# Authentication and Authorization

# Goals

- Any user may see the photos
- Only an authenticated user may add a new photo
- Only the owner of a photo may delete it

## Authentication

- Go to https://docs.microsoft.com/en-us/aspnet/core/security/authentication/scaffold-identity?view=aspnetcore-6.0&tabs=visual-studio#scaffold-identity-into-a-razor-project-without-existing-authorization and follow the instructions to scaffold Identity into a Razor project without existing authorization.
  At the end of the process, you should be able to register and logon to the site.

## Authorization

### Allow only authenticated users to add photos.

First, we need to add the concept of ownership to our `Photo` model, by adding a `SubmittedBy` property of type string to the Photo model.

```
 namespace PhotoSharingApplication.Shared.Entities;

public class Photo {
    public int Id { get; set; }
    public string Title { get; set; } = string.Empty;
    public string Description { get; set; } = string.Empty;
    public byte[] PhotoFile { get; set; }
    public string ContentType { get; set; } = string.Empty;
    public string SubmittedBy { get; set; } = string.Empty;
    public DateTime SubmittedOn { get; set; }
    public List<Comment>? Comments { get; set; }
}
```

Then, we need to save the User Name into our new `SubmittedBy` property during upload.

Authorization in ASP.NET Core is controlled with `AuthorizeAttribute` and its various parameters. In its most basic form, applying the `[Authorize]` attribute to a controller, action, or Razor Page, limits access to that component authenticated users.
Add the `[Authorize]` attribute to the `Upload` page.
In the `OnPost` method, add the following code:

```
 Photo.SubmittedBy = User?.Identity?.Name;
```

Modify the `PhotoDetailsPartial` partial view to include the name of the user who submitted the photo.
At this point you should be able to register a user, log on, upload a photo, and see the name of the user who submitted the photo.
When trying to upload a photo without logging on first, you should see the login page.

## Delete

We haven't implemented the deletion of a photo yet, so let's start by implementing this feature without any security involved, just to see it working first.
- Add a `DeletePhotoAsync` method to the `IPhotosService` interface. - Add a `DeletePhotoAsync` method to the `IPhotosRepository` interface. - Implement the `DeletePhotoAsync` method in the `PhotosService` class. - Implement the `DeletePhotoAsync` method in the `PhotosRepository` class. - Add a `Delete` Razor Page to the `Pages\Photos` folder of the Web project. - Add a `Photo` property - Implement the `OnGet` method by accepting an `id`, retrieving the photo through the service and setting the `Photo` property, eventually returning a `NotFound` if the Photo does not exist. - Implement the `OnPost` method by accepting an `id`, calling the `DeletePhotoAsync` method and redirecting to the `Index` action. - In the Delete.cshtml file, show the details of the photo and add a form to invoke the OnPost method. By this point, you should be able to delete a photo, even if you're not logged on or if you're not the owner of a photo.

### Allow only the owner of a photo to delete it.

Authorization strategy depends upon the resource being accessed. Consequently, the photo must be retrieved from the data store before authorization evaluation can occur.

Attribute evaluation occurs before data binding and before execution of the page handler or action that loads the document. For these reasons, declarative authorization with an `[Authorize]` attribute doesn't suffice. Instead, you can invoke a custom authorization method — a style known as *imperative authorization*.

Authorization is implemented as an `IAuthorizationService` service and is registered in the service collection. The service is made available via dependency injection to page handlers or actions. In the constructor of the `DeleteModel` page, add an `IAuthorizationService authorizationService` parameter and save it into a private readonly field.

`IAuthorizationService` has two `AuthorizeAsync` method overloads: one accepting the resource and the policy name and the other accepting the resource and a list of requirements to evaluate.

In both the `OnGet` and `OnPost` methods, retrieve the `Photo`, then call the `AuthorizeAsync` method with the `Photo` and the `PhotoDeletion` policy. Return a `ForbidResult` if the authorization fails. Continue with the normal operations if the authorization succeeds.

Writing a handler for resource-based authorization isn't much different than writing a plain requirements handler. Create a custom requirement class, and implement a requirement handler class.

The handler class specifies both the requirement and resource type.

```
public class PhotoOwnerAuthorizationHandler : AuthorizationHandler<PhotoOwnerRequirement, Photo> {
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
                                                   PhotoOwnerRequirement requirement,
                                                   Photo photo) {
        if (context.User.Identity?.Name == photo.SubmittedBy) {
            context.Succeed(requirement);
        }

        return Task.CompletedTask;
    }
}

public class PhotoOwnerRequirement : IAuthorizationRequirement { }
```

Register the requirement and handler in `Program.cs`:

```
builder.Services.AddAuthorization(options => {
    options.AddPolicy("PhotoDeletionPolicy", policy => {
        policy.RequireAuthenticatedUser();
        policy.Requirements.Add(new PhotoOwnerRequirement());
    });
});

builder.Services.AddSingleton<IAuthorizationHandler, PhotoOwnerAuthorizationHandler>();
```

At this point, you should be able to delete a photo only when logged on with the user that submitted the photo in the first place. When trying to delete a photo without logging on first, you should see the login page. When trying to delete a photo that you don't own, you should see an `Access Denied`.

### Modify the UI based on the current user identity

To inject the authorization service into a Razor view, use the @inject directive:

```
@using Microsoft.AspNetCore.Authorization
@inject IAuthorizationService AuthorizationService
```

Use the injected authorization service to invoke AuthorizeAsync in exactly the same way you would check during resource-based authorization:

```
bool userIsAuthorized = (await AuthorizationService.AuthorizeAsync(User, Model, "PhotoDeletionPolicy")).Succeeded;
```

## Resources

- https://docs.microsoft.com/en-us/aspnet/core/security/authentication/scaffold-identity?view=aspnetcore-6.0&tabs=visual-studio#scaffold-identity-into-a-razor-project-without-existing-authorization
- https://docs.microsoft.com/en-us/aspnet/core/security/authorization/resourcebased?view=aspnetcore-6.0
- https://docs.microsoft.com/en-us/aspnet/core/security/authorization/views?view=aspnetcore-6.0
- https://docs.microsoft.com/en-us/aspnet/core/security/authorization/limitingidentitybyscheme?view=aspnetcore-6.0