File Upload and Download

Goals

So far, we stored only the metadata of a photo. It's time to tackle the upload and download of a file, so that we can expand our feature to include the actual image.

The first step will be to update the Photo Entity and the DB schema.

For the Upload feature we're going to:

- Modify the `Upload` Content Page:
  - Add an html tag to select a file on the user device.
  - Modify the form to send the file along with the metadata
- Modify the `Upload` Page Model:
  - Add a property to store the file sent through the form
  - Read the content of the file and store the necessary data into our Photo Entity

For the Download feature - We're going to need a Minimal API to retrieve the file: we'll map a route to a method that will return the file if the photo exists or a notfound otherwise. - We're going to update the content pages to show an image whose source will be the result of a call to the Minimal Api - We're going to refactor the pages to avoid repeating ourselves.

Update the Photo Entity

There are two pieces of information that we need to store: the file bytes and the file content type. We can store the first as an array and the second as a string. The Entity becomes:

```
namespace PhotoSharingApplication.Core.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; }
}
```

Now we can update the schema of the DB by going to the Package Manager Console and executing the following commands: `Add-Migration PhotoFile`

follwed by:

`Update-Database`

Upload Feature

- Modify the `Upload` Content Page:
  - Add an html tag to select a file on the user device.
  - Modify the form to send the file along with the metadata

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

- Modify the `Upload` Page Model:
  - Add a property to store the file sent through the form
  - Read the content of the file and store the necessary data into our Photo Entity before saving it to the DB

```
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; }

    [BindProperty]
    public IFormFile FormFile { get; set; }

    public UploadModel(IPhotosService photosService) {
        this.photosService = photosService;
    }
    public void OnGet() {
    }
    public async Task<IActionResult> OnPostAsync() {
        using (var memoryStream = new MemoryStream()) {
            await FormFile.CopyToAsync(memoryStream);
            Photo.PhotoFile = memoryStream.ToArray();
            Photo.ContentType = FormFile.ContentType;
        }

        await photosService.AddPhotoAsync(Photo);
        return RedirectToPage("./Index");
    }
}
```

At this point, you should be able to select pictures from your device and upload the in the DB.

Download

A Minimal API is what we can use to retrieve the file and send it to the client.
Let's map a route in GET to receive the id of the photo to download and return the file if the photo exists or a notfound otherwise. To make use of the `IPhotosService`, we can just ask for it the method itself.
In `Program.cs` of the `PhotoSharingApplication.Web` project, between the `app.UseAuthorization();` and `app.MapRazorPages()` lines, add the following code:

```
app.MapGet("/photos/image/{id:int}", async (int id, IPhotosService photosService) => {
    Photo? photo = await photosService.GetPhotoByIdAsync(id);
    if (photo is null || photo.PhotoFile is null) {
        return Results.NotFound();
    }
    return Results.File(photo.PhotoFile, photo.ContentType);
});
```

Now let's update the content pages to show an image whose source will be the result of a call to the Minimal Api. Let's start simple and just show the image in both the `Index` and `Details` content pages. Add the following code to the `Index.cshtml` file, inside the `foreach` loop:

```
<img src="/photos/image/@item.Id" />
```

Add the following code to the `Details.cshtml` file:

```
<img src="/photos/image/@Model.Photo.Id" />
```

If you run the application, you should be able to see the pictures you uploaded in the previous step.

As you've seen, the `Index` and `Details` content pages are now very similar. We can refactor them to avoid repeating ourselves.

Refactor the content pages

We can move the repeated code into a `Partial View`. The only problem is that the `Index` Page should display a link to the `Details` page for each photo in the list, while the `Details` page should not. This means that the partial view needs to know if hte `Details` link should be desplayed or not.
One technique we can use is to use a variable in the `ViewData` bag. We'll set this variable in the `Index` and in the `Details`, then read it in the `Partial View`.

Let's create a new `Partial View` called `_PhotoDetailsPartial.cshtml` and add the following code to it:

```
@using PhotoSharingApplication.Core.Entities
@model Photo

<div>
    <h3>@Model.Title</h3>
    <p>@Model.Id</p>
    <p>@Model.Description</p>
    <img src="/photos/image/@Model.Id" />
    @if (ViewData["ShowDetailsButton"] is bool showDetailsButton && showDetailsButton) {
        <a asp-page="./Details" asp-route-id="@Model.Id">Details</a>
    }
</div>
```

The `Index.cshtml` can now become

```
@page
@using PhotoSharingApplication.Core.Entities
@model PhotoSharingApplication.Web.Pages.Photos.IndexModel
@{
    ViewData["ShowDetailsButton"] = true;
}

@foreach (Photo item in Model.Photos)
{
    <partial name="_PhotoDetailsPartial" model="item" view-data="ViewData"></partial>
}
```

While the `Details.cshtml` becomes:

```
@page "{id:int}"
@model PhotoSharingApplication.Web.Pages.Photos.DetailsModel
@{
    ViewData["ShowDetailsButton"] = false;
}
<partial name="_PhotoDetailsPartial" for="Photo" view-data="ViewData"></partial>
```

Running the application should give the same results as before, but our code now follows the `DRY` principle.

Lessons learned

- TagHelpers
- ViewData
- PartialViews
- Upload Files
- Download files

- Minimal API

References

- https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/built-in/?view=aspnetcore-6.0

- https://docs.microsoft.com/en-us/aspnet/core/mvc/views/partial?view=aspnetcore-6.0
- https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/built-in/partial-tag-helper?view=aspnetcore-6.0
- https://docs.microsoft.com/en-us/aspnet/core/mvc/models/file-uploads?view=aspnetcore-6.0
- https://docs.microsoft.com/en-us/aspnet/core/fundamentals/minimal-apis?view=aspnetcore-6.0