Caching, State and SignalR

Goals

- Improve the performances by caching the image
- Store Favorite photos in Session
- Build a chat with SignalR to chat about a photo

Improving the performances

We currently get unnecessary information from our db way too many times.
By splitting our model in two parts, one for the metadata and one for the image, we can start to improve the performances.
Our minimal API will only get the image in order to return a file, while every other part of the application will only get the metadata from the db.
This is already going to improve the performances.
The next step will be to cache the image server side. This will allow us to load the image from the db only once, and then return it to the user.

Table splitting

To use table splitting the entity types need to be mapped to the same table, have the primary keys mapped to the same columns and at least one relationship configured between the primary key of one entity type and another in the same table.
Let's add a new `Image` Entity. We will have an Id and we will move the `PhotoFile` and `ContentType` properties from the `Photo` entity to the `Image` entity:

```
public class Image {
    public int Id { get; set; }
    public byte[] PhotoFile { get; set; }
    public string ContentType { get; set; } = string.Empty;
}
```

The `Photo` Entity becomes:

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

The required configuration needs to be added to the `OnModelCreating` of our `PhotoSharingDbContext`, which becomes:

```
protected override void OnModelCreating(ModelBuilder modelBuilder) {
    modelBuilder.Entity<Photo>()
        .Property(b => b.Title)
        .HasMaxLength(100);
    modelBuilder.Entity<Photo>()
        .Property(b => b.Description)
        .HasMaxLength(250);

    modelBuilder.Entity<Photo>(
        photoBuilder => {
            photoBuilder.ToTable("Photos");
            photoBuilder.HasOne(o => o.Image).WithOne().HasForeignKey<Image>(o => o.Id);
        });

    modelBuilder.Entity<Image>(imageBuilder => imageBuilder.ToTable("Photos"));

    modelBuilder.Entity<Image>()
        .Property(b => b.ContentType)
        .HasMaxLength(30);

    modelBuilder.Entity<Comment>()
        .Property(b => b.Title)
        .HasMaxLength(100);
    modelBuilder.Entity<Comment>()
        .Property(b => b.Body)
        .HasMaxLength(250);
}
```

Let's also add a `DbSet<Image>` to our `PhotoSharingDbContext`:

```
public DbSet<Image> Images { get; set; }
```

Now we need to change the `OnPostAsync` method of the `Upload` feature to use the `Image` entity:

```
using (var memoryStream = new MemoryStream()) {
    await FormFile.CopyToAsync(memoryStream);
    Photo.Image = new() { ContentType = FormFile.ContentType, PhotoFile = memoryStream.ToArray() };
}
```

We also need to create a new validator for the `Image` entity:

```
using FluentValidation;
using PhotoSharingApplication.Shared.Entities;

namespace PhotoSharingApplication.Shared.Validators;

public class ImageValidator : AbstractValidator<Image> {
    public ImageValidator() {
        RuleFor(image => image.ContentType).NotEmpty();
    }
}
```

And change the validator for the `Photo` entity to use the `ImageValidator`:

```
using FluentValidation;
using PhotoSharingApplication.Shared.Entities;

namespace PhotoSharingApplication.Shared.Validators;

public class PhotoValidator : AbstractValidator<Photo> {
    public PhotoValidator() {
        RuleFor(photo => photo.Title).NotEmpty().MaximumLength(100);
        RuleFor(photo => photo.Description).NotEmpty().MaximumLength(250);
        RuleFor(photo => photo.Image).NotEmpty().SetValidator(new ImageValidator());
    }
}
```

Let's add a new method on the `IPhotosService` and `IPhotosRepository` interfaces to get an `Image` given an Id:

```
Task<Image?> GetImageByIdAsync(int id);
```

Let's implement it in the `PhotosService`:

```
public Task<Image?> GetImageByIdAsync(int id)=> photosRepository.GetImageByIdAsync(id);
```

And in the `PhotosRepositoryEF`:

```
public async Task<Image?> GetImageByIdAsync(int id) => await dbContext.Images.FirstOrDefaultAsync(p => p.Id == id);
```

Now that we have the methods in place, we can use them in our Minimal Api, lacated in the `Program.cs` of our `PhotoSharingApplication.Web` project:

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

If you have written tests for the minimal Api, you're going to have to change those as well.

At this point, you should be able to start the application and nothing should break. The difference so far is that the queries that we execute are more efficient, since we only load the data we need in each operation.

Caching

ASP.NET Core supports several different caches. The simplest cache is based on the `IMemoryCache`. `IMemoryCache` represents a cache stored in the memory of the web server.
The in-memory cache can store any object, as key-value pairs.
In-memory caching is a service that's referenced from an app using Dependency Injection. Request the `IMemoryCache` instance in the constructor. You can write code that uses TryGetValue to check if an item is in the cache. If an item isn't cached, a new entry is created and added to the cache with Set.

One strategy we can use, is to create a new interface `IPhotosServiceCache` that inherits from the `IPhotosService` interface, then create a `PhotosServiceCache` class that inherits from `PhotosService` and implements the `IPhotosServiceChace` interface. This class will depend on an `IMemoryCache` instance. The `GetImageByIdAsync` will use the cache to store and retrieve the data.
Our Minimal Api will depend on the new interface.

- In the `PhotoSharingApplication.Web` project, add a new folder `Services`
- In the `Services` folder, add a new file `IPhotosServiceCache.cs`
- In the `IPhotosServiceCache.cs` file, add the following code:

```
using PhotoSharingApplication.Core.Interfaces;

namespace PhotoSharingApplication.Web.Services {
    public interface IPhotosServiceCache : IPhotosService {
    }
}
```

- In the `Services` folder, add a new file `PhotosServiceCache.cs`
- In the `PhotosServiceCache.cs` file, add the following code:

```
using Microsoft.Extensions.Caching.Memory;
using PhotoSharingApplication.Core.Interfaces;
using PhotoSharingApplication.Core.Services;
using PhotoSharingApplication.Shared.Entities;
using PhotoSharingApplication.Shared.Validators;

namespace PhotoSharingApplication.Web.Services {
    public class PhotosServiceCache : PhotosService, IPhotosServiceCache {
        private readonly IMemoryCache cache;

        public PhotosServiceCache(IPhotosRepository repository, PhotoValidator validator, IMemoryCache cache) : base(repository, val
        public async Task<Image?> GetImageByIdAsync(int id) {
            string key = $"image-{id}";
            Image? image;
            if (!cache.TryGetValue(key, out image)) {
                image = await base.GetImageByIdAsync(id);
                // Set cache options.
                var cacheEntryOptions = new MemoryCacheEntryOptions()
                    // Keep in cache for this time, reset time if accessed.
                    .SetSlidingExpiration(TimeSpan.FromMinutes(10));

                // Save data in cache.
                cache.Set(key, image, cacheEntryOptions);
            }
            return image;
        }
    }
}
```

In the `ServiceCollectionExtensions.cs` file, add the following code:

```
services.AddScoped<IPhotosServiceCache, PhotosServiceCache>();
```

In `Program.cs`, change the dependency of the MinimalApi from `IPhotosService` to `IPhotosServiceCache`:

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

If you run the application, everything should work as expected, but you may notice an improvement in the performances.

Store Favorite photos in Session

The next feature we want to add consists in giving users the chance to add a photo to their favorites, regardless of their login status. This will be possible by clicking on a button on the Details page. Then we're going to have a new page where they will be able to browse through their favorite photos.
We want to keep the session as small as possible, so we will store a HashSet with the just the id of the photos in the session. To retrieve the photos, we will need a new method on the service and repository to return a list of photos given a list of id.

Adding items in session

We're going to add a new button in the `Details` page of the `Pages\Photos` folder in the `PhotoSharingApplication.Web` project. The button will invoke a new method which will store the id of the photo in the session. Let's also give a name to this method so that we can clearly see its purpose: `OnPostAddToFavorites`.

- In the `Details.cshtml` file, add the following code:

```
<form method="POST">
    <input type="submit" asp-page-handler="AddToFavorites" value="� Add To Favorites ♥" class="btn btn-secondary" />
</form>
```

In the `Details.cshtml` file, we will add an `OnPostAddToFavorites` method accepting an `int id`.

This method will try to retrieve a `HashSet<int>` from session, creating a new instance if it doesn't find it.
Then, it will add the id to the set and store the set back in the session.
It will also retrieve the photo so that the page can be rendered correctly.

Session state is accessed from a Razor Pages PageModel class with `HttpContext.Session`. This property is an `ISession` implementation.

The `ISession` implementation provides several extension methods to set and retrieve integer and string values. The extension methods are in the `Microsoft.AspNetCore.Http` namespace.
All session data must be serialized to enable a distributed cache scenario, even when using the in-memory cache. String and integer serializers are provided by the extension methods of `ISession`. Complex types must be serialized by the user using another mechanism, such as JSON.

So not only do we need to implement the OnPostAddToFavorites method, we also need to add two extension methods to the `ISession` interface.

This is the `OnPostAddToFavorites` method:

```
 public async Task<IActionResult> OnPostAddToFavorites(int id) {
     string key = "favoritePhotos";
     HashSet<int> favorites = HttpContext.Session.Get<HashSet<int>>(key) ?? new();
     favorites.Add(id);
     HttpContext.Session.Set(key, favorites);
     Photo = await photosService.GetPhotoByIdAsync(id);
     return Page();
}
```

These are the extension methods:

```
 using System.Text.Json;

namespace PhotoSharingApplication.Web.Sessions;

public static class SessionExtensions {
    public static void Set<T>(this ISession session, string key, T value) => session.SetString(key, JsonSerializer.Serialize(value))

    public static T? Get<T>(this ISession session, string key) {
        string? value = session.GetString(key);
        return value == null ? default : JsonSerializer.Deserialize<T>(value);
    }
}
```

To enable the session middleware, Progam.cs must contain:

- Any of the `IDistributedCache` memory caches. The `IDistributedCache` implementation is used as a backing store for session. We're going to use `DistributedMemoryCache` for this.
- A call to `AddSession`
- A call to `UseSession` The order of middleware is important. Call `UseSession` after `UseRouting` and before `MapRazorPages` and `MapControllers`.

At this point, you should be able to add favorites in session.

Retrive items from session

We will create a new page to show the favorites photos saved in Session.
The html of the page shares the same UI as the Index, so we will move the UI on a partial view and use that from both the Index and the Favorites pages.
The OnGet of the Favorites page will retreive the HashSet from session and pass it to a new method of the service, which will in turn invoke a new method of the repository, which will run a query to return the photos given the ids.

- Add a new `_PhotosList.cshtml` partial view to the `Pages\Photos` folder of the `PhotoSharingApplication.Web` project.
- Copy the code of the `Pages\Photos\Index.cshtml` file, modifying it so that it works with an `IEnumerable<Photo>`.

```
@model IEnumerable<Photo>
@{
    ViewData["ShowDetailsButton"] = true;
    ViewData["ShowDeleteButton"] = true;
    ViewData["ShowConfirmDeleteButton"] = false;
}
<div class="row row-cols-1 row-cols-md-2 g-4">
@foreach (Photo item in Model)
{
    <div class="col">
    <partial name="_PhotoDetailsPartial" model="item" view-data="ViewData"></partial>
    </div>
}
</div>
```

Modify the `Pages\Photos\Index.cshtml` file to invoke the partial view:

```
@page
@model PhotoSharingApplication.Web.Pages.Photos.IndexModel

<partial name="_PhotosListPartial" model="Model.Photos" view-data="ViewData"></partial>
```

At this point, if you run the application everything should still work as before.

Add a new `Favorites` Razor Page to your `Pages\Photos` folder. Set the content of the `cshtml` file to

```
@page
@model PhotoSharingApplication.Web.Pages.Photos.FavoritesModel

<partial name="_PhotosListPartial" model="Model.Photos" view-data="ViewData"></partial>
```

Replace the code of the `.cshtml.cs` file with the following code:

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

namespace PhotoSharingApplication.Web.Pages.Photos;

public class FavoritesModel : PageModel {
    private readonly IPhotosService photosService;

    public IEnumerable<Photo> Photos { get; set; }
    public FavoritesModel(IPhotosService photosService) {
        this.photosService = photosService;
    }
    public async Task OnGetAsync() {
        HashSet<int> favorites = HttpContext.Session.Get<HashSet<int>>("favoritePhotos") ?? new();
        Photos = await photosService.GetSetOfPhotosAsync(favorites);
    }
}
```

Now add the following code to both the `PhotoSharingApplication.Core.Interfaces.IPhotosService` and the `PhotoSharingApplication.Core.Interfaces.IPhotosRepository` interfaces

```
Task<IEnumerable<Photo>> GetSetOfPhotosAsync(IEnumerable<int> ids);
```

Add the following code to `PhotoSharingApplication.Core.Services.PhotosService`:

```
public Task<IEnumerable<Photo>> GetSetOfPhotosAsync(IEnumerable<int> ids) => photosRepository.GetSetOfPhotosAsync(ids);
```

Add the following code to `PhotoSharingApplication.Infrastructure.Repositories.PhotosRepository`:

```
public async Task<IEnumerable<Photo>> GetSetOfPhotosAsync(IEnumerable<int> ids) => await dbContext.Photos.Where(p => ids.Contains(p.
```

Add a new navigation link to the menu items in the `_Layout.cshtml` file.

```
<li class="nav-item">
    <a class="nav-link text-dark" asp-area="" asp-page="/Photos/Favorites">Favorites</a>
</li>
```

Run the application. Going to the favorites address should display an empty page, but after adding a couple of photos to the favorites, the page should display the photos.

Build a chat with SignalR to chat about a photo

The last feature of this chapter involves building a chat application with SignalR. Whenever users visit the Details page of a Photo, they can join a conversation with all the other users who are visiting the same photo.
Server side, we're going to build a SignalR hub which will receive messages from a user and broadcast them to all the other users.
Client side, we're going to build a Blazor Component containing a SignalR client which will receive messages from the server and display them in the chat window.

The Server Side SignalR Hub

Add a new class deriving from `Microsoft.AspNetCore.SignalR.Hub` to the `PhotoSharingApplication.Web` project. Implement two methods, one to have a client join a group and one to get a message from a client and broadcast to a specific group.
A group is a collection of connections associated with a name. Messages can be sent to all connections in a group. Groups are the recommended way to send to a connection or multiple connections because the groups are managed by the application. A connection can be a member of multiple groups. Groups are ideal for something like a chat application, where each room can be represented as a group. Connections are added to or removed from groups via the AddToGroupAsync and RemoveFromGroupAsync methods.
The code becomes:

```
using Microsoft.AspNetCore.SignalR;

namespace PhotoSharingApplication.Web.Chat;

public class ChatHub : Hub {
    public async Task SendMessage(string user, string message, int groupId) =>
        await Clients.Group($"photoId-{groupId}").SendAsync("ReceiveMessage", user, message);
    public async Task JoinGroup(int groupId) =>
        await Groups.AddToGroupAsync(Context.ConnectionId, $"photoId-{groupId}");
}
```

Add SignalR and Response Compression Middleware services to Program.cs

```
builder.Services.AddSignalR();
builder.Services.AddResponseCompression(opts => {
    opts.MimeTypes = ResponseCompressionDefaults.MimeTypes.Concat(
        new[] { "application/octet-stream" });
});
```

Use Response Compression Middleware at the top of the processing pipeline's configuration.

```
app.UseResponseCompression();
```

Add an endpoint for the hub.

```
app.MapHub<ChatHub>("/chathub");
```

The Blazor Component with the SignalR Client

Add the `Microsoft.AspNetCore.SignalR.Client` Package to the `PhotoSharing.Blazor.Client` project.
In the `Components` folder of the `PhotoSharing.Blazor.Client` project, add a new `ChatComponent` Razor Component. In the HTML section of the component, add two input fields where the user can write their own nicknames and messages respectively. Add a button to send the message to the hub. Also add a unordered list to display the messages coming from the server.

In the code section, during the OnInitializeAsync, use the `HubComnnectionBuilder` to build a hubConnection to the hub. Then, use the `HubConnection.StartAsync` method to start the connection and invoke the `HubConnection.InvokeAsync` to call the `JoinGroup`, sending the id of the Photo as a group Id.
Register a function to handle the `ReceiveMessage` method. This function will receive the message from the server and add them to a list, so that they can be shown in the unordered list.

When the user clicks the button, use the `HubConnection.InvokeAsync` method to send a message to the hub.
Dispose of the hub when the component is disposed.

The code becomes:

```
 @using Microsoft.AspNetCore.SignalR.Client
@inject NavigationManager NavigationManager
@implements IAsyncDisposable

<div class="form-group mb3">
    <label class="form-label">
        User:
        <input @bind="userInput" class="form-control" />
    </label>
</div>
<div class="form-group mb3">
    <label class="form-label">
        Message:
        <input @bind="messageInput" size="50" class="form-control"/>
    </label>
</div>
<button @onclick="Send" disabled="@(!IsConnected)" class="btn btn-primary">Send</button>

<hr>

<ul id="messagesList" class="list-group">
    @foreach (var message in messages)
    {
        <li class="list-group-item">@message</li>
    }
</ul>

@code {
    [Parameter]
    public int PhotoId { get; set; }

    private HubConnection? hubConnection;
    private List<string> messages = new List<string>();
    private string? userInput;
    private string? messageInput;

    protected override async Task OnInitializedAsync()
    {
        hubConnection = new HubConnectionBuilder()
            .WithUrl(NavigationManager.ToAbsoluteUri("/chathub"))
            .Build();

        hubConnection.On<string, string>("ReceiveMessage", (user, message) =>
        {
            var encodedMsg = $"{user}: {message}";
            messages.Add(encodedMsg);
            StateHasChanged();
        });

        await hubConnection.StartAsync();
        await hubConnection.SendAsync("JoinGroup", PhotoId);
    }

    private async Task Send()
    {
        if (hubConnection is not null)
        {
            await hubConnection.SendAsync("SendMessage", userInput, messageInput, PhotoId);
        }
    }
    public bool IsConnected =>
        hubConnection?.State == HubConnectionState.Connected;

    public async ValueTask DisposeAsync()
    {
        if (hubConnection is not null)
        {
            await hubConnection.DisposeAsync();
        }
    }
}
```

To use the component, add the following code to the `Details` page of the `Pages\Photos` folder of the `PhotoSharingApplication.Web` project:

```
<component type="typeof(ChatComponent)" render-mode="WebAssemblyPrerendered" param-PhotoId="@Model.Photo.Id"/>
```

Try running the application, opening multiple tabs of the same photo and see the chat messages. Tabs open on a different photo should not receive those messages.

Resources

- https://docs.microsoft.com/en-us/ef/core/modeling/table-splitting
- https://docs.microsoft.com/en-us/aspnet/core/performance/caching/response?view=aspnetcore-6.0
- https://docs.microsoft.com/en-us/aspnet/core/fundamentals/app-state?view=aspnetcore-6.0
- https://docs.microsoft.com/en-us/aspnet/core/razor-pages/?view=aspnetcore-6.0&tabs=visual-studio#multiple-handlers-per-page
- https://docs.microsoft.com/en-us/aspnet/core/tutorials/signalr-blazor?view=aspnetcore-6.0&tabs=visual-studio&pivots=webassembly
- https://docs.microsoft.com/en-us/aspnet/core/signalr/hubs?view=aspnetcore-6.0
- https://docs.microsoft.com/en-us/aspnet/core/signalr/groups?view=aspnetcore-6.0