Razor Pages

Goals

In this lab, we're going to build a first (simple) version of our Razor Pages.

- **Index** - Razor page with list of all photos
- **Upload** - Razor page to create a new photo
- **Details** - Razor page to see an existing photo

Setup

The associations of URL paths to pages are determined by the page's location in the file system.
The runtime looks for Razor Pages files in the Pages folder by default.
Index is the default page when a URL doesn't include a page.
We want to create - a page mapped to the URL `/photos` - a page mapped to the URL `/photos/upload` - a page mapped to the URL `/photos/details` This means we need to create a `Photos` folder under the `Pages` folder of the `PhotoSharingApplication.Web` project.

Index

Inside the `Photos` folder, create a `Index` Razor Page. This will create two files: `Index.cshtml` and `Index.cshtml.cs`.
In the latter file, we find a class named `IndexModel` that inherits from `PageModel`.
This class needs a public property named `Photos` that is a list of `Photo` objects. We need it so that the content page can display the list of photos as html.
In order to fill the list from the photos coming from the backend logic we built in the previous lab, we need to explicitly declare a dependency on the IPhotosService in the constructor, save the parameter in a private field and use the field during Get.
In the class that was generated for us, we find a method named `OnGet` that is called when the page is loaded. Since we need to invoke an asynchronous method, we need to change the signature of the method to `public async Task OnGetAsync()`. In this method, we use the service to fill the `Photos` list.
The code below shows the Razor Page code.

```
 using Microsoft.AspNetCore.Mvc.RazorPages;
using PhotoSharingApplication.Core.Entities;
using PhotoSharingApplication.Core.Interfaces;

namespace PhotoSharingApplication.Web.Pages.Photos;

public class IndexModel : PageModel {
    private readonly IPhotosService photosService;

    public IEnumerable<Photo> Photos { get; set; }
    public IndexModel(IPhotosService photosService) {
        this.photosService = photosService;
    }
    public async Task OnGetAsync() {
        Photos = await photosService.GetAllPhotosAsync();
    }
}
```

Now, onto the content page, where we can use a `foreach` to display a `div` for each item in the `Model.Photos` property.

```
 @page
@using PhotoSharingApplication.Core.Entities
@model PhotoSharingApplication.Web.Pages.Photos.IndexModel
@{
}

@foreach (Photo item in Model.Photos)
{
    <div>
        <h3>@item.Title</h3>
        <p>@item.Id</p>
        <p>@item.Description</p>
    </div>
}
```

If you run the application now and navigate to `/photos`, you should see the the list of photos.

Details

Create a `Details` Razor page and add the `IPhotosService` as a dependency in the constructor.
Add a propertyof type Photo.
Since we need to know which photo we want to show, we need a parameter in the URL tha we can map to a parameter in the `OnGet` method which, once again, need to be `async`.
We cannot just return a `Task` as we did in the `Index` page, though, because this time we need to take two different paths, depending on the existance of the photo: - If the photo cannot be found, we should return a `NotFoundResult` - If the photo does exist, we can return the `Page` The code becomes:

```
 using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using PhotoSharingApplication.Core.Entities;
using PhotoSharingApplication.Core.Interfaces;

namespace PhotoSharingApplication.Web.Pages.Photos;

public class DetailsModel : PageModel {
    private readonly IPhotosService photosService;

    public DetailsModel(IPhotosService photosService) {
        this.photosService = photosService;
    }
    public Photo? Photo { get; set; }
    public async Task<IActionResult> OnGet(int id) {
        Photo = await photosService.GetPhotoByIdAsync(id);
        if (Photo is null) {
            return NotFound();
        }
        return Page();
    }
}
```

In the `.cshtml` content page, let's configure the `Route` so that the `id` parameter can be appended as to the URL instead of being passed as a query string.
Then, let's display the photo's title and description.

```
 @page "{id:int}"
@model PhotoSharingApplication.Web.Pages.Photos.DetailsModel
@{
}
<div>
    <h3>@Model.Photo.Title</h3>
    <p>@Model.Photo.Id</p>
    <p>@Model.Photo.Description</p>
</div>
```

If you run the application and navigate to `/photos/1`, you should see the details of the photo with id 1.

Upload

The `Upload` Razor page also need the usual dependency on the `IPhotosService`, specified in the constructor.
We also need to add a property of type `Photo` to hold the photo we are uploading. This time, though, we need to add the `[BindProperty]` attribute to the property.
The `OnGet` method does not need to retrieve anything, so we can leave it empty.
The `OnPost` method will be called when the user submits the form. This is where we need to save the photo. When we're done, we can redirect to the `Index` page.

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using PhotoSharingApplication.Core.Entities;
using PhotoSharingApplication.Core.Interfaces;

namespace PhotoSharingApplication.Web.Pages.Photos;

public class UploadModel : PageModel {
    private readonly IPhotosService photosService;

    [BindProperty]
    public Photo Photo { get; set; }

    public UploadModel(IPhotosService photosService) {
        this.photosService = photosService;
    }
    public void OnGet() {
    }
    public async Task<IActionResult> OnPostAsync() {
        await photosService.AddPhotoAsync(Photo);
        return RedirectToPage("./Index");
    }
}
```

The `.cshtml` content page contains a form that submits to the `/photos/upload` URL using the `post` method. In the form, we add an `input` bound to the `Photo.Title` property and a `textarea` bound to the `Photo.Description` property. Plus, of course, a submit button.

```
@page
@model PhotoSharingApplication.Web.Pages.Photos.UploadModel
@{
}
<form method="post">
    <input asp-for="Photo.Title" />
    <textarea asp-for="Photo.Description"></textarea>
    <input type="submit" />
</form>
```

If you run the application and navigate to `/photos/upload`, you should see the upload form. You can type in the title and description and submit the form. You should be redirected to the `Index` page after the photo is saved, where you should see the data you just added.

Lessons learned

- Dependency injection in Razor Pages
- Page Routing
- Page Actions
- Model Binding

References

- https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-6.0
- https://docs.microsoft.com/en-us/aspnet/core/razor-pages/?view=aspnetcore-6.0&tabs=visual-studio
- https://docs.microsoft.com/en-us/aspnet/core/tutorials/razor-pages/razor-pages-start?view=aspnetcore-6.0&tabs=visual-studio