Setting up the Environment

Using .NET

https://dotnet.microsoft.com/en-us/download

Using Node.js

Start with "Chocolaty"
https://chocolatey.org/install#individual
Get version number here:
https://nodejs.org/en/
Use NVM-Windows to manage node.js installs through PowerShell
https://joachim8675309.medium.com/installing-node-js-with-nvm-4dc469c977d9
Follow the instructions above
NOTE: after "refreshenv" command, close PowerShell and reopen
NOTE: type out version number, ex: "nvm install 16.10.0" instead of "$version"

Refreshing Node and npm

Start with the command: npm cache clean --force
Use the installer .msi to "Remove" node

Go to /Program Files/nodejs and remove all references
Go to %AppData%Roaming/npm and remove all references
Go to “settings/system/About/Advanced System Settings/, open “Environment Variables”, remove any references to “npm” or “node”
Restart, reinstall using .msi installer

Using Visual Studio Code

https://code.visualstudio.com/
C# for VS Code
C# Extensions
Material Icon Theme
File >> Preferences >> Settings : Search “exclude” to hide certain file types and folders

Using Postman

https://www.postman.com/downloads/

Building a Walking Skeleton

Dotnet CLI

CD to desired location, dir to list contents, mkdir to make a new folder
dotnet –info : display the currently installed dotnet SDK’s on system
dotnet -h : get help, list all commands for dotnet
dotnet new -h : get list of all ways to use the ‘new’ command;

Create a new project

dotnet new sln : create a new solution (keeps name of the parent folder)
dotnet new webapi -o API : create new web API in folder called ‘API’
dotnet sln add API/ : add the solution to the API

In launchSettings.json : change “applicationUrl” to "https://localhost:5001;http://localhost:5000"

Running the App

Open VS terminal (ctl + `), cd into the API, type “dotnet run” and build the project
Stop the app: ctl + c, ‘clear’ to clear console

Certificate and Viewing in the Browser

Type: dotnet dev-certs https –trust
https://localhost:5001 will be the basic address, however the API controller determines the "route". Check the API controller for the route to find the app. Example: weatherforecastcontroller has a [Route("[controller]")], where "controller" gets replaced with "Weather Forecast"
NOTE: changing the name of the "public class" in the controller will change the address that is needed to visit to view the controller.
NOTE: "add dotnet watch" run in terminal to do hot reloads when working on files

Creating an Entity

Most applications will contain "users"; so create a new folder called "Entities" (entities are abstractions of physical things, like a user). The 'user' entity will contain attributes that a user will have.
Create a new class called "AppUser" in the "Entities" folder

Prop Shortcut

type 'prop' and press tab to auto-gen property; type 'prop' and select 'propfull' for full property snippet
Hint: use "Id" or "id" for a user id so that entity framework can see it and identify it as an 'id' property

Introducing Entity Framework

Entity Framework provides "DbContext" class as the primary class used to interact with the database. EF allows us to write "link queries". SQLite for development is cross-platform and does not require a server, it just adds a file that serves as the server during dev. EF allows querying, tracks changes, saves the db, provides concurrency and transactions, and caching, built-in conventions (default rules that automatically configure the EF schema), configurations, and migrations. EF basically will look at the code and create the appropriate database for us. This is called "code first", create code then create database through migrations.

Installing EF Package

Get extension "Nuget gallery", ctl+shift+p to open Nuget gallery, search for Microsoft.EntityFrameoworkCore and find the package that matches the runtime that is installed (get the version that has .Squlite attached). Tick the box to install with API project, click install.

Create dbContext class

New folder >> Data; new class DataContext, derive from DbContext in EF (be sure to check for error, should be missing a using statement; use the lightbulb hint, or add “using Microsoft.EntityFrameworkCore;)
Create a constructor in the class (use the shortcut to create constructor with (options))
Create a prop called DbSet<AppUser> Users; bring in the using directive
Go to the Startup.cs class and focus to ConfigureServices method; we need to create a connection string and add some services:

```
services.AddDbContext<DataContext>(options => {
        options.UseSqlite(“Connection String”);
});
services.AddControllers();
```

NOTE: will need several using statements to support the above lambda expression and service

Add the connection string to Sqlite to allow connection; connection string is ‘safe’ to be inside a file that is public, so the CS can be in appsettings.Dev.json. Add the ConnectionStrings at the top of the file. The CS is the name of the file where the db is stored.

“Private” Shortcut

go to preferences >> settings, search “private” and add ‘_’ to autogenerated private fields

To get the connection string

install a nuget tool (dotnet-ef), find the version to match the version of ef installed on machine, copy line of code from website. Install via VS Code terminal (doesn’t matter where, it’s global). Then from the API folder in terminal, use the ‘dotnet ef migrations add InitialCreate -o Data/Migrations’
(get the Microsoft.EntityFrameworkCore.Design package from nuget gallery)
Check out the “InitialCreate.cs” file now located in the Data file; should have two methods now for creating and dropping the table; notice how it added the properties from “AppUser”

In terminal, use “dotnet ef database update” to create the database

NOTE: if getting an error about calling SQLitePCL.Batteries.Init()., go to nuget and install the *Microsoft.EntityFrameworkCore.Sqlite*, NOT the *Microsoft.EntityFrameworkCore.Sqlite.Core*

New Extension: sqlite

data base (db file) will be located in the API folder. Sqlite extension will be in the Explorer pane, use this to inspect the file. NOTE: database will take on the name of the project, so plan accordingly! This cannot be changed easily

In the Sqlite explorer window right click “users” and select “New Query (insert) and then add new values to the table via SQL commands

VALUES(1, “Angelina”);

Etc…

Highlight all commands, right click, and select “Run selected query” to run the code; data is now in the table! Right click “users” and select “view table” to confirm

Adding our own API Controller

API controllers require a few things to be identified as controllers; [ApiController] at the start, followed by a [Route()] which is how the user gets to the controller from the client. Controller needs to derive from a controller base, “ControllerBase”.
“Route” will essentially be the “web address” the user needs to type to get to the controller, so “api/[controller]” signifies “www.websiteaddress.com/api/[name of controller]”

[HttpGet]

Basically options that the controller is capable of returning. You can add in parameters that are passed in from the user in {}, example [HttpGet(“{id}”)] will require the user to enter an “id” for this particular method of the controller

The method names of the controller are what is used to access each method in the web address; for instance, if the method is called “GetUsers”, in the controller at the above Route, then the user will type in “.../api/users”.

If the controller requires input, then the input is added at the end of the request, ie “...api/users/1”

NOTE: Error in displaying DB

sometimes this doesn’t work right away… unknown as to why! Things to try:

Restart VS Code

Manual save files

Restart browser

Delete database file and rebuild

Ensure .db file name is correct in appsettings.Development.json “Data source=[name].db”

HINT: if Sqlite is not opening the database, in VS Code, there may be an issue with the database itself; delete and rebuild using “dotnet ef db update” command in terminal; save and restart VS Code

Making Asynchronous Code

Code should be able to run in parallel to ensure scalability. Use the “Async” keyword, as well as “Async” versions of various commands like “ToListAsync()”

Async

Wrap methods in “Task<[Return type]>”, and tell returns to “await”
Ex:

```
Public async Task<ActionResult<blah>> GetInfo()
    Return await var.info.ToListAsync();
```

Also consider shorthand version:

```
Public async Task<ActionResult<blah>> GetInfo() => await _var.info.ToListAsync();
```

Walking Skeleton: Angular Integration

Creating the Angular App

NOTE: be careful about third party packages with the latest release of Angular; check compatibility.

To install:

Command prompt: navigate to folder for best results >> npm install -g @angular/cli@12
In this case “12” is the version we are installing; again, check for compatibility with the project
Type “ng –version” to check version number of Angular

Create a new angular project: ng new client – strict false (turning off strict is for the tutorial project); “yes” to routing, “CSS” for style sheets
There will be a new folder in the project called “client”, navigate to this folder in the terminal to use Angular

Uninstall and reinstall Angular

Use the command npm uninstall -g @angular/cli
Reinstall with npm install -g @angular/cli@XX.X

NOTE: pretty much any install using npm can use the extension “@XX.X” to specify a version to install

Running the Angular Project / Bootstrap

Within the “client” folder in the project, in terminal type “ng serve”

NOTE: Error with Digitally Signed Files

If an error appears saying that the file nodejs\ng.ps1 cannot be used bc it is not digitally signed, this means the computer is preventing the run of unsigned scripts. To fix this:
In terminal type “Get-ExecutionPolicy” to see a list of policies
Type “Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser”
The project should compile and run now

NOTE: the first time Angular is run, it may take a few minutes and appear to hang; wait it out

Check that it’s running: http://localhost:4200 will display the generic Angular page

NOTE: when making changes, if “ng serve” produces a compile error be sure you saved the most recent changes manually

Main.ts contains “platformBrowserDynamic” which bootstraps the “app.module.ts” file, which then bootstraps the Angular components and other modules (the BrowserModule is bootstrapped to allow our page to be displayed in the browser

Angular Components are the main building blocks of the application

VS Code Extensions for Angular

Angular Language Service
Angular Snippets
Bracket pair colorizer

HTTP Requests in Angular

Transfering the HTTP requests control to Angular

Add a module to imports from Angular “HttpClientModule” (import {HttpClientModule} from ‘@angular/common/http’ in the app.module.ts file

Create a constructor of private http: HttpClient {} in the app.component.ts
Add an ngOnInit function that calls another function to “get” data (getUsers())
Http belongs to “this” so “this.http.get([url of api])
The above line is an “observable” which needs to be subscribed to
… .subscribe(response => { this.users = response;} to send data back

NOTE: Deprecated Subscribe

Use the following syntax for subscribe (called ‘observer pattern’):
… .subscribe({
next: response => this.users = response,
error: error => console.log(error)
})

Adding CORS support

Cross Origin Resource Sharing

blocks all requests that do not originate from the same place
Resources must supply a header to the browser to say it’s ‘okay’
Startup.cs >> ConfigureServices method, add services.AddCors(); (anywhere)

However in Configure method, the following must go after UseRouting
app.UseCors(policy =>
policy.AllowAnyHeader().AllowAnyMethod().WithOrigins("http://localhost:4200"));

ERROR after allowing CORS

Ensure that “dotnet run” has been started as well, otherwise the API is not running and cannot field any requests

In Firefox, open developer tab, go to “Network”, select the file for the get request (“users”) and check the Headers for “200 ok” and check the “Response” tab for db results

Displaying Fetched Data

In the html file data can be interpolated through Angular using {{ }}
Likewise, Angular supplies a number of methods to fetch and display data using these braces
<li *ngFor=’let user of users’> will allow the fetching of “users” into a list item (looped through the list)
The list item will look like >{{user.id}} - {{user.userName}} - *etc*

NOTE: Angular is Case Sensitive

Check casing if items are not displaying properly (compiler will not throw an error)

Bootstrap / Font Awesome

Bootstrap

For this project we want Angular bootstrap, found here as ngx bootstrap
https://valor-software.com/ngx-bootstrap/#/documentation#getting-started
Check documentation for compatibility; typical lifecycle of bootstrap is 6 months behind angular releases

NOTE: check that the version of ngx-bootstrap you want is compatible with the version of Angular installed

> ** The specific version of ngx-bootstrap can be installed by adding "@X" at the end of the command below, where 'X' is the version number, e.g. '9'.

Helpful links:

> Issues using incompatible versions:
> https://github.com/valor-software/ngx-bootstrap/issues/6481
> Ngx-bootstrat Getting Started:
> https://valor-software.com/ngx-bootstrap/#/documentation#getting-started
> Bootstrap: https://getbootstrap.com/

Use command "ng add ngx-bootstrap" from "client" folder in VS Code terminal
Bootstrap contains all the css classes we need so that we don't need to write our own. For this project we used Angular 12, and we are preferring bootstrap 4 and ngx-bootstrap 7. Later on we will update code to use bootstrap 5.
package.json file updated dependencies
Angular.json file updated "styles" section
app.module.ts file updated "imports" to include "BrowserAnimationsModule" from bootstrap

Font Awesome

In terminal use command "npm install font-awesome"
Check package.json "dependencies" for "font-awesome"

Stop Angular and dotnet from running (Ctl+C), then restart both; check browser (refresh in necessary, more than once if necessary); font displaying data should be different, more modern

Using HTTPS in Angular

Adding a trust certificate

Follow the instructions in the text document inside the “StudentAssets” folder located within the project. Select “Machine Level” for location; follow the rest of the instructions. Done.
This certificate is good for 20 years from September 2020.

Create a new folder in “client” within project. Copy “server key” and “server” files to this folder.
In “angular.json” file go to “serve{ “ type

```
"options":{
        "sslKey": "/ssl/server.key",
        "sslCert": "./ssl/server.crt",
        "ssl": true,
        "browserTarget": "test:build"
```

Ensure the commands above are typed correctly!
NOTE: some of the above options may already exist. “Options” should be after “builder” but before “configuration”
Restart dotnet and angular, navigate to https://localhost:4200 and check that certificate if valid

NOTE: Fixing the Missing Data in Browser

Upon completing the above, you may notice the title of the page appears but the data previously displayed (contents of the database) does not. Again, this is a CORS issue; check in the “Startup.cs” file and modify the “app.useCors” policy to accept connections from “https://localhost:4200”

Extension: .Net Watch

Installing this extension in VS Code will keep dotnet running (press F5 to start, Shift+F5 to stop); updates to dotnet will update in browser automatically.

Authentication

NOTE: Where to Start?

When an idea for an application presents itself, where do you start? Start with requirements:
For this application…

- Users should be able to log in
- Users should be able to register
- Users should be able to view other users
- Users should be able to privately message other users

Storage of Passwords

Passwords are stored in the database; best way to store them is through encryption. Passwords should be hashed and salted for best practice. This ensures that even if users have the same password, they will produce different hashes.

Password salt is randomly generated and stored alongside the users password in the database.

ASP.NET Identity

Does a lot of the work for us; will be used later on

NOTE: storing pw salt in DB is not best practice for release

Update User Entity

Add in properties for both “PasswordHash” and “PasswordSalt” in the AppUser entity.

NOTE: “dotnet ef migrations add [name of migration]”

Use the above command any time a new property is added to an entity; afterwards, update the DB with “dotnet ef database update”

Create Controller for Login / Register

Create a base Api controller; AppUsers class will inherit from this controller
Create an AccountController class that will contain a method called “Register”; Register will be an [HttpPost], use HttpPost any time something is going to be added through the API endpoint.

Data can be sent up in the request via a variety of places (header, body, etc).
Consider what items will be needed to create a “user” when registering
Consider where it will come from, as well; API Controller, however, is smart enough to know where the inputs are coming from.

NOTE: “Using” keyword

“Using” ensures that a class will be disposed of after it’s done being used. Any class using the “dispose” method will implement the “IDisposable” interface, which must be provided
Question: can ANY class be used this way, provided it inherits from IDisposable?

System.security.cryptography supplies the HMACSHA512 class which contains the “computeHash” method. This method takes in a byte array; user password must be converted to the byte[] using Encoding.UTF8.GetBytes(password). HMACSHA512 also supplies the “Key” object, which is a randomly generated byte[] key that can be used for the “salt” of the password.

Adding data to the DB

See: "AccountController.cs"
Using the DataContext class (which derives from DBContext), a variable is created ("_context") to supply the database with the data to insert. Entity Framework uses this class to read the data and insert it appropriately (after migrations have been completed).

> "Users" are entities; DataContext class contains "Users"; _context is the instance of the DataContext class used to pass new users to the database
>
> "_context.Users.Add(user)" only places the "user" object in memory (Entity tracks it, but has not yet placed it in the DB.
>
> "_context.SaveChanges()" will save the data to the database

NOTE: Async

Create the method asynchronously, and use "await _context.SaveChangesAsync()" instead

NOTE: Postman Error

When testing the above method, ensure that your request is set to "POST", not "GET" or else a 405 error will be generated!

DTO's

Debugger in VSCode

Start by selecting a breakpoint in the code; open the Debugger from the side menu. From the dropdown select "DotNetCore Attach" and press the green arrow "play" button. This will open the file search… in the case of the api, we are looking for the API.exe file (the executable that Windows is running during this session).

I don't think there's a limit to what can be attached, but ensure the correct .exe is attached based on what is being tested. In the case of web API's, the API.exe will typically be the .exe to choose.

In PostMan, run the script that produced the error (in the above case, it was the "register user via body"). Use the debugger output once the breakpoint is reached to investigate the problem.

In the above error, body query's are passed as "strings" but must be instead passed as objects.

Data Transfer Objects

DTO's can be used to hide objects, flatten objects, and avoiding circular references (where entities reference other entities that reference the original entity)

We'll start by making a folder for the DTOs. DTO's in this folder will be .cs classes and should be named based on how they are being used; in this case "RegisterDTO"

NOTE: DTO's can be mapped easily, so camel case versus other cases is not an issue

Add in the properties that the DTO will be transferring; username and password, as public strings. Change the "Register" method in AccountController.cs to take in one DTO. Input variables will now be "[DTOvar].username" and "[DTOvar.password".

NOTE: Username Checking

This is a good place for a "helper method" that will check for username uniqueness. Use "_context.Users.AnyAsync()" to check if the username exists. Likewise, pass the expression (x => x.UserName == username.ToLower())" into the "AnyAsync( )" method.

Using an ActionResult allows us to return different Http status codes (like "BadRequest")

Adding Validation

Users should not be able to enter a null string for anything. Validation can be added anywhere, but the best place is within the DTO bc those are the properties we are receiving in the body of the request.

Within the DTO simply add a "data annotation" of "[Required]" for all entries

NOTE: There are a variety of data annotations

Like for [email] or [phone number]

Adding Login

Logic for the login should use another DTO ("loginDto" with the same params as the previous DTO).
When user attempts to login we need to validate whether they are in the database or not. For this we want to use "SingleOrDefaultAsync<>" in order to find within the database the user that is attempting to login. This will throw an exception if there is more than one element in the "sequence".

Un-hashing the Password

For this we essentially work backwards using HMACSHA512(). The original call to this method passes no arguments and generates the secret key; the overload passes in one argument, the secret key that is to be used to decode the given password.

Using var hmac = new HMACSHA512([salt]);

the salt is stored in the “user” var created at the very beginning of the method

user.PasswordSalt

Now we work backwards…

var computedHash =
hmac.ComputeHash(Encoding.UTF8.GetBytes(loginDto.Password));

If the password in the DTO is the same as what’s in the database, then the user is successfully logged in; use a “for loop” to check this, byte by byte in the byte arrays.

NOTE: we are not comparing the actual passwords!

We are

- Storing the user-entered information in a DTO
- Checking that the username in the DTO exists within the Database (“SingleOrDefaultAsync”)
- If it does, we take the hash of the entered password, using the salt stored at that user location in the DB
- We compare it to the hashed password stored in the DB for that user
- If they match, success, user is authenticated

NOTE: Clear the Database

Stop the app and run “dotnet ef database drop”, say “y”
Rebuild with “dotnet ef database update”

Authentication via JSON WebTokens

Web Tokens allow a session with an API to stay open without having to login every time a user wants to make a request to the API

WebToken Structure

3 parts

- Header - contains algorithm and type of token; algorithm is used to encrypt the signature in the 3rd part of the token
- Payload - information about our “claims” (who we are); user claims to be something. Also contains 3 time stamps: not before, expires, and issued
- Signature - encrypted by server using key that never leaves the server; this is the only part that’s encrypted

Token cannot be modified in anyway, or else it would change the overall structure of the Token and thus the API will not accept it

Token's are stored locally by the user and sent to the API every time a request is made (any time the user wishes to access information that is protected).

Adding a Token Service

To implement Tokens we'll use an Interface ("single responsibility principle" is used to create a service that's only responsibility will be to create a token for the user. The account controller will handle issuing the tokens).

The interface will implement a "CreateToken" method

Create a separate folder in the API for "Interfaces" as well as another for "Services"; the TokenService will be created here.

Since a JSON token is just a string, the TokenService method "CreateToken" will take in a "User" and return a "string" for that user.

NOTE: Add the Service to "Startup.cs"

We want the TokenService to start with the API, so we need to add it in "ConfigureServices" dependency injector. Best option for this is AddScoped for an Http Request because the service will be scoped to the lifetime of the Http request. A new instance is created when the request is made, and when the request is finished the service is disposed of. Consider using this all the time for requests.

An interface is NOT strictly necessary, but it is good for 1. Testing (interfaces are easy to mock during testing), and 2. Best practice

This would work just fine with just the tokenService with no interface.

Adding the Create Token Logic

For this we'll need a NuGet extension, so go to NuGet Gallery and get
    System.Identity.Model.Tokens.JWT by Microsoft; tick the box to install to the API
This will be used to make the "key" in the TokenServices class
    private readonly SymmetricSecurityKey _key;
NOTE: Token Key is created in the next lesson

Creating the Token:

    Start by creating a list of "claims" (list so that more items can be added later if necessary
    Create credentials ("creds") by encrypting the Token Key
    Create the tokenDescriptor with a lambda that inputs:
        Subject (claims)

Expires (when the token expires)
Signing Credentials (creds)
Create a tokenHandler
Create the token via the tokenHandler.CreateToken(tokenDescriptor)
Return the token via tokenHandler.WriteToken(token)

User DTO and Returning the Token

We want to now pass back the token instead of the user and their password (when logging in) For this we'll create a new DTO ("UserDto") with the properties "Username" and "Token" (both strings). Then in our AccountController "register" and "login" methods we'll change the return type to "UserDto".

We also need to add the ITokenService to the AccountController properties
private readonly ITokenService _tokenService;
Don't forget to initialize in the constructor of AccountController!

In order to return a UserDto object we will create the new object at the return statement; it will look like:

```
return new UserDto {
    Username = user.UserName;
    Token = _tokenService.CreateToken(user)
};
```

This runs the methods as before, but now at the end a UserDto is created with the username front the AppUser object, and a token is created via the ITokenServices interface we brought in earlier.

Create the Token Key

The token key should be a long string of unguessable text, stored in the appsettings.Development.json file (for now)

NOTE: token key complexity

The token key should be a long string of random characters, not a simple sentence (even for development). Additionally, keys that lack complexity will not be accepted by the app in some circumstances. Use this website
https://www.allkeysgenerator.com/Random/Security-Encryption-Key-Generator.aspx
To generate a strong token key. Use 512-bit for best security and compatibility.

NOW try and login a registered user, see that the token is returned instead of the password! Verify the token at https://jwt.io/ (copy and paste)

Adding Authentication Middleware

Authenticate user requests with this stage. In the controllers that access information (“HttpGet”) add the [Authorize] tag before the method.

To make use of this we need the nuGet package
Microsoft.AspNetCore.Authentication.JwtBearer
Once we’ve done that we need to add the token validation service in “services.cs”:

```
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme).AddJwtBearer(options
=> {
                options.TokenValidationParameters = new TokenValidationParameters{
                ValidateIssuerSigningKey = true,
                IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(_config["TokenKey"])),
                ValidateIssuer = false,
                ValidateAudience = false,
                };
        });
```

In this case we are not validating the issuer (The API) nor the Audience (Angular), since we are developing right now.

In the “Configure” method we need to add “app.UseAuthentication()” BEFORE UseAuthorization and AFTER UseCors!

Requesting user information now will result in a 401 unauthorized error

Postman Check

- Login as a user that exists
- Copy the token that is returned
- Paste the token in the Header of the “get user” request from Postman
    - Type Authorization in “key”
    - Type “Bearer ” in “Value”
    - Paste the token after “Bearer “
- Try again

Adding Extension Methods for Housekeeping

Extension methods allow us to create methods to existing types without creating a new derived type or modifying the original type.

Extension classes and methods must be Static
Extension methods are basically methods in a class that contain a bunch of stuff you expect to call on over and over, so instead of retyping code over and over you can make one call to the extension method and the method will do whatever it does as many times as it's called

In this way, we place all the startup services that we write into one extension method, and call that method with the "_config" property as a parameter.

We can do this with any of the startup services, or really anything we expect to be using multiple times

NOTE: be mindful of what you are returning from the extension method; in this case we want to return our "services" which is an "IServiceCollection".

NOTE: must always specify "this" before the type that is being extended, so in this case "this.IServiceCollection services" since in this case "services" is being extended, followed by an "IConfiguration config" which is being passed in as a parameter):

```
public static IServiceCollection AddIdentityServices(this IServiceCollection services,
IConfiguration config)
```

Section 5: Client Login and Register

Creating a Nav Bar

Angular Shortcut

“ng g -h” shows the Angular “generate” command and all the things that can be generated. For this part we want to generate a “component”, so “ng g c nav” will suffice (‘nav’ is simply what we are calling the component, we will still need to build it).

NOTE: Placement

Navigate to the “/src/app” folder before generating a component so that the component is added to this folder

The files will be generated in their own folder (several, including a .css and a .html), and the component “NavComponent” will be added to the app.module.ts file

Creating the Bar

Start at “Bootstrap” (getbootstrap.com) and go to “Examples”. Search for a component that you want to use (in this case we found “Carousel” in the “Custom Components” section). In Firefox press F12 or right-click to “inspect”, find the section you want to copy / use (nav, in this case), select, right-click, “copy > outer HTML”, and paste the code into the [name].component.html file in the /src/app/[name] folder.

For this site the file is”nav.component.html”.

The Main Page

The main page of the site exists on the “app.component.html” page of the project. In this project we want to add the component we just created so we first open the “nav.component.ts” file to see what the component is called in order to add it.

In this case it is called “app-nav” (this is found after “selector:”). Within the app.component file, add: <app-nav></app-nav> to add the component to the site!

NOTE: Nav bar looking squished

The nav bar may originally appear squished; this is due to several tags found within the copied html file. There are ways to fix this with different tags:

Alongside “navbar-nav” add “mr-auto”.
In form class replace with “form-inline mt-2 mt-md-0”

NOTE: responsiveness

This site as it is is not built to be responsive; that will have to be built in later

Angular Template Forms

Angular provides a “forms module” that we need to import in the app.module.ts FormsModule (uses “import {FormModule} from ‘@angular/forms’;

In the nav.component.ts we will add a “login()” method that returns the user to the console (this is done via a “model” type of “any” declared before the method)

In the nav.component.html we need to update the “form” to be a “#loginForm=”ngForm” which means it is now an Angular form; we also need to add (ngSubmit)=”login()” where “login()” is the name of the method we are using to take in user login information (found in the nav.components file). This method can be named anything, and can be changed to another method if we want to, for whatever reason.

Each input needs a

name=”username”
[(ngModel)]=”model.username”

The second line above allows two-way binding, meaning data can be (received) and [sent back]. The “model” is the type “any” that we are expecting in the form, and the model will have two properties: username and password.

Shortcut ngModel

Type “a-ngmodel” to have the line automatically fill with [(ngModel)]=””

Test this out by typing in a username and password into the box in the browser and check that both are returned to the console

Services

In the /src/app/ folder create a new folder for services, “_services” (‘_’ just to keep the folder at the top). Inside that folder use Angular CLI to generate (g) a new service (s) called “account”

ng g s account –skip-tests

Services are “singleton”, as in there is only one instance made during the duration the user is in the application. We’ll use the “account” service to make requests to the API. We need to set a baseUrl = ‘https://localhost:5001/api/’;

We also need to inject the Http client into the account service by adding “private http: HttpClient” into the constructor of the account service

Then add the login(model:any) method to the service
The method should return this.http.post(this.baseUrl + ‘account/login’, model)
This is a “post” request, so the ‘model’ is what we are sending up to the server. This is how the service will contact the api with the information from the browser that the user inputs. The model will be sent to the service which will forward it’s information to the api for login

Injecting services

In the nav.component constructor add “private accountService: AccountService
The login method now, instead of printing the console, will instead run this.accountService.login() and pass in “this.model” as a parameter. This calls the account.services.ts “login” method and returns an observable.

Testing

In order to test this we add a few extra items

- First we must subscribe to the observable that is being returned
  .subscribe(response)
- We then want the response to be logged to the console
  .subscribe(response => {console.log(response);}
- We can also add a temporary property of type Boolean called “logged in” to be set to “true” when a user logs in, but this isn’t strictly necessary
- We also want to log any error to console in the event there is one
  …response);}, error => console.log(error);

To test, try and log in a user! The console will show the user and the token that is returned. It should also show errors for bad usernames and bad passwords.

Conditionals

Conditionals allow us to show or hide items on the page based on certain conditions (like whether a user is logged in or not). For this example, we will use the “loggedIn” property from “nav.component” file.

In the nav html file…

- To show a component, add “ *ngIf="loggedIn" ” to the div
- To hide a component, add “ *ngIf="!loggedIn" “ to the div

Angular Bootstrap Components

Adding a Dropdown

This is done without using jquery
Go to https://valor-software.com/ngx-bootstrap/#/components/dropdowns?tab=overview
Make sure the left side menu is visible, go to “components > dropdowns”
In the API tab, find “installation” and review the “imports” that are needed for what you want to do
You can look at the “examples” tab to see what is possible

Paste the appropriate “import” in the imports section of “app.module.ts”
Paste the appropriate module in the “imports :[]” section for modules
In this case, “BsDropdownModule.forRoot()”
The “forRoot()” tells the module to load all services / components that need to initialized with the “root” module (@NgModule)
Several things need to be added to the component.html file as well, in the “dropdown” class
“Dropdown” class must be given the label ‘dropdown’
The button that will open the dropdown needs a tag “dropdownToggle” (before the text that will be the link for the dropdown)
The “dropdown-menu” class needs a ‘*dropdownMenu’ added

NOTE: automatic logout

If the dropdown-toggle class has an ‘href=””’ before the class tag, it will automatically log out the user once the toggle is clicked:

```
<a href="" class="dropdown-toggle text-light" dropdownToggle>Welcome</a>
```

Versus:

```
<a class="dropdown-toggle text-light" dropdownToggle>Welcome</a>
```

Adding Styling

Any link class that should be a pointer, create a css styling that says “cursor: pointer”
Bootstrap styling: “mt-[number 1 - 5]” as a margin to the item
Add this as a class to the html code

Observables

What are they? Relatively new, introduced in Angular v2, are lazy collections of multiple values over time. Can be used to stream data, but we’ll be using them for Http req’s and when we want components to observe values so that when values change, they respond accordingly.
If no subscribers to the observables, then the observable doesn’t do anything
Observables can be canceled bc they deal with streams of data
They emit multiple values over time

RxJS: reactive extensions for JavaScript

subscribe() function parts:

What to do next
What to do if there's an error
What to do when complete
.subscribe(members => { this.members = members} , error => {console.log(error);}, () => {console.log('completed')})

It is said that if you subscribe to something, then you should unsubscribe from it
Angular handles this when it's an http request, though

Persisting the Log-in

Here we'll use the "pipe" operator so we can do something with the response we get back from the API. The basic idea is to have a user log in, and then have the log-in persist while the browser is still open.

The browser has a "local storage" (accessed with "localStorage.setItem()")
Using JSON.stringify, we can store the string of the user information (in this case, the username and the token created). While this string exists in storage, the user will remain "logged in"
In the login() method in the account.service file, we put the user into an observable (here, called "currentUserSource" which is a "ReplaySubject<User>(1)"; the '1' signifies that only one user will be in there, and this observable will be set with the current user that logs in).
ReplaySubject will send back the last object OR however many objects that are specified

To create the observable:

currentUser$ = this.currentUserSource.asObservable();

In the app.component we need to have a method that checks to see if there is a string in localStorage… if so, JSON.parse the string to a 'user' object, then call setCurrentUser(user) from "accountService".

Using the Async Pipe

The above method has potential for memory leaks due to subscribing to an observable but never unsubscribing.
In the nav.component we should add a currentUser Observable<user>
Then we can remove the boolean "logged in" variable, as well as wherever it is referenced within the component. In the ngOnInit method we now call the accountService.currentUser$ observable and place its value into this.currentUser$.

Using the Pipe

In the nav component we want to access the accountService.currentUser$ observable to modify the page in the event the user logs off or whatever, so we use the async pipe with the following syntax:

*ngIf="accountService.currentUser$ | async"

Make sure the accountService is “public” in its constructor within the nav.component file

Adding a Homepage

Within the “app” folder use the Angular CLI to create a “home” folder with components:

ng g c home –skip-tests

g = generate, c = component

Build the home screen in the html file that was created

Use Bootstrap to build the page

https://getbootstrap.com/docs/3.4/

Click to hide feature

In the “component.ts” file for the component being edited, create a boolean that starts as “false”. Create a method to set the boolean to the opposite of whatever it currently is. Then in the “component.html” use *ngIf=”boolean” and *ngIf=”!boolean” in various components to turn them on or off on the page. Create a <button (click)=”booleanToggle” > to allow a button to change the value of the boolean and thus display or hide certain items.

Adding a Register Form

Add another component from the CLI called “register”

We’ll want a method to handle “registration” and another for “canceling” so we can call them from the html form.

Placing the component on the page

Just add the component to the part of the page that you want it to display in

<app-register></app-register> within a div

Parent to child communication and Reverse

Currently, the home component is the parent of the register component, and the home component is a child of the app component; we want to pass information down from home to register.

Within the following child component call…

<app-register></app-register> within a div

… the variables needed to pass information down can be added…

<app-register [variableFromOtherComponent] = “variable”></app-register> within a div

In order to bring in data from another source, use “@Input() variableName: type;” in the child component. Then use the variable in the template (html).

Interpolation

When needing to display interpolated data, use the format {{variable.property}} within the <option> list…

```
<div class="form-group">
    <label>Who is your favorite user?</label>
    <select class="form-control">
        <option *ngFor="let user of usersFromHomeComponent"
        [value]="user.userName">
            {{user.userName}}
        </option>
    </select>
</div>
```

The above creates a list of users where the data is pulled from the home.component.

Going from child to parent

We need a method in the parent that takes in the same type of variable that is being emitted from the child. Then set the appropriate variable to the value that is passed in.
In the case of our app, we want the cancel button to tell the home component that we no longer wish to display the registration form.

Passing data up uses the “@Output()” call along with an “EventEmitter()” from Angluar.core.
When we do something (like click a button) we want to “emit” a value of some sort.
Syntax: @Output() registerFormCancel = new EventEmitter();

```
this.doSomething.emit(value);
```

Place the above in a method. Then we’ll call it

Note: data transmit

Receiving data goes in brackets, sending data goes in parenthesis (within html template)

Registering Users via the Form

Here we will essentially do the same as we did for the log in method, as we consider a user registering to be “logged in”. Therefore we will make use of the .pipe(map(user => {} operation.

Section 6: Routing in Angular

Angular provides routing in the application that allows for page updates, instead of using conditionals. A routing solution lets a user navigate between different components in a Single Page Application

Routes are defined in the “app-routing.module.ts” in the array “Routes”

Constructing a route

{path: ‘’, component: HomeComponent}, …

path= the path for the route
Ex: ‘ ‘ for ‘home’, ‘members’ for the ‘members component’, or ‘members/:id’ for the ‘members component’ but with a parameter of ‘id’

Component= the component name

NOTE: wildcard in routing

path: ‘**’ acts as a wildcard for when nonsense is passed in as a URL for the app
Wildcard routes need a ‘pathMatch’ parameter at the end to specify how the wildcard is handled.
“full” means it will match the full URL before getting to the wildcard section of the URL
“default” means it will match the first URL that matches the beginning of the URL

Wildcard structure:
{path: ‘**’, component: [whateverComponent], pathMatch: ‘full’}

Routing

To implement routing in the app.component.html, add <router-outlet></router-outlet> in the body where the components will be expected to appear.

To link to routes, remove “hrefs” for links and instead, add “routerLink”
routerLink=”/messages”

Active Links

Add

routerLinkActive=”active”

to links to show them as active when clicked (this is a Bootstrap style, so you’ll need Bootstrap installed for this to work)

Routing Users

When a user does something that an observable is watching, they can be automatically routed to a specific component. In the component that will be handling this service, you must first inject the “Router” in the constructor. Then, the router can be used in the observable:

this.router.navigateByUrl(‘/urltonavigateto’)

NOTE: this can also be done in a regular method w/o an observable

Example usage: a user logs in, and the router.navigateByUrl changes the component of the SPA from the ‘home’ page to ‘userProfile’.

Toasts

NOTE: errors

Toasts can be a fun and interactive way to notify users of something. For instance, if an error occurs…

error: error => console.log(error)

… the error object will be logged to the console in the above example. A toast notification can show this error on the page, though, and is much more user-friendly. The API should send back info about the error from its controller. For example, in a user log-in attempt, if the password is invalid, the API will return “Unauthorized(“Invalid Password”)”, in which case the error is “Invalid Password”. This error appears in the console as “error”.

Check out https://github.com/scttcper/ngx-toastr for a great toast extension. View the README for documentation on how to implement it.

NOTE: versions

Be sure to read the documentation to determine the correct version to download. Use the ‘@XX’ param at the end of the command to force a specific version

npm install ngx-toastr@15

NOTE: toast properties

Things like position on screen and others go in the “ToastrModule.forRoot()” import in the app.module.ts file.

Route Guards

Adding a route guard

Angular CLI:

ng g g _guards/auth –skip-tests

Select “canActivate”

Creates a folder called _guards with the “auth.guards.ts” file inside

File is injectable but likely won’t need to be injected into anything. It will stay alive for the life of the app. Multiple guards are possible (“canActivate” function) where users will have to match all in order to move forward.

For our app, we will just return an Observable<boolean> that will be connected to the AccountService where a ‘user’ is logged in or logged out. To do this we must inject the AccountService into the auth.guard.ts file.

The Observable<boolean> that is returned is obtained from the accountService instance variable for ‘users’ (currentUser$, which is itself an observable from the accountService, tracking the user that is logged in or ‘null’ if none is logged in).

In auth.guard canActivate() we check the currentUser$ observable using the ‘.pipe’ method, where we ‘map(user => {}’. If ‘user’ has a value, ie a user is logged in, the return “true”, else return “false”.

NOTE: this is a good place to use a ‘toast’ for the error

If a user is *not* logged in but still tries to access parts of the site hidden behind the auth.guard, then the toast will notify them.

Implementing

In the app-routing.module, be sure to add ‘canActivate: [AuthGuard]’ to each route that needs to be protected by auth.guard

```
{path: 'members', component: MemberListComponent, canActivate: [AuthGuard]},
```

NOTE: this is not security!

All the auth.guard does is ‘hide’ things from the user. Real security needs to take place at the API level.

Dummy Route

Used for creating single guards for multiple routes in the application. This is done in the app-routing.module in “Routes” array.

Create a “path:” with parameters:

runGuardsAndResolvers: ‘always’
canActivate: [AuthGuard]
children: [… paste here all routes to be guarded…]

<ng container>

HTML container that can take conditionals from angular and apply them to whatever is inside the container. For example, placing a list of buttons inside the container so that if the user is logged in they are visible, but if not logged in they are not visible.

Adding a theme

Go to https://bootswatch.com/ to choose a super simple theme to use.

npm install bootswatch

In angular.json > styles, add the css file

"node_modules/bootswatch/dist/**[theme-name]**/bootstrap.css"

…where “theme name” is the name of the theme you wish to use. Adjust colors and styles as necessary in the app templates (html)

Accessing items

When a user logs in we would like to show their name in the “Welcome ‘user’” callout. To do this, we can gain access to the “currentUser$” observable in the accountService object.

First by wrapping “accountService.currentUser$ | async” in parenthesis:

*ngIf="(accountService.currentUser$ | async)

then adding on “as user”

*ngIf="(accountService.currentUser$ | async) as user"

we now have access to the “user” property and all parts contained within, such as “username”

Using angular {{user.username}} we can show the user's name anywhere in the code.
To specify a style to show the name, such as "title case", we use the '|' within the {{}}

```
Welcome {{user.username | titlecase}}
```

Shared Module

Tidying up the imports section of the app.module.ts file. Create a "module" with

```
ng g m _modules/shared
```

NOTE: "flat"

Using the command above (ng g m…) will produce a folder within a folder (/module/shared/shared.module.ts). If you don't want that and would rather the "shared.module.ts" file be within the "module" folder with no extra nests, you can use the "--flat" command tag:

```
ng g m _modules/shared –flat
```

In this new file, we can import some of the other imports from the app.module file, like third-party extensions. Add desired imports to the "imports" array, under "CommonModule".
In the app.module original, add the "SharedModule" to the imports array so that it gets imported
Add array for "exports" in shared.module.ts; here, export that modules that were imported above. This tells the module to send those out to the main module when called.

Section 7: Error Handling

The application is currently running in "Development" mode. This can be seen in the launchSettings.json file (in the API section). Changing the environment to "Production" will change the build to a production level app, meaning errors will be minimal when executed through Postman or otherwise.

Terminal will still show the errors in full.

Handling Errors

Old way was to create a try-catch block foreach method that could throw an exception, but this is tedious due to the fact that every method has the potential to throw an exception. Instead, we want to handle all errors at the highest level of the program to catch all potential issues.

Exception Handling Middleware

The “program.cs” file does not list it, but there is already middleware running automatically to handle exceptions. We see this in our responses when testing errors.

Creating our own middleware

Start with creating a class in the API for exceptions (“ApiException”) that will contain three items:

- statusCode
- Message
- Details

These will get passed to the the constructor when an exception is encountered, and the ApiException object will be used to handle the exception by a custom Middleware class

We then want to create a class that will handle the exception, called ExceptionMiddleware. This is essentially a controller (but don’t put it in the controller folder) that will be called when an exception is encountered. This controller will use the ApiException object to output information to wherever it needs to go, and will do different things based on the environment we’re developing in.

NOTE: InvokeAsync

All middleware is expected to have a method called “InvokeAsync” this is important!

The ExceptionMiddleware will make use of the Ilogger class, the REquestDelegate class, and the IHostEnvironment class.

- ILogger allows us to log the error
- IHostEvironment allows us to see what environment the app is running in (development, or otherwise
- RequestDelegate is used in the InvokeAsync method to try and go to the next command in the chain (it will fail, since if we’re here we’ve encountered an exception and can’t go anywhere but to “catch”

The InvokeAsync method uses a try-catch block, with the catch doing the heavy lifting for the exception.

- Log the error
- Generate a response from HttpContext
- Generate a status code
- Determine the environment we’re running in
- Create an ApiException object that passes the:
    - Status code
    - Message
    - Details from the StackTrace

Serialize the ApiException to json
Write the response asynchronously

Example:

```
public async Task InvokeAsync(HttpContext context){
            try
            {
                await _next(context);
            }
            catch (Exception ex)
            {
                _logger.LogError(ex, ex.Message);
                context.Response.ContentType = "application/json";
                context.Response.StatusCode =
(int)HttpStatusCode.InternalServerError;

                var response = _env.IsDevelopment()
                    ? new ApiException(context.Response.StatusCode,
ex.Message, ex.StackTrace?.ToString())
                    : new ApiException(context.Response.StatusCode,
ex.Message, "Internal Server Error");

                    var options = new
JsonSerializerOptions{PropertyNamingPolicy = JsonNamingPolicy.CamelCase};

                    var json = JsonSerializer.Serialize(response,
options);

                    await context.Response.WriteAsync(json);
            }
        }
```

Using the Middleware

In order to use this custom middleware, we must override the default middleware in the program.cs file. To do this we add the line

app.UseMiddleWare<ExceptionMiddleware>();

To the program.cs file after the app = builder.Build() line

NOTE: Testing Errors

To test errors and the above mentioned middleware, we can create a controller specifically to generate errors on purpose. We can call this “buggyController” and fill it with methods that will generate errors like null-reference exceptions and not found errors.

Testing Errors in the Client

Just like with the API we can add routes to test the various errors. For this, we can generate a new component that is specifically for errors.

The component “test-error” will have methods that call the “buggyController” in the API via observables
Example:

```
get404Error() {
    this.http.get(this.baseUrl + 'buggy/not-found').subscribe({
      next: response => console.log(response),
      error: error => console.log(error)
    })
  }
```

Be sure to inject HttpClient class as “http”, and se the “baseUrl” as the base URL for the application.

The html template will just be a page of buttons that each call the various error methods from the component.

```
<button class="btn btn-secondary me-3" (click)="get404Error()">Test 404 Error</button>
```

This component should be added to the routing module so that the route is accessible through other components

```
{path: 'error', component: TestErrorComponent}
```

Add to the navbar:

```
<a class="nav-link" routerLink="/error" routerLinkActive="active">Errors</a>
```

Adding an Error Interceptor

Angular offers a way to generate an “interceptor” via “ng g interceptor”. Apps often use an interceptor to set default headers on outgoing requests.

The Angular Interceptor helps us to modify the HTTP Request by intercepting it before the Request is sent to the back end. It can also modify the incoming Response from the back end. The Interceptor globally catches every outgoing and incoming request at a single place. We can use it to add custom headers to the outgoing request, log the incoming response, etc.

NOTE: tutorial

https://www.tektutorialshub.com/angular/angular-httpclient-http-interceptor/

We'll create the interceptor in its own folder and use it to intercept errors with the "intercept" method. This method takes a request (as an HttpRequest) and a next (as an HttpHandler). This in turn is set as an Observable, and the method uses the pipe operator to catchError:

```
 intercept(request: HttpRequest<unknown>, next: HttpHandler):
Observable<HttpEvent<unknown>> {
    return next.handle(request).pipe(
      catchError((error: HttpErrorResponse) => {
```

At the end of this method, after performing whatever necessary checks on the error, we need to

```
  throw error;
      }
```

Using the interceptor

To ensure we use the interceptor, we must add it to the app.module file in the "providers" array as {provide: HTTP_INTERCEPTORS, useClass: ErrorInterceptor, multi: true}. "Multi" allows us to continue to use other error handling middleware in addition to this custom interceptor.

The Angular interceptor can handle pretty much any error that the API sends back to us.

Section 8 - Extending the API

DateTime extension

This simple extension method can calculate a rudimentary age of a user

NOTE: this is very simple

This method is not a perfect method, and there are more accurate albeit more complicated methods out there. This is just meant for 99% of cases.

```
public static int CalculateAge(this DateOnly dob)
{
        Var today =DateOnly.FromDateTime(DateTime.UtcNow);
        Var age = today.Year - dob.Year;
        If (dob > today.AddYears(-age)) age–;
        Return age;
}
```

In the event “DateOnly” doesn’t work, use “DateTime instead:

From the entity calling the method add the following:

```
Public int GetAge()
{
        Return DateOfBirth.CalculateAge();
}
```

Where “DateOfBirth” is a public property of DateOnly;

Entity Framework Relationships

In order to connect entities to one another they need to have a foreign key that is generated by ef. We do this by adding the desired child entity to the parent entity.

NOTE: thinking about relationships

It helps to think about the relationship between entities in reverse. For example: if users of the app can upload photos, it may make sense to have the “user” entity contain the “photo” entity, but actually we would reverse this. The “photo” entity should contain the “user” entity, since one user can have many photos, but each photo can only be attributed to one user.

Adding the entity

Considering the above example of “user” and “photo”, the “photo” entity would contain the following:

```
public int AppUserId { get; set; }
public AppUser AppUser { get; set; }
```

This will link a photo to a specific user.

To help us get the photos for a user in our app, we can add

```
public List<Photo> Photos { get; set; } = new();
```

But this is not mandatory. If you choose to leave this out, it may be helpful to add the db for the entity (in this case, “photo”) to the DBContext as a DBSet.

NOTE: Updating the DB

When updating the db, ensure that the migration builder has the correct configuration for the entities. For instance, if you get an error about inserting a null value into a property that cannot take a null, change the migration to accept nulls. This should only be temporary, since ultimately we don’t want there to be null values in certain things.

Generating Seed Data

We can quickly generate seed data for our app by using the website JSON Generator.
Once we have data generated, we want to place it in a file within our application so that we can automatically seed the database when we start the application. Create a file called “UserSeedData.json” in the “Data” folder, and copy all generated data into the file.

Seed Method

In the Data folder we’ll make a new file called “Seed.cs” that will contain the method needed to properly seed the data. This file will need access to the DataContext but won’t require a constructor. Instead the static method will take in the DataContext when it is instantiated immediately, as it will only be called once anyways. Check if there are already users in the database first; if there are, return.

```
public static async Task SeedUsers(DataContext context){
    if (await context.Users.AnyAsync()) return;
```

Otherwise, we’ll open the file “UserSeedData.json”, set the JSON options to ignore case, and then user the JsonSerializer to Deserialize the data

```
var userData = await File.ReadAllTextAsync("Data/UserSeedData.json");
var options = new JsonSerializerOptions{PropertyNameCaseInsensitive = true};
var users = JsonSerializer.Deserialize<List<AppUser>>(userData);
```

NOTE: doesn't always work

An error was encountered when attempting to update the database via ef database-update in which certain data could not be inserted

Seeding the Data

To seed the data from above, first we should drop the database with the "drop" command.

In the "program.cs" file, after "app.MapControllers();", we can set the following up:

```
using var scope = app.Services.CreateScope();
var services = scope.ServiceProvider;
```

Which will give us a ServiceProvider that allows us to access the DataContext class within the program.cs file.

Then we can try to seed the database:

```
try
{
    var context = services.GetRequiredService<DataContext>();
    await context.Database.MigrateAsync();
    await Seed.SeedUsers(context);
}
```

We'll use the MigrateAsync to build the database via all the previous migrations, and then we'll call the "Seed" class's "SeedUsers" method while passing in the context we get from the ServiceProvider.

In the event of an exception, we can catch:

```
catch (Exception ex)
{
    var logger = services.GetService<ILogger<Program>>();
    logger.LogError(ex, "An error occurred during migration");
}
```

This will inform us of an issue or exception during migration, and show us what that issue is.

Repository Pattern

This development pattern essentially creates an abstraction within an abstraction. Currently we connect directly to the DbContext class to retrieve items from the database. Adding a repository class that we connect to the DbContext class through abstracts the DbContext class from the rest of our code, which ultimately makes it easier to test when we get to that phase.

Step 1: Interface

We'll construct an interface to represent the repository. This will have methods like "GetUsersAsync" and "GetUserByIdAsync(int id)", among others

Step2: Class

Construct the class that inherits from the interface. This class will inject the DbContext class and access the database through it. We can "eager load" items from multiple entities that are connected through relationships here.

```
public async Task<AppUser> GetUserByUsernameAsync(string username)
        {
                return await _context.Users
                        .Include(p => p.Photos)
                        .SingleOrDefaultAsync(x => x.UserName == username);
        }
```

Step 3: Add as Service

In the services extension class, add

```
services.AddScoped<IUserRepository, UserRepository>();
```

Similar to how we added the ITokenService earlier

NOTE: async

All methods here should be asynchronous

Step 4: Update the Controller

The controller now needs to be updated to use the repository. Inject the repository into the controller instead of the DbContext class. The methods that call for all users or single users will now just call the corresponding method in the repository that is injected.

DTO's

At this point we may find that returning certain types of lists return an error when the entities have a relationship with one another. For example, if a "user" can have "photos" and the photos entity contains a "user" entity to link it, then the request for users will result in a cyclic loop as each user contains a photo and each photo contains a user etc etc. To counter this, we will use a DTO for both entities.

In the “user” DTO we can leave out properties that we don’t want to return to the browser (like password hashes or other private info).

In the “photo” DTO we can leave out the “user” property.

AutoMapper

To map the entity to the DTO, we can use AutoMapper to automatically map each property to the corresponding DTO property.

To install, search in NuGet for “AutoMapper”, select the one with Microsoft in the name. Install into the project.

AutoMapper Profile

In a separate folder we’ll create a file “AutoMapperProfiles”, and in that class we’ll inherit from “Profile”, a part of “using AutoMapper”. We’ll have a public AutoMapperProfiles() constructor.

In that constructor we can use the “CreateMap” function where we list what we’re mapping from first, followed by what we’re mapping to.

```
CreateMap<AppUser, AppUserDTO>();
```

Configure the Controller

Back in the controller we inject an IMapper class of “mapper”, which is provided by “using AutoMapper”.

Now when we request a list of “users” we can map the users to the DTO, removing the cyclic error we had before:

```
public async Task<ActionResult<IEnumerable<MemberDTO>>> GetUsers()
{
        var users = await _userRepository.GetUsersAsync();
        var usersToReturn = _mapper.Map<IEnumerable<MemberDTO>>(users);
        return Ok(usersToReturn);
}
```

First we request the list of users and the photos associated with each (eager loading). Then we map the users to the “MemberDTO” (users DTO). Finally, we return the “Ok” to satisfy the “ActionResult” return type.

NOTE: AutoMapper Configuration

AutoMapper can be configured to map specific properties to other properties that it may not recognize. For example, if the AppUserDTO contains an item called “Url” for the url of the main photo of the user, but this Url is originally found in the “photo” entity, we can return to the AutoMapperProfiles.cs file and configure it thusly:

```
CreateMap<AppUser, MemberDTO>()
        .ForMember(dest => dest.PhotoUrl, opt => opt.MapFrom(src =>
        src.Photos.FirstOrDefault(x => x.IsMain).Url));
```

Here we use the “ForMember” function and specify a destination to be mapped to (PhotoUrl) found in the AppUserDTO. Then we specify the options with “MapFrom”. We specify the source, which is the “Photos” list found in the AppUserDTO (a list of PhotoDTO objects) and specifically look through the list to find the first PhotoDTO object that has the “IsMain” property as “true”, and select the URL from this PhotoDTO.

This could also be done with other types of values, just use the “==” operator instead:

```
…src.Photos.FirstOrDefault(x => x.Name == “photoName”).Url));
```

AutoMapper Queryable Extensions

When a database query is sent to the database, currently it sends back data that we do not wish to have in our response (password hashes, etc) because we are returning an AppUser from the db. To fix this, we can make user of queryable extensions from AutoMapper. These include the “Project” method, which projects our AppUser object onto a DTO object automatically.

As an example, we currently have:

```
public async Task<AppUser> GetUserByUsernameAsync(string username)
    {
        return await _context.Users
        .Include(p => p.Photos)
        .SingleOrDefaultAsync(x => x.UserName == username);
    }
```

This eager loads the photos and returns the AppUser object, where it is then mapped to the MemberDTO in the UserController. We can adjust this by using the queryable extensions instead:

```
public async Task<MemberDTO> GetMemberAsync(string username)
    {
        return await _context.Users
        .Where(x => x.UserName == username)
        .ProjectTo<MemberDTO>(_mapper.ConfigurationProvider)
        .SingleOrDefaultAsync();
```

```
    }
```

The Where clause finds the user in the database, and the ProjectTo uses automapper to project the user to the DTO once it is found. This whole method is completed first, then the result is returned to the user controller, where no mapping needs to be done.

Additionally, for getting all users from the database, we can perform the same action:

```
public async Task<IEnumerable<MemberDTO>> GetMembersAsync()
    {
        return await _context.Users
        .ProjectTo<MemberDTO>(_mapper.ConfigurationProvider)
        .ToListAsync();
    }
```

This will return a list of the DTO's, already mapped appropriately.

NOTE: internal methods

One other reason the entity is returned initially is because we have the method GetAge within our AppUser entity. This method is responsible for calling the extension on DateTime to calculate the user's age. In order to circumvent this, we can comment out the original method call in AppUser, and instead call the extension method from the AutoMapper configuration file (hence the reason we use _mapper.ConfigurationProvider).

In the AutoMapperProfiles.cs file, in the CreateMap method that maps an AppUser to it's DTO, we use the "ForMember" method again:

```
.ForMember(dest => dest.Age, opt => opt.MapFrom(src =>
    src.DateOfBirth.CalculateAge()));
```

The destination is set as Age, the options are set as the src being DateOfBirth.CalculateAge(). So the CalculateAge method is called when the DateOfBirth property is reached, and the return value is mapped to the Age property in the DTO.

Section 9 - User Interface

Using Typescript

Typescript has type safety. When you want to ignore type safety you can use the "any" type, but that's typically unsafe and only good for development.

Typescript is basically javascript with stronger types. To keep type safety in objects, create an interface:

```
Interface Car {
        Color: string;
        Model: string;
        topSpeed: number;
}
```

Then we declare an object by:

```
Const car1: Car = {
        Color: ‘red’;
        Model: ‘WRX’;
        topSpeed: 100
}
```

We can have type safety in functions as well
const multiply = (x: number, y: number) => { return x * y }

Creating the Member Interface

Adding “export interface Member { } will give us autocompletion and better compiling with errors before we compile.

To quickly convert from JSON to .ts we can use a website like
https://transform.tools/json-to-typescript

Adding the Member Service

First, we can remove hard coded url’s by going to the src/environments/environment.ts and environment.prod.ts. The first is for development, and the second is for production.

In these files we can add

```
apiUrl: 'https://localhost:5001/api/'
```

Into the non-dev environment, and

```
apiUrl: ‘/api’
```

Into the production environment

Then anywhere we have a url needed (like in a service) we can call “environment.apiUrl”

To use the interface in a service, we can call it with an observable like this:

```
getMembers() {
        return this.http.get<Member[]>(this.baseUrl + 'users', this.getHttpOptions());
}
```

where getHttpOptions is a separate method that sets the headers for us. In this case we want the header to be the token, giving us authorization and authentication:

```
getHttpOptions() {
        const userString = localStorage.getItem('user');
        if(!userString) return;
        const user = JSON.parse(userString);
        return {
                headers: new HttpHeaders({
                Authorization: 'Bearer ' + user.token
                })
        }
 }
```

We first get the token out of local storage, then we check if it's even there. Then we parse the JSON token into an object, and return it as the header to the method that called for it.

NOTE: temporary

This is just a temporary way to do this, just to test it and ensure the return list is working. We can test it in the component.

NOTE: not-found error

Ensure that the correct endings are put onto the url after "baseUrl":

```
baseUrl + '/users' + username
```

Getting the List of Members

After injecting the service into the appropriate component, within the component we can make a method called "load" that will use the service methods to load the list into an array within the component.

```
loadMembers() {
        this.memberService.getMembers().subscribe ({
        next: members => this.members = members
        })
 }
```

Creating Member Cards

To display the members on cards we'll use bootstrap "cards" in a new component called "member-card". The component needs an

```
@Input() member: Member | undefined
```

in order to be able to pass the member down from the member-list component to the member-card component. In the member-list template (html) we call this component with

```
<app-member-card [member]="member"></app-member-card>
```

where the [member] = “member” is how we are passing the member down to the card

Build the card

```
<div class="card mb-4" *ngIf="member">
    <div class="card-img-wrapper">
        <img src="{{member.photoUrl}}" alt="{{member.knownAs}}"
        class="card-img-top">
    </div>
    <div class="card-body p-1">
        <h6 class="card-title text-center mb-1">
            <i class="fa fa-user me-2"></i>
            {{member.knownAs}}
        </h6>
        <p class="card-text text-muted text-center">{{member.city}}</p>
    </div>
</div>
```

The *ngIf checks if the member exists before we try to load it, since we said the member *could* be undefined (in the @Input() in the component for the card). Then we can build the card.

Hover Effect

To add a hover effect we can modify the .css file for the component. Start with

```
.card:hover img {
    transform: scale(1.2, 1.2);
    transition-duration: 500ms;
    transition-timing-function: ease-out;
    opacity: 0.7;
}
```

Target the card, specifically the image, and what happens when a user hovers over it. We’ll transform it up, over a period of 500 milliseconds, and it will ease-out for the animation. Additionally we’ll change the opacity to 0.7.

To reset the card, we use:

```
.card img {
    transform: scale(1.0, 1.0);
    transition-duration: 500ms;
    transition-timing-function: ease-out;
}
```

To keep the image inside the card instead of having it overflow outward, we use:

```
.card-img-wrapper {
        overflow: hidden;
}
```

Adding Buttons

To add some animated buttons that the user can click on is easy. When a user hovers over the image, we want some buttons to nicely transition over the image that the user can click on.

NOTE: Absolute Positioning

We'll want to position the buttons absolutely in the card instead of relatively.

Absolute positioning means something needs to be positioned against a relative position element. We want the buttons positioned absolutely so that we can set a specific place where they start and end within the card.

First we'll add buttons in the html file of the component:

```
<li class="list-inline-item">
        <button class="btn btn-primary btn-sm"><i class="fa fa-user"></i></button>
</li>
```

Since we're using font-awesome we can use the class "i" and "fa" to find an icon that fits. In the above example we use the icon "user".
The buttons are part of an unordered list, and each button has a class of "list-inline-item", a bootstrap class that sets the buttons to be in-line with one another.

The <ul> that the buttons are in has a few classes we'll use to animate and style the buttons. These are homegrown classes:

```
<ul class="list-inline member-icons animate text-center">
```

We first add the "relative" keyword to the card so that the buttons can be absolute.

```
.card-img-wrapper {
        overflow: hidden;
        position: relative;
}
```

```
.member-icons {
        position: absolute;
        bottom: -30%;
        left: 0;
        right: 0;
        margin-left: auto;
        margin-right: auto;
        opacity: 0;
}
```

```
.card-img-wrapper:hover
.member-icons {
        bottom: 0;
        opacity: 1;
}

.animate {
        transition: all 0.3s ease-in-out;
}
```

The member-icons class is the entire list of buttons that appears.

We target the hover of the card-img-wrapper to move the member-icons (buttons) up into position when we hover over the wrapper. The animation is an ease-in-out transition. The member-icons class starts the buttons at -30%, meaning they are technically on the page but not visible bc they are out of the bounds of the wrapper.

Authentication Interceptor

Earlier in “Adding the Member Service” we explored a quick and dirty way to get the token from the local storage and use it for authentication. Now we will replace the method with an interceptor.

The AccountService service has currentUser$ as an observable, so when a user logs in the currentUser$ holds all the necessary information for authentication. We can make user of this in an interceptor.

```
ng g interceptor _interceptors/jwt –skip-tests
```

The jwt interceptor needs the AccountService injected into it. Then we can add one method:

```
this.accountService.currentUser$.pipe(take(1)).subscribe({
    next: user => {
        if (user){
            request = request.clone({
                setHeaders: {
                    Authorization: `Bearer ${user.token}`
                }
            })
        }
    }
})
```

The “take(1)” dissolves the need to unsubscribe from the observable, as it only takes one instance of it and then drops it. We check if there is a user (“if(user)”), then we’ll take the request from inside the intercept method, create a clone of it so we can use the “setHeaders” method, and set the headers to an Authorization header.

Using the backticks (` `) is an alternative method of concatenation.

Routing the Detail Page

Update the buttons in the card to include the route:

```
<button routerLink="/members/{{member.userName}}"...
```

In the app.routing module, ensure the MemberDetailComponent route is set correctly:

```
{path: 'members/:username', component: MemberDetailComponent}
```

In the member.service file, ensure that the route to the api is correct:

```
getMember(username: string){
    return this.http.get<Member>(this.baseUrl + 'users/' + username);
 }
```

In the member-detail component, we need to add a property of “Member” that can be also undefined, since the file loads before we actually have a member.

We’ll inject the memberService, as well as an “ActivatedRoute” as “route”

We can create a method called “loadMember” that will handle fetching the member from the api and loading it into the “member” property. This will subscribe to the observable in the memberService.

Start by checking that we have a user with the given username:

```
const username = this.route.snapshot.paramMap.get('username');
    if(!username)
    {
        console.log("error");
        return;
    }
```

If we do, then we can assign the “member” property with the member we get from the api

```
this.memberService.getMember(username).subscribe({
    next: member => this.member = member
    })
```

In the “detail” component html, we can check that this is working with a simple

```
<h1 *ngIf=”member”>{{member.knownAs}}</h1>
```

The member name should display on the page

Styling the Details Page

NOTE: bootstrap class “col”

The col style represents 12 columns across the page. A class of “col-4” takes up one third of the page. “Col-12” takes up the full width of the page.

We can add tabs with the “TabsModule.forRoot();” added to the shared.module. This requires an import of ngx-bootstrap/tabs.

Don’t forget to export the TabsModule in the shared.module!

Building Tabsets

To build a set of tabs, first create a div for the tabset to go. Then you use <tabset> to create, and within the tabset you’ll use <tab> to create a tab.

```
<tabset class="member-tabset">
        <tab heading="About {{member.knownAs}}">
                <h4>Description</h4>
                <p>{{member.introduction}}</p>
</tabset>
```

The above creates a set with one single tab. More tabs can be added

Photo Gallery

Use this gallery for a photo gallery. Follow the instructions to install.

NOTE: errors while installing packages

Sometimes errors appear when trying to install packages that require certain dependencies but it’s because you’ure using a new version of Angular or etc.

To work around this, add “--force” or “--legacy-peer-deps” when installing. Should work.

Don’t forget to add to the shared.module file, both import and export

This package has a lot of things to do, but we’ll focus on the “options” and the array of photos that we’re displaying. In the module where the photos will be displayed we want a property of galleryOptions and a property of galleryImages, both arrays.

In the OnInit we’ll set the options, there are a variety of settings to add but here are the basics:

```
        this.galleryOptions = [
                {
                width: '500px',
                height: '500px',
                imagePercent: 100,
                thumbnailsColumns: 4,
                imageAnimation: NgxGalleryAnimation.Slide,
                preview: false
                }
```

We will also need to popular the galleryImages array with our images, so we will make a method to do that that loops through the member's photos and adds them one at a time (if the member exists)

```
if (!this.member) return [];
    const imageUrls = [];
        for (const photo of this.member.photos) {
            imageUrls.push({
                small: photo.url,
                medium: photo.url,
                big: photo.url
            })
        }
    return imageUrls;
```

The names "small", "medium", and "big" cannot be changed.

To use in the html template, in the tab of our choosing, add the gallery:

```
<ngx-gallery [options]="galleryOptions" [images]="galleryImages"
class="ngx-gallery"></ngx-gallery>
```

The class is added in the css file for this component as

```
.ngx-gallery { display: inline-block; margin-bottom: 20px;}
```

NOTE: Loading Images

The getImages() method (above) should be called during the "loadMember()" method call, not in the OnInit. If the latter is done, it is possible that the member data has not been retrieved completely before the getImages is called, meaning that the "if(!this.member)" will be true, and an empty array will be returned instead! Call the getImages from the loadMember in the same code block:

```
this.memberService.getMember(username).subscribe({
    next: member => {
        this.member = member,
        this.galleryImages = this.getImages();
        }
    })
```

Section 10: Updating Resources

Adding an Edit component

```
ng g c members/member-edit
```

We want to be able to access the accountService to find which member is logged in, and then access the memberService to get all the member details to populate the form. Inject both into the constructor.

Also in the constructor, subscribe to the currentUser$ observable:

```
this.accountService.currentUser$.pipe(take(1)).subscribe({
        next: user => this.user = user
        })
```

Then load the member into the component

```
loadMember(){
        if (!this.user) return;
        this.memberService.getMember(this.user.username).subscribe({
        next: member => this.member = member
        })
 }
```

Call the loadMember method from the ngOnInit()

Adding the Form

Use the <form> tag in the html template. Label it as #editForm

```
<form #editForm=”ngForm (ngSubmit)=”updateMember()”></form>
```

The button to submit should be inside the form, however if it is not, we can link the button to the form through an “id” tag:

```
<form #editForm="ngForm" id="editForm" (ngSubmit)="updateMember()">
```

Then tell the desired button which form to link to through the id:

```
<button type="submit" form="editForm">Save Changes</button>
```

Adding Info

Adding the following to the information box at the top will ensure it only appears if the form has been edited by the user:

```
<div class="alert alert-info" *ngIf="editForm.dirty">
        <p><strong>Information: </strong>Any unsaved changes will be lost.</p>
</div>
```

Accessing the Form inside the Component

To access the form from the html template inside the component (for updating information) we use the following property:

```
@ViewChild(‘editForm’) editForm: NgForm | undefined;
```

Now we have access to the editForm methods that would normally only be in the html template. This lets us use editForm.reset to reset the form automatically after something happens (like the user submits the form

```
this.editForm?.reset(this.member)
```

Resets to the member values

Adding Can Deactivate

This feature allows interaction with the rest of the, to listen for when a user tries to navigate away from the page after making changes to the form but without saving them yet.

For this we'll use a "guard"

```
ng g g _guards/prevent-unsaved-changes –skip-tests
```

Deselect: can activate
Select: can deactivate

In the guard, set the the CanDeactivate<> to the component you wish to implement

```
CanDeactivate<MemberEditComponent>
```

And where it says "component:" set that to the same component. For this guard we are only going to return a boolean, and more importantly we're going to set a javascript pop-up to warn the user anyways

```
if (component.editForm?.dirty) {
    return confirm('Are you sure you want to continue? Any unsaved changes will be
    lost');
    }
    return true;
```

Because we set the component above, we have access to the editForm inside it.

In the routing module we need to modify the route to implement the guard:

```
{path: 'member/edit', component: MemberEditComponent, canDeactivate:
[PreventUnsavedChangesGuard]}
```

NOTE: Host Listener

This method only stops a user from losing changes if they navigate away within our app, but what about within the browser? A host listener allows this to happen.

Host listener is provided by Angular, and is implemented in the desired component thusly:

```
@HostListener('window:beforeunload', ['$event']) unloadNotification($event:any) {
    if (this.editForm?.dirty){
    $event.returnValue = true;
    }
 };
```

This code gets placed outside the constructor with the rest of the properties of the component. The pop up that appears is browser specific, and cannot be adjusted.

Updating the Database (backend)

As always, start with a DTO with the properties we want to save only. Make them match the entity's properties.

Add the map to Automapper

```
CreateMap<MemberUpdateDTO, AppUser>();
```

In the controller, create a method that returns an async task<ActionResult> and takes in the DTO we just created.

Because the controller inherits from our BasAPIController, which inherits from ControllerBase, we have access to the "ClaimsPrinciple" method within it. We access this with "User" (standard keyword and unrelated to our code specifically).

```
var username = User.FindFirst(ClaimTypes.NameIdentifier)?.Value;
```

The above line returns the claim value as a long string that includes a line "NameIdentifier", which points to the username of the user. This username is used to set the value of "username".

Once we have the username, it's all pretty standard from there:

```
var user = await _userRepository.GetUserByUsernameAsync(username);
if (user == null) return NotFound();
_mapper.Map(memberUpdateDto, user);
if (await _userRepository.SaveAllAsync()) return NoContent();
return BadRequest("Failed to update user");
```

Get the user from the db, check if they exist, map the dto to the user, save the user, return nothing if successful, return badrequest if not.

This method returns a bad request if no updates were done but the user tries to update anyways.

Updating the Database (front end)

First add the method to the service:

```
updateMember(member: Member) {
    return this.http.put(this.baseUrl + 'users', member);
}
```

So we set the url and then pass the "member" up as the body of the request.

We'll call this method from the component, and it will return an observable. The observable won't have anything in it, but we can move our toast and reset form calls into the "next" anyways:

```
this.memberService.updateMember(this.editForm?.value).subscribe({
        next: _ => {
                this.toastr.success('Profile updated successfully');
                this.editForm?.reset(this.member);
        }
})
```

this.editForm?.value contains all the information from the form. Because the information matches the information (via "name" in the form) as the "Member" interface, we can pass this to the member.service as a "Member" object. Only passed information will be updated though, and other information won't be touched / need not be sent.

Adding a Loading Indicator

Check out this spinner here: https://github.com/Napster2210/ngx-spinner
View the demo for other options. Use ng add ngx-spinner@X.X.X where "x.x.x" is the version to install based on version of Angular being used

NOTE: package install issues

Like previous examples, there may be issues with other packages that are installed, so use "npm install ngx-spinner –legacy-peer-deps" to force install

Add the spinner .css to the angular.json "styles" section. Import the "NgxSpinnerModule" and export it as well in the shared.module file. Here is where you'll reiterate the type of spinner:

```
NgxSpinnerModule.forRoot({
        type: 'line-scale-party'
})
```

We'll create a service that runs the spinner when an http request is sent. We'll use an interceptor to check http requests and then the service will run with a method called "busy" that shows the spinner. Another method called "idle" will reset the spinner to invisible when it's not needed.

```
busy() {
        this.busyRequestCount++;
        this.spinnerService.show(undefined, {
        type: 'line-scale-party',
        bdColor: 'rgba(255,255,255,0)',
        color: '#333333'
        })
```

```
idle() {
        this.busyRequestCount--;
        if (this.busyRequestCount <= 0) {
        this.busyRequestCount = 0;
        this.spinnerService.hide();
        }
  }
```

}	

In the new interceptor we need to import the above service first. Then we can call it when the intercept happens. We'll call the "busy" method first. Then in the return statement we can "pipe" to call the idle function to stop the spinner.

```
return next.handle(request).pipe(
    delay(1000),
        finalize(() => {
        this.busyService.idle()
        })
    )
```

In the above example we add a delay of 1 second so that we can see the spinner in action on our local machine, as requests don't take that long.

List the interceptor in the providers section of the app.module file

```
{provide: HTTP_INTERCEPTORS, useClass: LoadingInterceptor, multi: true}
```

Then add the component to the app.component.html section

```
<ngx-spinner type="line-scale-party">
    <h3>Loading...</h3>
</ngx-spinner>
```

NOTE: Error "static ɵcmp: i0.ɵɵComponentDeclaration<NgxSpinnerComponent"

This error was caught during compiling. The version of ngxSpinner 14.0.0 has some sort of error in the component.d.ts file. The fix was to downgrade to version **13.1.1 with npm install ngx-spinner@13.1.1 --legacy-peer-deps**, followed by removing the "forRoot" and arguments from the NgxSpinnerModule import in the shared.module file.

NOTE: Changing the Icon

Remove the "type="" " from the app.component call for the spinner. Then the **service** is the only place the spinner type is declared, and it can be changed there without refactoring other files.

Storing the State

Since components get destroyed when a user navigates away from the page, we lose the loaded information. However, services are not destroyed, they persist through the life of the application. So we can store the user data in a service and get it from there instead of constantly pulling from the db.

Similar to how we have a members array “members[]” in the component, we can use the same technique in the members.service. In the getMembers method we’ll first check that the members array is populated with members, and then if it is we’ll return it as an observable:

```
if (this.members.length > 0) return of(this.members);
```

If it’s not, then we’ll get the members from the API first, then use the pipe to map them to the array, then return them.

```
return this.http.get<Member[]>(this.baseUrl + 'users').pipe(
    map(members => {
    this.members = members;
    return members;
    })
    )
```

We can do the same for a single member “getMember” as well, by adding a “find”:

```
const member = this.members.find(x => x.userName === username);
    if (member) return of(member);
```

Finally, we can do the same in the updateMember method:

```
return this.http.put(this.baseUrl + 'users', member).pipe(
    map(() => {
        const index = this.members.indexOf(member);
        this.members[index] = {...this.members[index], ...member}
        })
    )
```

In the return map we don’t get anything back from the api so it is blank (), however we want to find the index of the member in the array. Once we have that, we use the “spread operator” (...) to map the “member” values to the “member[index]” value and set the “this.members[index]” values to the new value.

Now, to use these refactored methods we need to adjust the component that uses them, so in the member-list component we’ll first change the “members” array to an observable:

```
members$: Observable<Member[]> | undefined;
```

Then in the ngOnInit we can set the observable by calling this.memberService.getMembers(), since that method returns an observable.

Then in the component that shows the member-list, we’ll replace the single “member” in the ngFor loop with the observable and the async pipe:

```
<div class="col-2" *ngFor="let member of members">
```

becomes:

```
<div class="col-2" *ngFor="let member of members$ | async">
```

Section 11: Photo Upload Service

Cloud Service and API Configuration

Use “Cloudinary”, for a free cloud service account. Log in with Github. In the dashboard find the “Cloud Name”, “API Key”, and “API Secret”.

Install into app via nuget package manager, search for “Cloudinary.net”

Configure the API

In the appsettings.json file we need to add a section for the above mentioned items:

```
"CloudinarySettings": {
        "CloudName": "XXX",
        "ApiKey": "XXX",
        "ApiSecret": "XXX"
 }
```

Copy the required items to the “XXX” above. These should be kept secret, so not in a public repo.

Then we’ll make a helper called “CloudinarySettings” where we’ll declare properties named after the ones above, all public strings.

In applicationServiceExtensions, add the service:

```
services.Configure<CloudinarySettings>(config.GetSection("CloudinarySettings")
);
```

Build an interface that has two methods:

```
Task<ImageUploadResult> AddPhotoAsync(IFormFile file);
Task<DeletionResult> DeletePhotoAsync(string publicId);
```

Create an API Service that implements the interface. The service should have private property:

```
private readonly Cloudinary _cloudinary;
```

It will need a constructor that creates an account object that we can then use to upload and delete photos:

```
public PhotoService(IOptions<CloudinarySettings> config)
    {
        var acc = new Account
        (
            config.Value.CloudName,
```

```
                config.Value.ApiKey,
                config.Value.ApiSecret
        );

    _cloudinary = new Cloudinary(acc);
    }
```

So, we add the info to the appsettings file (1) and then create a helper that will access it (2). We add the helper in the extensions (3), using Configure<"helper_name">(config.GetSection("appsettings_section"). Then we build an interface (4) and implement it in the services file (5). Then we use the IOptions in the constructor (6) to bring in the helper "CloudinarySettings", which then gives us access to the items in the appsettings.json file.

Updating the Users Controller

In the usersController we can add a method called AddPhoto that will take an IFormFile as a parameter, get the user via the ClaimsPrinciple item "User", and then upload the photo, as well as add the photo to the user db.

NOTE: ClaimsPrinciple Extension Method

At this point we created an extension method for the claims principle class that allows us to shorten the code needed to get the username:

```
public static string GetUsername(this ClaimsPrincipal user){
        return user.FindFirst(ClaimTypes.NameIdentifier)?.Value;
}
```

Now we can use this extension method in the userController, replacing

```
var username = User.FindFirst(ClaimTypes.NameIdentifier)?.Value;
var user = await _userRepository.GetUserByUsernameAsync(username);
```

with

```
var user = await _userRepository.GetUserByUsernameAsync(User.GetUsername());
```

NOTE: User

This works because "User" is a property of type ClaimsPrincipal in the ControllerBase (which we inherit from in our BaseAPIController class), and the ClaimsPrincipal class has a method called "FindFirst". "User" is available in all controllers that inherit from ControllerBase.

With our new Extension, we can get the username from the logged in user. Then we can simply call the AddPhotoAsync using the _photoService that we injected into the controller:

```
var result = await _photoService.AddPhotoAsync(file);
```

After checking for errors, we can create a new “Photo” object and set the Url and PublicId:

```
var photo = new Photo
    {
        Url = result.SecureUrl.AbsoluteUri,
        PublicId = result.PublicId
    };
```

We can check to see if the user has any photos already (because we are eager loading them from the userRepository) and if not, this photo can be set to the main photo.

Then we’ll add the photo to the Photos db, and then save. When we save, we’ll return a photoDto after mapping the Photo to the PhotoDto object

```
user.Photos.Add(photo);
if (await _userRepository.SaveAllAsync()) return _mapper.Map<PhotoDTO>(photo);
```

Using the CreatedAt Route

NOTE: Rest API’s

For our recent AddPhotos method in the usersController, which is an HttpPost, we are returning the objectDTO, but in a Rest API we would actually want to return a 201 Created, in addition to where the resource can be found

For this app, we don’t want to return the actual location of the resource (the photo) but instead the user page where you can view the photos.

To do that, we’ll change the return value of the method from

```
if (await _userRepository.SaveAllAsync())
    return _mapper.Map<PhotoDTO>(photo);
```

to…

```
if (await _userRepository.SaveAllAsync())
    {
        return CreatedAtAction(nameof(GetUser),
            new {username = user.UserName},
            _mapper.Map<PhotoDTO>(photo));
    }
```

Adding a Photo Editor Component

ng g c members/photo-editor –skip-tests

In the component we want to receive the member from the parent, so we'll use an

```
@Input() member: Member | undefined;
```

Next in the html template we'll start with a very basic photo viewer so we can see that all is working. We can first check that we have a member (*ngIf="member"), then we'll loop through the photos (*ngFor="let photo of member.photos"). Then we can show each photo with

```
<img src="{{photo.url}}" alt="Photo of user" class = "img-thumbnail mb-1">
```

We also need to include the new component in the member-edit component so that it shows up, so we go to the tab for "Edit Photos" and add

```
<app-photo-editor [member]="member"></app-photo-editor>
```

This will pass the member down to the photo-editor component so it can be used.

Adding the Photo Uploader

Instead of writing this by hand we'll use a premade library called "ng2-file-upload" found here.

NOTE: error when compiling (again)

Similar to the last item we installed, there could be a compile error when installing the latest version of this regardless of the version of Angular. To circumvent this, use

```
npm install ng2-file-upload –save –legacy-peer-deps
```

to install the version that works. You can append the "@" tag after "upload" to specify a previous version (3 seems to work, just not "next").

Add to shared module

Import "FileUploadModule" and export "FileUploadModule". No "forRoot" necessary

Edit the component

In the photo-editor component this library needs some specific items in order to work, so start with properties:

```
uploader: FileUploader | undefined;
hasBaseDropZoneOver = false;
baseUrl = environment.apiUrl;
user: User | undefined;
```

We need the "FileUploader" to actually work, the "hasBaseDropZone" is used for the drag-n-drop feature, and the baseUrl and User are so we can route the file appropriately.

Start by getting the currentUser$ and subscribing

We'll need a method called "initializeUploader" where we'll build the FileUploader object with the necessary options:

this.uploader = new FileUploader({ url: this.baseUrl + 'users/add-photo', authToken: 'Bearer ' + this.user?.token, isHTML5: true, allowedFileType: ['image'], removeAfterUpload: true, autoUpload: false, maxFileSize: 10 * 1024 * 1024 });	Url to send the photos to Getting the authorization Standard File types allowed to upload Don't keep the files in memory Wait for user to click 'upload' Maximum file size (10mb here)

We'll also add the following to the above method for once the file is uploaded

```
this.uploader.onAfterAddingFile = (file) => {
        file.withCredentials = false
        }
```

Finally, we will have on more thing at the end for when the file is successfully uploaded:

```
this.uploader.onSuccessItem = (item, response, status, headers) => {
        if (response) {
        const photo = JSON.parse(response);
        this.member?.photos.push(photo);
                }
        }
```

The uploader

With the above in place, we can edit the html template to include a file uploader that users can interact with. The code on this page can be modified to suit needs, or kept as is, although some classes need to be changed or modified to display properly.

Setting the Main Photo (API)

This will be an HttpPut request with a parameter of {photoId}, so it will ultimately return a "NoContent()" if successful. To start, we'll get the user from the userRepository, followed by the

photo by the Id that is passed in. Both should use defensive coding to check that they are not null.

Check also to see if the selected photo is already the main photo, if so return a “BadRequest()”

Once we have a user and a desired main photo, then we’ll find the current main photo, set it to false, then set the new selected photo to true:

```
var currentMain = user.Photos.FirstOrDefault(x => x.IsMain);
if (currentMain != null) currentMain.IsMain = false;
photo.IsMain = true;
```

Then we can save the results and return “NoContent()”

Adding the Main Photo to the Navbar

We want the main photo to be available to the front end so first we’ll add a photoUrl property to the “userDto” entity in the API as “PhotoUrl”. We need to eager-load this property when we get a user on log-in, so in the “login” method we’ll chain an “include” on to the first call to get a user from the database:

```
var user = await _context.Users
        .Include(p => p.Photos)
        .SingleOrDefaultAsync(x => x.UserName == loginDTO.Username);
```

Now for the client side of things, we’ll add “photoUrl” of type ‘string’ to the user interface. Now it will be available to use on the frontend when we load the user through accountServices.

In the nav-component.html we can add an img tag in the navbar, with the source being the photoUrl, and an alternative picture in case a main photo does not exist yet:

```
<img src="{{user.photoUrl || './assets/user.png'}}" alt="user photo">
```

NOTE: User versus Account Controllers

The account controller handles security of registering and logging in only, and loading the user’s information once they are registered or logged in. The User controller, on the other hand, assumes a user is already logged in and therefore focuses on the activities a logged in user may partake in, like updating the profile, changing photos, etc. The user controller connects to the database via the userRepository class, which then connects to the database, adding an extra layer for testing purposes. The repository class is not technically necessary and all of it’s methods could be relocated to the user controller.

NOTE: Using DTO's

The use of DTO's allows us to add properties to entities that are passed around the application without having to modify the item in the database. For example, we now have access to the photoUrl property via the DTO, but this property is not a property of the AppUser entity. We only have two entities in our database, but we have (as of this writing) six DTO's to represent them in various ways.

Setting the Main Photo (Client)

Members are people who have filled out the information that is displayed for other users to see, so in the membersService we'll add a new method called "setMainPhoto" that takes in a number and returns an http request. This method will access the API method we created two sections ago:

```
return this.http.put(this.baseUrl + 'users/set-main/photo/' + photoId, {});
```

Note the empty curly brackets

This is because the "put" method expects at least two arguments, a url and a body. Since we don't have anything to send up, we just send an empty object. This can be done for a variety of methods that require this.

Photo Editor Component

Because users will want to edit their photos in the photo editor component, we'll add a method in there to do so. This method will take in a "Photo" object and connect to the memberService to set the main photo.

```
this.membersService.setMainPhoto(photo.id).subscribe({
    next: () => {
    if (this.user && this.member) {
        this.user.photoUrl = photo.url;
        this.accountService.setCurrentUser(this.user)
        this.member.photoUrl = photo.url;
        this.member.photos.forEach(p => {
            if (p.isMain) p.isMain = false;
            if (p.id === photo.id) p.isMain = true;
        })
    }
    }
    })
```

After checking that we have a user and a member, we'll

- Change the photoUrl for the user to be the new photoUrl of the new main photo
- Update the "user" object in accountService

We do the second thing because other components are subscribed to the user as an observable, and so setting the user this way updates all other subscribed services immediately

We also need to update the member photoUrl because that's where the main photo is displayed in the larger profile view. Finally, we loop through the photo array and change the newly selected photo to "isMain" and remove the "isMain" from the old main photo.

NOTE: Members as Observable

In the membersService we also declared the "members" array as an observable in the "getMembers" method (by using the 'of' keyword), so when the member is updated, so is the object in the array.

Deleting Photos (API)

Similar to the SetMainPhoto method in the userController, we'll add a DeletePhoto method below that.

Get the user from the repository
Get the photo to be deleted using FirstOrDefault and the photoId passed in
Check that the photo exists, or that the photo is not the main photo
Because we're using cloudinary we should also check that the photo has a publicId; if it does we'll delete from the cloud as well using:

```
var result = await _photoService.DeletePhotoAsync(photo.PublicId);
if (result.Error != null) return BadRequest(result.Error.Message);
```

Tell entity to remove the photo we found
SaveAllAsync and return Ok()

Deleting Photos (Client)

Starting in the membersService we'll make a method that calls the API with an http.delete:

```
return this.http.delete(this.baseUrl + 'users/delete-photo/' + photoId);
```

Note that this method does not require a body, so no {} is necessary

In the photo-editor component we'll add a method to handle deleting:

```
deletePhoto(photoId: number){
    this.membersService.deletePhoto(photoId).subscribe({
        next: () => {
            if (this.member) {
            this.member.photos = this.member.photos.filter(x => x.id !=
            photoId)
            }
        }
    })
 }
```

Buttons for editing photos

Finally, we can modify the buttons in the photo editor html template to allow users to set a main photo and to delete photos:

```
<button
        class="btn btn-sm"
        [disabled]="photo.isMain"
        (click)="setMainPhoto(photo)"
        [ngClass]="photo.isMain ? 'btn-success active' : 'btn-outline-success'"
>Main</button>

<button
        class="btn btn-sm btn-danger"
        [disabled]="photo.isMain"
        (click)="deletePhoto(photo.id)"
><i class="fa fa-trash"></i></button>
```

With the [disabled] we can disable the button based on the conditional of the photo being the main photo or not. We set the click event to follow the desired method and pass the appropriate parameter.

In the first button, we use a ternary conditional to check if the photo is the main; if it is then the button is filled, and if not it is an outline only.

Section 12: Reactive Forms

Reactive Forms Introduction

Reactive forms are component based; we create and control the form via the component instead of the html template, making validation easier as well as testing.

Start by importing ReactiveFormsModule in the app.module file

In the component to be using the reactive forms (in this case our “register” component) we’ll add a new property:

```
registerForm: FormGroup = new FormGroup({});
```

We’ll start simple by just having three things to input into, and we’ll initialize them in a new method:

```
initializeForm() {
```

```
        this.registerForm = new FormGroup ({
        username: new FormControl(),
        password: new FormControl(),
        confirmPassword: new FormControl()

        })
    }
```

In the html template we can replace the “#registerForm=*ngForm” with “[formGroup]=”registerForm””

Then each input gets its own div with “formControlName” set to the appropriate name front he intializeForm method in the component:

```
<div class="mb-3">
        <input
        type="text"
        class="form-control"
        formControlName="username"
        placeholder="Username"
        />
 </div>
```

Adding Client-Side Validation

In the initializeForm method, for each FormControl object we can add parameters to pass in. The first can be an initial value, and then the next can be either a validator, or an array of validators:

```
Username: new FormControl(‘initial_value’, Validators.required)
```

The “Validators” covers literally every possible scenario, but custom validators can be constructed as well.

Custom Validator

A custom validator can be used to check that password and password confirmation are matching. We’ll start with a method to check this, a method that takes in a string (the original password) and returns a Validator as a function (ValidatorFn)

```
matchValue(matchTo: string): ValidatorFn {
```

In this method we’ll run the following:

```
return (control: AbstractControl) => {
    return control.value === control.parent?.get(matchTo)?.value ? null : {notMatching: true}
```

In the above, control is an abstract function that matches the value of the input with the value that was passed in. If they match, it returns ‘null’ for a valid form. If not, it returns an object called ‘notMatching’ set to ‘true’. (the name “notMatching” is not important here, just the value ‘true’)

We then call the above method in the validators array:

```
confirmPassword: new FormControl('', [Validators.required, this.matchValue('password')])
```

However, we still have an error where a user can change the initial password after confirming it and the form remains valid. To circumvent this we need to check if that field has been changed, and if so, revalidate. We do that by adding this at the end of the initializeForm method:

```
this.registerForm.controls['password'].valueChanges.subscribe({
    next: () =>
    this.registerForm.controls['confirmPassword'].updateValueAndValidity()
    })
```

Since this returns an observable we can subscribe to any changes that are made, and update the value and validity through the provided function.

Validation Feedback

To start, we want to use bootstrap classes to display errors to users when they have not entered text into the required fields, so we can add that in the “input” areas for each input:

```
<input
type="text"
[class.is-invalid]="registerForm.get('username')?.errors &&
registerForm.get('username')?.touched"
class="form-control"
formControlName="username"
placeholder="Username"
/>
```

Below that we’ll add a div that shows them what they did wrong using another bootstrap class “invalid-feedback”:

```
<div class="invalid-feedback">Please enter a username</div>
```

Now what about for fields that have more than one validator?

```
<div
class="invalid-feedback"
*ngIf="registerForm.get('password')?.hasError('required')"
>
Please enter a password
</div>
<div
```

```
class="invalid-feedback"
*ngIf="registerForm.get('password')?.hasError('minlength')"
>
Password must be at least 4 characters
</div>
```

Notice the conditional above, checking for the specific error “minLength”

For our custom validator, we use the “notMatching” name in the returned object, and this is where that name becomes important:

```
class="invalid-feedback"
*ngIf="registerForm.get('confirmPassword')?.hasError('notMatching')"
> Passwords do not match
```

In this div is where we check that specific item, and if it is set to true (instead of ‘null’ when the passwords match) then we will display this error to the user

Reusable Text Input

The above of course is too much text and will only get larger as more inputs are added, so creating a reusable input component is best.

Start by creating a new component. Where it says “implements OnInit” we can remove the “OnInit” and replace it with “ControlValueAccessor”, which is an interface, and will need use to implement the required methods.The methods that are added, however, can be left blank, because the form will pass through the methods without us having to do anything inside them. All the methods are controlled via the form control.

In the constructor we’ll set an NgControl using the “@Self” decorator:

```
constructor(@Self() public ngControl: NgControl)
```

NOTE: Why “@Self”?

When we inject something into a constructor Angular checks to see that it’s been used recently (in memory). When we inject this NgControl, however, we don’t want to reuse another control that is already in memory, we want to make sure that this NgControl is unique to the inputs that we’re updating in the DOM.

We need to initialize the NgControl in the constructor:

```
this.ngControl.valueAccessor = this;
```

So we’re setting the valueAccessor of the ngControl to the component we are currently in

We’ll also add some properties that we can reuse, namely:

```
@Input() label = '';
@Input() type = 'text';
```

We can set the ‘label’ for the input and the type of input for each instance of the component when we use it

We’ll also add a single method at the end of the component that will return the NgControl object as a FormControl object to the html template, making it easier to access the items within it.

```
get control(): FormControl {
    return this.ngControl.control as FormControl
 }
```

Now, in the html template we will build the template form that we can reuse for each input item. The idea is that, using the binding provided in angular, we can send up the type of input and the label to the component and have it rendered on the page. We’ll also add all validators in the template, using the conditional *ngIf to check what kind of error we get and then displaying the appropriate message.

Starting with the input:

```
<input
    type="{{ type }}"
    [class.is-invalid]="control.touched && control.invalid"
    class="form-control"
    [formControl]="control"
    placeholder="{{ label }}"
 />
```

For each invalid-feedback error we can just use a conditional check. The following checks if the validator error has to do with the minimum length of a password:

```
<div class="invalid-feedback" *ngIf="control.errors?.['minlength']">
    {{ label }} must be at least
    {{control.errors?.['minlength'].requiredLength}} characters
</div>
```

Notice the use of interpolation and binding to provide text for the ‘label’ and ‘requiredLength’ values

Now, in the component that will contain the inputs we can add a simple component call instead of the full-on div text that we had once before. Each input would look like:

```
 <app-text-input
    [formControl]="$any(registerForm.controls['password'])"
    [label]="'Password'"
    [type]="'password'"
 ></app-text-input>
```

The “formControl” is cast as “any” and bound to the ‘password’ control
The “label” is shown as text “Password”
The “type” is typically default to “text” but in this case we wan’t security, so we set it to ‘password’

Note the single quotes for the label and type, these are necessary to pass them to the component.

Form Builder Service

Reactive Forms provides this service that makes things a little shorter for us.

In the original register component where we initialize the form, we have:

```
this.registerForm = new FormGroup ({
        username: new FormControl('', Validators.required),
        password: new FormControl('', [
        Validators.required,
        Validators.minLength(4),
        Validators.maxLength(8)]),
        confirmPassword: new FormControl('', [Validators.required,
this.matchValue('password')])

        });
```

We’ll change this by using a FormBuilder object ‘fb’ that we inject into the component

```
this.registerForm = this.fb.group({
        username: ['', Validators.required],
        password: ['', [
        Validators.required,
        Validators.minLength(4),
        Validators.maxLength(8)]],
        confirmPassword: ['', [Validators.required, this.matchValue('password')]]

        });
```

This doesn’t save a lot of text, but is typically how form groups are created out in the wild, so be aware of it.

Adding More Fields

This can simply be done by copying down what we already have in place in the component:

```
gender: ['male', Validators.required],
username: ['', Validators.required],
```

```
knownAs: ['', Validators.required],
dateOfBirth: ['', Validators.required],
city: ['', Validators.required],
country: ['', Validators.required],
```

Note the default value of ‘male’ in gender; this is because we will use a radio button for this item and radio buttons are difficult to validate, so we’ll just set a default value.

To make the radio button in the html we can use the following:

```
<div class="mb-3">
    <label style="margin-right: 10px">I am a: </label>
    <label class="form-check-label">
    <input type="radio" class="form-check-input" value="male"
formControlName="gender"> Male
    </label>
    <label class="form-check-label">
    <input type="radio" class="form-check-input ms-3" value="female"
formControlName="gender"> Female
    </label>
 </div>
```

The formControlName=”gender” allows the binding of the input value into the form

Everything else can be a copy of the original text input field, with the labels changed appropriately as well as the control binding.

```
<app-text-input
    [formControl]="$any(registerForm.controls['knownAs'])"
    [label]="'Known As'"
 ></app-text-input>
```

Adding a Reusable Date Input

Browsers provide a generic looking date picker, but it varies from browser to browser and we want more control over this. If we simply set the “type” to “date” in the html template then the date picker is automatic, but still looks generic. So we’re going to use ngx-bootstrap to custom make a datepicker item.

Start here for information about the datepicker item.

We’ll make a new component called “date-picker”. Then in the shared module we’ll import BsDatepickerModule.forRoot() and export it as well. Make sure the “import” at the top populates, else you may need to copy it from the above website.

In the new component we are going to treat it the same as the text-input template component by implementing the ControlValueAccessor. We'll have properties coming in as well:

```
@Input() label = '';
@Input() maxDate: Date | undefined;
```

We'll also have a bsConfig property

```
bsConfig: Partial<BsDatepickerConfig> | undefined;
```

The constructor will also make use of the @Self() public ngControl: NgControl import, and then we'll instantiate it:

```
this.ngControl.valueAccessor = this;
this.bsConfig = {
        containerClass: 'theme-red',
        dateInputFormat: 'DD MMMM YYYY'
}
```

We'll also make a bsConfig object that sets the theme of the calendar as well as the date format

Just like with the last component we need a "get contro()" method to cast the ngControl object to a FormControl object.

To display the date picker, in the html of the component we'll set up the input as follows:

```
type="text"
[class.is-invalid]="control.touched && control.invalid"
class="form-control"
[formControl]="control"
placeholder="{{label}}"
bsDatepicker
[bsConfig]="bsConfig"
[maxDate]="maxDate"
```

Users can type in their date if they want, and validation will occur based on whether the box has been touched and if it's invalid. Class and placeholder are standard, and formControl is still "control" (using the "get control" method). bsDatePicker lets the browser know to use that library, the config property is sent through, the maxDate property is sent as well.

To add this to our register component we'll change the dateOfBirth input from an <app-text-input> to be:

```
<app-date-picker
[formControl]="$any(registerForm.controls['dateOfBirth'])"
[label]="'Date of Birth'"
[maxDate]="maxDate"
 ></app-date-picker>
```

Finally, in the register component we want to set the maxDate property that we pass to the datePicker component:

```
maxDate: Date = new Date();
```

and initialize when component is made:

```
this.maxDate.setFullYear(this.maxDate.getFullYear() - 18);
```

So when the register component is built, it will calculate the appropriate maximum date, then pass that down to the datePicker component, where it can be accessed by the component's html. The progression is:

Note on Property Binding

First, check out this website for more info.

Note that the Property Binding uses the following Syntax: [binding-target]="binding-source"
So the "item" is assigned to the [target]

In our datePicker case, maxDate is a property of the bs-datepicker.component.d.ts class, meaning it is part of the object and has a purpose / method. We want to pass the maximum date down to this property from the register component. We do that by

first setting the maxDate to a specific date in the register component
Then we pass that to the <app-date-picker> component through
[maxDateToSet] = "maxDateFromRegisterComponent"
The date-picker.ts component gets this date, and then sends it to the date-picker.html
The date-picker.html sends the maxDate to the class for use via
[maxDate] = "maxDateToSet"

NOTE: AutoDate

It appears that with the above validation, the date will automatically default to the youngest date without going over the max in the event a user inputs a date lower than the max via typing directly into the form.

Updating the API for Registration

In the registerDTO we'll add the properties for the items a user is to submit in the register form. We also need to add a new AutoMapper profile that maps the RegisterDTO to the AppUser:

CreateMap<RegisterDTO, AppUser>();

NOTE: nulls in Entity

To prevent issues of not allowing nulls to be inserted to the database, you can set the values of the properties that are not submitted during registration. Do this in the AppUser Entity by setting null values to either empty strings or the equivalent default value. A better option would be to allow nulls but at this point it's just easier to set values when created.

In the Account Controller in the API we'll inject the IMapper class to start, as "_mapper". Now, in the register method we can map the registerDTO object to a new AppUser. That's it. If we want

to add a property to the UserDTO we can do that (such as the “KnownAs” property) and now that we have access to it we can return it along with the username and token.

Test with Postman by registering a new user and then getting the user by username.

Client Side Registration

In our register method inside the register component, we want to reroute the user to the “members” page once they register, so we need to inject a Router first. Then we can say

```
this.router.navigateByUrl('/members')
```

We still have a toast notification but it’s not needed any longer, so we’ll remove it and replace it with an array of errors from the server. We’ll use a string array property to start:

```
validationErrors: string[] | undefined;
```

and then for the errors we’ll say

```
error: error => {
    this.validationErrors = error
    }
```

We also need to update the method to use the registerForm that we are now implementing:

```
this.accountService.register(this.registerForm.value).subscribe({
```

Then we can add the validation error list to the html just above the buttons in the form:

```
<div class="row" *ngIf="validationErrors">
<ul class="text-danger">
<li *ngFor="let error of validationErrors">
{{error}}
</li>
</ul>
</div>
```

DateOfBirth

The DateOfBirth item can cause us some issues because of timezones, as well as the DateTime object containing time stuff as well. We only want a date to be shown. So we’re going to strip the time stuff from the date that is input in the form.

We’ll start by adding a method to just get the date out of a dateTime object:

```
private getDateOnly(dob: string | undefined) {
if (!dob) return;
let theDob = new Date(dob);
return new Date(theDob
```

```
        .setMinutes(theDob.getMinutes()-theDob.getTimezoneOffset())).toISOString().slice(0,
10);
  }
```

The above method will convert the string it receives to a date object, then remove all time stuff and only leave the date stuff.

The Register Method

Now that we have a method to modify the date, and we have a registerForm to handle the forms, let's modify our register method one last time to reflect these things:

```
        const dob = this.getDateOnly(this.registerForm.controls['dateOfBirth'].value);
        const values = {...this.registerForm.value, dateOfBirth: dob};
        this.accountService.register(values).subscribe({
        next: () => {
        this.router.navigateByUrl('/members')
        },
        error: error => {
        this.validationErrors = error
        }
        })
```

We use the spread operator to insert the new dob item into the dateOfBirth value from the register form.

We can disable the register button on the page with a simple:

```
        [disabled]="!registerForm.valid"
```

We can also disable the submission of the form itself unless it is valid

```
        (ngSubmit)="registerForm.valid && register()"
```

We do this in the event a user uses a tool like Postman to circumvent the UI entirely. It is important to have server-side validation for cases like this.

Update the photo for a new user

When a user registers they will have a default photo showing, and they may want to update this. This updated photo should automatically show on the areas where a photo would populate (nav bar and edit photos). We can do this in the photo-editor component by adding a check in teh onSuccessItem method:

```
        if (photo.isMain && this.user && this.member) {
        this.user.photoUrl = photo.url;
        this.member.photoUrl = photo.url;
```

```
this.accountService.setCurrentUser(this.user);
}
```

So if it's the only photo it will be set to main, we have a user and member, so we can set the photoUrl to the uploaded photo, same with the member photoUrl. Then we can update the user observable with setCurrentUser. This should show the photo immediately.

Section 13: Pagination

Pagination in the API

Pagination is a great way to minimize the amount of work the server and API does by sending just a small amount of the data at a time, and also restricting the user to only a certain amount of data at a time. To start, we'll build pagination in the API

In the "Helpers" folder we'll created:

PagedLists.cs
PaginationHeaders.cs
UserParams.cs

and in the extensions folder we'll create an extension for

HttpExtensions

The PagedLists.cs file will make use of generics <T> and derive from List<T> so we can make use of list methods. The PagedList object itself will take in an IEnumerable of items, a count of items, a pageNumber, and a pageSize. The items will be added to the PagedList with the "AddRange(items)" method in the ctor.

The PagedList object will also have a static method that creates the PagedList object by taking in an IQueryable<T> (which we'll call "source"), an int for the pageNumber, and an int for the pageSize.

A count will be obtained by counting the IQueryable (source.CountAsync())
The items we want to return will be taken from the source using "Skip" and "Take":

```
var items = await source.Skip((pageNumber - 1) *
        pageSize).Take(pageSize).ToListAsync();
```

Skip: calculates the number of entries to skip in the source by calculating what page number we're on multiplied by the page size
Take: takes the number of entries specified in the pageSize var

The method then returns the PagedList<T> object with items, count, pageNumber, and pageSize

We'll return to this later…

In the PaginationHeaders.cs file, simply want to create the headers for pagination so we can pass the values back and forth in the UsersController in the API

```
public PaginationHeader(int currentPage, int itemsPerPage, int totalItems,
int totalPages)
        {
            CurrentPage = currentPage;
            ItemsPerPage = itemsPerPage;
            TotalItems = totalItems;
            TotalPages = totalPages;
        }
```

Finally, we'll look at the UserParams.cs file, where we want to set the defaults for pagination and create a method to handle custom page sizes:

```
private const int MaxPageSize = 50;
public int PageNumber { get; set; } = 1;
private int _pageSize = 10;
public int PageSize
{
        get => _pageSize;
        set => _pageSize = (value > MaxPageSize) ? MaxPageSize : value;
}
```

For the PageSize property, we can get it as normal, but we can set it with a ternary operation, where the user passes a "value" and that value is compared to the MaxPageSize.

For the last file, HttpExtensions.cs, we want to create the headers that the client side can understand in JSON format.

Because this is an extension we'll make it static. We are extending the **HttpResponse** class, so we must include that in the method as the first argument:

```
public static void AddPaginationHeader(this HttpResponse response,
PaginationHeader header){
```

We'll also take in a PaginationHeader object as "header". Then in the method we'll convert it to JSON camelCase and add it to the response.Headers:

```
var jsonOptions = new JsonSerializerOptions{PropertyNamingPolicy =
        JsonNamingPolicy.CamelCase};
response.Headers.Add("Pagination", JsonSerializer.Serialize(header, jsonOptions));
```

Finally, we'll add the following line:

response.Headers.Add("Access-Control-Expose-Headers", "Pagination");

The client expects the above, and it will come into play when we add pagination to the Angular side

Implementation

We'll modify the IUserRepository method "GetMembersAsync" to pass a PagedList instead of the current IEnumerable:

Task<PagedList<MemberDTO>> GetMembersAsync(UserParams userParams);

Note the "userParams" passed in as well.

In the UserRepository.cs file, we'll modify the above method as well to reflect the change in the Interface. We'll create a var "query" that we get by accessing the context.Users db, project it to a MemberDTO using AutoMapper as usual. We'll add the method .AsNoTracking() which will help with efficiency because we don't want to track any changes yet.

We can now return the PagedList through the method CreateAsync, passing in the "query" as the items, userParams.PageNumbers, and userParams.PageSize. This will in turn create the PagedList object and return it…

1.

```
            return await PagedList<MemberDTO>.CreateAsync(query,
userParams.PageNumber, userParams.PageSize);
```

2.

```
public static async Task<PagedList<T>> CreateAsync(IQueryable<T> source,
            int pageNumber, int pageSize)
        {
            var count = await source.CountAsync();
            var items = await source.Skip((pageNumber - 1) *
pageSize).Take(pageSize).ToListAsync();
            return new PagedList<T>(items, count, pageNumber, pageSize);
        }
```

3.

```
public PagedList(IEnumerable<T> items, int count, int pageNumber, int
pageSize)
        {
            CurrentPage = pageNumber;
```

```
            TotalPages = (int) Math.Ceiling(count / (double) pageSize); //
round up
            PageSize = pageSize;
            TotalCount = count;
            AddRange(items);
        }
```

The Controller

In the UsersController, we'll change the GetUsers method to reflect our recent changes by first returning a PagedList instead of an IEnumerable. We'll also pass in the userParams, from the query string. We tell the method this by adding it in brackets:

```
public async Task<ActionResult<PagedList<MemberDTO>>>
GetUsers([FromQuery]UserParams userParams)
```

Now we can get just a set of users that we want to return

```
var users = await _userRepository.GetMembersAsync(userParams);
```

The response will now contain the headers we need, created with the new AddPaginationHeaders method:

```
Response.AddPaginationHeader(new PaginationHeader(users.CurrentPage,
        users.PageSize, users.TotalCount, users.TotalPages));
```

This makes use of our new extension method, which takes in a new PaginationHeader object. The "users" var we created contain our results from the modified GetMembersAsync method in the UserRepository class, which is a PagedList, containing all the necessary properties to create a paged list (CurrentPage, PageSize, TotalCount, and TotalPages)

Return of "users" will return the selected list of users, as well as the headers specified.

Testing

In Postman, sending {{url}}/api/users?pageNumber=2&pageSize=3 will return a selected section of users based on the query string. The body will contain just the users, while the headers tab will contain the header parameters, including the Pagination parameters.

Front-end Pagination

The four properties in the PaginationHeaders.cs file need to be given an interface in the client-side of things so that Angular can recognize what they are and use them appropriately. We'll make a file pagination.ts and export the Pagination interface with these four properties:

```
export interface Pagination {
        currentPage: number;
        itemsPerPage: number;
        totalItems: number;
        totalPages: number;
}
```

The names of the properties must match the names in the PaginationHeader.cs file. These values will be populated from the values passed through in the API, specifically in the response from the controller (see above).

We'll also export a class to make use of the above properties as well as a generic list of the members we get back from the response:

```
export class PaginatedResult<T> {
        result?: T;
        pagination?: Pagination;
        }
```

The Service Implementation

In this service we want a new property to use, a property of

```
paginatedResult: PaginatedResult<Member[]> = new PaginatedResult<Member[]>;
```

which will make use of the exported class from the pagination.ts file.

In the membersService.ts file we'll modify the getMembers() method by adding some parameters to pass in:

```
(page?: number, itemsPerPage?: number)
```

which we'll be getting from the query string. We'll use a variable "params" which is an object of HttpParams(), which allows us to set query string parameters along with the Http request. We'll append the parameters we got to the params variable:

```
params = params.append('pageNumber', page);
params = params.append('pageSize', itemsPerPage);
```

Now we'll return the http.get<Member[]> array just like before, but we'll add in a property of "observe" so that we can observe the response and check the params that are passed in (the page number and the number of items to display on that page).

```
return this.http.get<Member[]>(this.baseUrl + 'users', {observe: 'response',
params}).pipe(
```

Just like previously we want to map the response

```
map(response => {
        if (response.body) {
        this.paginatedResult.result = response.body;
        }
```

The above maps the members array to the “result” property of the class, which is a generic. We then map the headers to the the same object, paginatedResult:

```
const pagination = response.headers.get('Pagination');
        if (pagination) {
        this.paginatedResult.pagination = JSON.parse(pagination);
        }
```

Now that our paginatedResult object has both headers and a body of members (as “T”) we can return it.

The Component

The component currently uses an observable<Member[]> of members$ but we can’t use that yet. We’ll fix that later, but for now, to test, we’ll use a simple Member[] array of “members”. We’ll also add two properties, “pageNumber = 1” and “pageSize = 5”. Finally, we’ll add a pagination of type “Pagination” from the exported class.

In the loadMembers() we’ll call the getMembers from our memberService, passing in the pageNumber and pageSize values. The response we receive from the getMembers method will have a body (the members in an array) and a header (the values for the page parameters):

```
this.memberService.getMembers(this.pageNumber, this.pageSize).subscribe({
        next: response => {
                if (response.result && response.pagination) {
                        this.members = response.result;
                        this.pagination = response.pagination;
```

So when a user clicks on the “members” link, the component will call the getMembers method in the memberService, which will then call the API at the listed url. The memberService getMembers() will “watch” for the params in the ‘response’ and return it as the pagination property in the paginatedResult object, with the result property of the paginatedResult being the list of members to return.

NOTE: currently, the “pagination” property does nothing, we will use it in the next section

Pagination Interaction

Now it's time to add a way for users to change which page they are seeing, so we'll start with Angular Bootstrap and get a pagination component to add in

Reading the documentation, we'll first need to add the import to the shared.module.ts file

```
import { PaginationModule } from 'ngx-bootstrap/pagination';
[imports]:      PaginationModule.forRoot()
[exports]:      PaginationModule
```

In the html template for the member-list component, we'll add a row of buttons for pagination. Since we are using a Pagination object we called "pagination" we'll check for that first

```
*ngIf="pagination"
```

Then we can build the button row

```
<pagination
        [boundaryLinks]="true"
        [totalItems]="pagination.totalItems"
        [itemsPerPage]="pagination.itemsPerPage"
        [maxSize]="10"
        [(ngModel)]="pagination.currentPage"
        (pageChanged)="pageChanged($event)"
        previousText="&lsaquo;"
        nextText="&rsaquo;"
        firstText="&laquo;"
        lastText="&raquo;">

</pagination>
```

Explanation:

"Boundary links" show buttons to go to the end or beginning
"totalItems" shows the total amount of items we have from the db
"itemsPerPage" tells how many things we want to show on a page max
"maxSize" is for the number of number links to show in the bar
"ngModel" is our two-way binding to the currentPage property
"(pageChanged)" is an event emitter that we'll use to call the pageChanged method

The last few items are standard and we don't change those for this model

In the component.ts file we'll create the pageChanged method so that when the event happens (when the page is changed) we can tell it what to do.

```
pageChanged(event: any){
        if (this.pageNumber !== event.page) {
        this.pageNumber = event.page;
        this.loadMembers();
```

```
        }
    }
```

For this method we'll take in the event of type "any". The event will have a "page" property which we'll compare to the current page first. Then, provided they are not the same, we'll change the pageNumber property in our component to match the clicked page number from the button row. Then we'll reload the members.

Adding Filtering to the API

Adding username and gender filtering

Because the userParams object is being passed around, we'll add two extra params to that to start:

```
public string? CurrentUsername { get; set; }
public string? Gender { get; set; }
```

NOTE: nullable issue

Add the '?' to the above to ensure there are no errors when the API is called to get the users, or else it may throw a 400 error where the params are needed for the return

In the UsersController we'll still return the users as usual, but we'll use the userParams object to pass some info about the logged in user, using the above added properties.

To start, we'll get the current user

```
var currentUser = await
_userRepository.GetUserByUsernameAsync(User.GetUsername());
userParams.CurrentUsername = currentUser.UserName;
```

Then we'll get the gender of the current user, and then set the userParams gender property to the opposite:

```
if (string.IsNullOrEmpty(userParams.Gender))
 {
   userParams.Gender = currentUser.Gender == "male" ? "female" : "male";
  }
```

Finally, in the UserRepository we'll modify our query in the GetMembersAsync method, using the userParams we pass in. Start by making a variable that we can use queries on:

```
var query = _context.Users.AsQueryable();
```

Then we'll add queries to the "query" var:

```
query = query.Where(u => u.UserName != userParams.CurrentUsername);
```

```
query = query.Where(u => u.Gender == userParams.Gender);
```

In the return, we'll modify what we have (the 'query' var) to do our mapping and not track (basically what we are already doing but now we're doing it in the return instead)

```
return await PagedList<MemberDTO>.CreateAsync(
        query.AsNoTracking().ProjectTo<MemberDTO>(_mapper.ConfigurationProvider),
        userParams.PageNumber,
        userParams.PageSize);
```

Test with Postman. All opposite gender users should be returned when no query is passed in. All queries should return only users that are not the current user.

Adding Age Filtering

Once again we'll add new properties to the UserParams.cs file

```
public int MinAge { get; set; } = 18;
public int MaxAge { get; set; } = 100;
```

NOTE: Setting the values negates the need to add a 'null' option

Since the UserRepository.cs file is where our queries are being sent from, we'll modify the getMembers method to calculate the desired DOB that we want to return, based on the input from the query string.

```
var minDob = (DateTime.Today.AddYears(-userParams.MaxAge - 1));
var maxDob = (DateTime.Today.AddYears(-userParams.MinAge));
```

minDob refers to the lowest date of birth, or the oldest age we'll accept
maxDob refers to the most recent date of birth, or the youngest age we'll accept

Then we can just query it like the others, adding it to our "AsQueryable()" object "query":

```
query = query.Where(u => u.DateOfBirth >= minDob && u.DateOfBirth <= maxDob);
```

When the userParams are passed to the controller via the UserController "GetUsers" method, the query is mapped to the UserParams object and passed around as "userParams".

Adding Properties to the DTO

The original goal was to display the user's "knownAs" property in the "welcome" message, so we'll add some properties to the DTO to pass that up to the client.

Starting in the DTO we'll add:

```
public string KnownAs { get; set; }
```

```
public string Gender { get; set; }
```
Now we can assign these in the accountController.cs in the Login and Register methods

In the “user” interface (user.ts) we must add these two properties as strings so they are available in the components that use this “User” object. Then in the components we can use “user.knownAs” in place of “user.username”.

Filtering in the Client

Similar to what we did in the API, we’ll have a file that we’ll call “userParams” that will contain the minAge, maxAge, pageNumber, and pageSize properties, and we’ll pass that through the members.service.ts methods to return the desired filtered results.

We’ll make the UserParams a class instead of an interface, so that we can use a constructor within it to initialize some of our values (in this case, ‘gender’). To do this we’ll follow the same logic we did in the API version of this:

```
this.gender = user.gender === 'female' ? 'male' : 'female';
```

The getMembers method will now switch from

```
getMembers(page?: number, itemsPerPage?: number)
```

to

```
getMembers(userParams: UserParams)
```

For the HttpParams that we construct in this method, we can now call userParams.pageNumber and userParams.pageSize. This section of code can also be placed in it’s own method to clean up the code:

```
private getPaginationHeaders(pageNumber: number, pageSize: number) {
    let params = new HttpParams();

    params = params.append('pageNumber', pageNumber);
    params = params.append('pageSize', pageSize);

    return params;
 }
```

Also in the getMembers method, we have another set of code that can be extracted.

NOTE: Using generics in methods

Currently we declare a property of paginatedResults as a type of PaginatedResult<Member[]>. However, we can remove this from the top of the MemberService class and instead make the method accept a generic type that can be passed in where necessary.

Because we still have a Members[] array, that will be what we pass as the generic type in the method call:

```
this.getPaginatedResults<Member[]>(this.baseUrl + 'users', params);
```

The above method call can be made without passing in a <generic> object as well, although it won't work as the method relies on having the ability to get something from this object. However, we can potentially pass any generic object and, provided it has the necessary properties, the method will work. We just need to modify the method to instantiate the passed generic as a PaginatedResult:

```
private getPaginatedResults<T>(url: string, params: HttpParams) {
    const paginatedResult: PaginatedResult<T> = new PaginatedResult<T>;
```

Editing the Params

Once we have the params returned to us with the correct headers, we can add the params from the userParams variable:

```
let params = this.getPaginationHeaders(userParams.pageNumber,
userParams.pageSize);
    params = params.append('minAge', userParams.minAge);
    params = params.append('maxAge', userParams.maxAge);
    params = params.append('gender', userParams.gender);
```

We'll return the paginatedResults as before:

```
return this.getPaginatedResults<Member[]>(this.baseUrl + 'users', params);
```

Remember, 'params' are HttpParams() which is a request/response body

Adding the Filtering to the Component

We'll need to have access to the AccountService so that we can subscribe to the currentUser$ observable, and we'll pull the user and use their properties to populate the userParams properties that match (in this case, 'gender' so far).

```
this.accountService.currentUser$.pipe(take(1)).subscribe({
    next: user => {
    if (user) {
    this.userParams = new UserParams(user);
    this.user = user;
```

Now that we have a userParams property in our class we can use it in place of "pageNumber" in the pageChanged method:

```
pageChanged(event: any){
    if (this.userParams && this.userParams?.pageNumber !== event.page) {
    this.userParams.pageNumber = event.page;
```

```
this.loadMembers();
```

The highlighted portion above is meant to check that the userParams exists so that no errors are thrown.

The loadMembers method can now be modified to pass the userParams to the memberService.getMembers method:

```
if (!this.userParams) return;
    this.memberService.getMembers(this.userParams).subscribe({
```

Again, we must check that the userParams exists / has value before calling the method.

Now, the getMembers is called with the newly created userParams object that contains the user's gender, as well as the minAge, maxAge, pageSize, and pageNumber properties.

Adding Filter Controls

Users will be able to select which gender they want to see, so we'll need a list of the genders available in the member-list component

```
genderList = [{value: 'male', display: 'Males'}, {value: 'female', display: "Females"}];
```

We also want to be able to reset the filters, so we'll make a method that does that when called from the form:

```
resetFilters(){
    if (this.user) {
    this.userParams = new UserParams(this.user);
    this.loadMembers();
```

This method first checks that we have a user, then sends that user up to make new userParams, and then runs the loadMembers method with the new userParams

HTML Filter Form

We have ageMin, ageMax, and gender to filter by, so each will get their own input / list to select. This will be a form and will only display if we have a "userParams" object available:

```
<div class="container mt-3" *ngIf="userParams">
    <form #form="ngForm" class="d-flex mb-3" (ngSubmit)="loadMembers()"
    autocomplete="off">
```

NOTE: d-flex

D-flex ensures that all items in this form are in-line

When the submit button is pressed, it will run loadMembers() with the given parameters from the form, courtesy of two-way binding:

```
<div class="d-flex mx-2">
    <label class="col-form-label">Age from: </label>
    <input
        type="number"
        class="form-control ms-1"
        style="width: 70px"
        name="minAge"
        [(ngModel)]="userParams.minAge">
</div>
```

Do the same as above for the maxAge property

Select List

For gender, we need to call the list we made in the component file, called genderList

```
<select name="gender"
    style="width: 130px"
    class="form-select ms-1"
    [(ngModel)]="userParams.gender">
        <option *ngFor="let gender of genderList" [value]="gender.value">
        {{gender.display}}
        </option>
</select>
```

“Option” uses a for list to give us the display names in a dropdown list and the value of the dropdown item.

Finally, we’ll add two buttons, one for submitting and the other for resetting the form

```
<button class="btn btn-primary ms-1" type="submit">Apply Filters</button>
<button (click)="resetFilters()" class="btn btn-info ms-1" type="submit">Reset
Filters</button>
```

When the “apply filters” button is pressed it calls the form’s submit method which we told to run the loadMembers method. The other button, “reset”, will call the resetFilters method in the component

Sorting Users

Sorting on the API

Starting in the UserParams.cs file, we'll add another property we call "OrderBy", which will be a string and we'll set its default value to "lastActive", one of the properties of a member / user.

In the UserRepository.cs file, in the GetMembersAsync method, we can add another query in the form of a switch statement (in case we want to add more options later on).

```
query = userParams.OrderBy switch
        {
                "created" => query.OrderByDescending(u => u.Created),
                _ => query.OrderByDescending(u => u.LastActive)
        };
```

"OrderByDescending" does exactly what it says based on the parameter passed in the parenthesis. The "_" stands in for the default value.

Adding an Action Filter

Updating the LastActive Property

Middleware can technically do this, but it will act on every request that passes through to the API. An Action Filter can be more selective about what we apply the action to.

First let's make a helper class called LogUserActivity. This will derive from "IAsyncActionFilter" and give us access to ActionExecutingContext (for when we want to perform an action before / during something) and ActionExecutionDelegate (for when we want to perform an action after something).

In this case we'll use the ActionExecutionDelegate, "next()". We'll use this in a var called resultContext. The resultContext has access to the HttpContext object, which gives us access to the User.Idenity.IsAuthenticated method, so we can access that to check that we have an authenticated user logged in:

```
if (!resultContext.HttpContext.User.Identity.IsAuthenticated) return;
```

Then we'll get the username of the user

```
var username = resultContext.HttpContext.User.GetUsername();
```

NOTE: GetUsername

This is an extension method that we made to extend the ClaimsPrincipal class. It takes in a user that is registered in System.Security.Claims and returns the NameIdentifier property.

Now that we have the username we can access the user in the db through the UserRepository, change their LastActive property, then SaveAllAsync to update the db:

```
var repo = resultContext
        .HttpContext
        .RequestServices
        .GetRequiredService<IUserRepository>();
var user = await repo.GetUserByUsernameAsync(username);
user.LastActive = DateTime.UtcNow;
await repo.SaveAllAsync();
```

Now that that's built we need to tell the application about it, so in the ApplicationServiceExtensions we'll add:

```
services.AddScoped<LogUserActivity>();
```

We *could* use this newly created action inside the UserController, but since the action is already checking if we're authenticated we can simply use it in the BaseAPI controller. That way we only need to specify it in one place. At the top of the BaseAPIController we'll add

```
[ServiceFilter(typeof(LogUserActivity))]
```

Testing this in Postman requires logging in as a user, and then getting that user. When we do that we can see a large query being sent up just to make this minor update, so next we'll look how we can improve that efficiency.

Making The ActionFilter More Optimal

Currently the ActionFilter is calling the GetUserBayUsernameAsync method from the UserRepository.cs class. This method returns the user, but also joins the user db with the photos db, causing extraneous data to be returned and resulting in a larger-than-necessary query. A more ideal scenario uses the GetUserByIdAsync method, which just returns the user. To use this we'll need to add the user id property to our token, and modify the ClaimsPrincipleExtension to offer a user id as a return value as well.

In the TokenService.cs we currently create a token that contains just the username as "NameId". We'll modify that to pass the Id instead, while still passing the username as well:

```
var claims = new List<Claim>
        {  new Claim(JwtRegisteredClaimNames.NameId, user.Id.ToString()),
           new Claim(JwtRegisteredClaimNames.UniqueName, user.UserName)
```

Now our token will contain the user id as well

Previous Payload	New Payload
{	{

"nameid": "marion", "nbf": 1688257177, "exp": 1688861977, "iat": 1688257177 }	"nameid": "1", "unique_name": "marion", "nbf": 1688313651, "exp": 1688918451, "iat": 1688313651 }

In order to avoid refactoring existing code, we'll adjust the ClaimsPrinciplesExtensions methods to offer up the id in addition to the name. In "Claims", "Name" is the username, and "NameIdentifier" is the id, so we'll modify the two methods to reflect this:

```
public static string GetUsername(this ClaimsPrincipal user)
        {
            return user.FindFirst(ClaimTypes.Name)?.Value;
        }

        public static string GetUserId(this ClaimsPrincipal user)
        {
            return user.FindFirst(ClaimTypes.NameIdentifier)?.Value;
        }
```

Finally, we modify the LogUserActivity.cs to get the user id instead of the username:

```
var userId = resultContext.HttpContext.User.GetUserId();
```

Because this returns a string, but we need to send up an int, we'll parse it:

```
var user = await repo.GetUserByIdAsync(int.Parse(userId));
```

Completed:

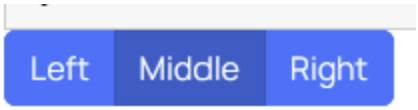
```
var userId = resultContext.HttpContext.User.GetUserId();

            var repo =
resultContext.HttpContext.RequestServices.GetRequiredService<IUserReposito
ry>();
            var user = await repo.GetUserByIdAsync(int.Parse(userId));
            user.LastActive = DateTime.UtcNow;
            await repo.SaveAllAsync();
```

NOTE: Querying

The above fix does not reduce *all* traffic, as when a user is retrieved initially there is still a join from the photo db. This fix merely removes a *second* join in the second query.

Adding Filtering Buttons to the Client

We'll make a set of buttons that are essentially radio buttons for filtering



To use the orderBy functionality we need to add an orderBy property to the userParams.ts file

We'll need to import an ngx library first, into the shared.module.ts file:

```
import { ButtonsModule } from 'ngx-bootstrap/buttons';
Import = ButtonsModule.forRoot()
Export = ButtonModule
```

In the member-list component html file we'll add the buttons to show in-line with the other filters. Each button will be given a type of "button" so they don't confuse the form, and name of "orderBy" because we'll be using two-way binding. The ngModel will be set to userParams.orderBy. The btnRadio is set to the value want to pass when the button is clicked, and then the (click) event will call "loadMembers()".

```
<button
    type="button"
    class="btn btn-primary"
    name="orderBy"
    [(ngModel)]="userParams.orderBy"
    btnRadio="lastActive"
    (click)="loadMembers()">
    Last Active
</button>
```

Finally we need a way to work with this in the members.service file, so where we have getMembers working with the userParams, we'll add one more "params" (since technically this is being passed as a query string):

```
params = params.append('orderBy', userParams.orderBy);
```

Fixing the Time Display

Angular allows '|' (pipes) to modify parameters, so in the member-detail component where we have

```
<p>{{member.lastActive}}</p>
```

We can change this to

```
<p>{{member.lastActive | date }}</p>
```

Or even with modifiers

```
<p>{{member.lastActive | date: "longDate" }}</p>
```

However, there is an ngx module that we can use that will show a count-up timer instead, to give real-time values to the amount of time since a user was active. We can find it here.

Install using npm install ngx-timeago@X –legacy-peer-deps
Note the correct version is installed for the working version of Angular.

Add the following to the shared.module:

```
import { TimeagoModule } from "ngx-timeago";
TimeagoModule.forRoot() (import)
TimeagoModule (export)
```

NOTE: Error Encountered

The first time attempting to use this there was a major error that broke several other files, including an html file. If the error is encountered, discard git changes and start again. Ensure git is being updated regularly. Ensure to add the import at the top of the shared.module file.

Now we can add “timeago” to the html for the date parameters:

```
<p>{{member.lastActive | timeago }}</p>
```

You may notice that the time is funky, either showing too long ago or, weirder, in the future. This is because the time in the server is set to UTC, but the browser is looking for local time. UTC time is typically passed with a ‘Z’ at the end to indicate it is in fact UTC, and so we can add a ‘Z’ to the end of the time through concatenation.

```
<p>{{member.lastActive + 'Z' | timeago }}</p>
```

Now we can see that the time is updated accordingly

Restoring Caching for Members

Because of our pagination, we no longer pull from the cache when we query the database. However, each pagination object userParams that we pass around the client has a unique key, so if we can access this key then we can store *this* in the cache, access it when needed, and display that without making db calls.

Start in the memberService.ts…

We can see the contents of the userParams object by using

```
console.log(Object.values(userParams).join('-'));
```

In the console we’ll see something like this:

```
18-99-1-5-lastActive-female
```

What we can do is use this as a key by using the javascript object map function. This gives us access to ‘get’ and ‘set’.
Since the getMembers method makes use of the userParams to filter, we’ll use that method to set the key value for the new property “memberCache = new Map();”

Start by storing the userParams object in a response:

```
const response = this.memberCache.get(Object.values(userParams).join('-'));
```

Then check that the response has something in it. If so, then just return without sending up a new query:

```
if (response) return of(response);
```

Where we were returning the paginatedResults, we can add the ‘pipe’ so we can set new key-value pairs in the memberCache property

```
return this.getPaginatedResults<Member[]>(this.baseUrl + 'users', params).pipe(
    map(response => {
    this.memberCache.set(Object.values(userParams).join('-'), response);
    return response;
```

NOTE: the response

The key-value pair in the memberCache property is that the key is the joined userParams, and the value is the paginated result. The line “const response = …” looks for the key in the memberCache property, if it exists, then the value is set as the response. Then the response is returned.

Restoring Caching for Member Details

The memberCache contains the user objects, but they are deep deep deep inside the object. We can get the individual users out of this object though using the “reduce” function and the spread operator (...).

In the getMember method in the memberService.ts file, we’ll add

```
const member = [...this.memberCache.values()]
    .reduce((arr, elem) => arr.concat(elem.result), []);
```

Here we’ll make the “member” object from taking the memberCache values and spreading them out, then reducing it down to individual objects of members. The ‘arr’ is the array, and the ‘elem’ is the element within the original object. This is kind of like a for() loop that concatenates the resulting objects into an array.

Now, only the member objects are stuffed into the array. We can view it in the console with

```
console.log(member)
```

Since member is an array

Now that we have an array of users we can use array methods on it, like ‘find’.

```
const member = [...this.memberCache.values()]
    .reduce((arr, elem) => arr.concat(elem.result), [])
    .find((member: Member) => member.userName === username);
```

Then we can check if the member exists in the array and return it if it does:

```
if (member) return of(member);
```

Remembering Filters

If we are looking at a user and go back to ‘matches’ we lose the previous query and it resets the filters. What we would ideally like is that the previous query stays in existence even if the component gets destroyed through navigation.

In the member-list-component we are setting and storing the userParams, so we can do the same thing in the memberService.

We’ll start by injecting the accountService into the memberService, which is totally okay so long as no circular references are made. In the member service we’ll now take care of the userParams property in the constructor:

```
this.accountService.currentUser$.pipe(take(1)).subscribe({
    next: user => {
        if (user) {
            this.userParams = new UserParams(user);
            this.user = user;
    }
```

The above was moved from the member-list component.ts file, where it is no longer needed. The accountService in that file is also not needed any longer.

Back in memberService we’ll add some helper methods:

```
getUserParams() {
    return this.userParams;
  }


  setUserParams(params: UserParams){
    this.userParams = params;
  }

  resetUserParams() {
    if (this.user) {
```

```
      this.userParams = new UserParams(this.user);
      return this.userParams;
    }

    return;
  }
```

Now in the member list component we can call these methods to get the user params and reset them.

The constructor has
	this.userParams = this.memberService.getUserParams();

loadMembers now has
	loadMembers() {
		if (this.userParams) {
		this.memberService.setUserParams(this.userParams);

The resetFilters method now is:
	resetFilters() {
		this.userParams = this.memberService.resetUserParams();
		this.loadMembers();

Section 14: Adding the Like User Feature

Adding a “Likes” Entity

For this section we want to add a new entity, the “UserLike”, which will hold the id’s of users that like another user and the id’s of the users that are liked by those users. The AppUser will also be stored for each id, so this entity will contain:
	public AppUser SourceUser { get; set; }
	public int SourceUserId { get; set; }
	public AppUser TargetUser { get; set; }
	public int TargetUserId { get; set; }

The AppUser entity will also contain two more lists, one of LikedByUsers and another of LikedUsers. Each list will be of the UserLike entity.
	public List<UserLike> LikedByUsers { get; set; }
	public List<UserLike> LikedUsers { get; set; }

Entity framework *does* have the ability to construct the tables based off of these conventions, but the resulting tables and foreign keys are weirdly named and a little disjointed. We'll create the tables ourselves in the DataContext class instead.

To do this we'll override the method "OnModelCreating" in the DbContext class. We'll add this method to the DataContext.cs file.

```
protected override void OnModelCreating(ModelBuilder builder)
```

It will need the following line to start:

```
base.OnModelCreating(builder);
```

Then we'll build the table, which is essentially a join table:

```
builder.Entity<UserLike>()
        .HasKey(k => new { k.SourceUserId, k.TargetUserId });
```

And then we'll build the association between SourceUser and TargetUser with two more parameters about the table:

1	2
builder.Entity<UserLike>() .HasOne(s => s.SourceUser) .WithMany(l => l.LikedUsers) .HasForeignKey(s => s.SourceUserId) .OnDelete(DeleteBehavior.Cascade);	builder.Entity<UserLike>() .HasOne(s => s.TargetUser) .WithMany(l => l.LikedByUsers) .HasForeignKey(s => s.TargetUserId) .OnDelete(DeleteBehavior.NoAction);

With the above in place we can add a new migration (dotnet ef migrations add AddUserLikes) When we run, we will see a new table created, the UserLike table.

Adding a Likes Repository

We want to be able to shape the data for use on the client, so we'll start by making a DTO for Likes, a "LikeDTO" that will contain the properties we'll want displayed in a card for the user. This DTO will be for users that the user has liked, or for users that have liked the user. It will be interchangeable, and will contain properties for: ID, UserName, Age, KnownAs, PhotoUrl, and City.

Interface

To build the repository we'll follow the same procedures as we did previously, by creating an interface for the repo. This interface will start with three methods that all return Tasks.

To get the user likes generally:

```
Task<UserLike> GetUserLikes(int sourceUserId, int targetUserId);
```

To get an AppUser with their “likes” lists:

```
Task<AppUser> GetUserWithLikes(int userId);
```

And to get a list of UserLikes but with an option to see either 1: the users that a user has liked or 2: the users that have liked the user (passed as “predicate”):

```
Task<IEnumerable<LikeDTO>> GetUserLikes(string predicate, int userId);
```

The third method above will likely be our most used for use with the front end

Finally, we’ll add the service to the ApplicationServiceExtensions class as

```
services.AddScoped<ILikesRepository, LikesRepository>();
```

Easy.

Implementing the Likes Repository

Because a UserLikes entity has technically two keys, we can search for a specific UserLike by passing in BOTH keys:

```
return await __context.Likes.FindAsync(sourceUserId, targetUserId);
```

The above is in the GetUserLikes() method in the repo

For the next method, “GetUserLikes”, we’ll use a predicate to determine which set of users the user would like to see: those that they liked, or those that liked them.

To start we’ll build up a query

NOTE: AsQueryable()

Using this method does not send out to the database yet. Instead, this is a way to build our query without accessing the database, as provided in System.Linq

```
var users = _context.Users.OrderBy(u => u.UserName).AsQueryable();
var likes = _context.Likes.AsQueryable();
```

If users want who they liked:	If users want who liked them:

if (predicate == "liked")
{
likes = likes.Where(like => like.SourceUserId == userId);
users = likes.Select(like =>
	if (predicate == "likedBy")
	{
	likes = likes.Where(like => like.TargetUserId == userId);
	users = likes.Select(like =>

like.TargetUser); }	like.SourceUser); }

Now we'll hit the database with the built up query, and return the LikeDTO with properties mapped (by hand):

```
return await users.Select(user => new LikeDTO
    {
        UserName = user.UserName,
        KnownAs = user.KnownAs,
        Age = user.DateOfBirth.CalculateAge(),
        PhotoUrl = user.Photos.FirstOrDefault(x => x.IsMain).Url,
        City = user.City,
        Id = user.Id
    }).ToListAsync();
```

For the last method in the repo, we'll just return a single user with all of their likes:

```
return await _context.Users
    .Include(x => x.LikedUsers)
    .FirstOrDefaultAsync(x => x.Id == userId);
```

The Controller for Likes

Let's make a new controller!
LikesController.cs, and it will derive from BaseAPIController so that it can be found at /api/likes

Import the userRepository and the likesRepository so we can use both

We'll make an [HttpPost("{username"}] method where the username is passed in through the address:

/api/likes/angelina

This method creates a new entry in the join table, but is not creating a new user or updating a user, so we won't be returning anything other than statuses

Steps:

Get the id of the currently logged in user and parse it to an int
Find the user that we're trying to like
Get the user and their likes from the likesRepository

```
var sourceUserId = int.Parse(User.GetUserId());
var likedUser = await _userRepository.GetUserByUsernameAsync(username);
var sourceUser = await _likesRepository.GetUserWithLikes(sourceUserId);
```

Next: Error checking

See if the username even exists

Check that the user is not trying to like themselves

```
if (likedUser == null) return NotFound("User not found");
if (sourceUser.UserName == username) return BadRequest("You cannot like yourself");
```

Next: Check that the user hasn't already liked this user

```
var userLike = await _likesRepository.GetUserLikes(sourceUserId, likedUser.Id);
if (userLike != null) return BadRequest("You already like this user");
```

Finally: create a new UserLike, add it to the LikedUsers list in the users AppUser object, save the db, return Ok();

```
userLike = new UserLike
{
        SourceUserId = sourceUserId,
        TargetUserId = likedUser.Id
};
sourceUser.LikedUsers.Add(userLike);
if (await _userRepository.SaveAllAsync()) return Ok();
```

To get the users that the logged in user has liked, we'll use an [HttpGet] method that returns a list of likeDTO's and takes in the predicate as a string

```
var users = await _likesRepository.GetUserLikes(predicate, int.Parse(User.GetUserId()));
return Ok(users);
```

Likes Function in Angular

We already have a button for "likes" on the user cards, so we'll add a method to call when it is clicked. In the members.service.ts file we'll add two methods:

```
 addLike(username: string) {
        return this.http.post(this.baseUrl + 'likes/' + username, {});
  }
```

And

```
getLikes(predicate: string) {
        return this.http.get(this.baseUrl + 'likes?predicate=' + predicate);
}
```

Since the like button is in the card, we need to add a link to the memberService in the card-component.ts. Inject the MemberService, as well as the ToastrService. The method “addLike” will handle the like request from the card button:

```
addLike(member: Member) {
    this.memberService.addLike(member.userName).subscribe({
    next: () => this.toastr.success('You have liked ' + member.knownAs)
```

The member is the member who’s like button was clicked. Pass their username to the addLike method in the memberService. Send a toast message to confirm the like was successful.

In the card itself, give the button a (click) event to call the addLike method, and pass the member object to the method.

Adding the Likes Component

We want to be able to list both members we’ve liked and members who’ve liked us. The controller in the API for getUserLikes returns an array (IEnumerable) of users who’ve either liked me, or who’ve I’ve liked. We can work with this in the client, specifically if we receive it as an array and use it to populate our member-cards component.

In “Lists” we’ll make sure we have a members array set to undefined and a predicate set to ‘liked’ as the default. Our constructor needs to import the MemberService as well. Then we can make a method to “loadLikes”:

```
this.memberService.getLikes(this.predicate).subscribe({
    next: response => {
    this.members = response
```

We’ll call this in the ngOnInit, so that we go to the memberService and call the API, get back the IEnumerable, and then respond back to “lists” to populate the members[] array.

NOTE: MemberService getLikes

The http.get currently does nothing with the response from the API, so we’ll modify that slightly so it has something to store the gotten IEnumerable in:

```
return this.http.get<Member[]>(this.baseUrl + 'likes?predicate=' + predicate);
```

Now that we’re getting a response, we can use binding in the html template to populate the member-cards that we’ll display. We can make radio buttons to switch between “liked” and “likes” on the fly:

liked	likedBy
<button class="btn btn-primary"	<button class="btn btn-primary"

btnRadio="liked" [(ngModel)]="predicate" (click)="loadLikes()">Members I like</button>	btnRadio="likedBy" [(ngModel)]="predicate" (click)="loadLikes()">Members who like me</button>

We used “liked” and “likedBy” in the repository, so when a specific button is clicked the btnRadio value is sent as the new value of “predicate”, which is passed to the method “loadLikes()”

Then below that we can load all the cards that match the predicate stipulation:

```
<div class="row mt-3">
    <div class="col-2" *ngFor="let member of members">
        <app-member-card [member]="member"></app-member-card>
    </div>
</div>
```

Section 15: Adding a Messages Feature

Setting Up The Entities For Messaging

Typical flow of adding a features starts with the API, and specifically with adding an entity for that feature. In this case we’ll add a “Message” entity with the following properties:

```
public int Id { get; set; }
public int SenderId { get; set; }
public string SenderUsername { get; set; }
public AppUser Sender { get; set; }
public int RecipientId { get; set; }
public string RecipientUsername { get; set; }
public AppUser Recipient { get; set; }
public string Content { get; set; }
public DateTime? DateRead { get; set; }
public DateTime MessageSent { get; set; } = DateTime.UtcNow;
public bool SenderDeleted { get; set; }
public bool RecipientDeleted { get; set; }
```

“Content” is the message content, everything above that should be explanatory. The last two bools are there for deleting the message from the db, which will only happen if *both* send and receiver delete the message from their ends.

In AppUser entity we’ll add two more lists for the messages, one for “sent” and another for “recieved”, so a user will have access to both sets:

```
public List<Message> MessagesSent { get; set; }
 public List<Message> MessagesReceived { get; set; }
```

This will be used similar to the “Liked” and “LikedBy” lists.

Finally, we’ll set up the Join table in the DataContext file by first adding the entity as a db

```
public DbSet<Message> Messages { get; set; }
```

Then we’ll use the builder to build the db according to what we need. In this case a many-to-many relationship again, similar to the Likes.

Recipient message	Sender message
builder.Entity<Message>() .HasOne(u => u.Recipient) .WithMany(m => m.MessagesReceived) .OnDelete(DeleteBehavior.Restrict);	builder.Entity<Message>() .HasOne(u => u.Sender) .WithMany(m => m.MessagesSent) .OnDelete(DeleteBehavior.Restrict);

Setting Up The Message Repository

Just like we’ve done before, we’ll start with an Interface called IMessageRepository that will have the methods we wish to support in our MessageRepository:

```
void AddMessage(Message message);
void DeleteMessage(Message message);
Task<Message> GetMessage(int id);
Task<PagedList<MessageDTO>> GetMessagesForUser();
Task<IEnumerable<MessageDTO>> GetMEssageThread(int CurrentUserId, int
recipientId);
Task<bool> SaveAllAsync();
```

We’ll use a MessageDTO so we can control what properties we want to transfer to the client. It will contain most of the same properties from the entity but will remove the AppUser properties as well as the “Deleted” props. We’ll add in strings for the senderPhotoUrl and recipientPhotoUrl so we can display those next to each message.

The MessageRepository will start with the simpler methods (we’ll return to the more complex ones later on). After importing the DataContext class:

```
public void AddMessage(Message message)
        {
        _context.Messages.Add(message);
```

…and…

```
        public void DeleteMessage(Message message)
                {
                _context.Messages.Remove(message);
```

…and…

```
        public async Task<Message> GetMessage(int id)
                {
                return await _context.Messages.FindAsync(id);
```

…and…

```
        public async Task<bool> SaveAllAsync()
                {
                return await _context.SaveChangesAsync() > 0;
```

Now to just add this service to our AppExtensions:

```
        services.AddScoped<IMessageRepository, MessageRepository>();
```

AutoMapper Profiles for Messages

Because the Message Entity is missing some things that AutoMapper won't be able to work out on its own, we'll make an AutoMapper Profile with specific instructions on how to map something.

Mainly, the main photo is not something that is easily understood by AutoMapper. It is however smart enough to get the username and userId from the AppUser that is passed in, which helps. So we'll add a new profile in AutoMapperProfiles:

```
        CreateMap<Message, MessageDTO>()
```

…followed by…

```
        .ForMember(d => d.SenderPhotoUrl,
                o => o.MapFrom(s => s.Sender.Photos.FirstOrDefault(x => x.IsMain).Url))
```

…which will map the main photo from the sender to the senderPhotoUrl property…

```
        .ForMember(d => d.RecipientPhotoUrl,
                o => o.MapFrom(s => s.Recipient.Photos.FirstOrDefault(x => x.IsMain).Url));
```

… and then do the same for the recipient.

We also need another DTO for the CreateMessage object we need to have, but no mapper profile will be necessary because it will only contain two properties for now:

```
        public string RecipientUsername { get; set; }
        public string Content { get; set; }
```

When a user goes to create a message they will populate it with those two items. Once the message is sent it will be transferred to the MessageDTO.

Adding a Message Controller

Let's build a controller! Start by injecting the IUserRepository, the IMessageRepository, and the IMapper. Don't forget to derive from the BaseAPI Controller!

We'll start with an [HttpPost] method that returns a Task<ActionResult<MessageDTO>>. It will take in a CreateMessageDTO object.

The idea here is that a message will be sent from the body of the request to api/messages, in this format:

"recipientUsername": "marion",
"content": "Test message 1 from Isabella to Marion"

First we'll get the username of the currently logged in user, and then check it against the recipient username (users can't send messages to themselves)

Then we'll get the sender from the _userRepository and the recipient username the same way. If the recipient doesn't exist we can return notFound().

With all that, we can create a new Message object! Then we'll add the message to the db via _messageRepository.AddMessage(message).

Finally, we can save the changes, and return the MessageDTO after it is mapped from the new message object.

```
var username = User.GetUsername();

      if (username == createMessageDto.RecipientUsername.ToLower())
            return BadRequest("You cannot send messages to yourself");

      var sender = await _userRepository.GetUserByUsernameAsync(username);
      var recipient = await
_userRepository.GetUserByUsernameAsync(createMessageDto.RecipientUsername);

      if (recipient == null) return NotFound();

      var message = new Message
      {
         Sender = sender,
         Recipient = recipient,
         SenderUsername = sender.UserName,
         RecipientUsername = recipient.UserName,
         Content = createMessageDto.Content
      };
```

```
        _messageRepository.AddMessage(message);

        if (await _messageRepository.SaveAllAsync()) return
Ok(_mapper.Map<MessageDTO>(message));

        return BadRequest("Failed to send message");
```

Test in postman.

NOTE: Testing Messages

In order to test messages from and to different users you must log in as them and ensure the correct recipient is in the body of the message.

Getting Messages from the Repo

Just like with the Likes we want to return a paginated list of messages, so we'll make a MessageParams.cs file that derives from the PaginatedParams class.

NOTE: PaginatedParams

Recall that this class sets the max page size, page number to start on, and the page size for the page.

The MessageParams file will contain a username and container property, with the container property set to a default "Unread". This will ensure unread messages (those that have a "messageReadDate" of "null")

Update the IMessageRepository file to reflect the change above. The GetMessagesForUser will now take in a MessageParams object

In the MessagesController we will add a method called GetMessagesForUser that will take in a MessageParams object [FromQuery]

NOTE: Nulls in Queries

The "username" property for the messageParams object will be populated in the method; be sure it is set to be nullable in the MessageParams class or else you will receive a 400 error

In this method, we will

- Get the username from system.security (from the token)
- Go to the _messageRepository.GetMessagesForUser and pass the messageParams
- Construct a response with a PaginationHeader using the default values from the PaginationParams
- Return the messages and the header

…which will look like…

```
messageParams.Username = User.GetUsername();
var messages = await _messageRepository.GetMessagesForUser(messageParams);
Response.AddPaginationHeader(new PaginationHeader(messages.CurrentPage,
messages.PageSize, messages.TotalCount, messages.TotalPages));
return messages;
```

Finally in the MessageRepository we will build our query so that when the controller calls for the messages, we can return what is requested (unread, inbox, or outbox):

Build the query AsQueryable();
Use a switch statement for the three options for the messages to return
Project the messages to a MessageDTO using autoMapper
Return a PagedList of MessageDTO's

…which codes as…

```
var query = _context.Messages.OrderByDescending(x =>
x.MessageSent).AsQueryable();
query = messageParams.Container switch
{
"Inbox" => query.Where(u => u.RecipientUsername == messageParams.Username),
"Outbox" => query.Where(u => u.SenderUsername == messageParams.Username),
_ => query.Where(u => u.RecipientUsername == messageParams.Username &&
u.DateRead == null)
};
var messages = query.ProjectTo<MessageDTO>(_mapper.ConfigurationProvider);
return await PagedList<MessageDTO>.CreateAsync(messages,
messageParams.PageNumber, messageParams.PageSize);
```

NOTE: PagedList

Recall that the PagedList object contains the custom method CreateAsync which takes a generic list <T> as the source of items, the page number, and the page size. It uses the built-in list method "Count" to count the number of items in the source, skips the appropriate number based on the page number and page size, takes the appropriate number of items based on the page size, and then makes them into a list. Then it returns the PagedList as a List<T>, the number of items total, the page number, and the page size.

Test in postman. Ensure that a user is logged in before sending the query.

Getting the Message Thread for Two Users

We want to get the thread of messages between two users, specifically the logged in user and another user that is specified in the query. First, to make things easier we'll update the IMessageRepository interface method "GetMessageThread" to take in strings for the currentUserName and recipientUserName (as opposed to ints for their respective id's)

The Controller will get a new [HttpGet] with a route parameter of "thread/{username}" to represent the other (recipient) username to query. This method will return the same as the previous method, an ActionResult<IEnumerable<MessageDTO>>.

First we'll get the currently logged in user's username. Then we'll pass that and the username we get from the query to the MessageRepository method "GetMessageThread":

```
return Ok(await _messageRepository.GetMessageThread(currentUsername,
username));
```

To the the message thread from the repository we will need to query a whole bunch of stuff from the db, starting with the current user and their photos, and the recipient user and their photos as well:

```
var messages = await _context.Messages
    .Include(u => u.Sender).ThenInclude(p => p.Photos)
    .Include(u => u.Recipient).ThenInclude(p => p.Photos)
```

…then we'll tack on a "Where" clause that looks for any messages in which the current user is the recipient AND the recipient user is the sender, OR the current user is the sender and the recipient is the recipient. Basically any messages between them and them alone.

```
.Where(
    m => m.RecipientUsername == currentUserName &&
    m.SenderUsername == recipientUserName ||
    m.RecipientUsername == recipientUserName &&
    m.SenderUsername == currentUserName
    )
```

… and then we'll order them in reverse, so newest at the top, and make it a list:

```
.OrderByDescending(m => m.MessageSent)
.ToListAsync();
```

We'll next check for unread messages

```
var unreadMessages = messages.Where(m => m.DateRead == null &&
    m.RecipientUsername == currentUserName).ToList();
```

…and use a "foreach" loop to mark them as read (since we're opening them up now):

```
if (unreadMessages.Any())
{
```

```
            foreach (var message in unreadMessages)
            {
            message.DateRead = DateTime.UtcNow;
            }

            await _context.SaveChangesAsync();

        }
```

Finally, return the mapped messages to the method that called (the controller).

```
        return _mapper.Map<IEnumerable<MessageDTO>>(messages);
```

Setting Up the Angular App for Messaging

Before we set this up, we're going to move some of the reusable methods for Pagination into their own file so we can access them from any service that is going to use them. In the memberService.ts file we'll grab the getPaginatedResults and getPaginatedHeaders methods, cut and paste them into their own file, paginationHelper.ts

NOTE: Making methods usable outside of a file

In order to make these methods usable outside of the file, we can just change the "private" tag for the methods to "export function". Then, in the file we want to use the methods in, like memberService.ts, we can simply import the new file:

```
        import { getPaginatedResults, getPaginationHeaders } from './paginationHelper';
```

…and then remove any references of "this." from any calls to the functions.

In the event a function in the new helper method requires something from the previous file (in this case, the HttpClient) we can add that as a parameter in the method signature.

Also, don't forget to import missing imports to the new file!

The MessageService

First we want to export an interface called "Message" to be our DTO for the client; this file can go in the "models" folder. Next, we'll use "ng g s _services/message –skip-tests" to create a message.service.ts file in the _services folder.

We'll need the baseUrl in this file as environment.apiUrl, as well as an injected HttpClient.

We can set a getMessages method that takes in a page number, page size, and container string, all so we can create a paginated response. From here we'll call the

getPaginationHeaders method in our new “helper” file, append the “container” to the result of the call, and then call the getPaginatedResults with an array for “Message”, the url to the API, the params, and the HttpClient object:

```
getMessages(pageNumber: number, pageSize: number, container: string)  {
  let params = getPaginationHeaders(pageNumber, pageSize)
  params = params.append('Container', container);
  return getPaginatedResults<Message[]>(this.baseUrl + 'messages', params, this.http);
```

The above method will get called from the messages.component.ts file. That file will set the default container, pageNumber, and pageSize, and pass them to getMessages. getMessages returns the observable “getPaginatedResults” so we can subscribe to it and wait for a response, setting the messages array and pagination object accordingly.

```
loadMessages()  {
        this.messageService.getMessages(this.pageNumber, this.pageSize,
        this.container).subscribe({
                next: response => {
                this.messages = response.result;
                this.pagination = response.pagination;
```

We can add a pageChanged method as well in preparation for the pagination buttons:

```
pageChanged(event: any) {
        if (this.pageNumber !== event.page) {
        this.pageNumber = event.page;
        this.loadMessages();
```

Don’t forget to call the “loadMessages” function in the ngOnInit!

Adding the Message Thread to the Client

For the thread to show, we’ll make a new component that specifically handles this, and it will be a child component of the member-detail. Start with “ng g c members/member-messages –skip-tests” and make a new component for the message thread.

Starting in the message.service file we’ll add a new method called “getMessageThread” that will take a string username. It will return an observable:

```
getMessageThread(username: string)  {
        return this.http.get<Message[]>(this.baseUrl + 'messages/thread/' + username);
```

In the component we created above we just need one method called “loadMessages”, which will use the injected messageService to getMessageThread while passing the username.

```
this.messageService.getMessageThread(this.username).subscribe({
```

```
next: messages => this.messages = messages
```

To use this method we'll just insert some temporary code into the html:

```
<p *ngFor="let message of messages">{{message.content}}</p>
```

…and then insert the component into the tab for "messages" in the member-detail component

```
<app-member-messages [username]="member.userName"></app-member-messages>
```

Notice the [username] being passed downward to the component from the parent (member-detail). In the member-messages component we'll need to add an Input in order to use this:

```
@Input() username: string | undefined;
```

Now we can pass the username from the "member" in the detail component down to the username string in the member-messages component.

Styling the Message Tab

Notes on styling:
Use *ngIf to display specific parts of the page based on whether we have certain elements yet
Use to place items like images at various parts of a list

```
<span class="chat-img float-end"
```

Create custom classes to change item size and color and margins

Activating the Messages Tab

Two things to look at here:

- If a user clicks on another user's profile but never goes to their messages tab, then why load the messages? That's a waste of resources
- If a user clicks on certain buttons we would like to automatically activate the messages tab when they land on the user's page

Starting with option 1, let's first dissect what we're currently doing.

When we click on a user and go to their details page, we load the messages as well, regardless of which tab we're looking at. This happens because the "loadMessages" method in the member-messages component is called when the member-details page is loaded up.

What we can do is make a new method in the member-details component that will look to see what tab is activated currently, and if the tab 'Messages' is activated, then we'll load the messages. Otherwise we won't.

```
onTabActivated(data: TabDirective)  {
        this.activeTab = data;
        if (this.activeTab.heading === 'Messages')  {
        this.loadMessages();
```

Each tab sends out an event that we can pass up to the component, so we'll call the "onTabActivated" method that way…

```
<tab heading="Messages" (selectTab)="onTabActivated($event)">
```

Each tab gets the above decorator, but only the 'Messages' tab will send up 'Messages' to the "ontabActivated" method. When that method gets the 'Messages' string, it will call "loadMessages, which we moved to the member-detail component

```
loadMessages() {
        if (this.member){
        this.messageService.getMessageThread(this.member.userName).subscribe({
        next: messages => this.messages = messages
```

Note: ViewChild

We want to be able to see the tab set of the html, so we need two things: an id for the tab set and a ViewChild property in the component:

```
<tabset class="member-tabset" #memberTabs>
@ViewChild('memberTabs') memberTabs?: TabsetComponent;
```

We will also need a few other properties to be able to use the "ontabActivated" method

```
activeTab?: TabDirective;
messages: Message[] = [];
```

This is because we moved the "loadMessages" method to this component

We'll still need to pass the messages down to the component, so we'll do that in the html:

```
<app-member-messages [messages]="messages"
[username]="member.userName"></app-member-messages>
```

Now, with the memberService injected into the member-detail component, we can access the message thread through "loadMessages", send the messages down to the component <app-member-messages>, and only do this when a user clicks on the specific tab of 'Messages'

Using Query Params

To route to a specific tab or place

We'll start with something more simple: routing the "Messages" button on the user's page to the "Messages" tab on their page. For this, we'll use a method that reads the memberTabs property that we created in the last lesson. First we'll check that we have a memberTabs object, then we'll use the "find" method to match it with a heading that we'll pass it from the html component:

```
selectTab(heading: string)  {
if (this.memberTabs) {
this.memberTabs.tabs.find(x => x.heading === heading)!.active = true;
```

…and in the html…

```
<button (click)="selectTab('Messages')" class="btn btn-success">Messages</button>
```

…we'll just pass 'Messages' to the method on the click event. Simple.

Routing from Other Components

We can add [queryParams] to router links in the html. For example, if we want to pass 'Messages' for the tab, like we did above, we can do that by adding it after the routerLink for the item we wish to link from. Here we do it for the envelope icon that is on each user's picture in the "Matches" section:

```
<button
    routerLink="/members/{{member.userName}}"
    [queryParams]="{tab: 'Messages'}"
    class="btn btn-info btn-sm">
    <i class="fa fa-envelope"></i></button>
```

NOTE: View comes after component construction

HTML loads after Javascript

Passing queryParams in the above example requires a separate method to handle. In our member-detail component we'll add a script in the ngOnInit to handle this when it gets passed in. Why here? Because that's the component we end up on, where the messages tab is located.

```
this.route.queryParams.subscribe({
    next: params => {
    params['tab'] && this.selectTab(params['tab'])
```

Unfortunately this won't work as-is because the javascript component loads before the html component. Since the above relies on the "selectTab" method, which checks to see that we

have a memberTabs object (which we don't yet, it's in the html which hasn't loaded yet) then we either don't go to the messages tab, or we get an error (if we try to circumvent typescripts failsafes).

This all happens because of the @ViewChild property that allows us to view the tabs in the child, the html template

```
@ViewChild('memberTabs') memberTabs?: TabsetComponent;
```

We can modify this, though, to allow us access beforehand. Currently it's dynamic, but we can make it static.

```
@ViewChild('memberTabs', {static: true}) memberTabs?: TabsetComponent;
```

Now, the issue with *this* method is that it requires the "member" to be loaded in order to load the html components in the view. That means that even with access to the view, we can't yet see or access the tabs because the tabs' loading is dependent on use getting the member, which we won't have until the component loads.

So how to fix it? We'll use a "route resolver"

Route Resolver

A route resolver is injectable, and it is provided in 'root' meaning it is available when the app starts up. It implements a resolve method, which basically gets executed before a user is routed to a new page. In this case we want to execute the getMember function so that we can then create the component that has the member details and the messages. Route resolvers execute asynchronously and only after processing the call will it route to the url requested. Thus, component initialization will wait until the callback is completed.

We'll start by making a resolver using "ng g r _[folder]/[file_name] –skip-tests"

In the resolver file, we'll add a constructor that injects the memberService, bc we need the member. The "resolve" method returns an observable, so we'll return the "Member" observable:

```
resolve(route: ActivatedRouteSnapshot): Observable<Member> {
    return this.memberService.getMember(route.paramMap.get('username')!)
```

We need to let the the routing module know that we will be using the resolver, so we add it to the route that we want to implement it on. In this case, the MemberDetailComponent:
Before:

```
{path: 'members/:username', component: MemberDetailComponent},
```

After:

```
{path: 'members/:username', component: MemberDetailComponent, resolve: {member: MemberDetailedResolver}},
```

In the memberDetail component we'll make a few changes, since we no longer need the "loadMember" method. In ngOnInit we can substitute the call for

```
this.loadMember();
```

…with…

```
this.route.data.subscribe({
    next: data => this.member = data['member']
```

The "data" is "member", which we call for in the appRoutingModule addition. In the "resolve" method of the the resolver we are getting the member by the username, basically the exact same way we did in the "loadMember" method originally:

Original:

```
const username = this.route.snapshot.paramMap.get('username');
this.memberService.getMember(username)
```

Resolver:

```
this.memberService.getMember(route.paramMap.get('username')!)
```

With the removal of the "loadMember" method we lose the ability to get the gallery images, but not really. We can just move the "this.galleryImages = this.getImages();" to the ngOnInit method.

One more thing: we need to declare our member as an empty object to avoid issues with nulls.

```
member: Member = {} as Member;
```

That line above allows us to remove the conditional "*ngIf="member"" from our html template. Now, we should be able to access the messages tab directly by clicking on the message button in the matches list.

Active Link

Currently, when we go to a member, we are still showing the "Matches" route highlighted, even though that's not where we're at. This has to do with the router link matching the first part of the route and saying "close enough". We can disable this behavior though, by going to the nav component html file and adding

```
[routerLinkActiveOptions]="{exact: true}"
```

…to the routerlink we wish to change attribute for (in this case, "/members"

Sending Messages

Now that our messages are loading the way we want, we need a simple way to send them, and we already have a form box set up for that. We'll start by making a method for sending messages in the messageService file:

```
sendMessage(username: string, content: string) {
    return this.http.post<Message>(this.baseUrl + 'messages',
```

```
        {recipientUsername: username, content})
```

This will take the username of the recipient and the content of the message, and send it up to the API, which will return an http post to confirm.

We'll call this method from the member-messages component with a similarly named "sendMessage" method that takes no parameters, because it will get what it needs from the html.

```
this.messageService.sendMessage(this.username, this.messageContent).subscribe({
    next: message => {
        this.messages.push(message);
        this.messageForm?.reset();
```

In order for the method to work, though, we'll need to add a new property

```
    messageContent = '';
```

…and a new view child…

```
    @ViewChild('messageForm') messageForm?: NgForm
```

The messageContent will be binded to the html, and the ViewChild('messageForm') will give us access to the form in the html.

HTML Send Form

Currently the form is very simple with no functionality:

```
<form>
    <div class="input-group">
        <input
            type="text"
            class="form-control input-sm"
            placeholder="Send a private message">
        <div class="input-group-append">
        <button class="btn btn-primary"
            type="submit">Send</button>
        </div>
    </div>
</form>
```

We're going to first change the <form> to

```
    <form #messageForm="ngForm" (ngSubmit)="sendMessage()" autocomplete="off">
```

The “#messageForm=”ngForm”” tells the component it is an Angular form with the appropriate methods attached to it (like “resetForm”). The ngSubmit tells it what to do when the submit button is pressed.

Then we can add the following to the <input>

```
name="messageContent"
required
[(ngModel)]="messageContent"
```

This is where the messageContent is bound to the component using the “name” and the “ngModel” decorators.

Now, messages can be sent and should appear in the list immediately!

Loading…

Fixing the photo weirdness

We can make use of a simple trick to hide content until stuff is loaded fully. In the messages component we’ll add a property “loading” and set it to “false”, then in our “loadMessages” method we’ll set it to “true” immediately, run the method, then set loading back to “false” at the end of the method.

In the html, where we want to hide content, we can simply add [hidden]=”loading” or *ngIf=”!loading”, and content will hide and show with the changing of the loading property. Easy.

Deleting Messages: API

The system that we have set up and wish to continue with is one where if a sender deletes a message but the recipient does not, then the message still populates in the recipients inbox and thread but not in the sender’s. The same is true for the opposite scenario. However, if *both* sender and recipient delete the message, then we will delete it from the database completely.

To support this functionality our message entity has two properties for each deleting, so in the API we’ll make a new [HttpDelete] function in the message controller:

```
[HttpDelete("{id}")]
public async Task<ActionResult> DeleteMessage(int id)
```

The message id will be taken in as a parameter. We want to first check that the user who is deleting the message is authorized to do so, so we’ll get the currently logged in user, get the message, and compare:

```
var username = User.GetUsername();
var message = await _messageRepository.GetMessage(id);
```

```
if (message.SenderUsername != username && message.RecipientUsername !=
username) return Unauthorized();
```

With authorization, we'll now check *who* is deleting the message, and change the appropriate property of the message:

```
if (message.SenderUsername == username) message.SenderDeleted = true;
if (message.RecipientUsername == username) message.RecipientDeleted = true;
```

Finally, if both of the above properties are marked "true", then the message can be removed from the database:

```
if (message.RecipientDeleted && message.SenderDeleted)
        {
                _messageRepository.DeleteMessage(message);
        }
```

Save and return Ok();

```
if (await _messageRepository.SaveAllAsync()) return Ok();
```

Updating methods to return correct messages

Now that we have a method that marks the messages as deleted by one or the other user, we still want to return messages to the user that didn't delete them, so we'll update two methods in our messageRepository: GetMessageForUser and GetMessageThread

In GetMessageForUser, we need to update the queries for "Inbox", "Outbox" and "unread" ("_"):

```
"Inbox" => query.Where(u => u.RecipientUsername == messageParams.Username
        && u.RecipientDeleted == false),
"Outbox" => query.Where(u => u.SenderUsername == messageParams.Username
        && u.SenderDeleted == false),
_ => query.Where(u => u.RecipientUsername == messageParams.Username
        && u.RecipientDeleted == false
        && u.DateRead == null)
```

The idea here is that in your inbox, you are the recipient of those messages, so if they are deleted by you then they won't appear. Your outbox contains messages that you sent, so you are the sender of those and again if you deleted them then you won't see those. Unread works the same as the inbox.

We'll update the GetMessageThread as well, specifically the Where clause for getting messages by adding conditionals for recipient or sender deleted:

```
.Where(
        m => m.RecipientUsername == currentUserName &&
        m.RecipientDeleted == false &&
        m.SenderUsername == recipientUserName
```

```
            ||
            m.RecipientUsername == recipientUserName &&
            m.SenderDeleted == false &&
            m.SenderUsername == currentUserName
    )
```

This checks if the message was or was not deleted, if it was not, then it will be shown

Deleting Messages: Client

A relatively easy set of methods to add and that's it. In the message.service we'll add a deleteMessage method that takes in the id of the message and routes to the API

```
    deleteMessage(id: number) {
        return this.http.delete(this.baseUrl + 'messages/' + id);
```

We don't get anything back because it's a deletion. We'll call this method front he messages.component file with a deleteMessage method that also takes in an id:

```
    deleteMessage(id: number) {
        this.messageService.deleteMessage(id).subscribe({
        next: _ => this.messages?.splice(this.messages.findIndex(m => m.id == id), 1)
```

We call the message.service method and pass the id. Once that's done we then go into the messages property that we have and remove the message at the index of the id we passed in. We remove just that message, hence the '1' at the end.

Because it's an observable it will update the view automatically, but not the pagination.

Finally we call the method to delete from the delete button in the html:

```
    <button
        (click)="$event.stopPropagation()"
        (click)="deleteMessage(message.id)"
        class="btn btn-danger">Delete
    </button>
```

The first click event is to tell the browser to ignore the fact that clicking on the row will result in a page change to the messages tab. That way clicking the delete button just deletes the message and that's it. The second click event handles the deteletion.

Section 16: Identity Role and Management

Setting up the Entities

MS Identity is battle-hardened, tested, and reliable compared to what we've built thus far for managing users logging in and registering. We can still add Identity to our application fairly easily though, with just a few small modifications to the classes we've set up in the API already.

Starting the AppUser.cs file, we're going to now inherit from "IdentyUser"

```
public class AppUser : IdentityUser<int>
```

Note the <int> at the end. Identity framework typically uses strings for id's, but since we want to use ints for id's we can specify that by adding it there.

We'll make another class in the Entities folder called "AppUserRole" which will derive from "IdentityUserRole<int>". This class will effectively be our "Join" table between AppUser and AppRole classes (not created yet).

```
public class AppUserRole : IdentityUserRole<int>
    {
    public AppUser User { get; set; }
    public AppRole Role { get; set; }
```

Finally we'll create the AppRole class, which will derive from "IdentityRole<int>" and will contain all the roles a user could take on.

```
public ICollection<AppUserRole> UserRoles { get; set; }
```

We will also add this same property to the AppUser class

We set things up this way so we have better control over which roles users can have, and ideally we can change roles in the future from the front end if needed.

We've broken the app, so we'll fix the issues in the AccountController and Seed files. Both files refer to "passwords hashes" and etc, but now that we're using Identity, we no longer need those. Any errored lines can be commented out.

NOTE: AppUser properties to remove

Because we are using Identity, we no longer need an "Id", "Username" "PasswordHash" or "PasswordSalt" property, and these can be removed as well.

Configuring the DB Context

NOTE: dotnet –info

Get the version of dotnet that is running currently. This is needed for the nuget package we'll be installing

Open nuget gallery and search for "microsoft.aspnetcore.identity" and select the ".EntityFrameworkCore" version. Install.

Once that is installed we will change the DataContext class to derive from "IdentityDbContext". However, because we created so many new classes and we want to control how they are id'd, we need to specify them in this declaration.

```
public class DataContext : IdentityDbContext<AppUser, AppRole, int,
IdentityUserClaim<int>, AppUserRole, IdentityUserLogin<int>,
IdentityRoleClaim<int>, IdentityUserToken<int>>
```

The above declaration tells the IdentityDbContext about our AppUser and AppRole classes, that we want to use an "int" for their id's, and then lists the other related classes that we'll be using and what type of id they will use as well. They need to be in the order listed to ensure proper functionality.

Now we can comment out the DbSet<AppUser>, since Identity is handling that now

To create our join table we will construct it in the builder like we did with the UserLikes:

```
builder.Entity<AppUser>()
        .HasMany(ur => ur.UserRoles)
        .WithOne(u => u.User)
        .HasForeignKey(ur => ur.UserId)
        .IsRequired()
        .OnDelete(DeleteBehavior.Cascade);



builder.Entity<AppRole>()
        .HasMany(ur => ur.UserRoles)
        .WithOne(u => u.Role)
        .HasForeignKey(ur => ur.RoleId)
        .IsRequired();
        .OnDelete(DeleteBehavior.NoAction);
```

These commands connect the AppUser with the AppRole class, with the foreignKey of UserId that joins them together.

NOTE: Migration Error

If using something other than SqLite, the above *italicized* commands must be added (similar to the issue with userLikes) or else there will be an error generated related to “Cascading” when the migration is implemented on the next run.

Configuring the Startup Class

The next step is to configure “Identity” in the extension method “IdentityServiceExtensions”. In this class we can add the services.AddIdentityCore<AppUser>(options => … and specify a bunch of options:

```
services.AddIdentityCore<AppUser>(options =>
{
        options.Password.RequireNonAlphanumeric = false;
})
```

NOTE: Other Options

options.Passoword allows for a variety of password options to be specified. options.User also allows for similar options, so take a look and see what you may need!

Finally, at the end of the above command we’ll add the following commands:

```
.AddRoles<AppRole>()
.AddRoleManager<RoleManager<AppRole>>()
.AddEntityFrameworkStores<DataContext>();
```

The above must be in the listed order so they are added correctly. For instance, for “AddRoleManager<RoleManager<AppRole>>()” we must add the type RoleManager first to let identity know that that is the type of AppRole.

Refactoring and Adding a New Migration

Our seed class does not work just yet, so we’ll comment out the line:

```
await Seed.SeedUsers(context);
```

…in the program.cs file. Now we add a migration, and then dotnet run to update our databases.

Updating the Seed Method

Now that we’ve implemented Identity Core we can re-seed our database with the same data, but refreshed to use Identity’s password hashing system.

In the seed.cs file we're going to remove the DataContext injection and replace it with UserManager from Identity Core.

```
public static async Task SeedUsers(UserManager<AppUser> userManager)
```

The first command where we check to see if there's any users already in the database uses the DataContext; we'll swap that out for the UserManager object we injected:

```
if (await context.Users.AnyAsync()) return;
```

…becomes…

```
if (await userManager.Users.AnyAsync()) return;
```

We can remove the using var hmac line because we'll allow Identity to handle passwords, as well as the other lines regarding passwordHash and passwordSalt. Adding the users is done with:

```
await userManager.CreateAsync(user, "Pa$$w0rd");
```

…and since the above method also saves the user to the database, we can remove the "await context.SaveChangesAsync();" method as well.

Back in our Program.cs file we can re-enable the Seed.Seedusers line, but this time we need to pass "UserManager" instead of "DataContext". We'll create that with:

```
var userManager = services.GetRequiredService<UserManager<AppUser>>();
```

…followed by…

```
await Seed.SeedUsers(userManager);
```

Now that we've completely modified how users will be stored we need to drop the database via "dotnet ef database drop" and say "y" when prompted. Then "dotnet watch –-no-hot-reload" and reseed. If all is working, we will have a brand new database with the same users as before.

Updating the Account Controller

Now to update the account controller on the API to use Identity Core. We'll replace the DataContext with a UserManager object first. Everywhere we use _context we'll change it to use _userManager instead, like in the UserExists method:

```
return await _userManager.Users.AnyAsync(x => x.UserName == username.ToLower());
```

In the Register method we can remove the _context.Users.Add() method and replace it with a var to get a result from CreateAsync()

```
var result = await _userManager.CreateAsync(user, registerDTO.Password);
```

Here we pass the user and the password that was created by the user, Identity will create the user in the database for us after hashing the password. The result will store any errors for us that we can return if necessary.

In the Login method we'll do something similar:

```
var result = await _userManager.CheckPasswordAsync(user, loginDTO.Password);
if(!result) return Unauthorized("Invalid password");
```

This result is just a boolean, so it either logs in or not. We can return the message to the client and it will show as a toast.

That's it. Now we can run and test the application. The filters at this point are acting a little funny but perhaps that is something we'll address in a later lesson.

Adding Roles to the App

To add roles to the users that we seed, we'll first drop the database

```
dotnet ef database drop
```

Then in the seed.cs file we'll inject a RoleManager<AppRole> object as roleManager. This will give us access to creating a table of roles for users. Then we can make a list of roles to insert into the table:

Create roles	Insert roles

var roles = new List<AppRole>
{
new AppRole{Name = "Member"},
new AppRole{Name = "Admin"},
new AppRole{Name = "Moderator"},
};
	foreach (var role in roles)
	{
	await roleManager.CreateAsync(role);
	}

Note that the roleManager gives us access to a CreateAsync method for creating roles in the table.

Then in our foreach loop for users, we can add each user to the table and then assign them a role using the roleManager:

```
await userManager.AddToRoleAsync(user, "Member");
```

The first argument is the user getting the role, and the second argument is the role they get.

We'll also create an admin user

```
var admin = new AppUser
{
        UserName = "admin"
};
```

NOTE: null properties and database creation

Once again, any properties that are not nullable will be required. The above statement only works if all properties can be null for an AppUser object.

We'll then create the admin user and give them some roles:

```
await userManager.CreateAsync(admin, "Pa$$w0rd");
await userManager.AddToRoleAsync(admin, "Admin");
await userManager.AddToRoleAsync(admin, "Moderator");
```

Add the roleManager

In the program.cs, the SeedUsers method now takes in a second argument. Create a role manager object…

```
var roleManager = services.GetRequiredService<RoleManager<AppRole>>();
```

…and then add it to the method…

```
await Seed.SeedUsers(userManager, roleManager);
```

Now we can rebuild the database with dotnet ef run. Check for errors relating to:

- Cascading
  - See above for fix; you must drop the database and remove the last migration and create a new migration
- INSERT fails
  - Possibly due to a null property not allowed. Again, check entities and redo migration
- Foreign-key conflicts
  - This will be due to using the same key for two different entities. Check the join table creating in the dataContext, ensure that each entity in the join has it's own unique foreign key

Adding the roles to the JWT Token

When a user registers, we'll give them a role of "Member". In addition, when they register or log in, we'll call our token service and add their "role" to the token so we can access it. To do this we'll start in the TokenService.cs file and inject the UserManager class as userManager. Now we can use it to get the user's role with…

```
var roles = await _userManager.GetRolesAsync(user);
```

…and add it to the token with…

```
claims.AddRange(roles.Select(role => new Claim(ClaimTypes.Role, role)));
```

The "claims" object provides for us a way to access the role as a Role, so the token knows that it is in fact a role. The "Select" will project the elements we get back from the ClaimTypes.Role call and place it in the token.

NOTE: await and async now needed

Because the userManager makes use of “GetRolesAsync” we must add “async” to our method. Where the method once returned a string, it will now return a Task<string>. We’ll need to update the interface to reflect this as well.

Likewise, in the AccountController, everywhere we call for the _tokenService.CreateToken method we’ll need to add “await” to the front.

While in the AccountController, we’ll take this moment to add each new member that registers to the “Member” role:

```
var roleResults = await _userManager.AddToRoleAsync(user, "Member");
if (!result.Succeeded) return BadRequest(result.Errors);
```

Test this in Postman by logging in as a user, copying the token that is returned, and pasting it into jwt.ms to view the “role” : “Member” line. If this does not appear, restart the API.

Policy Based Authorization

Policy based auth is a bit more flexible than role-based.

We’ll start by making a new controller called AdminController that will derive from our BaseAPIController. We’ll add two methods in it that we can use to test policy-based authorization.

```
[Authorize(Policy = "RequireAdminRole")]
        [HttpGet("users-with-roles")]
        public ActionResult GetUsersWithRoles()
        {
        return Ok("Only admins can see this");
        }
```

```
[Authorize(Policy = "ModeratePhotoRole")]
        [HttpGet("photos-to-moderate")]
        public ActionResult GetPhotosForModeration()
        {
        return Ok("Admins or moderators can see this");
        }
```

Note the “Authorize(Policy…) line. These strings are the names that the policy will use when determining if a user can access these methods.

In the IdentityServiceExtensions, we’ll add the Authorization policy after the Authentication service. Authentication comes first, and then authorization. We’ll pass the AddAuthorization method some options that will tell the app what policy name requires which roles to access:

```
options.AddPolicy("RequireAdminRole", policy => policy.RequireRole("Admin"));
options.AddPolicy("ModeratePhotoRole", policy => policy.RequireRole("Admin",
"Moderator"));
```

These policies can now be used throughout the API. Test in postman by logging in as a “member” user and trying to access. It should return 403 Unauthorized. Logging in as admin and accessing will return a 200 OK and the string.

Returning a List of User and their Roles

Currently our GetUsersWithRoles method in the AdminController just returns a string, but we can modify this to actually return a list of users and their roles. We need access to our UserManager<AppUser> class, so we’ll add a constructor for that

NOTE: Injection of classes

Need to access a class of something in a controller or otherwise? Create a constructor that injects that thing as a parameter.

Okay, now that we have a _userManager object to access the users and their roles, let’s modify the method to do that. We’ll get the users and store them in an object “users”. When we get the users we’ll order them by their username first. Then we’ll use “Select” to project them into an anonymous object. This object allows us to return just the items we want, so even though the users have lots of properties, we just want to see a couple.

```
var users = await _userManager.Users
        .OrderBy(u => u.UserName)
        .Select( u => new
        {
                u.Id,
                UserName = u.UserName,
                Roles = u.UserRoles.Select(r => r.Role.Name).ToList()
        })
        .ToListAsync();
```

NOTE: Projection to Properties

In the above code, “UserName =” is used to give the u.UserName property a tag in the returned JSON. However, this can be anything you want, or can be left blank because u.UserName returns the user’s username and gives it its default label. For instance, the following:

```
{
        u.Id,
        u.UserName,
```

```
        })
```
… would still return the “userName” labeled as “userName” in the JSON. This does not work for “Roles” though, because the roles are projected into a new list, and they need to be stored somewhere. The name “Roles” could be changed to whatever, but the list must be assigned to a variable in order for the code to compile.

There are other options for the u.UserRoles (like “count” and etc), but for our purposes here we just want the names of the roles, so we’ll just return those.

NOTE: Matching to the interface export in the client

It can be tempting to make changes as mentioned above, but be careful of this! If your exported interface in the client uses specific terms, then changing them here will cause errors with interpolation later on.

For example:

u.Id, UserName = u.UserName, Roles = u.UserRoles.Select(r => r.Role.Name).ToList()	Our interface in the client uses “username” and “roles” as property names. Using this code will result in these values being populated correctly.
u.Id, u.UserName, Role = u.UserRoles.Select(r => r.Role.Name).ToList()	Because “username” is not specified, and “Roles” is changed to “Role”, the interface will not match the properties up correctly, and will result in empty properties on the client side. This means that interpolation will not work.

Now we can return an Ok(users) and test in postman. Be sure to log in as admin first!

Editing User Roles

Now it’s time to add an endpoint that allows us to edit roles for users using a query string. We’ll start with our authorize policy and an httpPost with the endpoint:

```
[Authorize(Policy = "RequireAdminRole")]
[HttpPost("edit-roles/{username}")]
```

Our method will take in the username to change roles in as a string, as well as the roles we wish to add or remove from the user as a query string:

```
public async Task<ActionResult> EditRoles(string username, [FromQuery]string roles)
```

Therefore our endpoint address will look like:

{{url}}/api/admin/photos-to-moderate

Now for some logic to get the roles from the string, split them by the comma that will separate them and turn them into an array. Then we'll get the user via the userManager object we have (provided by Identity). We'll get their roles as well:

```
if (string.IsNullOrEmpty(roles)) return BadRequest("You must select at least one role");
var selectedRoles = roles.Split(",").ToArray();
var user = await _userManager.FindByNameAsync(username);
if (user == null) return NotFound();
var userRoles = await _userManager.GetRolesAsync(user);
```

Now we'll add the roles to the user, only if they are not part of the roles already (using the "except" method)

```
var results = await _userManager.AddToRolesAsync(user,
selectedRoles.Except(userRoles));
if (!results.Succeeded) return BadRequest("Failed to add to roles");
```

Then, we'll remove the roles that are *not* part of the list that was passed in:

```
results = await _userManager.RemoveFromRolesAsync(user,
userRoles.Except(selectedRoles));

if (!results.Succeeded) return BadRequest("Failed to remove from roles");
```

Essentially we are just passing in a list of new roles for the user to be a part of. If a role is not in the list in the query, then it is not added to the user. To check our work we'll return the list:

```
return Ok(await _userManager.GetRolesAsync(user));
```

Adding and Admin Component and Admin Guard

Now that we have a policy and an admin account and roles, we want to be able to access these things within the client. However, the component should not be accessible by members who are not admin or moderators, so we'll have to make a guard to block access.

Start by making a new component (ng g c admin/admin-panel –skip-tests). We'll add the routing to the app-routing.module with: {path: 'admin', component: AdminPanelComponent}. Then in the nav-component we'll add a link in the nav bar to 'admin':

```
<a class="nav-link" routerLink="/admin" routerLinkActive="active">Admin</a>
```

Easy enough. Now for the guard.

To use the guard we'll use the token that the client has access to. Real security happens on the API, so the token is validated there. On the client side we can decode the token and look at what's in it. Currently, the token for each user contains their roles, either as a string (one role) or

an array (multiple roles). We'll access that portion of the token on the client side, then, and use it to determine if the user has authorization to access the Admin panel.

NOTE: The API handles validation

The user token is validated long before they are able to access the admin panel on the client side. This kind of security is essentially hiding a safe behind a curtain. The API is the safe, the client is just a curtain hiding stuff that is still there, but the user can't see it because of their roles.

On the client side we have the token in the User interface, so we'll access the token from the user in the account.service.ts file.

```
getDecodedToken(token: string) {
    return JSON.parse(atob(token.split('.')[1]));
```

Here we'll take the token, parse it, use atob to allow the token to be split at the period, which will give us an array of three items. We want the second, so we call it at index [1] and return it.

We'll call the above method from the setCurrentUser method, get the roles out of the token, and store them in the user.

```
user.roles = [];
const roles = this.getDecodedToken(user.token).role;
```

In the token string that we've passed back we are looking for "role", hence the '.role' at the end.

Because we don't know if the user is part of many roles or just one, we don't know if we have an array or just a string, so we'll check for that and, if we have an array, we'll set it to user.roles. Otherwise, we'll take the single role and push it into the roles[], so either way we are working with an array:

```
Array.isArray(roles) ? user.roles = roles : user.roles.push(roles);
```

Now that we have all that taken care of, we have a way to access the roles in the user observable. We'll make use of another guard to ensure only certain roles are allowed access to the admin-panel. Create the guard with (**ng g guard _guards/admin –skip-tests**) and also say "CanActivate" at the prompt.

In the app-routing we'll add to the path: 'admin':

```
{path: 'admin', component: AdminPanelComponent, canActivate: [AdminGuard]},
```

Now to configure the guard. We only need to return an Observable<boolean> so the other returns can be removed. We'll create a constructor to bring in our account service and also toastr.

```
constructor(private accountService: AccountService, private toastr: ToastrService) {}
```

In the canActivate method we'll access the user observable (we don't need to subscribe, guards automatically subscribe for us) so we can skip to "pipe" and "map" to get the roles from the user. If the user has the role of 'Admin' or 'Moderator' then we return true:

```
canActivate(): Observable<boolean> {
    return this.accountService.currentUser$.pipe(
    map(user => {
    if (!user) return false;
    if (user.roles.includes('Admin') || user.roles.includes('Moderator')) {
    return true;
```

…otherwise, we'll return false…

```
else {
    this.toastr.error('You cannot enter this area')
    return false;
```

Now we can test this by logging in as a standard user and try to access the admin panel (should get a toast) and logging in as an admin / moderator and seeing the panel when we navigate to it.

Adding a Custom Directive

We now want to hide the Admin link from users that shouldn't have access to it. So far we've been using directives (ngIf) to block links from appearing in the navbar, but we have the option to create our own directives.

ng g d _directives/has-role –skip-tests

In the new file that is created we see a "selector" element with the value '[appHasRole]'. This works very similar to the *ngIf statements from before. In this case, we'll call our directive with:

```
*appHasRole='["Admin","Moderator","Etc…"]'
```

…and this will allow us to remove certain things from the dom.

To do this, we'll need to add a few things to the directive we just created. First in the class declaration we need to implement OnInit. We'll also take an @Input as a string array from wherever it's being called, and we need access to our user object

```
export class HasRoleDirective implements OnInit {
  @Input() appHasRole: string[] = [];
  user: User = {} as User;
```

Next, in the constructor we need to have a viewContainerRef object, a templateRef object, and the accountService…

```
constructor(private viewContainerRef: ViewContainerRef,
private templateRef: TemplateRef<any>, private accountService: AccountService)
```

In the constructor we need to subscribe to the user and save the current user in the directive…

```
this.accountService.currentUser$.pipe(take(1)).subscribe({
    next: user => {
    if (user) this.user = user
```

Finally, in the ngOnInit method we will check the user for roles. If any of the roles that have been passed to us from the component are in the user's roles list, then we'll display the item in the html that has the *appHasRole tag…

```
if (this.user.roles.some(r => this.appHasRole.includes(r))) {
this.viewContainerRef.createEmbeddedView(this.templateRef);
```

…otherwise, we won't display it…

```
this.viewContainerRef.clear()
```

Now, in the html component we wish to use this in, we'll just add the following to whatever div, li, or other container that we want to show only to admins…

```
*appHasRole="['Admin','Moderator']"
```

Adding the Edit Roles Component

We'll start by making a service and two new components. We'll create an admin.service, a user-management component, and a photo-management component.

The service will exist to serve data to the components, mainly getting the user information from the api for us. We'll set a base url and inject the HttpClient. Then we'll have a method that will get the users and their roles from the API and return them:

```
getUsersWithRoles() {
return this.http.get<User[]>(this.baseUrl + 'admin/users-with-roles');
```

Starting in the user-management component we'll inject our adminService, and we'll create a "users" property that will be an empty array of User[]. Then we'll add a method to connect to the service to get the users and place them in the array:

```
getUsersWithRoles() {
    this.adminService.getUsersWithRoles().subscribe({
    next: users => this.users = users
```

Don't forget to call the above method inside ngOnInit()

In the user-management html we'll just build a table for now that populates with the usernames and the roles for each user.

NOTE: Getting the right properties from the API

See this note here about ensuring that the client-side object interface properties match what is exported from the API. If interpolation is not working but the API call is returning the correct values, check that the values are being returned with the right property names.

Setting up Modals

A modal is like a mini pop-up on the page, and we'll make use of this from ngx bootstrap here.

We want to use the Component Modal so we can pass info to it. We'll start in the shared.module and import the necessary stuff:

```
import { ModalModule } from 'ngx-bootstrap/modal';
ModalModule.forRoot()
ModalModule
```

In the user-management component we'll add an object of bsModalRef and we'll also import that modalService from BsModalService:

```
bsModalRef: BsModalRef<RolesModalComponent> = new
BsModalRef<RolesModalComponent>();
private modalService: BsModalService
```

Create a component with ng g c modals/roles-modal –skip-tests. In here we'll first just copy the html template from ngx-bootstrap:

```
<div class="modal-header">
<h4 class="modal-title pull-left">{{title}}</h4>
<button type="button" class="btn-close close pull-right" aria-label="Close"
(click)="bsModalRef.hide()">
<span aria-hidden="true" class="visually-hidden">&times;</span>
</button>
</div>
<div class="modal-body">
<ul *ngIf="list.length">
<li *ngFor="let item of list">{{item}}</li>
</ul>
</div>
<div class="modal-footer">
<button type="button" class="btn btn-default"
(click)="bsModalRef.hide()">{{closeBtnName}}</button>
</div>
```

The above can be found in the component 'tab'.

There are some errors because it's looking for

A title
A list
A closeBtnName

…so in the component of the roles-modal we'll add those as class properties:

```
title = '';
list: any;
closeBtnName = '';
```

We'll also inject the bsModalRef in the constructor.

Back in the user-management component we'll create a method to call the modal with. The general way to call it is with

```
this.bsModalRef = this.modalService.show(RolesModalComponent);
```

…but we want to send some info to the modal as well, which we can do with an "initialState" parameter. First we'll set the initial state options:

```
const initialState: ModalOptions = {
    initialState: {
        list: [
            'Do thing',
            'Another thing',
            'Something else'
            ],
        title: 'Test Modal'
    }
```

Then modify the call:

```
this.bsModalRef = this.modalService.show(RolesModalComponent, initialState);
```

We also need to give the closeBtnName a name:

```
this.bsModalRef.content!.closeBtnName = 'Close';
```

Now in our html, for the "Edit roles" button we can call this with a click event:

```
<td><button (click)="openRolesModal()" class="btn btn-info">Edit roles</button></td>
```

Editing Roles

Now that we have modal functionality, we'll populate the modal with the data we wish to display. Starting in the role-modal we'll change the generic class properties to our custom properties:

```
username = '';
availableRoles: any[] = [];
selectedRoles: any[] = [];
```

We'll also add a method that will either remove or add a role to the "selectedRoles" array by checking if the role has just been checked / unchecked:

```
updateChecked(checkedValue: string) {
const index = this.selectedRoles.indexOf(checkedValue);
```

```
        index !== -1 ? this.selectedRoles.splice(index, 1) :
this.selectedRoles.push(checkedValue);
```

The above method will be called whenever a checkbox in the html is interacted with, and the role associated with that checkbox will be sent as the parameter.

In the html we can change the “modal-body” to be an ngForm

```
        <form #rolesForm="ngForm" id="rolesForm">
        <div class="form-check" *ngFor="let role of availableRoles">
```

In the form we’ll add a ngFor loop to loop through the array of available roles and display them with a checkbox next to each. The checkbox will call the “updateChecked” method from above whenever a user checks or unchecks the box:

```
        <input type="checkbox"
                class="form-check-input"
                [checked]="selectedRoles.includes(role)"
                value="role"
                (change)="updateChecked(role)"
                [disabled]="role === 'Admin' && username === 'admin'">
        <label>{{role}}</label>
```

The last line is to disable the ability for an admin to uncheck themselves as ‘admin’.
Now that we have the basics, we can get more specific with what we show in the modal. First, in the admin.service we will add a method to return a post request to connect to the API to edit the roles of the user we pass in with the roles we pass in as well:

```
        updateUserRoles(username: string, roles: string[]) {
                return this.http.post<string[]>(this.baseUrl + 'admin/edit-roles/'
                + username + '?roles' + roles, {});
```

NOTE: remember that POST requests require an object

Now in the user-management component we can add a class property of “availableRoles”, which normally we would gather from the API but in this case we only have three so it’s not necessary:

```
        availableRoles = [
                'Admin',
                'Moderator',
                'Member'
         ]
```

In the openRolesModal we’ll modify our initialState and populate it with actual values. We’ll also add in a ‘user’ as a parameter for the method so we have access to the **username**:

```
        openRolesModal(user: User) {
                const config = {
                        class: 'modal-dialog-centered',
                        initialState: {
```

username: **user.username**,
availableRoles: this.availableRoles,
selectedRoles: [...user.roles]

NOTE: it is important to ensure the initialState properties are named correctly

They should match the class properties in the roles-modal, or else they will not display properly

We can test that the roles populate correctly right now.

We don't have any logic to change the roles yet, so let's add that. In the openRolesModal, at the end we had a method for the closeBtnName. We'll replace that with a method for onHide (when the Modal is hidden either through the Submit button or through the close button):

```
this.bsModalRef.onHide?.subscribe({
next: () => {
const selectedRoles = this.bsModalRef.content?.selectedRoles;
```

Here we are creating an array of the selectedRoles from the roles that are selected by the user. We get this from the roles-modal component class property "selectedRoles".

We need to check to see if any roles have been checked / unchecked (they will show up in the selectedRoles array as either added or removed) and then compare it with the array of roles the user already has. If they are the same we won't make an API call. This situation happens if a user opens the modal but then closes it without making a selection.

We'll make a private method to compare the two arrays. It will convert them to strings and see if they are equal:

```
private arrayEqual(arr1: any[], arr2: any[]) {
return JSON.stringify(arr1.sort()) === JSON.stringify(arr2.sort());
```

Now we can call this method and use it to check if the user.roles need to be updated or not:

```
if (!this.arrayEqual(selectedRoles!, user.roles)) {
    this.adminService.updateUserRoles(user.username, selectedRoles!).subscribe({
        next: roles => user.roles = roles
    })
```

Finish it off by setting the user roles to the new list of roles.

Section 17: SignalR

Adding a Presence Hub

SignalR offers features that allow us to send live notifications and updates to users based on things like other users logging in, sending and receiving messages, etc. We'll implement all of this in our application, starting with a "live presence" that notifies users when other users log in.

To start this we'll go to our API and create a new folder called "SignalR" and in it place a new class called "PresenceHub". This class will derive from "Hub" which imports from Microsoft.AspNetCore.SignalR.

In this class we'll have two methods to start: one that will handle when a user connects and another that will be for when a user disconnects.

```
public override async Task OnConnectedAsync()
        {
        await Clients.Others.SendAsync("UserIsOnline", Context.User.GetUsername());
        }
```

…and…

```
public override async Task OnDisconnectedAsync(Exception? exception)
        {
        await Clients.Others.SendAsync("UserIsOffline", Context.User.GetUsername());

        await base.OnDisconnectedAsync(exception);
        }
```

In the above method, "Clients" refers to any users that are connected to the API. The "SendAsync" method is labeled with the way we want to call it ("UserIsOnline", etc), and the Context object gives us access to the username of the user that just logged in (or logged out).

To use this service, we'll add it to our ApplicationServiceExtensions class as:

```
services.AddSignalR();
```

And to let the program know that the presence hub exists, we'll add it in our Program.cs file:

```
app.MapHub<PresenceHub>("hubs/presence");
```

The "hubs/presence" will be the endpoint to which it can be accessed.

Now we need to deal with authenticating users so that we can use them in the hub and ensure they have appropriate permissions. Because SignalR does not use http requests, but instead

web sockets to communicate with the client, we'll do this in a different way than what we're used to.

We'll start in the IdentityServiceExtensions class where we AddAuthentication. We'll add another options, this time of "Events", and we'll use this to request an "access_token". This will grant us the access token (the bearer token essential) needed to access the "/hubs" section of the API.

```
options.Events = new JwtBearerEvents
{
        OnMessageReceived = context =>
        {
                var accessToken = context.Request.Query["access_token"];
                var path = context.HttpContext.Request.Path;
                if (!string.IsNullOrEmpty(accessToken) &&
                        path.StartsWithSegments("/hubs"))
                {
                        context.Token = accessToken;
                }
                return Task.CompletedTask;
        }
 };
```

NOTE: access_token must be typed as shown, no changes

NOTE: /hubs must match what we called the endpoint in the presence class

To use this we'll add ".AllowCredentails()" in the app.UseCors method in our Program.cs file.

Client side SignalR

Now that we have a presenceHub class on the API we want to call the two methods inside it when users connect or disconnect, and here's how we'll do that. First we'll install the necessary libraries for SignalR into the client with **npm install @microsoft/signalr –legacy-peer-deps**. Next we'll add the endpoints to our environment.ts and environment.prod.ts files as "hubUrl" so we can call them in the new service we're going to create.

We'll build a new service called presence.service and add two class properties to it:

```
hubUrl = environment.hubUrl
private hubConnection?: HubConnection;
```

We'll also inject our ToastrService into this service so we can send toasts to users about other users logging in (this is only temporary). In the service we need two methods: one to handle

when a user is connected, and the other to handle when users are disconnected. We'll eventually call these methods from the accountService.

For the first method, 'creatHubConnection(user: User), we'll use the hubConnection class property to build a new hub connection with the url of "https://localhost:5001/hubs/presence", and it will carry an access token that we pull from the user.token. We'll set the hub to automatically try reconnecting every few seconds by default as well:

```
this.hubConnection = new HubConnectionBuilder()
        .withUrl(this.hubUrl + 'presence', {
        accessTokenFactory: () => user.token
        })
        .withAutomaticReconnect()
        .build();
```

Then we can start the hubConnection with "start()" and catch any errors that get thrown

```
this.hubConnection.start().catch(error => console.log(error));
```

In the presenceHub class on the API we had two methods and each had a tag in it of "UserIsOnline" or "UserIsOffline". We'll access those with the following methods:

```
this.hubConnection.on('UserIsOnline', username => {
this.toastr.info(username + ' has connected');
})

this.hubConnection.on('UserIsOffline', username => {
this.toastr.warning(username + ' has disconnected');
})
```

These two commands essentially are listening for when a user connects or disconnects from the hub in the API. If a user connects, then the first command is called and a toaster notifies all users that the user has connected. Likewise, if the user disconnects from the hub in the API the second method "OnDisconnectedAsync" is called, and the 'UserIsOffline' method is called. Each time, the method in the API hub returns the username so that it can be used in the toast that is displayed.

When a user logs off, the stopHubConnection will handle disconnecting from the hub as well:

```
stopHubConnection() {
        this.hubConnection?.stop().catch(error => console.log(error));
}
```

Finally, in the accountService we'll call each of the above methods when either a user logs in or a user logs out:

In the 'setCurrentUser' method: this.presenceService.createHubConnection(user);

In the ‘logout’ method: this.presenceService.stopHubConnection();

Test functionality by logging in as a user, then in a new container tab (firefox) logging in as another user. Try logging out as well.

Adding a Presence Tracker

So currently we have ways to see if users connect or disconnect to the hub, but we have no way of knowing *which* users are connected or disconnecting, ie if a user is *still* connected or not. This has not been implemented by Microsoft because it’s possible that the application could live on multiple servers, meaning the hub can exist on multiple servers, and therefore a user may connect to a server that other users are not, and so the other users would not see the first user as connected or “online”. There are some options here, like a service called “Redis” or having the online-status persist in the database. We’ll implement something different using a dictionary. Note that this method is not scalable at all, it is meant for one server total.

We’ll start with a new class in the SignalR folder called PresenceTracker.cs, and in it we’ll create a Dictionary object that uses a <string> of the username as the key, and a List<string> as the connection id’s.

- ➔ private static readonly Dictionary<string, List<string>> OnlineUsers =
  new Dictionary<string, List<string>>();

Every time a user connects they get a unique connection id. We use a List<string> for those id’s because a user may connect via browser and then also via the phone, meaning they are connected by more than one device, necessitating more than one connection id. We want to account for this.

We’ll next create two methods to add or remove users from the Dictionary:

- ➔ public Task UserConnected(string username, string connectionId)
- ➔ public Task UserDisconnected(string username, string connectionId)

NOTE: Dictionary is not thread-safe

So we’ll “lock” the dictionary any time we want to add or remove an entry

To add a user to the dictionary…

- ➔ lock(OnlineUsers)
  {
  if (OnlineUsers.ContainsKey(username))
  {
  OnlineUsers[username].Add(connectionId);
  }

```
            else
            {
            OnlineUsers.Add(username, new List<string>{connectionId})
            }
        }
```

…and to remove the connectionId and / or the user…

```
➔ lock(OnlineUsers)
    {
            if (!OnlineUsers.ContainsKey(username)) return Task.CompletedTask;
            OnlineUsers[username].Remove(connectionId);
            if (OnlineUsers[username].Count == 0)
            {
            OnlineUsers.Remove(username);
            }
    }
```

After each method we can return Task.CompletedTask;

We want to be able to see all users who are online, so we'll add one more method called GetOnlineUsers():

```
➔ string[] onlineUsers;
    lock(OnlineUsers)
    {
            onlineUsers = OnlineUsers.OrderBy(k => k.Key).Select(k => k.Key).ToArray();
    }
    return Task.FromResult(onlineUsers);
```

This method returns a string[] of all connected users in order by username.

Now we need to tell the program about this class, so in the ApplicationServiceExtensions we'll add "**services.AddSingleton<PresenceTracker>();**"

Adding and Removing when a user Connects or Disconnects

Back in the PresenceHub we can now inject the PresenceTracker, and use its methods. In the OnConnectedAsync method we'll call the tracker.userConnected, and then refresh all online users for the client:

```
➔ public override async Task OnConnectedAsync()
    {
    await _tracker.UserConnected(Context.User.GetUsername(),
    Context.ConnectionId);
    await Clients.Others.SendAsync("UserIsOnline", Context.User.GetUsername());
    var currentUsers = await _tracker.GetOnlineUsers();
```

await Clients.All.SendAsync("GetOnlineUsers", currentUsers);
}

Likewise for the OnDisconnectedAsync method:

➔ public override async Task OnDisconnectedAsync(Exception? exception)
{
await _tracker.UserDisconnected(Context.User.GetUsername(), Context.ConnectionId);
await Clients.Others.SendAsync("UserIsOffline", Context.User.GetUsername());
var currentUsers = await _tracker.GetOnlineUsers();
await Clients.All.SendAsync("GetOnlineUsers", currentUsers);
await base.OnDisconnectedAsync(exception);
}

Now with the API taken care of, we can use these methods on the client side to show when users are online.

Displaying online Presence

Because we want to update whenever a user is on or offline we'll start with an observable

Create the Observable

In the presence.service.ts file we'll create an observable in the class properties:

private onlineUsersSource = new BehaviorSubject<string[]>([]);
onlineUsers$ = this.onlineUsersSource.asObservable();

This observable will stream the list of users that are online. To use this in the member card, we'll inject the presenceService as a public injection so that we can use it in our html component.

Now in the card html, where we have the user icon, we'll place the icon into a span that has a custom class and the class will only display if the member's username is included in the onlineUsers$ observable in the presenceService:

➔ <span **[class.is-online]="(presenseService.onlineUsers$ | async)?.includes(member.userName)"**>

Now for the style. In the css file we'll add an animation:

@keyframes fa-blink {

0% {opacity: 1;}
50% {opacity: 0.4;}

```
        100% {opacity: 1;}
    }
```

…and then the class properties…

```
    .is-online {
        animation: fa-blink 1s linear infinite;
        color: rgb(1, 189, 42);
    }
```

This should produce a green blinking user icon. We can do the exact same thing in the member.detail component and html. First inject a public presenceTracker so we can use the async pipe again. Then in the html, just below the main photo we'll add:

```
<div class="mb-2" *ngIf="(presenceService.onlineUsers$ |
async)?.includes(member.userName)">
<i class="class fa fa-user-circle text-success"> Online now!</i>
</div>
```

…to show a green user icon and text that reads "Online now!".

Adding a Message Hub

Unlike the presence hub, we don't need to send a message to all users, but just the user the message is intended for. For this, we can use "signalR groups". Let's first create a message hub class and inject the messageRepository.

We'll make methods similar to the presenceHub, with an OnConnectedAsync and an OnDisconnectedAsync.

When a user connects to the message hub, we need to get the username of the user they are messaging (the "otheruser") so we'll first get an httpContext and use it to get the other user via the query string. Then we'll make a group name for the two users that includes the usernames in alpha-order (via a private method), and then we'll create a group using that name and the connection id:

```
➔  var httpContext = Context.GetHttpContext();
   var otherUser = httpContext.Request.Query["user"];
   var groupName = GetGroupName(Context.User.GetUsername(), otherUser);
   await Groups.AddToGroupAsync(Context.ConnectionId, groupName);
```

Next we'll get the messages from the messageRepository, and then send the thread through the SignalR Group:

```
➔ var messages = await
   _messageRepository.GetMessageThread(Context.User.GetUsername(), otherUser);
➔ await Clients.Group(groupName).SendAsync("RecieveMessageThread", messages);
```

The GetGroupName method (referenced above) simply takes the two usernames and compares them via “CompareOrdinal” which returns <0 if stringA is less than stringB (lower in alpha) and >0 if the opposite.

➔ private string GetGroupName(string caller, string other)
{
var stringCompare = string.CompareOrdinal(caller, other) < 0;
return stringCompare ? $"{caller}-{other}" : $"{other}-{caller}";
}

Just like with the presenceHub we need to reference this hub in our program class:

➔ app.MapHub<MessageHub>("hubs/message");

Adding the Send Message Method to the Hub

We have a SendMessage method in the messagesRepository class, and most of that will work find for what we need to do in the hub, so we can copy that method straight over to the hub and just import or modify what we need to in order to get it working.

Firstly it will be

➔ public async Task SendMessage(CreateMessageDTO createMessageDto)

We don’t have access to the User in here but we can use Context.User.GetUsername() instead, so that will work where we need it.

Also, since this isn’t an API controller we don’t have access to http methods so we can’t return “Ok” or “Bad Request”, but in this case we can use Exceptions instead:

➔ throw new HubException("You cannot send messages to yourself");
➔ throw new HubException("Not found");

We’ll need to bring in the userRepository and AutoMapper so be sure to inject those in the constructor.

Finally, the last method where the message is sent from, we’ll modify to remove the http and instead use the “Clients.Group” from the hub:

```
if (await _messageRepository.SaveAllAsync())
        {
                var group = GetGroupName(sender.UserName, recipient.UserName);
                await Clients.Group(group).SendAsync("NewMessage",
                _mapper.Map<MessageDTO>(message));
        }
```

With that taken care of we should be good to start working in the client now.

Adding the Hub Connection to the Message Service

In the messageService on the client side we need to add the hub connection like we added the presence connection before. We'll start by adding the hubUrl and the HubConnection.

- ➔ hubUrl = environment.hubUrl;
- ➔ private hubConnection?: HubConnection;

We will also need an observable that we can subscribe to so we can update the components with new messages:

- ➔ private messageThreadSource = new BehaviorSubject<Message[]>([]);
- ➔ messageThread$ = this.messageThreadSource.asObservable();

Now we need to create the hub connection, so we'll do that and take in the User and a string for the other username. We'll build the hub connection with the hub url (to which we'll add 'message?user=' and the other username). Then we'll add the user token and build.

- ➔ this.hubConnection = new HubConnectionBuilder()
  .withUrl(this.hubUrl + 'message?user=' + otherUsername, {
  accessTokenFactory: () => user.token
  })
  .withAutomaticReconnect()
  .build();

Then we'll start the hub connection, and then call the method from the MessageHub in the API and subscribe to the observable messageThreadSource:

- ➔ this.hubConnection.start().catch(error => console.log(error));
- ➔ this.hubConnection.on('RecieveMessageThread', messages => {
  this.messageThreadSource.next(messages);

We also need to be able to stop the connection, so we'll add that method as well:

- ➔ stopHubConnection() {
  this.hubConnection?.stop();

Refactoring the Message Components to use the Hub

Our next step is to refactor the client side area where the messages get shown to the user. Since this is in our member detail page we'll start in that component. When the onTabActivated method is called and the "Messages" heading is passed, then we need to connect to the message hub, so we'll add a call to the messageService.createHubConnection() there. We need to pass the user and the the user that they are talking to to this method though, meaning we need access to the user object. So we'll add the following:

1. Class properties:
    a. user?: User;
2. Constructor injection:
    a. private accountService: AccountService

3. Constructor:
    a. this.accountService.currentUser$.pipe(take(1)).subscribe({
    b. next: user => {
    c. if (user) this.user = user;
4. onTabActivated:
    a. if (this.activeTab.heading === 'Messages' && this.user) {
    b. this.messageService.createHubConnection(this.user, this.member.userName);
    c. } else {
    d. this.messageService.stopHubConnection();

We also need to ensure we close the hub connection if we navigate away from the message tab, so we'll implement OnDestroy for the component and when it's called we will stop the hub connection:

➔ export class MemberDetailComponent implements OnInit, OnDestroy

…and then…

➔ ngOnDestroy(): void {
this.messageService.stopHubConnection();

NOTE: attempting to close a connection that is not open will result in a crash

We must code defensively, so in the messageService.ts, in the stopHubConnection method we'll check if the hubConnection exists first, then close it if it does:

➔ stopHubConnection() {
if (this.hubConnection) {
this.hubConnection.stop();

The member detail component needs to be adjusted as well. In the tabset in the html we are no longer getting the [messages] and passing them to the app-member-messages component, so we can remove that:

➔ <app-member-messages [messages]="messages" [username]="member.userName"></app-member-messages>

…becomes…

➔ <app-member-messages [username]="member.userName"></app-member-messages>

The member-messages component itself will now get the messages from SignalR using the messageService, so we'll ensure the injected service is public so that we can use it directly in the html, as well as remove the @Input() messages: Messag[] property since we won't be using it. Then in the html of the component we'll replace the 'messages' call in the conditionals to messageService.messageThread$ and use the async pipe:

➔ <div *ngIf="messages.length === 0">

…becomes…

➔ <div *ngIf="(messageService.messageThread$ | async)?.length === 0">

…and likewise for the second ngIf. For the ngFor we'll use the observable as well:

➔ <li *ngFor="let message of messages">

…becomes…

➔ <li *ngFor="let message of (messageService.messageThread$ | async)">

We can test at this point, and all should be well.

NOTE: method not found error: restart API

Tracking the Message Groups

Currently the message functionality is working as expected, except that when a user messages another user within the same group the message should be marked as "read" because they are both online and messaging back and forth. For this, we need an easy way to see that they are in the same group (same hub connection essentially), and to be able to see when one user leaves the group.

An easy and relatively scalable way to do this is to store group names and connection id's in a database. When a group is made, store it. When someone leaves the group, remove the group from the database. Easy enough (in theory).

We'll make some new entities to do this. Each entity will have an empty constructor to satisfy the requirements of Entity Framework.

Group entity	Connection entity

public class Group
{
public Group()
{
 	
}
 	
public Group(string name)
{
Name = name;
}
 	
[Key]
public string? Name { get; set; }
public ICollection<Connection>
Connections { get; set; } = new
	public class Connection
	{
	public Connection()
	{
	
	}
	
	public Connection(string connectionId,
	string username) {
	ConnectionId = connectionId;
	Username = username;
	}
	
	public string? ConnectionId { get; set;
	}

List<Connection>(); } 	public string? Username { get; set; } }

The group entity doesn't get an id, but we want the group Name to be the key, so we can tag it with [Key].

Add them in the DataContext class:

- ➔ public DbSet<Group> Groups { get; set; }
- ➔ public DbSet<Connection> Connections { get; set; }

Update the IMessageRepository with the following:

- ➔ void AddGroup(Group group);
- ➔ void RemoveConnection(Connection connection);
- ➔ Task<Connection> GetConnection(string connectionId);
- ➔ Task<Group> GetMessageGroup(string groupName);

…and then flesh out the methods in the messageRepository…

```
public void AddGroup(Group group)
        {
        _context.Groups.Add(group);
        }
```

```
public void RemoveConnection(Connection connection)
        {
        _context.Connections.Remove(connection);
        }
```

```
public async Task<Connection> GetConnection(string connectionId)
        {
        return await _context.Connections.FindAsync(connectionId);
        }
```

```
public async Task<Group> GetMessageGroup(string groupName)
        {
        return await _context.Groups
                .Include(x => x.Connections)
                .FirstOrDefaultAsync(x => x.Name == groupName);
        }
```

Update the MessageHub with Group Tracking

We want to use the above schema to determine if a user has read or not read a message based on whether they are on the messages tab or not.

So we'll start in the MessageHub and add two new private methods, AddToGroup and RemoveFromMessageGroup:

- ➔ private async Task<bool> AddToGroup(string groupName)
- ➔ private async Task RemoveFromMessageGroup()

The AddToGroup method will be tasked with adding the group and the connection to the Connections database. We'll do this by getting the group and the connection. If the group doesn't exist we'll create a new Group. We'll add the connection to the group, and then save the database:

➔
```
var group = await _messageRepository.GetMessageGroup(groupName);
var connection = new Connection(Context.ConnectionId, Context.User.GetUsername());

if (group == null)
{
        group = new Group(groupName);
        _messageRepository.AddGroup(group);
}

group.Connections.Add(connection);

return await _messageRepository.SaveAllAsync();
```

For the RemoveFromMessageGroup we'll get the connection id first, then just remove it from the database:

➔
```
var connection = await _messageRepository.GetConnection(Context.ConnectionId);
_messageRepository.RemoveConnection(connection);
await _messageRepository.SaveAllAsync();
```

Now in the OnConnectedAsync method in the MessageHub we can await AddToGroup(groupName) and add the newly created group to the database.

In the SendMessage method, we'll refactor some code to get the groupname, get the group, check that the recipient is connected to the group (basically if the group is in the database) and if so, set the "read" receipt to DateTime.UtcNow:

➔
```
var groupName = GetGroupName(sender.UserName, recipient.UserName);
var group = await _messageRepository.GetMessageGroup(groupName);
if (group.Connections.Any(x => x.Username == recipient.UserName))
{
        message.DateRead = DateTime.UtcNow;
}
```

One last thing, we need to remove the user if they disconnect from the group, so in the OnDisconnectedAsync we'll call await RemoveFromMessageGroup():

➔ public override **async** Task OnDisconnectedAsync(Exception? exception)
{
    await RemoveFromMessageGroup();
    await base.OnDisconnectedAsync(exception);
}

NOTE: reconnecting to the same group

As the api restarts over and over, the OnDisconnectedAsync method may not get called, meaning that group's will persist in the database. We want to set the Connections database to refresh when the server restarts, so in the Program.cs file we can do this with a simple SQL command:

➔ await context.Database.ExecuteSqlRawAsync("TRUNCATE TABLE [Connections]");

And with that, once a user navigates away from the messages tab, their messages will show as "unread" to the user that sent them, until the recipient returns to the messages tab.

Dealing with UTC Date Formats

Currently we deal with the weird times in the client, but that all changes now. We can use automapper on the API to specify how we want to format the dateTime object. In the AutoMapperProfiles we will add the following:

➔ CreateMap<DateTime, DateTime>().ConvertUsing(d => DateTime.SpecifyKind(d, DateTimeKind.Utc));

This will convert the times for pretty much all times that we're using in the app, except for any times that are nullable (like the messages being read or not). For that, we'll create a nullable map for times…

➔ CreateMap<DateTime?, DateTime?>().ConvertUsing(d => d.HasValue ?
    DateTime.SpecifyKind(d.Value, DateTimeKind.Utc) : null);

Now all times should display properly in the browser.

Notifying users when they receive a message

The presence tracker is already tracking whether a user is online and connected, so we can start there with a method to get the connectionIds for a user.

We'll start by getting all the connection ids for a user. This is for if they are connected to the app through multiple devices, we want to send them a notification to all devices within the app. Once we have all the ids associated with them we'll return that as a list of strings:

➔ public static Task<List<string>> GetConnectionsForUser(string username)
{
List<string> connectionIds;
lock (OnlineUsers)
{
connectionIds = OnlineUsers.GetValueOrDefault(username);
}
return Task.FromResult(connectionIds);
}

We still need to be able to send the message from the message hub though, while retaining access to the presence hub. We can inject the latter into the former in the constructor, as:

➔ IHubContext<PresenceHub> presenceHub

With that injected, we can go to our SendMessage method where we determine if the user is on the Messages tab and therefore reading the message in real time. After the 'if' statement that determines this, we can now add an 'else' that will get the connections and send the message to those instead. Essentially, if the user is not on the message tab but is still connected to the presence hub, we'll send them a toast notification.

➔ else
{
var connections = await
PresenceTracker.GetConnectionsForUser(recipient.UserName);
if (connections != null)
{
await
_presenceHub.Clients.Clients(connections).SendAsync("NewMessageReceived"
,
new {username = sender.UserName, knownAs = sender.KnownAs});
}
}

Now with the API taken care of we can adjust our code in the client to show a toast when a message is received. In the presence.service we need to inject our router so we have access to the routes (specifically the messages route). Then we'll add to the createHubConnection:

➔ this.hubConnection.on('NewMessageReceived', ({username, knownAs}) => {
this.toastr.info(knownAs + ' has sent you a new message! Click here to see it')
.onTap
.pipe(take(1))
.subscribe({

```
next: () => this.router.navigateByUrl('/members/' + username + '?tab=Messages')
})
})
```

The NewMessageReceived must match what we called it in our API. The ‘onTap’ method turns the toast into a link, and the observable allows us to navigate to the message tab.

An problem that arises from this is that when a user is viewing another user’s profile and they receive a notification for a message, the toast link takes them to the messages of the currently-being-viewed user, not the sending user, so we’ll fix that in the member-detail component.

We’ll inject the router again, and in the constructor we’ll add the following:

➔ this.router.routeReuseStrategy.shouldReuseRoute = () => false;

…which will take care of that problem.

Optimizing the Presence

Currently we are making a lot of API calls and queries. When we connect to the presence hub, we send a notification to all users that are connected, which is fine. However, we’re also getting a list of users and user “Client.All” and sending that list to every client that’s connected, including the current user, which doesn’t seem efficient. Really, only the currently connecting user (Caller) needs the list of all online users.

In the presence tracker, for the UserConnected and UserDisconnected methods, we can return a boolean of true or false to denote whether a user has connected or not. Then, in the PresenceHub we can receive that boolean and determine whether we need to let clients know that a user has connected or not.

1. public Task<bool> UserConnected(string username, string connectionId)
    a. bool isOnline = false;
    b. (else) OnlineUsers.Add(username, new List<string>{connectionId});
       isOnline = true;

2. return Task.FromResult(isOnline);

So we’ll declare a bool, set it to false, and only change it to true if a completely new user connects. If the user is already connected and they connect again through a different device, no notification will be sent.

For the UserDisconnected we’ll do something similar:

1. public Task<bool> UserDisconnected(string username, string connectionId)
    a. bool isOffline = false;

b. if (!OnlineUsers.ContainsKey(username)) return Task.FromResult(isOffline);
c. if (OnlineUsers[username].Count == 0)
{
OnlineUsers.Remove(username);
isOffline = true;
}

2. return Task.FromResult(isOffline);

Now in the presence hub we can change our await methods to store the returned bool, and then only SendAsync("UserIsOnline"...) if we get back the "true":

- var isOnline = await _tracker.UserConnected(Context.User.GetUsername(), Context.ConnectionId);
- if (isOnline)
- {
  - await Clients.Others.SendAsync("UserIsOnline", Context.User.GetUsername());
- }

…and in Disconnected…

- var isOffline = await _tracker.UserDisconnected(Context.User.GetUsername(), Context.ConnectionId);
- if (isOffline)
- {
  - await Clients.Others.SendAsync("UserIsOffline", Context.User.GetUsername());
- }

Now on the client side, we can go to the presence.service and remove the toasts from before and replace them with subscriptions to update the usernames in the onlineUsers$ observable:

- this.hubConnection.on('UserIsOnline', username => {
  this.onlineUsers$.pipe(take(1)).subscribe({
  next: usernames => this.onlineUsersSource.next([...usernames, username])
  })

…and…

- this.hubConnection.on('UserIsOffline', username => {
  this.onlineUsers$.pipe(take(1)).subscribe({
  next: usernames => this.onlineUsersSource.next(usernames.filter(x => x !== username))
  })

Finished Touches on Messages

Currently if a user switches between users they have messaged, the previous message board shows briefly. This is because the message array in the message.service does not reset right away, so we'll add that. Additionally we'll add loading while this happens.

In the message.service constructor, add "private busyService: BusyService" to access the loading screen. Then in the creatHubConnection add "this.busyService.busy()" to call it.

Below that, in the same method, where "this.hubConnection.start()" ends we'll make use of the "finally" clause in observables. After the "catch" line we can add

➔ .finally(() => this.busyService.idle());

This will reset the busy service back to idle when nothing is happening, after messages have been loaded.

Below this in the stopHubConnection, we can add

➔ this.messageThreadSource.next([]);

…to set the message thread to empty.

Adding a Spinner to the Send Message Button

Now to add a small loading icon in the component, we'll add a variable "loading" and set it to "false". Then in sendMessage we can add "this.loading-true" to set the loading spinner, then after the message has been sent we can stop the spinner with "finally(() => this.loading=false)"

➔ sendMessage() {

```
if (!this.username) return;
this.loading=true;
this.messageService.sendMessage(this.username, this.messageContent).then(()
=> {
this.messageForm?.reset();
}).finally(() => this.loading=false);
}
```

To show this spinner on the button when a message is being sent, we can change the code of the send button to be:

➔ <button [disabled]="!messageForm.valid || loading"

```
class="btn btn-primary" type="submit">
Send <i *ngIf="loading" class="fa fa-spinner fa-spin"></i>
</button>
```

…which will check if "loading" is true, and if it is, it will disable the button AND show the spinner from font-awesome.

While here, we can also fix the message box to show the input at the bottom of the page. We can just move the code

#scrollMe
style="overflow: scroll; height: 535px;"
[scrollTop]="scrollMe.scrollHeight"

… to be inside the div for the “card-body” so that the card body controls the scrolling and the message box is at the bottom of it.

Section 18: Unit of Work / Finished Touches

What is the Unit of Work?

Unit of Work pattern is a way of implementing an API with a transactional approach. Every request to the API is seen as a transaction, and the unit of work controller facilitates these transactions for us. Currently each of our controllers has a DataContext class injected into it, and each controller serves the caller in some way, and then the controller saves the changes. This isn’t the best approach, though, because with multiple dataContexts there could be multiple transactions happening at once, resulting in data inconsistency.

Entity framework tracks all database changes for us, so the Unit of Work pattern makes use of this by having only one class that contains a SaveChanges method, and all controllers are injected into that one class. The data requests are passed to this one class as parameters, the appropriate controller is called, and when control passes back to our one class then changes are finally saved.

Implementing the Unit of Work Pattern

First, you could say the dbcontext is an abstraction of the database (which it is), and we’ve created a further abstraction with our repository classes. This is the repository pattern This pattern of work is good because it allows us to test much easier, as well as keeping many important methods out of the controller classes related to the repositories.

The unit of work pattern is an abstraction of these abstractions. It works by acting as the main tunnel through which the repositories are accessed through, and it handles all of the transactions with the database and then saves them at the end (like the saveChanges method). This pattern waits for all transactions and then saves once, and if one transaction fails, they all get canceled and the database gets rolled back.

To implement this pattern, we’ll create a new Interface called IUnitOfWork that contains the following methods:

- ➔ IUserRepository UserRepository {get;}
- ➔ IMessageRepository MessageRepository {get;}
- ➔ ILikesRepository LikesRepository {get;}
- ➔ Task<bool> Complete();

- ➔ bool HasChanges();

Then we'll make a class called UnitOfWork (in the Data folder) that implements the above interface. We'll need to inject our DataContext and Mapper classes, and then use them to create instances of our repositories:

- ➔ public IUserRepository UserRepository => new UserRepository(_context, _mapper);
- ➔ public IMessageRepository MessageRepository => new MessageRepository(_context, _mapper);
- ➔ public ILikesRepository LikesRepository => new LikesRepository(_context);

Likewise the last two methods will be for saving the changes, and tracking the changes:

public async Task<bool> Complete()
{
return await
_context.SaveChangesAsync() > 0;
}
	public bool HasChanges()
	{
	return
	_context.ChangeTracker.HasChanges();
	}

In the applicationServiceExtensions class we can remove the services for UserRepository, MessageRepository, and LikesRepository, and instead add a service for our new UnitOfWork interface and class:

- ➔ services.AddScoped<IUnitOfWork, UnitOfWork>();

And now, to break the app!
Go to the IUserRepository and IMessageRepository and remove the "SaveAllAsync" methods. Then go to the corresponding classes and remove the implemented methods. VS Code should give fresh warnings about problems, so we'll fix those next.

Refactoring

We're now going to go to each controller one by one and refactor them to use the UnitOfWork class. We can first remove all Interfaces for the Repositories, and instead inject just one instance of the IUnitOfWork as "uow". This injection has access to all of the repositories anyways, so we can now use that instead:

- ➔ var likedUser = await _userRepository.GetUserByUsernameAsync(username);

…becomes…

- ➔ var likedUser = await _uow.UserRepository.GetUserByUsernameAsync(username);

Anywhere that we call the "saveAllAsync" method from the repositories (the one we removed already), we can replace that with "_uow.Complete()":

- ➔ if (await _uow.Complete()) return Ok();

NOTE: Shortcut: Shift+Ctl+L select all instances of highlighted text

One last thing… in the MessageRepository we still have an “await _context.SaveChangesAsync()” call in the GetMessagesThread method. We don’t want this, all responsibility of “saving” should be handled by the UnitOfWork class now. So we’ll remove that, but we still want to save around this time.

This method is called front he MessagesController, so we’ll remove that method from the controller first off because we aren’t going to use the API controller to get the thread anymore.

The MessageHub has a method that calls the MessageRepository method “GetMessageThread”
(mentioned above) and we still need to save the database after this call, but we removed the “_context.SaveChangesAsync()” already, so now what?

Easy. In the “OnConnectedAsync” method, where the GetMessagesThread is called, right after we return from that we’ll make use of our HasChanges method in the UnitOfWork class to check for changes, and then save them if they exist:

- ➔ if (_uow.HasChanges()) await _uow.Complete();

The wild part? Everything still works! Go test it!

Optimizing the Message Queries

Our queries to the database for messages is currently really big, but that’s because we are doing eager-loads on both the sender and recipient to get their photos. We don’t need to do this if we are using projection, though, we can refactor our GetMessageThread method in the MessageRepository to do this.

This method is returning a MessageDTO already, so projection is in place. We can change the “message” where we load all the stuff into a “query” instead. Then we’ll just project it to the DTO at the end:

Left → Right

var messages = await _context.Messages .Include(u => u.Sender).ThenInclude(p => p.Photos) .Include(u => u.Recipient).ThenInclude(p => p.Photos)	var query = _context.Messages
.ToListAsync();	.AsQueryable();
return _mapper.Map<IEnumerable<MessageDTO>>(	return await query.ProjectTo<MessageDTO>(_mapper.C

messages);	onfigurationProvider).ToListAsync();

So instead of storing all that stuff into the “messages” object and then returning all of the info as an IEnumerable, we’ll make a query that just gets the messages we need and then projects it into the MessageDTO object. The Message object has fields for all the stuff we need already, so there really wasn’t a point in eager loading all the stuff from before.

Optimizing Querying the List of Users

The query for getting all users by gender is very big currently, due to lots of stuff being fetched and a JOIN to the photos table. This query originates in the UsersController method “GetUsers”:

- ➔ var currentUser = await _uow.UserRepository.GetUserByUsernameAsync(User.GetUsername());
- ➔ userParams.CurrentUsername = currentUser.UserName;

But all we need from this is the user’s gender. We can get the username from the token, and then all we need is the user’s gender, so we can make a method that does just that and nothing else. This should help make the query much smaller.

In the IUserRepository we’ll add a new method, GetUserGender:

- ➔ Task<string> GetUserGender(string username);

In the UserRepository we’ll add the new method and define it:

- ➔ public async Task<string> GetUserGender(string username)
  {
  return await _context.Users
  .Where(x => x.UserName == username)
  .Select(x => x.Gender).FirstOrDefaultAsync();
  }

This will take the username we’ll get from the token, pass it to this method, then return the user’s gender. Then in the UsersController we’ll call the method in the GetUsers method:

- ➔ var gender = await _uow.UserRepository.GetUserGender(User.GetUsername());
- ➔ userParams.CurrentUsername = User.GetUsername();

Then in the proceeding if-statement we can just use the gender we get back for the comparison:

- ➔ userParams.Gender = currentUser.Gender == "male" ? "female" : "male";
  …becomes…
- ➔ userParams.Gender = gender == "male" ? "female" : "male";

Much cleaner and smaller!

Adding a Confirmation Service

Okay, just like any other time, if we want to do a thing, we need to add a service to do it. Here we want to add a modal to confirm navigation away from the “edit-profile” page, so we’ll create a service called “confirm” to do that. We’ll also need a component we’ll call “confirm-dialog”.

In the service we need a class property for BsModalRef<ConfirmDialogComponent>. We’ll also need to inject our BsModalService in the constructor. Then we’ll create a method called “confirm” that will get passed a bunch of strings, but we’ll load them with default values too:

➔ confirm(
title = 'Confirmation',
message = 'Are you sure you want to do that?',
btnOkText = 'Ok',
btnCancelText = 'Cancel'
)

The actual method body will create an initialState to pass to the modal when it’s built:

➔ const config = {
initialState: {
title,
message,
btnOkText,
btnCancelText
}

Then we’ll show the modal:

➔ this.bsModalRef = this.modalService.show(ConfirmDialogComponent, config)

In the modal component we’ll have all the same strings from above as class properties, with one extra called “result” set to “false”:

```
title = '';
message = '';
btnOkText = '';
btnCancelText = '';
result = false;
```

We’ll inject the BsModalRef, and then have two methods, one called “confirm” and the other called “decline”. They will be the same except confirm will set “result” to “true”:

```
confirm() {
    this.result = true;
    this.bsModalRef.hide();
```

```
decline() {
    this.bsModalRef.hide();
}
```

}	

Finally, we'll build the modal in the html:

```
<div class="modal-header">
        <h4 class="modal-title pull-left">{{title}}</h4>
</div>
<div class="modal-body">
        <p>{{message}}</p>
</div>
<div class="modal-footer">
        <button class="btn btn-success" (click)="confirm()">{{btnOkText}}</button>
        <button class="btn btn-danger" (click)="decline()">{{btnCancelText}}</button>
</div>
```

Note how we use interpolation to display the values that are passed to the modal.

Implementing into the Guard

In the confirm.service method "confirm" we want to return a boolean observable, so we'll be explicit and note that in the method signature:

```
➔ confirm(
  title = 'Confirmation',
  message = 'Are you sure you want to do that?',
  btnOkText = 'Ok',
  btnCancelText = 'Cancel'

  ): Observable<boolean> {
```

To return the observable, at the end of the method we'll add the following, in which we call the pipe method, map from rxjs, and return the result:

```
➔ return this.bsModalRef.onHidden!.pipe(
  map(() => {
  return this.bsModalRef!.content!.result
        })
  )
```

Finally, we'll go to our guard "prevent-unsaved-changes" and add a constructor to inject the confirmService:

```
➔ constructor(private confirmService: ConfirmService) {}
```

Then in the canDeactivate method we'll change the return type from "boolean" to "Observable<boolean>". Then instead of "return confirm()" we'll return the the confirm method from the confirmService (which will be an observable):

➔ return this.confirmService.confirm()

We also need to add “return of(true)” at the end in the event we get that far, in order to satisfy the return type of Observable.

Adding Scrolling to the Messages

This one is easy to implement, and we want to because the messages are getting quite long and will soon take too long to scroll to the bottom of the whole page to get to.

In the member-messages html, in the <ul> for the messages, we’ll give it an id of #scrollMe, which we’ll refer back to in a minute. Then just below that we’ll use a style=”overflow: scroll; height: 500px;”, followed by a [scrollTop]=”scrollMe.scrollHeight”. With this in place we can scroll our messages easily.

To quash any potential errors we’ll get in the console, in the component we can add:
➔ changeDetection: ChangeDetectionStrategy.OnPush,
…which will keep an Angular onChange error from cropping up.

Finally, when a user logs out we want to clear their info from the log-in fields, and we can do that in the login() method in the nav.component. After the this.router we can add this.model = {};, which loads the model with an empty object, leaving nothing to display in the inputs when we log out.

Section 19: Publishing

Preparing the App for publishing

In order to prepare the Angular side of the app for publishing we first need to replace all hard-coded url’s that start with “localhost”. Do a quick search in the ‘client’ folder for “localhost” and open any files that have it.

Replace ‘localhost’ with ‘environment.apiUrl’, from the “environment” file we set up earlier.

Next, in the Angluar.js file we need to specify where we want to place the built files. In the ‘build’ section we’ll specify the “outputpath” to be “../API/wwwroot”, which will be created when generating the production build. This will be where the build files are stored so that the API can serve them.

While in this file we can go to “budgets”, “initial”, and in “max warning / error” reset the values to be slightly higher (1mb and 2mb respectively). This will stop any errors when building in the next steps.

In the API Program.cs file, we also need to add

➔ app.UseDefaultFiles();

…and…

➔ app.UseStaticFiles();

…which will tell the API to look inside the default folder (wwwroot) and serve the content from inside there. Specifically, it wilĺ look for “index.html” (the default) and show that.

NOTE: Placement of the above files

These both need to be AFTER “UseAuthorization” and BEFORE “MapControllers”.

In the terminal use “ng build” and the application will build a new set of compressed files in the folder specified above (wwwroot). Now we can test our app by running the api (dotnet watch –no-hot-reload) and going to “https://localhost:5001”. The app will load with no issues!

However, after navigating to the “members” page and refreshing we are met with an error. This is due to routing, and we’ll tackle that next.

Angular Fallback Controller

We need to tell the API to fallback to the index when it doesn’t have any knowledge of the route it needs to take. To do this we’ll create a new controller called “FallbackController”, which will derive from “Controller” (NOT our baseAPIController), and in it we’ll have one single method:

➔
```
public ActionResult Index()
{
return PhysicalFile(Path.Combine(Directory.GetCurrentDirectory(),
        "wwwroot", "index.html"), "text/HTML");
}
```

This method tells the API to go to the wwwroot directory and find the HTML/text file “index” and run it.

Now in the Program.cs file we need to inform the API of this new controller, so we will add

➔ app.MapFallbackToController("Index", "Fallback");

… after the other Mappings.

Now we can reload the API and test it by refreshing the page… success!

Removing the Loading Delay

Remember when we added an artificial delay so we could see our loading animation? We're going to remove that delay now.

In the "loading.interceptor" file we want to replace
- ➔ delay(1000)

…with…
- ➔ (environment.production ? identity : delay(1000))

…and that's it. Rebuild the application and test.

Switching the DB Server to PostGres

PostGres is a better SQL server for production, so we'll convert our app to that from the currently used SQL Server (or SQL Lite, whatever).

To do this, we need to first install Docker (install docker desktop, it's easiest). With Docker installed, open a command terminal and navigate to the App's main folder (our app, not Dockers) and run "docker" to ensure it is installed correctly (we'll see a list of commands to use).

With docker installed correctly, we'll create a docker image that contains postgres with the parameters that we want to have in it, so we can use the following command to do that:
> docker run –name postgres -e POSTGRES_PASSWORD=postgrespw -p 5432:5432 -d postgres:latest

This will create the container with the latest version of postgres. It will run in detached mode (-d) and have a password of "postgrespw". I will listen on post 5432 and receive on the same port.

NOTE: don't make the password complex

Docker may change it automatically because it doesn't like complex passwords. This will make issues later, so keep it simple.

In docker desktop we should now see a container running called "postgres".

In VS Code, we'll also install the postgres extension by Chris Kolkman. It will make an icon on the right for use to access the database once it's connected, which we'll do next.

Changing the DB Server

In nuget gallery install Npgsql.EntityFrameworkCore.PostgresSQL by Shay and Austin. With that installed, we can prepare our app to use Postgres instead.

First we need to change the connection string in appsettings.Development.json:

- ➔ "Server=localhost; Port=5432; User Id=postgres; Password=postgrespw; Database=datinapp"

Double check spelling and syntax

Next, in the Ap[licationServiceExtensions file we need to remove the mention of SqlServer / SqLite, and replace it with “UseNpgsql” (which is now available bc of the nuget package we installed).

We can now drop the database (dotnet ef database drop) and delete the entire migrations folder. We’ll make a new migration (dotnet ef migrations add PostGresInitial -o Data/Migrations) and a new folder with new migrations will be created.

Now we’re not ready to run the app yet; there are a few things we need to fix in order for Postgres to work for us:

In the Program class, we need to remove the line “ExecuteRawSql()” because 1. It contains square brackets that psotgres doesn’t like, and 2. It’s really not all that great of practice to do this to the database. We’ll instead do this in the seed class:

First, add this method:

- ➔ public static async Task ClearConnections(DataContext context)
  {
  context.Connections.RemoveRange(context.Connections);
  await context.SaveChangesAsync();
  }

Next, in the Program class change:
await context.Database.ExecuteSqlRawAsync("TRUNCATE TABLE [Connections]");
…to…

- ➔ await Seed.ClearConnections(context);

NOTE: this is not as efficient as the original method

But still better than executing raw SQL. If there’s a lot of rows to remove at app startup, this will take some time.

One more issue to resolve, if we try to run we’ll get an error referring to not using UTC time somewhere, it’s in the Seed class where we are creating the users. We need to add the following lines:

- ➔ user.Created = DateTime.SpecifyKind(user.Created, DateTimeKind.Utc);
- ➔ user.LastActive = DateTime.SpecifyKind(user.LastActive, DateTimeKind.Utc);

… just after the line where we set the user.Username.ToLower().

NOTE: Error with Postgres and legacy

After completing the above steps and dropping the database and attempting to run the app, you may get an error that says:

- ★ Cannot write DateTime with Kind=UTC to PostgreSQL type 'timestamp without time zone'

In order to fix this issue, we need to enable ‘Legacy TimeStamp Behavior’ for Npgsql. To do this we:

1. Create a new class in the Data folder (or wherever the DataContext / DBContext class is)
2. Call the file “MyModuleInitializer”
3. Add the following code:
    a.
```
public static class MyModuleInitializer
{
        [ModuleInitializer]
        public static void Initialize()
        {
        AppContext.SetSwitch("Npgsql.EnableLegacyTimestampBehavior", true);
        }
}
```

After dropping the database again and restarting the app (with the postgres Docker container running) you should see the app run as normal.

Dockerizing the App

To get the app into Docker, we first need the Docker extension (from microsoft) so download and enable. Then in our API folder we’ll make two new files:

1. Dockerfile
2. dockerignore

In the Dockerfile, we’ll give the instructions to build the container. We need to tell it to get the dotnet sdk, run some commands from it, then get a slimmer version of the sdk to install into the container, and then build it. The commands look like:

```
FROM mcr.microsoft.com/dotnet/sdk:7.0 AS build-env
WORKDIR /app

# copy csproj and restore as dictinct layers
COPY *.csproj ./
RUN dotnet restore

# copy everything else and build
COPY . ./
RUN dotnet publish -c Release -o out

# build runtime image
```

```
# dotnet sdk is needed for 'restore' and 'publish' but is too large for the final image
FROM mcr.microsoft.com/dotnet/aspnet:7.0
WORKDIR /app
COPY --from=build-env /app/out .
ENTRYPOINT [ "dotnet", "API.dll" ]
```

Once this is done, we want to add two lines to the ‘dockerignore’ file:

- **/bin
- **/obj

We don’t need to copy these things, we’ll build them fresh when we make the container.

Now in the terminal, we can build the container:

- docker build -t [docker_user_name]/[app_name] .

Don’t forget the ‘.’ at the end, else you’ll get an error. The first build takes a while, but once it’s done the container should appear in the extension’s container list. You can run it with:

- docker run –rm -it -p 8080:80 [docker_username]/[app_name]:latest

However, you’ll probably get an error like this:

- System.ArgumentException: Host can't be null

…so let’s fix that.

Updating the Config to use Postgres

The hosting environment is in “production” (per the info when the container is run, even with the error). We need to change the configuration (temporarily) to see Docker working locally.

Start by taking the “ConnectionStrings” settings from appsettings.Development.json and pasting it into the appsettings.json file, below the “API Secret”. Change the “Server=localhost” to “Server=host.docker.internal”

NOTE: We’ll change this later and do something different for production

Now rebuild the docker image using the command from above. Try running again, and see if it works.

NOTE: drop the database again, to replicate deployment

We should see the data being inserted into the database now. We’ll get a couple warnings too regarding protected data being destroyed when the container is destroyed. This is okay though, we don’t need to persist the data that it’s referencing.

Now navigate to http://localhost:8080 and see if it works. Does it? Success!

Let’s push the image to Docker hub (online) using (in the terminal)

➔ docker push [docker_username]/[app_name]:latest

…and watch it go. Check it out on docker.com at your repository tab.

NOTE: if an error appears about authentication, use
- ➔ docker login

…and fill out the requested information. Then retry the push.

Creating the Config Files for Fly.io

In the terminal, cd out of the API directory and move up to the main app directory. There use the command:
- ➔ powershell -Command "iwr https://fly.io/install.ps1 -useb | iex"

flyctl will install. Check by typing ‘flyctl’ or ‘flyctl –help’

NOTE: Restart the computer if ‘fly’ or ‘flyctl’ does not work in the command line

Now we can use ‘fly launch –image [docker_username]/[app_name]:latest’
Answer the questions / give the app a name. Choose a server closest to you. Yes to setting up the postgres db, and no to scaling to zero. Choose the “development” machine at the choices.

Note the connection string for future reference:
Connection string:
postgres://postgres:zQMGNWu1q9zyuRH@lorenarms-datingapp-db.flycast:5432

Don’t create a dockerignore file (we already have one) and don’t deploy *yet*.

Once all that is done there will be a ‘fly.toml’ file in the app folder, open that up and take a look. First ensure the image name is correct. Additionally check the internal_port is ‘8080’. Go to the Dockerfile and under the first WORKDIR add:
- ➔ EXPOSE 8080

Next, in the fly.toml file add a section called [env] under the image name. In that section we’ll add:
- ➔ ASPNETCORE_URLS="http://+:8080"
- ➔ CloudinarySettings__CloudName="[coud_name]"
- ➔ CloudinarySettings__ApiKey="[api_key]"

Get the above info from the appsetting.json file.

We need to add the secret to this as well, but we do that in a special way so it’s not seen in the file. In the terminal type “fly secrets list” and a list of one item (the database url) should show up. We’ll add the api secret for cloudinary to this list as well, and also our token key.

Use “fly secrets set” followed by the key you want to set (name=key):
- ➔ Example: fly secrets set TokenKey=DUvyO9PM265jp3g8MTRA6jXvk3emIj0BXOtVBvK9aJFH1Gezs0ZBiLdJjSLol7qQ

No quotes!

NOTE: the TokenKey MUST be of sufficient length!

The video recommends using a 32 character key, but this will not be enough to generate the SHA512 key necessary to persist login. Instead, use a 64 character key (no special characters). If a shorter key was initially used and an error of:

- ★ Error: IDX10720: Unable to create KeyedHashAlgorithm for algorithm

… happens, simply use the above command (fly secrets set TokenKey=...) in the terminal. Fly should automatically update the re-deploy the app.

For the above (token key) we can set whatever random string we want to use. Just no special characters (letters and numbers only).

When the app is built and deployed, the toml file will be read, not the appsettings.json file.

Creating the Config Variables for Fly.io

In the ApplicationServiceExtensions file we'll remove

- ➔ services.AddDbContext<DataContext>(options =>
  {
  options.UseNpgsql(config.GetConnectionString("DefaultConnection"));
  });

… because we don't want to use this for deployment, only for local testing. In the Program.cs file we'll add a long script to check where the app is running first, and then use the appropriate database configuration:

```
var connString = "";
if (builder.Environment.IsDevelopment())
        connString = builder.Configuration.GetConnectionString("DefaultConnection");
else
{
// Use connection string provided at runtime by FlyIO.
        var connUrl = Environment.GetEnvironmentVariable("DATABASE_URL");

        // Parse connection URL to connection string for Npgsql
        connUrl = connUrl.Replace("postgres://", string.Empty);
        var pgUserPass = connUrl.Split("@")[0];
        var pgHostPortDb = connUrl.Split("@")[1];
        var pgHostPort = pgHostPortDb.Split("/")[0];
        var pgDb = pgHostPortDb.Split("/")[1];
        var pgUser = pgUserPass.Split(":")[0];
        var pgPass = pgUserPass.Split(":")[1];
```

```
        var pgHost = pgHostPort.Split(":")[0];
        var pgPort = pgHostPort.Split(":")[1];
        var updatedHost = pgHost.Replace("flycast", "internal");

        connString = $"Server={updatedHost};Port={pgPort};User
Id={pgUser};Password={pgPass};Database={pgDb};";
}
builder.Services.AddDbContext<DataContext>(opt =>
{
        opt.UseNpgsql(connString);
});
```

The above generates the appropriate connection string based on where the app is running.

Then we need to rebuild the Docker image (make sure docker desktop is running). Then re-push the image back up to Docker hub

Deploy the App

Making Changes and Redeployment

Any time changes are made, in order for the changes to take effect the app needs to be redeployed to fly.io.

NOTE: test all changes in local development first

To redeploy, follow these steps:

1. ng build (from client folder)
2. docker build (API folder)
3. docker push (API folder)
4. fly deploy (App main folder)