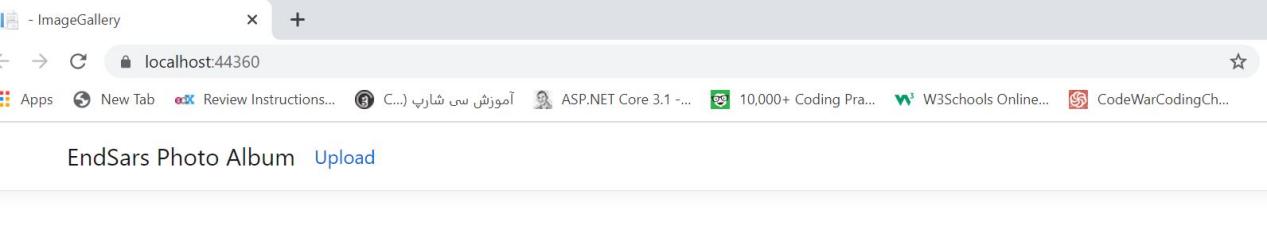
Go up to the configure method of the StartUp class and add this code snippet

services.AddScoped<IImageService, ImageService>();

**TEST ONE**

Save all and the launch the project to test what we have done

You should see a web browser similar to this below



If you have a similar web browser as shown above then you should congratulate yourself, else, try to go over the steps above a second time so as to trace where you missed out.

Now you can seethe Upload button successfully added but when we click it we get a 404 page not found error, so lets go and fix this.

**Image Upload Service**

We want when the end users click on Upload image button on the home page, we want the user to be directed to a directory where they can select a particular image or picture for upload and then when this has been done, we want the user to see the new image added on the home page.

At the backend, what we want to implement is such that when the user uploads an image, we want this image to be stored up in a remote cloud service like Cloudinary for fast retrieval and rendering. Then we also want this image path to be stored in our local database simultaneously.

At the beginning of this tutorial we have talked about how to get your Cloudinary account and cloudinary credentials set up. If you still haven't gotten yours then follow this link <https://cloudinary.com/documentation/cloudinary_get_started> .

To be able to use the Cloudinary service in our project, we need to have an interface that will hold the implementation of how we want this Cloudinary Api to function within our application.

So return back to the Web project, right click to add a folder name it Infrastructures.

Right click the Infrastructures folder then add an interface class, name it ICloudinaryImageUpload.cs. Inside this interface class add the code snippet below.

Task<string> UploadPicture(UploadImageModel model);

This is the interface that will expose our application to the Cloudinary Api remotely.

The code snippet above is a method declaration that will accept an UploadImageModel and return a task of strings.

We need to now write how this method should be implemented whenever it is called.

Now return back to Infrastructures folder right click and add a class name it CloudinaryImageUpload.cs.

This class has to implement the ICloudinaryImageUpload interface.

We need to create a constructor for the class CloudinaryImageUpload which will hold initialization of the Cloudinary credentials that we got when we created an account on Cloudinary.



**Adding Cloudinary Credentials to the Project**

Now we need to add the Cloudinary Api credentials that we got upon sign up into our proejct. We can hide this in the appsettings.json file and then use the DotNet IConfiguration interface to get it wherever and whenever we need to use it within our application. Now open the appsettings.json file and add the following code snippets as a json object.

"Cloudinary": {

"ApiKey": "put your Cloudinary Api key here",

"ApiSecret": "put your Cloudinary Api key secret",

"Cloud": "put your Cloudinary cloud name here"

}



Go to the CloudinaryImageUpload class and add the Cloudinary credentials to this class as a private properties

private string ApiKey { get; set; }

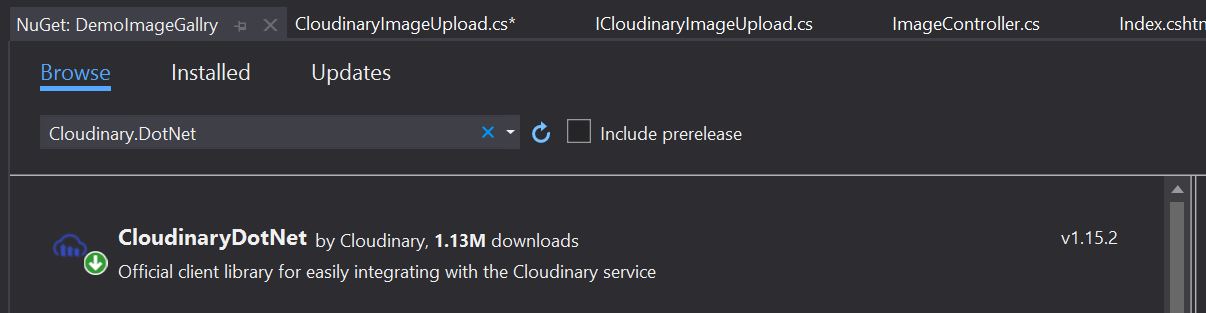
private string ApiSecret { get; set; }

private string Cloud { get; set; }

private Account Account { get; set; }

Right click on the web project and add the nugget package for Cloudinary by going to the browse box of the nugget package manager and type Cloudinary.DotNet

Select the right package see image below and click install, Ok and I accept as prompted by the installer wizard.



Now inject the IConfiguraton interface as a well as an instance of our database I.e. SimpleImageGalleryDbContext into the constructor class of the CloudinaryImageUpload class and create a private read only field.

Next we initialize all the private properties of the Cloudinary Api credentials as shown below, and bring in all missing references by using control period to see appropriate suggestions.

Your code should look the snippet below

private readonly IConfiguration configuration;

private readonly SimpleImageGalleryDbContext \_ctx;

public CloudinaryImageUpload(IConfiguration configuration, SimpleImageGalleryDbContext ctx)

{

this.configuration = configuration;

this.ApiKey = configuration["Cloudinary:ApiKey"];

this.ApiSecret = configuration["Cloudinary:ApiSecret"];

this.Cloud = configuration["Cloudinary:Cloud"];

this.Account = new Account { ApiKey = this.ApiKey, ApiSecret = this.ApiSecret, Cloud = this.Cloud };

\_ctx = ctx;

}

Now lets write the logic that will expose our application to the remote Cloudinary service.

Go to the UploadPicture method inside the CloudinaryImageUpload class.

Lets quickly summarize all the steps that we want this method to handle.

1. Open a connection to the Cloudinary Api
2. Convert the image into a base 64 string
3. Create a folder or a directory in the Cloudinary remote server using codes where we can later access all our uploaded images from the Cloudinary dash board that was automatically created for us when we signed up.
4. Pass this base 64 string to the Cloudinary function.
5. Lastly, call the Cloudinary UploadAsync function which will then expose and execute all the above steps for us.

Now because we also want to save this same image in our database, we will need to do step 6 below:

1. Create a new GalleryImage and then save this into the database.

public async Task<string> UploadPicture(UploadImageModel model)

{

var cloudinary = new Cloudinary(Account);

cloudinary.Api.Secure = true;

//reads the Image in the IFormFile into a bytes then we convert this to a base64 string

byte[] bytes;

using (var memoryStream = new MemoryStream())

{

model.UploadImage.CopyTo(memoryStream);

bytes = memoryStream.ToArray();

}

string base64 = Convert.ToBase64String(bytes);

var prefix = @"data:image/png;base64,";

var imagePath = prefix + base64;

// create a new ImageUploadParams object and assign the directory name

var uploadParams = new ImageUploadParams()

{

File = new FileDescription(imagePath),

Folder = "EndSars/img"

};

// pass the new ImageUploadParams object to the UploadAsync method of the Cloudinary Api

var uploadResult = await cloudinary.UploadAsync(@uploadParams);

// adds the new image to be uploaded to the database

var image = new GalleryImage()

{

Title = model.Title,

Created = DateTime.Now,

Url = uploadResult.Url.AbsoluteUri,

Tags = ParseTags(model.Tags)

};

\_ctx.Add(image);

await \_ctx.SaveChangesAsync();

return uploadResult.SecureUrl.AbsoluteUri;

}

The users of our application are most likely going to pass in tags when uploading images with spaces in between each tags so we need to format this to be in form of comma separated values.

We can just write a small private function to handle this for us and then call it in the method above see code snippet below

private List<ImageTag> ParseTags(string tags)

{

return tags.Split(",").Select(tag => new ImageTag

{

Description = tag

}).ToList();

}

We are done setting up the logic that will expose our application to the Cloudinary Api in the remote server.

Now when the Upload button is clicked, this will fire up the controller which will then make a call to this method and then this method will fire up a call to the remote Cloudinary Api.

To do all this we have to go back to the controllers folder and add a new controllers class name it ImageController.

Create a constructor of the ImageController and inject the ICloudinaryImageUpload interface service.

Create a private read only field so that this can be used to call up any available methods of the ICloudinaryImageUpload interface.



Inside this controller class, we need two methods lets call them Upload.

One will handle a Get Http action while the other will handle the POST Http action respectively.

public IActionResult Upload()

{

var model = new UploadImageModel();

return View(model);

}

[HttpPost]

public async Task< IActionResult> UploadNewImage(UploadImageModel model)

{

await \_imageUpload.UploadPicture(model);

return RedirectToAction("Index", "Gallery");

}

So lets quickly talk through what is happening here.

public IActionResult Upload(), this Http action method is to show the form that the user will file so as to be able to upload the image and then pass in other values like the description and all.

The public async Task< IActionResult> UploadNewImage method takes in an UploadImageModel then we use the \_imageUpload field to call up the UploadPicture method then pass in the model.

Lastly, we return a redirect to action response which basically is to take the user back to the home page where he or she could see the newly uploaded image.

We are all set and ready to run a second test of our application to see how far we are doing or not.

**TEST TWO**

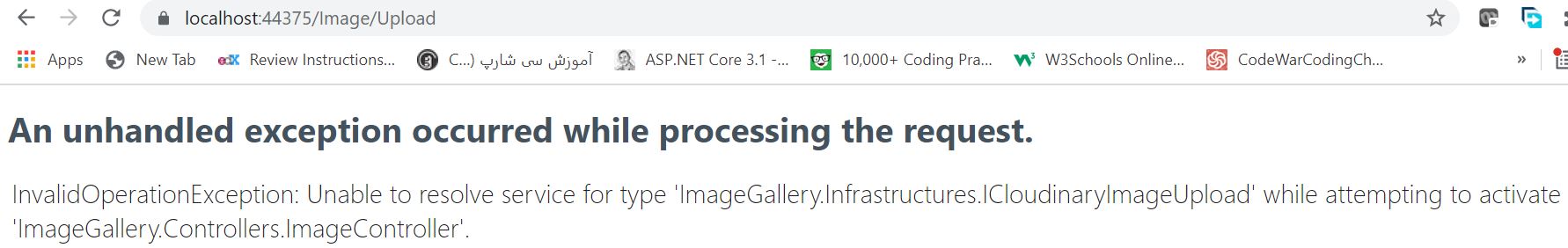
Run the application and click on the Upload button.

You should get this exception shown

# An unhandled exception occurred while processing the request.

InvalidOperationException: Unable to resolve service for type 'ImageGallery.Infrastructures.ICloudinaryImageUpload' while attempting to activate 'ImageGallery.Controllers.ImageController'.

See the figure below

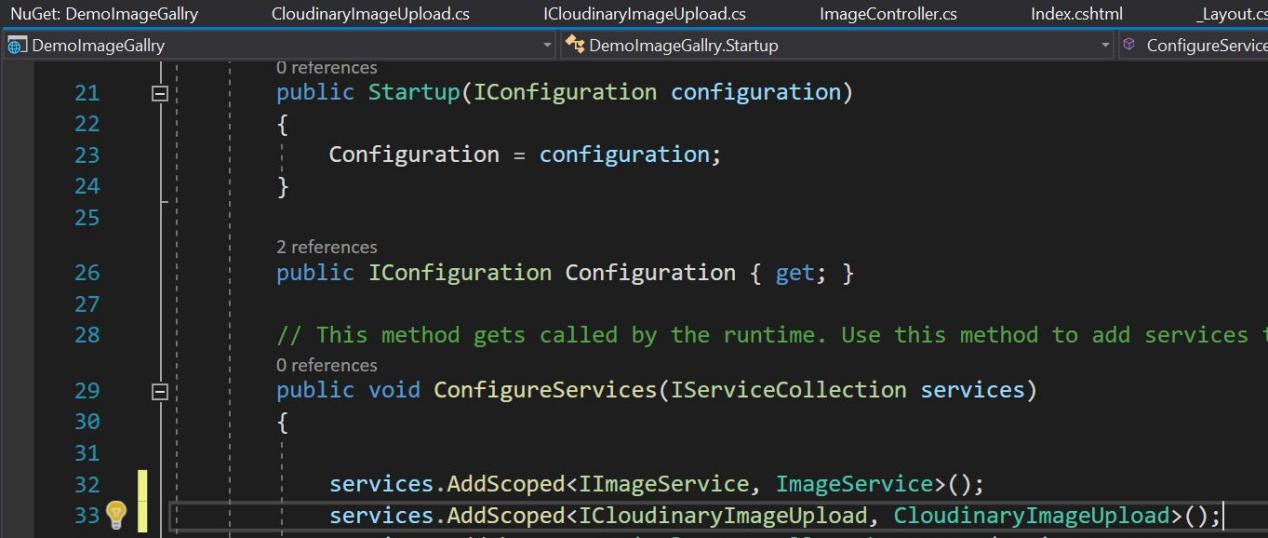


What the exception is saying is that we need to register the ICloudinaryImageUpload in the configure method of the StartUp class.

Now go to the configure method of the StartUp class and register the above Interface as a Scooped service see the code below

services.AddScoped<ICloudinaryImageUpload, CloudinaryImageUpload>();

Your code should look like the image below

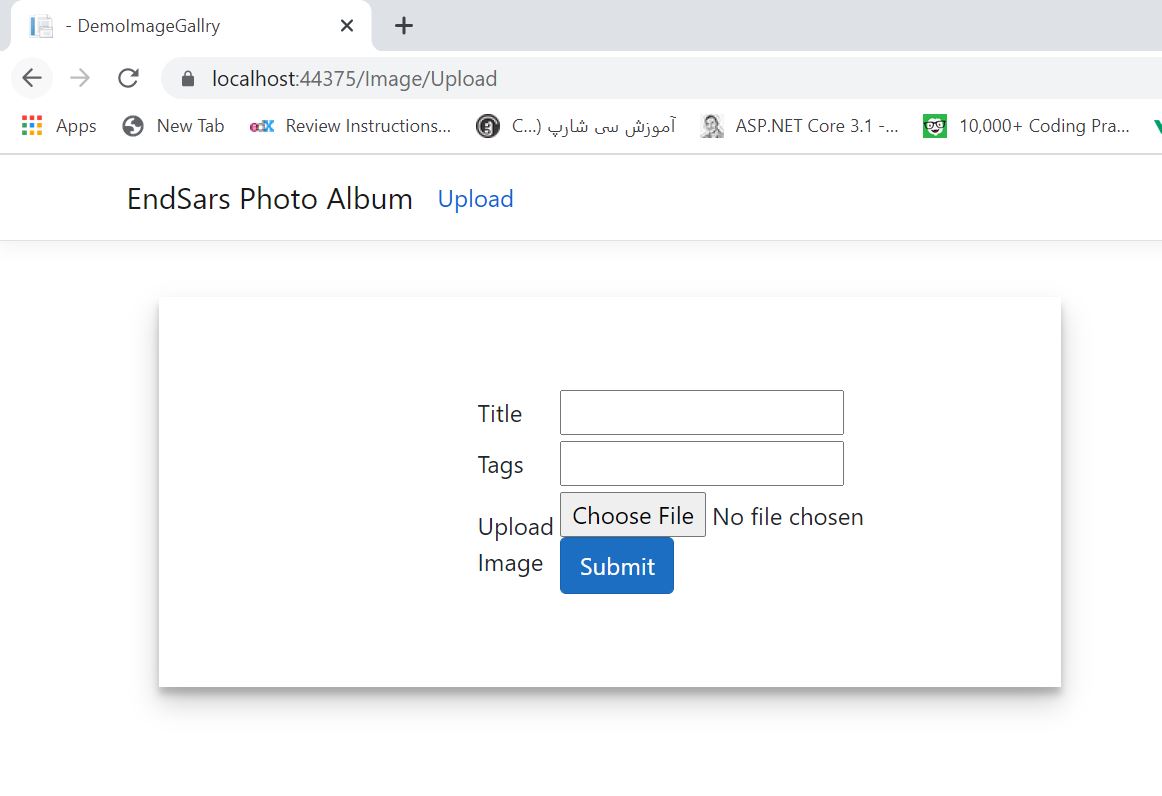


**TEST THREE**

Now we are ready to test our application again to see how we are doing.

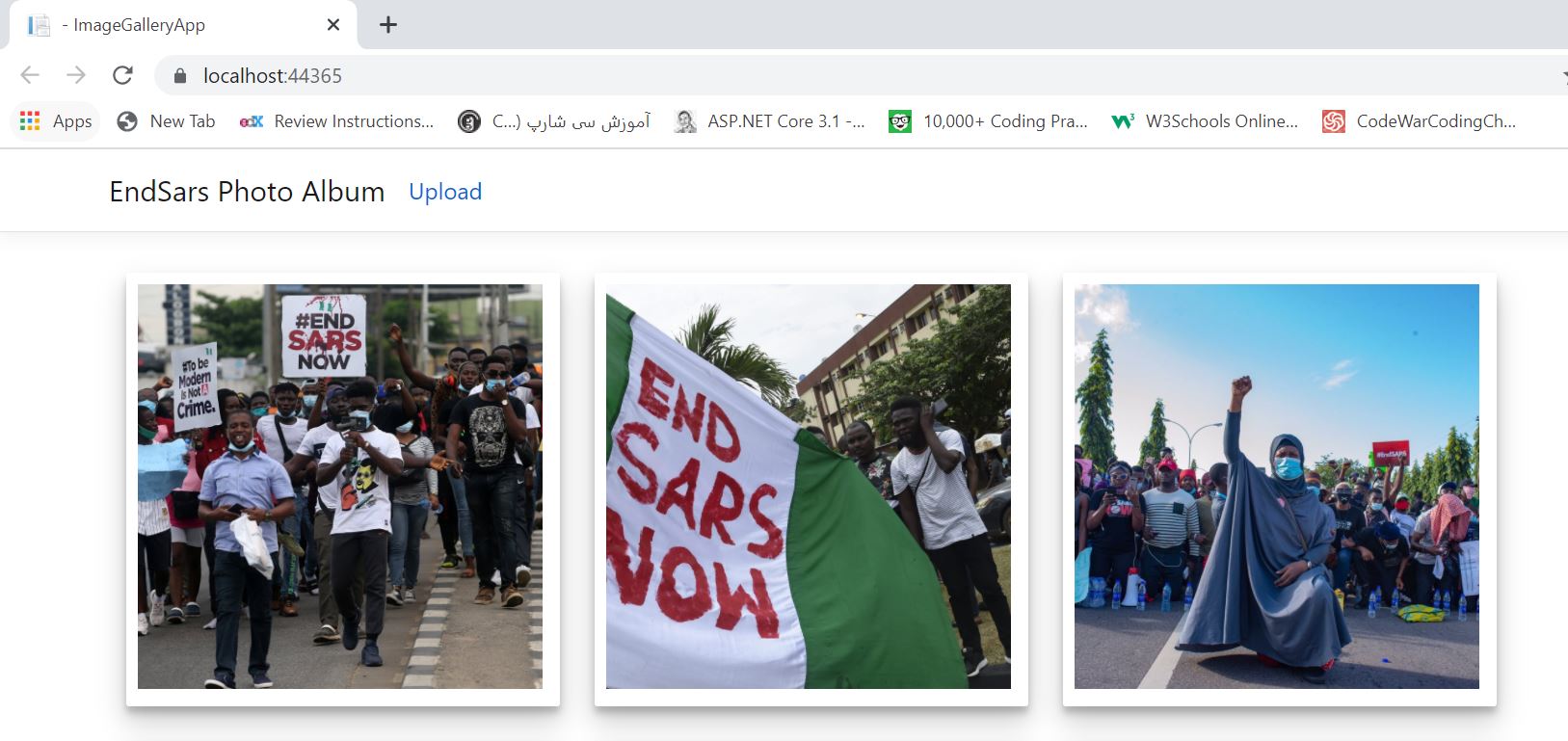
Run the application and then click the Upload button.

I will just pick up any random image on my laptop to upload just to test that we are fine up to this point.



Fill in the boxes, choose any image from your local machine and then clcik submit.

Below is how mine looks like’



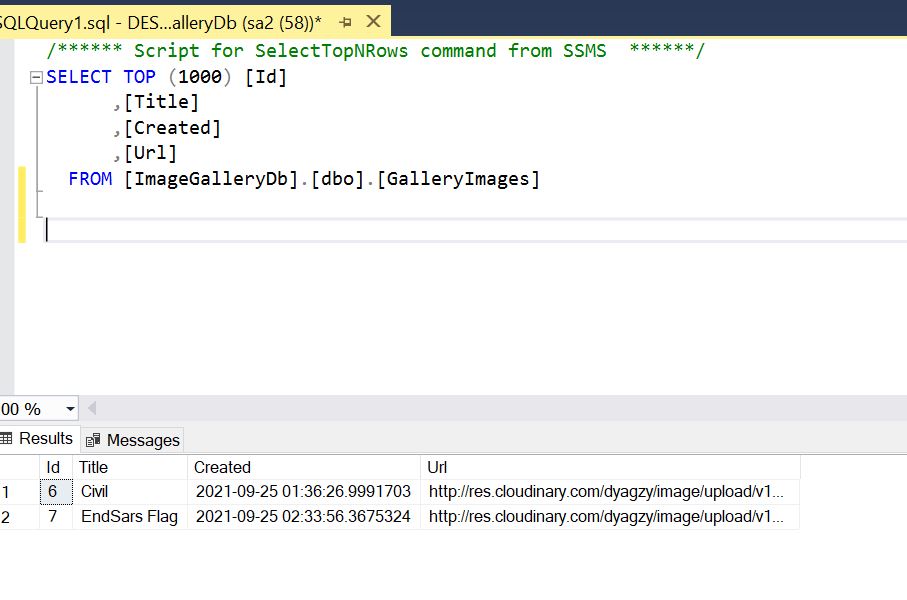
**TEST FOUR**

For test four, we simply want to confirm that our uploaded images are stored up in the Cloudinary server and also on our local database.

**DataBase Check**

So lets check the database first,

Launch your Sql database, navigate to your database and open the GalleryImages table to check

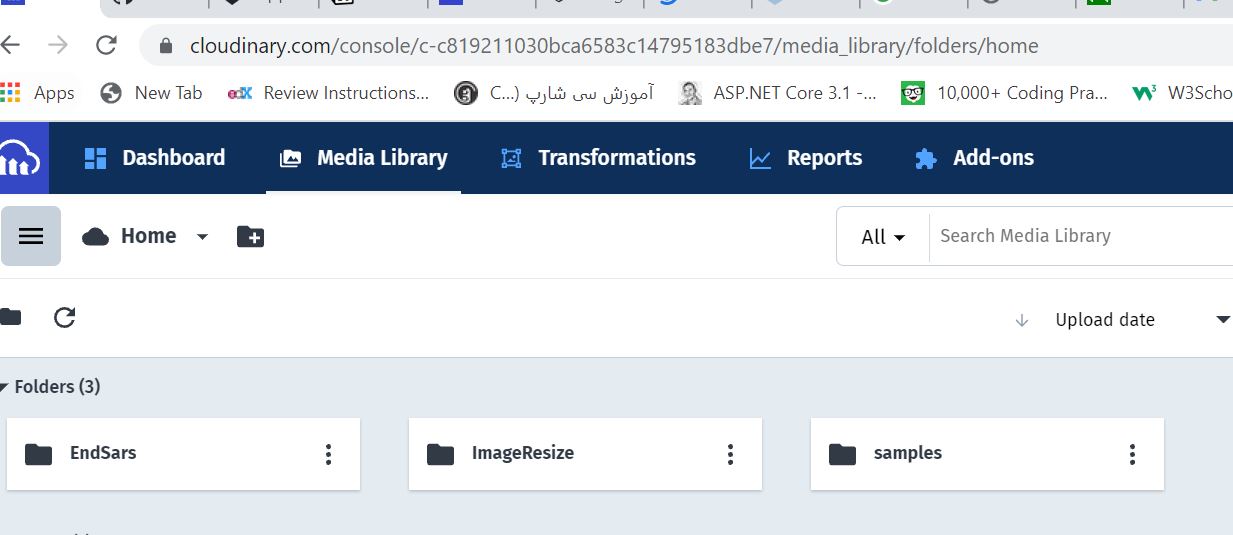


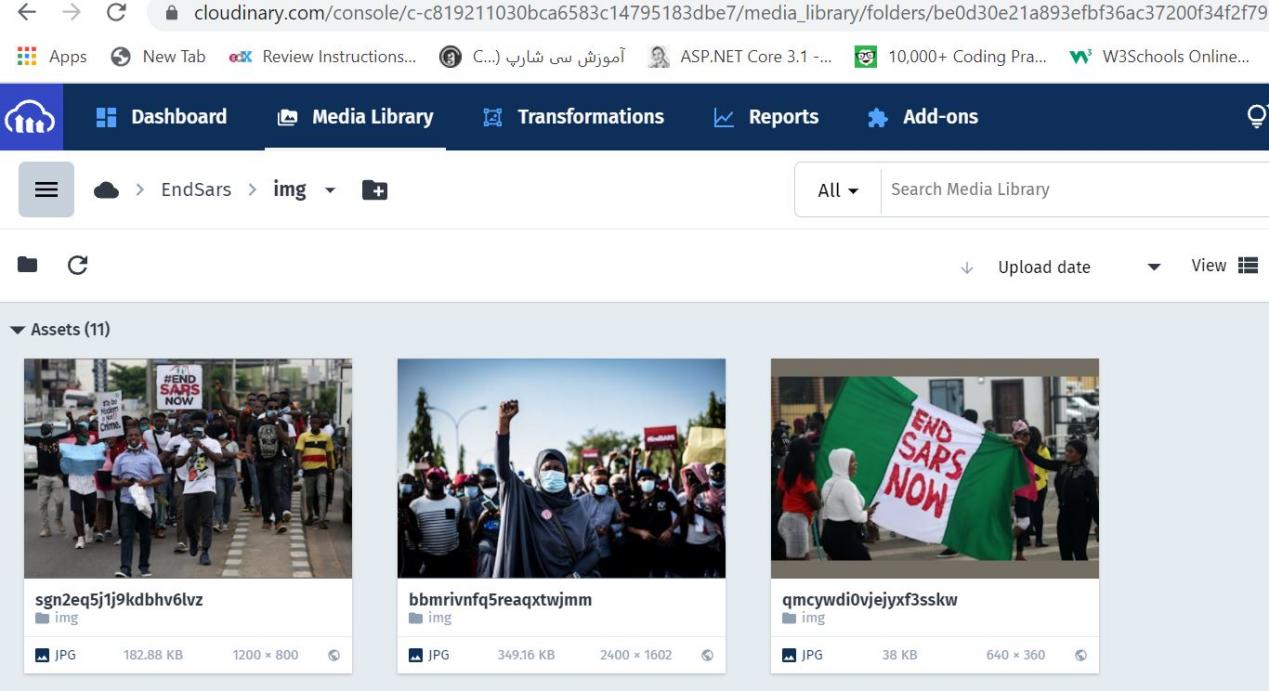
Just pay attention to the Url of the last two images, we can see the Url path to the images we uploaded to the Cloudinary server sitting right on there on our local database. You can copy this Url and past it into your browser to see the exact image.

**Cloudinary Check**

Next login into your Cloudinary account on the browser, click the Media Library button and you should see the name of the directory that you specified when we set up the CloudinaryImageUpload service.

In my case the name of this directory is EndSars, double click this folder and the new image you uploaded should be sitting right there.





At this point we are 85% done with our project.

Now we need to configure authentication.

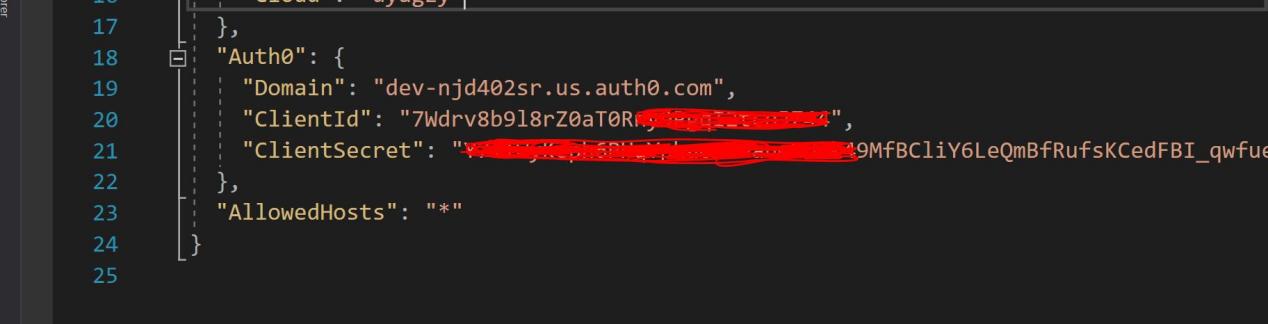
We want any user of this application to login into the application before having access to be able ti use the app.

**Configuring Authentication**

To configure the authentication we are going to use the Auth0 authentication Api, which we have already signed up for at the beginning of this tutorial.

The first thing we have to do is to add the Auth0 credentials i..e. Domain,ClientId, ClientSecret to the appsettings.json file

See the figure below



The next thing we will do is to go to the Web project and add the necessary nugget packages we need in other to be able to use theAuth0 authentication Api.

Go to the manage Nugget manager window and type this

Microsoft.Asp.NetCore.Authentication.OpenIdConnect, then install it.

Basically, the nugget packages above is what will inform our DotNet Core project that we want to use Auth0

Next register the Auth0 in the configure method of the StartUp class by calling the services.AddAuthentication() method.

You have to tell the Auth0 how you want it to work or how you want it to be set up by configuring various options on it. So next we have to configure options on the Authentication method by adding the codes below to the StarUp class.

services.AddAuthentication( options =>

{

options.DefaultAuthenticateScheme = CookieAuthenticationDefaults.AuthenticationScheme;

options.DefaultSignInScheme = CookieAuthenticationDefaults.AuthenticationScheme;

options.DefaultChallengeScheme = CookieAuthenticationDefaults.AuthenticationScheme;

})

Don’t forget to add any missing references.

Next chain the above with the AddCookies method and the AddOpenIdConnect method respectively.

Add the following snippet to the StartUp class

services.AddAuthentication(options =>

{

options.DefaultAuthenticateScheme = CookieAuthenticationDefaults.AuthenticationScheme;

options.DefaultSignInScheme = CookieAuthenticationDefaults.AuthenticationScheme;

options.DefaultChallengeScheme = CookieAuthenticationDefaults.AuthenticationScheme;

}).AddCookie()

.AddOpenIdConnect("Auth0", opt =>

{

// Set the authority to your Auth0 domain

opt.Authority = $"https://{Configuration["Auth0:Domain"]}";

// Configure the Auth0 Client ID and Client Secret

opt.ClientId = Configuration["Auth0:ClientId"];

opt.ClientSecret = Configuration["Auth0:ClientSecret"];

// Set response type to code

opt.ResponseType = OpenIdConnectResponseType.Code;

// Configure the scope

opt.Scope.Clear();

opt.Scope.Add("openid");

// Set the callback path, so Auth0 will call back to http://localhost:3000/callback

// Also ensure that you have added the URL as an Allowed Callback URL in your Auth0 dashboard

opt.CallbackPath = new PathString("/callback");

// Configure the Claims Issuer to be Auth0

opt.ClaimsIssuer = "Auth0";

opt.Events = new OpenIdConnectEvents

{

// handle the logout redirection

OnRedirectToIdentityProviderForSignOut = (context) =>

{

var logoutUri = $"https://{Configuration["Auth0:Domain"]}/v2/logout?client\_id={Configuration["Auth0:ClientId"]}";

var postLogoutUri = context.Properties.RedirectUri;

if (!string.IsNullOrEmpty(postLogoutUri))

{

if (postLogoutUri.StartsWith("/"))

{

// transform to absolute

var request = context.Request;

postLogoutUri = request.Scheme + "://" + request.Host + request.PathBase + postLogoutUri;

}

logoutUri += $"&returnTo={ Uri.EscapeDataString(postLogoutUri)}";

}

context.Response.Redirect(logoutUri);

context.HandleResponse();

return Task.CompletedTask;

}

};

});

The remaining codes we need to complete the Auth0 configurations can be find, simply copy it and paste into the StatUp class

<https://auth0.com/docs/quickstart/webapp/aspnet-core/01-login#configure-your-application-to-use-auth0>

Lastly, head over to the controllers folder and add a controller class name it AccountController

Then add the following snippet into the AccountController class

public class AccountController : Controller

{

public async Task Login(string returnUrl = "/")

{

await HttpContext.ChallengeAsync("Auth0", new AuthenticationProperties() { RedirectUri = returnUrl });

}

public async Task LogOut(string returnUrl = "/")

{

await HttpContext.SignOutAsync("Auth0", new AuthenticationProperties()

{

RedirectUri = Url.Action("Index", "Gallery")

});

await HttpContext.SignOutAsync(CookieAuthenticationDefaults.AuthenticationScheme);

}

}

Lastly, we need to add a login button to the existing navigation bar.

Open the \_Layout.cshtml file located in the Shared folder inside the Views folder.

Refer back to figure XXXX where we added the Upload button to the navigation bar, just after the closing div add the following codes snippet.

<ul class="nav navbar-nav navbar-right">

@if (User.Identity.IsAuthenticated)

{

<li><a id="qsLogoutBtn" **asp-controller**="Account" **asp-action**="Logout">Logout</a></li>

}

else

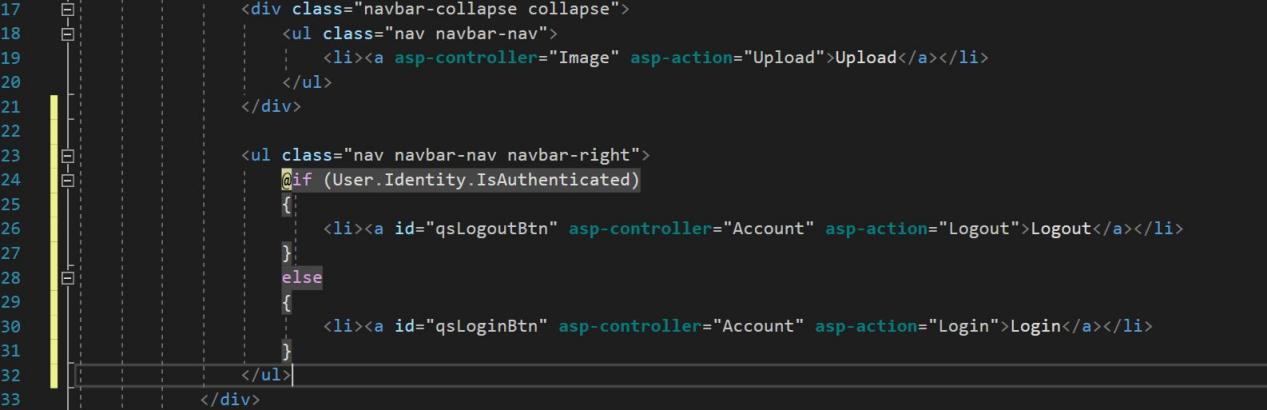
{

<li><a id="qsLoginBtn" **asp-controller**="Account" **asp-action**="Login">Login</a></li>

}

</ul>

So your code should look like this



**List of Installed Packages**

* MicrosoftEntityFrameworkCore (5.0.10)
* MicrosoftEntityFrameworkCore.SqlServer (5.01.0)
* MicrosoftEntityFrameworkCore.Tools (5.01.0)
* Microsoft.EntityFrameworkCore.Design.
* CloudinaryDotNet
* MicrosoftEntityFrameworkCore.Authentication.Cookies (2.2.0)
* MicrosoftEntityFrameworkCore.Authentication.OpenIdConnect (5.01.0)
* MicrosoftEntityFrameworkCore.Hosting

This is a photo automation app for human right activist. It was built using C# Asp.Net Core Mvc 5. For backend and frontend. It layers on the Cloudindary Api for remote image storage and transformations, and then it uses the Auth 0 Api for user management and login functionalities.

The first time I heard about the word Auth 0 was during my participation at this Hackathon. I have just about 4 months experience into software and using C#. The Auth 0 doc for DotNet was very easy and direct to read and apply. Implementing Auth 0 from a C# Asp.net core was super easy for me.

I am happy that as a newbie into software with less than 4 months of development experience, I was able to participate in my first hackathon ever, I was able to read Cloudinary and Auth 0 docs and to make sense of what the docs says and how to apply it. My app works , but am not able to complete my article due to the deadline. I am very happy that over the last two weeks I was able to build something I had never built before, read docs and use Apis I have never used before. Yes I did struggled with this for days but overall I am very happy and feel fulfilled and lastly to explore what it means to be a Technical Writer

I am happy that as a newbie into software with less than 4 months of development experience, I was able to participate in my first hackathon ever, I was able to read Cloudinary and Auth 0 docs and to make sense of what the docs says and how to apply it. My app works , but am not able to complete my article due to the deadline. I am very happy that over the last two weeks I was able to build something I had never built before, read docs and use Apis I have never used before. Yes I did struggled with this for days but overall I am very happy and feel fulfilled and lastly to explore what it means to be a Technical Writer. Lets I forget, I built my demo project 3 times because I kept having issues which were peculiar to a newbie like me but at the end I was able to develop a working app..