# Angular Basics

## Important Angular classes

app. module.ts

bootstrap: [AppComponent]

Root of all components for this application. All components roll up here

imports: [

[other, modules] to make them globally usable throughout the application

component - Provides data for the view templates

package. json - Manages dependencies + versions. One level higher than src

angular. json - The app configuration

node\_modules - 100's of listed dependencies listed out

webpack -Compiles code to JS + injects JS into index.html (5 files)

pollyfills. Ts – How you resolve unsupported browser issues. These load before the app and allow your Angular application to work with and display older browser versions that would not otherwise be supported. Just uncomment the stock browser stuff

# Angular 3rd party libraries

Wrapping 3rd party libraries as an Angular service.

## How to install bootstrap

npm install bootstrap font-awesome (installs both)

src>styles.css

@import "../node\_modules/bootstrap/dist/css/bootstrap.min.css";

@import "../node\_modules/font-awesome/css/font-awesome.min.css";

## Alertify

Alertifyjs.com - Notifications (success/failures) from browser

npm install alertifyjs DatingApp > spa

spa > src > styles

/\* You can add global styles to this file, and also import other style files \*/

@import "../node\_modules/bootstrap/dist/css/bootstrap.min.css";

@import "../node\_modules/font-awesome/css/font-awesome.min.css";

@import "../node\_modules/alertifyjs/build/css/alertify.min.css";

@import "../node\_modules/alertifyjs/build/css/themes/bootstrap.min.css";

@import "../node\_modules/bootswatch/dist/flatly/bootstrap.min.css";

ng g service alertify spa> src > app > \_services

spa> src > app > \_services> alertify.service.ts

import {Injectable} from '@angular/core';

import \* as alertify from 'alertifyjs';

@Injectable ({

  provided In: 'root'

})

export class AlertifyService {

  constructor () {}

  confirm (message: string, OkCallback: () => any) {

    alertify. Confirm (message, (evnt: any) => {

      if (evnt) OkCallback ();

//ok=if user clicks ok -> callback ...define in components

      else {} //Do nothing, cxl the callback, cxl the confirm

    });

  }

//Pass hard coded value from a component while calling this function

  success (message: string) {

    alertify. success(message);

//Assume you have injected this class into constructor

  }

if stream. subscribe(next=> {this.alertify.xxx ("pass hard coded msg");

  error (message: string) {

error => {this. alertify. error(error) ...pass error from stream

    alertify. error(message);

  }

  warning (message: string) {

    alertify. warning(message);

  }

  message (message: string) {

    alertify. message(message);

  }

}

spa> src > new file declare module 'alertifyjs'

spa> src > typings.d.ts

declare module 'alertifyjs'

Add “Alertify to the app config spa> tsconfig.app.json

spa> tsconfig.app.json

{

  "extends": "./tsconfig.base.json",

  "compilerOptions": {

    "outDir": "./out-tsc/app",

    "types": []

  },

  "files": [

    "src/main.ts",

    "src/polyfills.ts"

  ],

  "include": [

    "src/\*\*/\*.d.ts"

  ]

}

Example of using “Alertify” being used in a component

src > app > nav> nav.component.ts

import {AlertifyService} from "../\_services/alertify.service";

export class NavComponent implements OnInit {

  model: any = {};

  constructor (public authService: AuthService, private alertify: AlertifyService,

    private router: Router) {}

  ngOnInit (): void {

  }

  Login (): void {

    this. authService.login (this. model)

      . subscribe (

        next => {

//if successful, pass a success message to the browser

          this. alertify. success ("Logged in successfully"); },

        error => {this. alertify. error(error);}, If error, pass the error

        () => {this. router. navigate(['/members']);}

//alternate use complete (): void => after success req/resp

//this. router. navigate(['/members']) next=Req is successful + Resp has data from api

                                        route to this Url after all successful req/resp

      );

  }

## Angular JWT - Token helpers –

3rd party library to help manage tokens <https://github.com/auth0/angular2-jwt> … see usage: Injection

Jwt token helpers – decode the incoming token. Once decoded, you can see all the properties in the browser console. You can then use these token properties to use in your methods to enact logic

I.E. decodedToken .unique\_name, claims information, tokenDescriptor

-> this. decodedToken .exp , this.decodedToken.iat, this.decodedToken.nbf, this.decodedToken.nameid

npm install @auth0/angular-jwt

src > app.module.ts

import {JwtModule} from "@auth0/angular-jwt";

imports: [

    JwtModule.forRoot({

      config: {

        tokenGetter: tokenGetter,

        allowedDomains: ["localhost:5000"],

        //disallowedRoutes: ["localhost:5000/api/auth"],

      }

    })

  ],

In the example below. The helper service is being used in a “service” that is getting a token from the server

import {JwtHelperService} from "@auth0/angular-jwt"

@Injectable({

  providedIn: 'root'

})

export class AuthService {

  baseUrl: string = environment. apiUrl + "auth/";

  jwtHelper = new JwtHelperService();

  decodedToken: any;

  constructor (private http: HttpClient) {}

  login (model: any): Observable<void> {

    return this.http.post (this. baseUrl + "login", model)

      . pipe(

        map((response: any) => {

//incoming resp is a bearer (token: "token string")

          const user = response; //user = this (k,V)

          if (user) {//set token in same format on the browser local storage

            localStorage.setItem("token", user. token);

            this. decodedToken = this. jwtHelper. decodeToken (user. token);

//decode (v) token string and hold as var

            console.log (this. decodedToken);

//to see the fields of jwt decode method in browser

            /\* console. log("user: " + this.decodedToken.unique\_name);    --> fields comes from authController ...claims + tokenDescriptor

             this.decodedToken.exp , this.decodedToken.iat, this.decodedToken.nbf, this.decodedToken.nameid\*/

          }

        }) //--map transform one-at-a-time

      ); //--pipe transforms to Observable

  }

## NGX Bootstrap – w/ dropdowns & tabs

3rd party library to integrate bootstrap with Angular. Best practice is not to use jQuery but, pure Angular and integrate Angular based libraries.

<https://valor-software.com/ngx-bootstrap/#/dropdowns>

<https://valor-software.com/ngx-bootstrap/#/tabs>

ng add ngx-bootstrap DatingApp>spa

ng add ngx-bootstrap --component tabs

src > app.module.ts

import {BrowserAnimationsModule} from '@angular/platform-browser/animations'

import {BsDropdownModule} from 'ngx-bootstrap/dropdown';

import {TabsModule} from "ngx-bootstrap/tabs";

imports: [

    BrowserAnimationsModule,

    TabsModule. forRoot(),

    BsDropdownModule. forRoot(),

-After installed, just add some NGX specific key words to elements to use features.

<a>item1</> just groups these items that will be shown as drop-down options

<a>item2</>

src > app>nav>nav.component.html

<!--Drop Down menu will display if the User is logged in ...using NGXBootstrap-->

--<div “dropdown”> is the keyword needed to surround the elements indicating a dropdown

<div class="dropdown" \*ngIf="loggedIn ()" dropdown>

--<a “dropdownToggle”> key word what you click on to display dropdown options

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

                Welcome {{authService.decodedToken?.unique\_name | titlecase}}

            </a>

<div \*dropdownMenu> The inner items are <a> link as drop-down options.

\* this is a structural directive that changes the DOM directly

            <div class="dropdown-menu mt-3" \*dropdownMenu>

                <a class="dropdown-item" href="#"><i class="fa fa-user"></i> Edit Profile</a>

                <div class="dropdown-divider"></div>

                <a class="dropdown-item" (click)="logout ()"><i class="fa fa-sign-out"></i> Logout</a>

            </div>

src > app>members>member-details.ts

<! – Example of NGX bootstrap tabs -->

    <div class="col-sm-8">

      <div class="tab-panel">

        <tabset class="member-tabset">

          <tab heading="About {{user. knownAs}}">

            <h4>Description</h4>

            <p>{{user?. introduction}}</p>

</tab>

<tab heading="Messages">

            <p>Messages will go here</p>

          </tab>

        </tabset>

## ngx gallery

<https://www.npmjs.com/package/ngx-gallery>

Use this for install angular 8+ <https://github.com/kolkov/ngx-gallery>

npm install @kolkov/ngx-gallery –save

This is used to make photo galleries. See a bunch of organized pictures on a page that you can scroll, select

enlarge etc... the "Usage" section shows you how to paste data into your component + ngOnInit + html

import {NgxGalleryModule} from '@kolkov/ngx-gallery';

import { NgxGalleryOptions, NgxGalleryImage, NgxGalleryAnimation } from 'ngx-gallery';

@Component({

  selector: 'app-member-detail',

  templateUrl: './member-detail.component.html',

  styleUrls: ['./member-detail.component.css']

})

export class MemberDetailComponent implements OnInit {

  user: User;

  galleryOptions: NgxGalleryOptions[];

  galleryImages: NgxGalleryImage[];

  constructor(private userService: UserService, private alertify: AlertifyService,

    private route: ActivatedRoute) {}

  ngOnInit() {

    //Route Resolver

    this.route

      .data.subscribe(data => { this.user = data['user'] })

    this.galleryOptions = [

      {

        width: '500px',

        height: '500px',

        imagePercent: 100,

        thumbnailsColumns: 4,

        imageAnimation: NgxGalleryAnimation.Slide,

        preview: false

      }

    ];

    this.galleryImages = this.getImages();

  }

  getImages() {

    const imageUrls = [];

    for (const photo of this.user.photos) {

      imageUrls.push({

        small: photo.url,

        medium: photo.url,

        big: photo.url,

        description: photo.description

      })

    }

    return imageUrls;

  }

}

## Bootswatch –

Change theme of design free (through Bootstrap) https://bootswatch.com/help/

npm install bootswatch DatingApp>spa

spa>node\_modules These are the project dependencies-> will show "bootswatch"

spa>app>src>styles.css @import "../node\_modules/bootswatch/dist/flatly/bootstrap.min.css";

<> means check out the code. Only one change was made

<nav class="navbar navbar-expand-lg navbar-dark bg-primary"> changed to bg-primary to match theme better

## NGX bootstrap Date Picker

### Set-up

1. Get Started > Datepicker <https://valor-software.com/ngx-bootstrap/#/datepicker>

// RECOMMENDED app>app. module.ts

2a. import {BsDatepickerModule} from 'ngx-bootstrap/datepicker';

2b. import {BrowserAnimationsModule} from '@angular/platform-browser/animations';

import {BrowserAnimationsModule} from '@angular/platform-browser/animations'

import {BsDatepickerModule} from 'ngx-bootstrap/datepicker';

}

@NgModule ({

  declarations: [

…

  ],

  imports: [

    BrowserAnimationsModule,

    BsDatepickerModule.forRoot(),

3. Load CSS file as import node\_modules/ngx-bootstrap/datepicker/bs-datepicker.css

src>style.css

@import "../node\_modules/ngx-bootstrap/datepicker/bs-datepicker.css";

### Adding Datepicker to the component

In this scenario we are adding the datepicker to a DOB field within our registration component.

1. Bring in instance of Bootstrap datepicker Configuration

BsDatepickerConfig -> peek Definition. In looking at the class, a lot of fields are required but, we only want to deal with a single field (containerClass: string;) which is used to change the color theme of the datepicker container.

2. Use Partial Class to configure the BsDatepickerConfig

**Partial Class: Partial<…>** Surrounding a class makes all fields including the required fields optional. In this scenario we only want to implement a single property of the BsDatepickerConfig class .

type Partial<T> = { [P in keyof T]?: T[P]; }

import {BsDatepickerConfig} from 'ngx-bootstrap/datepicker';

export class RegisterComponent implements OnInit {

  model: any = {};

  registerForm: FormGroup;

  bsConfig: Partial<BsDatepickerConfig>;

ngOnInit (): void {

    this. bsConfig = { containerClass: "theme-red" }

3. Add bsDatepicker directive + config for the color

src>app>register>register.component.html

<div class="form-group">

    <input type="text" bsDatepicker [bsConfig] = bsConfig

[ngClass]="{'is-invalid’: registerForm. get('dateOfBirth'). errors &&

      registerForm. get('userName'). touched}" class="form-control" placeholder="Username" formControlName="dateOfBirth">

  </div>

## Ngx Paging

<https://valor-software.com/ngx-bootstrap/#/> -> documentation -> Paging … Read the Usage to know how to import and look through examples to see which one would look good on the site. We have already loaded ngx bootstrap so, we do not need to re-load through the CLI. Rather, just add needed imports at the app root level

### Set-up

Usage:

// RECOMMENDED … This is a module so, add “app. module. ts” imports

import {PaginationModule} from 'ngx-bootstrap/pagination';

@NgModule ({

imports: [PaginationModule. ForRoot(),]

})

export class AppModule () {}

Custom links content [](https://valor-software.com/ngx-bootstrap/) The html will display the numbered paging wherever pastes. Paste below the member-list. component so that, when the Client sees all profiles pictures. This will be displayed below the pictures



* [template](javascript:void(0);)

1. <pagination [boundaryLinks]="true" [totalItems]="77"
2. previousText="&lsaquo;" nextText="&rsaquo;" firstText="&laquo;" lastText="&raquo;">
4. </pagination>

### Connect template to component

The header from the Api response has our paging information. Before the component is loaded, the Url path is being captured by the “route resolver”. We are accessing this route, which has been Activated (Client has selected), which was caught be the resolver. When we access the “ActivatedRoute” we can subscribe (use the resolver’s observable who made the Api call and has the flow of data) to get the paging information.

The Api call made by the resolver is returning a PaginatedResult<User[]>. Remember, model. Pagination was a class. ts that we made to hold all the header + body of an api call from getUsers (result: T = body, pagination: Pagination = headers). The component tied to this template needs those fields to populate the numbered paging boxes.

export class MemberListComponent implements OnInit {

  users: User []; <--Has all body content Users []

  pagination: Pagination; <--Has all header fields

  constructor (private service: UserService,

    private alertify: AlertifyService, private route: ActivatedRoute) {}

  ngOnInit (): void {

    this. route. data. subscribe (data => { --> PaginatedResult<User[]>

      this. users = data['users'].result;

      this. pagination = data['users'].pagination;

    })

  }

# Cloudinary 3rd party storage

3rd party storage app used as a proxy between our server/api and Client. Data is stored on the cloud and returns headers with Url link. Client fetches Url from the cloud provider not our service. Api Key & Api Secret = Only we can upload photos. This allows us to authenticate the user before the photo is actually uploaded

1. Client uploads Photo to Api with Jwt --> 2. Our Api app Server uploads photo (w/ api key + secret) to Cloudinary

3. Cloudinary stores the photo & send response (Url + public Id of photo that was just loaded)

4. Take response info and save string values to DB (Photo Url + PublicId) stored in Users or Photos table

5. DB generates a SQL id that is stored with that photo

6. Controller gives 201 created response w/ location header

<https://cloudinary.com/>

Cloud name: dqipefkrt

API key:952234811161497

API secret: 452b3in3HbMF14ihj3x9cwvie90

Environment variable: cloudinary://952234811161497:452b3in3HbMF14ihj3x9cwvie90@dqipefkrt

## Cloudinary Set-up

api> appsettings.json Add Cloudinary configuration

"AllowedHosts": "\*",

  "CloudinarySettings": {

    "CloudName": "dqipefkrt",

    "ApiKey": "952234811161497",

    "ApiSecret": "452b3in3HbMF14ihj3x9cwvie90"

  }

api>helpers>CloudinarySettings Add class to hold config settings

namespace DatingApp.API.Helpers

{

    public class CloudinarySettings

    {

        public string CloudName {get; set;}

        public string ApiKey {get; set;}

        public string ApiSecret {get; set;}

    }

}

api>Startup.cs register Cloudinary as a service. Match up the class with the appsettings.json

services.AddCors();

services.Configure<CloudinarySettings>(Configuration.GetSection("CloudinarySettings"));

api>models>Photo.cs Capture the public Id from Cloudinary response when a photo is uploaded to cloud

public class Photo

    {

public string PublicId {get; set;}

--Our app is now different than our DB. We need them to be the same

❯ dotnet ef migrations add AddedPublicId-Photo-field

Build started... Build succeeded.

Done.

Api>csproj

<PackageReference Include="CloudinaryDotNet" Version="1.12.0"/>

--Add Cloudinary through Nuget package to the project dependencies

### Cloudinary Configuration

Remember that in the above we brought in 3 values required to configure. CloudName (the name of our cloud environment location on their site) Api Key and Api Secret (these are needed to access our env. and stuff)

How do we use these configs in our program?

appsettings.json is our global location to store configuration settings. So by registering them here, it is made available to the whole program. But, we still do not have a strongly typed (real class) that we can use to DI to places that we need to actually use these.

How can we inject these configs as a service in the Startup.cs file (IConfiguration Configuration)?

By using services.Configure <Class with Props matching configs> and (Configuration.GetSection(“..”)) which gets a section of appsetting.json. In this case we are asking for the whole object “CloudinarySettings": {“…”:”…”}

api>Startup.cs

services.Configure<CloudinarySettings>(Configuration.GetSection("CloudinarySettings"));

How can we inject the configured service into a class that we can use the configured instance?

When we use services.Configure to register. We cannot DI into our class the normal way. We need to use IOptions <Class with setting Props > to make a class instance

Once we have a local instance “\_cloudinaryConfig” which is the class we passed into IOptions. We “get” this instance the “Values” from the k:v string names in the appsettings.json file. (see below)

How can we send a fully configured new Cloudinary object with each request on this Controller?

Note that per Api docs this is done inside of the Constructor so, every http req/resp call from this Controller has these configs.

Also, an internal structure for our cloudinary set-up is done within an Account acct inside of this Constructor. Basically, we

1. Inject Cloudinary ConfigSettings (see above) 2. Use these settings to make a cloudinary Account acct 3. Pass this acct with the configs specific to our cloudinary environment to the stand-alone Cloudinary \_cloudinary member instance 4. \_cloudinary = A new cloudinary instance w/ all of our environment configs each and every time we receive a new request pulled in by this Api Controller

    [Authorize]

    [Route("api/users/{userId}/photos")]

    public class PhotosController : ControllerBase

    {

        private readonly IOptions<CloudinarySettings> \_cloudinaryConfig;

        private Cloudinary \_cloudinary;

        public PhotosController (IOptions<CloudinarySettings> cloudinaryConfig)

        {

            \_cloudinaryConfig = cloudinaryConfig;

            Account acc = new Account(

                \_cloudinaryConfig.Value.CloudName,

                \_cloudinaryConfig.Value.ApiKey,

                \_cloudinaryConfig.Value.ApiSecret

            );

            \_cloudinary = new Cloudinary(acc);

        }

### Additional fields

When cloudinary responds (after a photo is uploaded) from its server we have an additional field we need to track “PublicId” this is a uniqueID from Cloudinary. This did not exist in Cloudinary before the file was uploaded and saved (similar to a DB entry from our own DB).

Because we want to track this moving forward, we need to 1. Make a new Field in Photos.cs so that we have a place to capture and ultimately save to our Db as a column value for a Photo object. 2. Stop the running program and make a migration -> our DB knows about the change from our app -> dotnet ef migration add <…> + update

## Controller – Cloudinary

See the Constructor configurations from above. We also DI- IMapper & DatingRepo (not shown) the normal way

1. Authorize the request by comparing the UserId in the Url string <-> UserId from passed JwToken

[HttpPost]

        public async Task<IActionResult> AddPhotoForUser

        (int userId, [FromForm] PhotoForCreationDto photoForCreationDto)

        {

// Check if the current User is the one that passed the token to the server

// Trying to match passed id to what is in their token ... see authController line 79

// User = check the passed token and get info from it .. we are [Authorize] this request

            //

            if (userId != int.Parse(User.FindFirst(ClaimTypes.NameIdentifier).Value))

                return Unauthorized();

// Call the repo method to return a single user from the repo <-> DB based on Id

            //

            User userFromRepo = await \_repo.GetUser(userId);

### Using Image files

2. Grab the “photo/image” from the Dto using a getter method from its “file” Property

// call Dto instance that is being passed and get its File property (has photo)

            //

            IFormFile file = photoForCreationDto.File;

3. Use cloudinary ImageUploadResult() object to hold the response from Cloudinary

4. Use a conditional to make sure the file is not empty

5. Read the file and make sure the resource can be auto-closed when we are done with the OpenStreamreader()

//To hold the results we get back from Cloudinary.  ImageUploadResult= Cloudinary class

            //

            var uploadResult = new ImageUploadResult ();

            if (file.Length > 0)

            {

                // read this file into memory then dispose whne done

                //

                using (var stream = file.OpenReadStream())

                {

### Using Cloudinary Image Params

5. Ready the “ImageUploadParams()” with “stream” = the read photo file & Transformation of the image so that, this image is in a desired transformation state that is good for our Client site

6. Call \_cloudinary.Upload (all the params needed + transform stuff that we want) to push the Image to Cloudinary which, saves to our environment on their site and gives us a response header with a crap-load of info

using (var stream = file.OpenReadStream())

                {

// Populate the “uploadResult” from Cloudinary w/ the Photo from Client

// IFormFile as file which we will read as a stream, get name tied to this object

                    // Transform the Photo to meet our shape/size specs for the site

                    // Use all of these as params which we use to initialize uploadresult

                    // This is what we are going to pass to cloud storage

                    //

                    var uploadParams = new ImageUploadParams ()

                    {

                        File = new FileDescription (file.Name, stream),

                        Transformation = new Transformation ()

                            .Width(500).Height(500).Crop("fill").Gravity("face")

                    };

// Calls method to upload to 3rd party storage + store in local variable

                    //

                    uploadResult = \_cloudinary.Upload(uploadParams);

                }

            }

7. From the “uploadResult” = the response from Cloudinary with all the info. Capture the relevant info back into the Dto now that we have 1) Cloudinary Url of where the photo is being saved 2) Cloudinary Id of photo

### Ready Dto to save to DB

8. Convert the updated Dto data -> Photo. This requires that we have set up AutoMapper to map between the two. In this case “source”=Dto with updated info -> “destination”= Photo instance to capture the Dto data

9. Conditional to make this Photo the IsMain only if. No other Photo’s tied to this User have IsMain already

10. We have not saved to DB yet. 1st we need to Take the User object & use LinQ Add method to tell the DB. We are calling for Tracking of this particular User. Also, we are updating this User object to hold a new Prop “photo” so, this User is now fully updated + tracked. Ready to be saved to the DB

//These are to populate Dto w/ results returned back from Cloudinay

            //

            photoForCreationDto.Url = uploadResult.Url.ToString();

            photoForCreationDto.PublicId = uploadResult.PublicId;

            // map returned results from our Dto -> Photo object

            //

            Photo photo = \_mapper.Map<Photo>(photoForCreationDto);

            if (!userFromRepo.Photos.Any(u => u.IsMain))

            {

                photo.IsMain = true;

            }

            // Track changes to the User object which now has updated Photo

            // details that we are adding from Dto (has upload response + isMain)

            //

            userFromRepo.Photos.Add(photo);

11. Conditionally call a DB save from our Repo methods true = do all the stuff in {…} false = BadRequest()

### Returning response back to Client

12. now that save has been initiated. We have a lot of data that we don’t want to send back to the Client. We should have another Dto just for this response to the client “PhotoForReturnDto”

13. The fully saved Photo object needs to be “Mapped” to a Dto only showing the needed return info. “Source” =Photo.cs “Destination” = The Dto only showing info useful to the Client

            if (await \_repo.SaveAll())

            {

                // Convert updated Photo object to Dto because, we want the return

                // status code 201 also, include header info about this photo

                // Once saved the photo will have a DB generated Id

                //

                var phototoReturn = \_mapper.Map<PhotoForReturnDto>(photo);

                //Show location header of created resource

                // string routeName = Name [httpGet{"{id}"}, Nme = "GetPhoto"]

                // object routeValues, new object with values from passed userId +

                //                     photo(mapped Dto from above)

                // object value = The object being created

14. Returning CreatedAtRoute () 201 response code. We want to use these 201 codes because it allows us to add header information. This is 1 of 3 overloads. See above comments for details

                return CreatedAtRoute ("GetPhoto",

                    new { userId = userId, id = photo.Id }, phototoReturn);

            }

            else return BadRequest("Could not add the Photo");

        }

    }

# Angular Time ago pipe

<https://www.npmjs.com/package/ngx-timeago/v/1.0.4> use ngx for angular 9+ latest version

Make sure you only call this method in the root module of your application, most of the time called AppModule. This method allows you to configure the TimeagoModule by specifying a formatter, clock and/or an intl service. You should end up with code similar to this:

npm install ngx-timeago –save

**import** {BrowserModule} **from** '@angular/platform-browser';

**import** {NgModule} **from** '@angular/core';

**import** {TimeagoModule} **from** 'ngx-timeago';

@NgModule ({

  imports: [

    BrowserModule,

    TimeagoModule. forRoot()

  ],

  bootstrap: [AppComponent]

})

**export** class AppModule {}

This is how you do it with the **pipe**:

<div>{{1553683912689 | timeago: live}}</div>

And in your component define live (true by default).

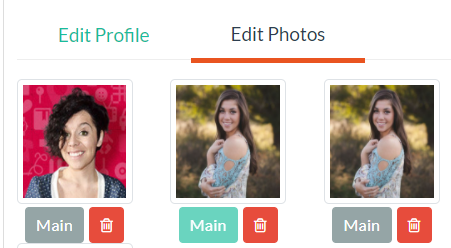
This is how you use the **directive**:

<div **timeago** [**date**]="1553683912689" [**live**]="live"></div>

# Angular Templates

## (Click)

In this scenario we are allowing the user to click on the “Main” button to select which photo should be marked as isMain = true from the User’s collection of photos. On-click the method subscribes to a service that calls the api and passes the needed info to make updates to the DB



Whichever photo has the isMain =True -> “active”

False = btn-secondary

[ngClass] = “style based on condition”

[disabled] = “based on condition true”

(click) =” execute this function on click”

<div class="col-sm-2"

\*ngFor="let p of photos">

<img src="{{p.url}}" class="img-thumbnail p1" alt="">

<div class="text-center">

      <button type="button"

class="btn btn-sm mr-1"

(click)="setMainPhoto(p)"

[ngClass]="p.isMain ?

'btn-success active’: 'btn-secondary'"

[disabled]="p.isMain">Main</button>

      <button type="button"

class="btn btn-sm btn-danger">

<i class="fa fa-trash-o"></i></button>

</div>

</div>

## Disabled

[disabled]="p. isMain">Main</button>

[disabled]= “! registerForm. valid">Register</button>

## ngClass

[ngClass]="p. isMain? 'btn-success active’: 'btn-secondary'"

<input [ngClass]="{'is-invalid': registerForm. get('knownAs'). errors && registerForm. get('knownAs').touched}"

## \*ngIf

<div class="invalid-feedback"

      \*ngIf="registerForm. get('country'). touched && registerForm. get('country').

hasError('required')">Country is Required

</div>

 <! --Only show NavBar categories if user is logged-in Boolean true-->

        <ul class="navbar-nav mr-auto" \*ngIf="loggedIn ()">

             <li>Cat1</li> <li>Cat2</li> <li>Cat3</li>

</ul>

# FormsModule - Login <form>

<form> is just html until you activate the Angular capabilities for:

change tracking -Angular tracks changes for forms and fields behind the scenes

two-way binding - Turn on FormsModule and use [(ngModel)]

validation and error handling -You can test in your form for validation and state values

## NgForm <form #...="ngForm"> using template reference

A directive exported from FormsModule once you import it. It is the way to turn on Angular functionality for a form and use ng directives like (\*ngIf, [(ngModel)] ):

1. import FormsModule from the @angular/forms into spa>src>app>AppModule

import {NgModule} from '@angular/core';

import {FormsModule} from "@angular/forms";

@NgModule({

  declarations: [],

imports: [

    FormsModule,

…]

})

export class AppModule {}

1. Make a template reference inside of the <form> … This allows you to reference the template as a variable that, can be access outside of the form tag

<! --If not logged in...user/pass login fields will appear.  Removed once logged in-->

-- Template reference #loginForm = variable to assign "ngForm" directive

<form #loginForm="ngForm" \*ngIf= “! loggedIn ()" class="form-inline my-2 my-lg-0">

--name= attribute used to assign and register the input + required

            <input class="form-control mr-sm-2”

 type="text" placeholder="UserName" required name="username"

                [(ngModel)] ="model. username">

            <input class="form-control mr-sm-2”

type="password" placeholder="Password" required name="password"

                [(ngModel)] ="model. password"> <--2-way binding to model: any={} in component

            <button [disabled]= “! loginForm.valid"

class="btn btn-success my-2 my-sm-0" type="submit"

                (click)="login()">Login</button>

</form>

### Angular Form States

dirty – A form starts out as $pristine but, as soon as the user interacts with any form field. The form itself has now been touched by the end-user …aka… “dirty” …The below <div> is not part of the form (completely separate)

<form #editForm="ngForm"

I.E the form template ref = “editForm” … we are saying that if, the user touches this form in any way …dirty=true then show/add whatever elements are inside of this <div> to the DOM

<div \*ngIf="editForm. dirty"

[disabled] – This element is not clickable unless the form is “dirty”. True= $pristine False=$dirty

<div class="card-footer" >

                <button [disabled]="!editForm. dirty" class=”…”> Save Changes</button>

                </div>

### Connect <form> to separate <button>

Problem: A form has changes but, the button is not part of the form because of the page layout. How would you connect the button on-click event to handling the form data?

1. The form needs:

a) template reference #...=”ngForm”

b) matching id=” …”

c) form submit action (ngSubmit)=” method ()”

2. Form should be connected to the button … tell the button which form template to reference

This will allow a button-click to act as a submission for this form

src>app>members>member-edit>member-edit.html

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

… button is not part of the form

<button [disabled]= “! editForm. dirty" form="editForm" class="…">Save Changes</button>

### @ViewChild method -> <form> state

Problem: On button-click some fields are now dirty. These dirty states are causing our \*ngIf=” …” statements to have kicked off, which is fine but, now we want the \*ngIf’s to revert back. We now want to change the state to $pristine (untouched) when the above updateUser () happens.

Goal: Have the method revert the “form state” back to $pristine…which updates the “form status”

1. The component.ts needs

a) @ViewChild decorator– To gain access to the <form>

i) param = (“form to access”, {static: true}) reference\_forThis: Type of Form

b) Use the “ViewChild” reference in the method to effect state within the <form>

i) form is being reset to $pristine

c) reset Parameter = Give reset values, when on-click (state they were at when “saved”) -> $pristine

src>app>members>member-edit>member-edit.ts

import {Component, OnInit, ViewChild} from '@angular/core';

import {NgForm} from '@angular/forms';

export class MemberEditComponent implements OnInit {

  user: User;

  @ViewChild ("editForm", {static: true}) editForm: NgForm;

updateUser () {

    this. editForm. reset(this.user);

  }

## Angular validation

Testing the states (field & form states) that Angular is keeping track of behind the scenes. I am using a <div> on the template so, I can see real-time. For example, the field states will change when I put in information vs. having the fields blank.

1. Add reference var to the end of each <input> field … This allows us to access the tag in our div below #password="ngModel" - Allows for 2-way binding so, you can see the field states
2. The form already has its own template reference so, we can access the form for validation in the below div by accessing #loginForm
   1. A form will be considered “invalid” if the “required” fields are not populated
   2. This test shows our form is only valid when fields are populated but, we need a way to stop the user from logging in with invalid (fill out required fields).
      1. **Disable the <button> if form is not valid**
      2. <button [disabled]= “! loginForm.valid"
      3. <button [attribute] = “expression” > …</button>
      4. Button is only available once the form is valid (input all required fields)
3. This <div> can be removed after testing is complete

--This <div> is just an example of testing state/validation inside of the template

<! -- form-states that Angular is keeping track of -->

<div>

Form Valid: {{#loginForm.valid}} --Required fields populated entire form?  T/F

Form Values: {{#loginForm.value | json}}  --Values populate as you type

<! -- field-states that Angular is keeping track of -->

Password Valid: {{password. valid}} --valid = required field is populated

Password Value: {{password.value}} --2-way binding value will be printed

</div>

### (click) vs (ngSubmit) – Handlin a click event

These two do the exact same thing. The only difference is that one is done on the <button> while the other is within the <form> see the difference in syntax below

<form #loginForm="ngForm"  (ngSubmit)="login()">

-OR-

<button …(click)="login()"> Login </button>

### Reading data from the browser

Console: Message you send to the log

Network: Where you see your Req/resp info .. Click on indiv. req to see req/resp info

Application: Storage- Local/Session/Cookies, Cache-



## The \*ngIf structural directive

### \*ngIf directive on a method

True = Add objects inside of this <div> to the DOM

False = Items in this <div> will not be displayed in the DOM

<! --Drop Down menu will display if the User is logged in ...using NGXBootstrap-->

        <div class="dropdown" \*ngIf="loggedIn()" dropdown>

Conditional method above in the html template “loggedIn()” is being used as a switch for this \*ngIf=" …" The login form is triggering this method on-click event when user signs in

Below is the component.ts that contains the method that is being executed in the template

<!—-Does the “token” exists in the local storage (T/F) -->

loggedIn (): boolean {

      const token = localStorage. getItem("token");

        return !!token;  !!== If (token) -> true || else (! token) -> false

  }

template reference

The \*ngIf=” …” is constantly running and listening. So, when conditions changes, the values are updated and the switch if flipped (True / False)

### \*ngIf directive on a field value

Alternate pattern for toggling

1. make a variable in the component, set as False to be initially hidden

export class HomeComponent implements OnInit {

  registerMode: boolean = false;

1. set this variable inside the \*ngIf="…"

    <div class="container" \*ngIf="registerMode"> --hidden since currently false

        <div class="row justify-content-center">

            <div class="col-4">

                <app-register> </app-register> --Once \*ngIf becomes true this child

            </div> component will be displayed

        </div>

</div>

1. set an event (click)="toggle()" that will toggle a variables value (True -or- False) on-click

<div class="container mt-5">

    <div style="text-align: center" \*ngIf= “! registerMode">

        <h1>Find your match</h1>

        <p class="lead">View your matches... Sign up! <p>

        <div class="text-center">

            <button class="btn btn-primary btn-lg mr-2" (click)="registerToggle()">Register</button> --Toggle changes “registerMode” F/T

            <button class="btn btn-info btn-lg">Learn more</button>

        </div>

    </div>

>

# Interfaces in TypeScript

Interfaces in TS == classes in C# but, structured differently …TS does not have constructors. You just make the fields + optional fields … Make TS Interface to match C# class structure … exact name match

xxx?: any = Optional filed must come after all required fields

Make a new folder “\_models” src>app>\_models

ng g i users to hold the Interfaces to match up with the classes

Example: In our case, we are using Dto's to only pass selected fields between the Spa < - > Api. So, we would replicate each Dto. For Dto's that are basically the same +/- a few fields.

public class UserForDetailedDto

{

public int Id {get; set;}

public string UserName {get; set;}

public string Gender {get; set;}

public int Age {get; set;

public string KnownAs {get; set;}

public DateTime Created {get; set;}

public DateTime LastActive {get; set;}

public string City {get; set;}

public string Country {get; set;}

public string Introduction {get; set;}

public string LookingFor {get; set;}

public string Interests {get; set;}

public string PhotoUrl {get; set;}

public ICollection<PhotosForDetailedDto> Photos {get; set;}

src>app>\_models>user.ts

export interface User {

    id: number;

    userName: string;

    gender: string;

    age: number;

    knownAs: string;

    created: Date;

    lastActive: Date;

    city: string;

    country: string;

    introduction?: string;

    lookingFor?: string;

    interests?: string;

photUrl: string;

photos?: Photo[];

}

# Reactive Forms

The difference between “Reactive Forms” and “Template Forms” is:

Template = Most of the code is done inside of the template (html)

Reactive = Most of the code is done in the component (so, html should be cleaner)

## How to set up:

1. make a variable property to hold the FormGroup

FormGroup: Tracks the value and validity states for instances of a FormControl

2. Import FormGroup

app>register>register. component. ts

import {FormGroup} from '@angular/forms';

export class RegisterComponent implements OnInit {

  model: any = {};

  registerForm: FormGroup; <--This holds the fields of your form

3. Create the form inside of this components ngOnInit’s lifecycle method

a. FormGroup contains FormControls (like fields of a form) as K, V pairs

app>register>register. component. ts

ngOnInit (): void {

    this. registerForm = new FormGroup ({

       username: new FormControl (), <--These are FormControl names

       password: new FormControl (), <--Must be the same as the html elem names

       confirmPassword: new FormControl ()

    })

  }

4. Register this module with app. module to let app know you are using “Reactive Forms Module”

app>app. module. ts

import {FormsModule, ReactiveFormsModule} from "@angular/forms";

imports: [

    FormsModule,

    ReactiveFormsModule,

    BrowserModule,

    HttpClientModule,

## Updating “Form Template” to Reactive

Template reference is replaced with [formGroup]

required is no longer needed b/c validation happens w/I the component

ngModel is not used b/c 2-way binding auto-happens using FormGroup

name= is replace with formControlName=

app>register. register. component. html

--Template reference variable not needed in the html b/c no longer a template form. The reference is in the ngOnInit. Instead use a “formGroup” with name

<form ~~#registerForm="ngForm” [~~formGroup]="registerForm"

(ngSubmit)="register ()">

  <h2 class="text-center text-primary">Sign Up</h2>

  <hr>

  <div class="form-group">

    <input type="text" class="form-control"

placeholder="Username"

~~required~~ ~~name="username"~~ formControlName="username"

~~[(ngModel)] ="model. username"~~>

  </div>

--required is no longer needed b/c validation happens w/I the component

--ngModel is not used b/c 2-way binding auto-happens using FormGroup

-- name= is replace with formControlName=

  <div class="form-group">

    <input type="password" class="form-control"

placeholder="Confirm Password"

~~required~~ ~~name="password"~~ formControlName="confirmpassword"

~~[(ngModel)] ="model. password">~~

  </div>

  <div class="form-group text-center">

    <button class="btn btn-success" type="submit mr-2">Register</button>

    <button class="btn btn-default" type="button" (click)="cancel()">Cancel</button>

  </div>

</form>

--use this area to test if the form is working correctly this far

<p>Form Value: {{registerForm. value | json}} </p>

<p>Form Status: {{registerForm. status | json}} </p>

## Checking the form

One way to make sure your form is working correctly is to test the values of the form itself (formName.value) as seen above. Which gets the values of the form and outputs to the console on submit. Notice the (ngSubmit)="register ()"> is done one the <form> … it could be on the button but, this is just preference

app>register. register. component. ts

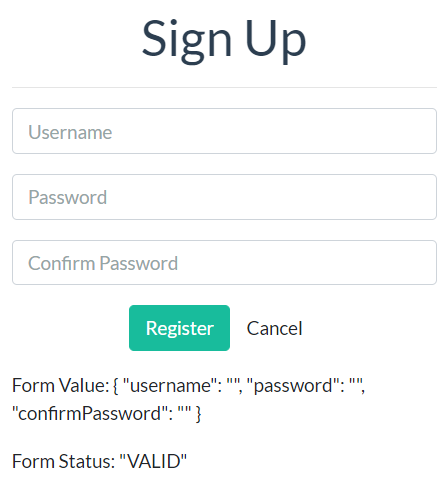
Register () {

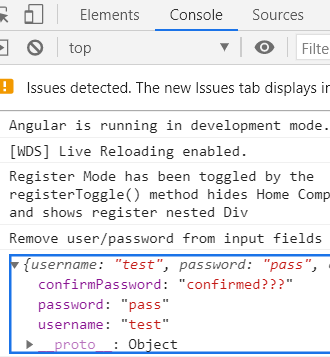
    console.log (this. registerForm. value)

--Log values of this form to the console

When using Reactive forms, you automatically get 2-way binding similar to ngModel so, when you start typing in the fields. The test values in <p> should be real time

  }





registerForm. status = Are all the fields required to be validated ok? At this point we have no validation so; the state is valid.

### Validating Reactive form

Angular provides validation as part of its framework through forms module so, you don’t have to go out and get 3rd party apps. The FormControl assigns validation by passing values that are checked against its FormControl params

Validators = Provides a set of built-in validators that can be used by FormControls.

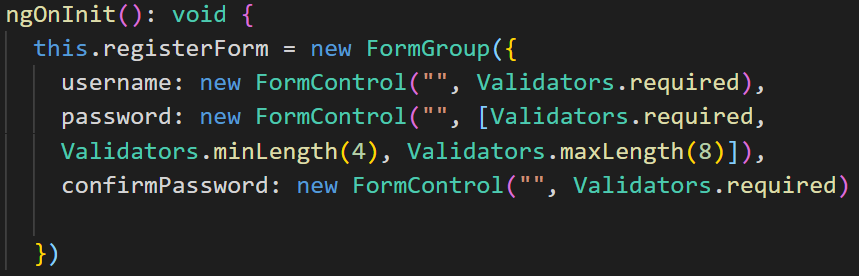
(alias) new FormControl (formState? : any,

  validatorOrOpts? ValidatorFn

  | ValidatorFn []

  | AbstractControlOptions, asyncValidator? : AsyncValidatorFn

  | AsyncValidatorFn []): FormControl



formState?: any = Do we want to give an initial value to form control (“hello” = field pre-populated with “hello”)

validatorOrOpts?: These are validators which are functions that process a FormControl () or collection of them. These have out-of-box validators using dot notation

### Making a custom Validator

Since this could affect multiple FormControls, this is a method that would need to be made done at the FormGroup level. In this example, this method is used to compare the password and confirmPassword fields.

**In reactive forms we are required to** **return null if the validator passes**, else set the validator key as an object (any word key) with a value of true. = invalid state

Now these fields see “Checking the form” section above will only be valid if these two fields match

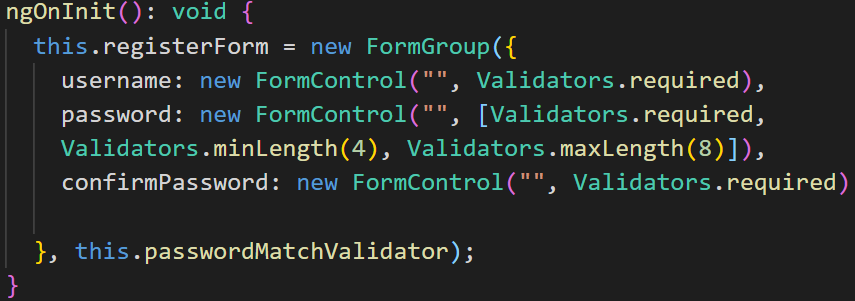
passwordMatchValidator (cv1: FormGroup) {

    return cv1.get("password").value === cv1.get("confirmPassword").value

      ? null: {mismatch: true};

  }

\*\* add () after methods … duh



Custom Validator methods using 'get' method (to get a specific named field of that FormGroup) will only take a path of the form control. So, to make the methods generic to take any input. See links below for examples in “Form Template”

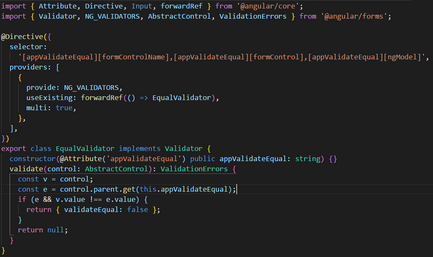
<https://scotch.io/tutorials/how-to-implement-a-custom-validator-directive-confirm-password-in-angular-2#toc-custom-confirm-password-validator>

<https://stackoverflow.com/questions/35474991/angular-2-form-validating-for-repeat-password>

replace with

}, {validator: this. matchValidator})

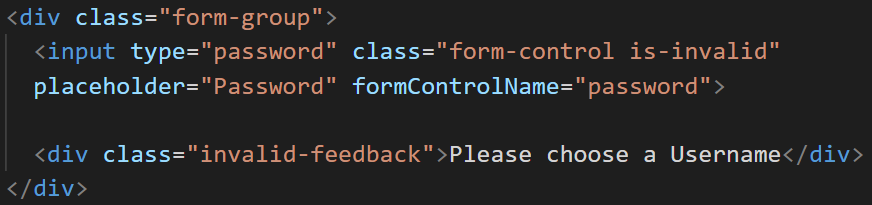
### forwardRef – Using generic values vs. named field values

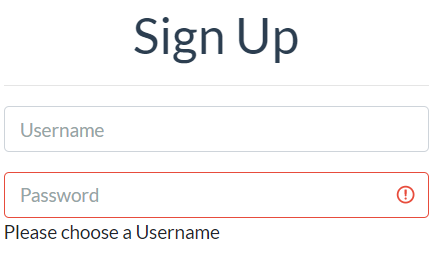


## Feedback to User

When the User does not meet the validation rules. They need some sort of feedback to let them know what is wrong. With bootstrap there are special classes that you use in pairs within the html.

Example: <input class =”is-invalid”> matches up with another element to display text if the field status is invalid <div class=” invalid-feedback”> The problem here is that, the error messaging is being displayed before the user has done anything b/c required + min-length. It would be better if the error was conditional





### Conditionally display errors

When we want a conditional error to display to a Client. Basically, we mean that if, certain condition/s exists. Only then should the field or form be invalid and an output message should be displayed. The way this is accomplished is by pairing [ngClass] in the <input> with \*ngIf=” …” as part of the output to Client <div>

[ngClass] = {‘display this class’: ‘for this field’. given this condition}

&& combines two statements so that, both conditions must be true in order for the [ngClass] to display its conditional class

Example 1: If a condition is met that the “registerForm” field ‘confirmPassword’ produces an error (error is generated by the Validators in the component). && the “registerForm” field ‘confirmPassword’ is “touched” by the Client -> the ‘invalid-feedback’ bootstrap class is initiated that makes the field have a red outline

|| is being used to create a possible 2nd scenario where

Example 2: If a condition is met that the “registerForm” field ‘confirmPassword’ is “touched by the Client”. && the form itself “registerForm” generates an error (error is generated by “passwordMatchValidator” … which is a method tied to the form) this. registerForm = new FormGroup ({…}, this. passwordMatchValidator); -> the ‘invalid-feedback’ bootstrap class is initiated that, makes the field have a red outline.

Remember, **returning null =validator passes**, else the validator method returns a K, V pair. The key for this method = ‘mismatch’ so, if this is returned the form itself is invalid and will. hasError

An individual field. errors <-> The form itself. hasError (‘…’) pair ***error type must be in all lowercase***

<div class="form-group">

    <input type="password"

--Example 1

[ngClass]="{'is-invalid': registerForm. get('confirmPassword'). errors &&

    registerForm. get('confirmPassword'). touched

--Example 2

    || registerForm.get('confirmPassword').touched && registerForm.hasError('mismatch')}" class="form-control"

      placeholder="Confirm Password" formControlName="confirmPassword">

Only show the is-invalid class pairing when the FormControl field or Form itself is in an errors state

The output <div> pairing uses an \*ngIf=” …” to only show an output if, errors have been generated by the form.

1) If the registerForm field. hasError (‘specific error’) -> provide ‘invalid-feedback’

If this particular field is causing, this particular error and causing the form in general to be invalid

2) The registerForm itself hasError (‘specific error’) -> provide ‘invalid-feedback’

The form itself is throwing an error causing the form to be invalid -> provide ‘invalid-feedback’

<! -- validation error outputs for confirmPassword field    -->

--We are pairing the ‘is-invalid’ class to the ‘invalid-feedback’ output class

    <div class="invalid-feedback"

\*ngIf="registerForm. get('confirmPassword'). hasError ('required')">

      Password is required

    </div>

    <div class="invalid-feeback" \*ngIf="registerForm.hasError('mismatch') &&

    registerForm. get('confirmPassword'). touched">

      Passwords must match

    </div>

# Reactive Form Build Service

Allows for easier syntax for building forms. FormGroup, FormControl are now not needed to create a new Form. This is the normal way of doing forms and comes out of the box by Angular as a service.

## Set-Up

1. Import FormBuilder

2. Inject FormBuilder into the Constructor

src>app>register>register. component .ts

import {FormBuilder, FormControl, FormGroup, Validators} from '@angular/forms';

constructor (private authService: AuthService, private alertify: AlertifyService,

    private router: Router, private fb: FormBuilder) {}

3. Create method to build a Form

We are no longer building the form in the ngOnInit () using FormGroup as the overall form and FormControl as the fields. The form + fields are created in a method -> that is called in the ngOnInit ().

Old way:

ngOnInit (): void {

    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. passwordMatchValidator);

## Reactive Form fields

New way:

registerForm: FormGroup;

ngOnInit (): void {

    this. createRegisterForm;

  }

createRegisterForm () {

    this. registerForm = this. fb. group ({

      gender: ["male"],

      username: ["", Validators. required],

      knownAs: ["", Validators. required],

      dateOfBirth: [null, Validators. required],

      city: ["", Validators. required],

      country: ["", Validators. required],

      password: ["",

        [Validators.required, Validators.minLength(4), Validators.maxLength(8)]

      ],

      confirmPassword: ["", Validators. required]

    }, {validator: this. passwordMatchValidator})

  }

Notice: We are still naming the form. The component has a field called “registerForm” that is of type FormGroup. So, the form group is still being created, just in a different way

By calling “fb. group” = new FormGroup **(** **{** fieldName**:** “state/empty/something”**,** [optValidators] }**)** vs.

this. fb. group ({ username: ["", Validators. required] notice [] vs. {} notation

passwordMatchValidator (cv1: FormGroup) {

    return cv1.get("password").value === cv1.get("confirmPassword").value

      ? null: {mismatch: true};

  }

# Parent -Child <-> Child-Parent:

This is how you pass data from one component to another so that, values can be made available across components. Pass values from Parent -> Child @Input () -or- Child -> Parent @Output ()

## Angular @Input @Output Property binding

### Parent Child @Input () Property

Pass values from a Parent component to a Child Component

Scenario: Parent has a \*ngFor that produces all User objects of an array that only the Parent has access to from it component.ts. “u” represents an individual User object. This User object comes from the parent. Assign this “u” to the Child by passing it in the Child’s template selector. [user] is a value that lives in the Child’s component.ts

src>app>members>member-list>member-list.component.html

 <div class="row">

        <div \*ngFor="let u of users" class="col-lg-2 col-md-3 col-sm-6">

 <! --Parent passes "input" to the [@Input variable name] of the child-->

            <app-member-card [user]="u"> </app-member-card>

        </div>

@Input () = Catch a value from Parent -> Child. The child now has access to the Parent’s user object which was passed in the Parent’s template and caught as a member variable of the Child in its component.ts

src>app>members>member-card>member-card. component.ts

@Component ({

  selector: 'app-member-card',

  templateUrl: './member-card.component.html',

  styleUrls: ['./member-card.component.css']

})

export class MemberCardComponent implements OnInit {

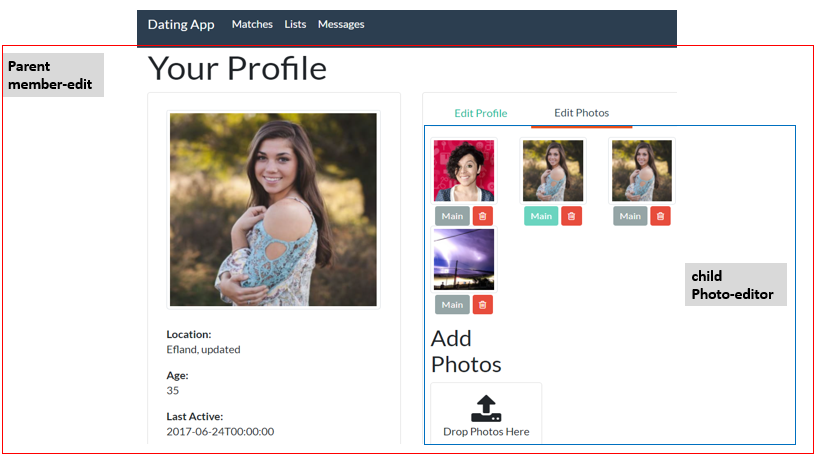
@Input() user: User;

### Child Parent @Output () Property

Pass values from the Child component to the Parent

Scenario: The Parent has access to a specific User object which it grabbed using a route resolver. When the Client clicked to change pages, the route changed and this route had the Id for a specific user which, is being used to create the Member-edit page (displays all details of this member).

Within this page there are tabs, one of which is called “Edit Photos” that has access to all photos from the member’s collection of Photo objects. The isMain property of Photos takes a single photo and displays it on the left-hand side of this member-edit page. This User object and its current isMain state comes from the Parent.



On the right-hand side of this member-edit page are all Photos objects tied to this User which, include isMain. The right-hand side is actually a Child component. It is here that the user interacts with our app to potentially change which photo is currently being displayed as isMain.

We want: The Clients selection in the child component to be reflected in the Parent component. This will update the state of an object in the Parent when, change (coming from the child) is made. Client clicks “main” button (in child) and the photo on the left-hand-side (property of the parent) should reflect the change we made. There are 4 parts that happen when passing values from a Child component to a Parent component

1. Create an @Output property in the Child which will be sent to the parent

2. Populate the Emit parameter with what we want to send. In the below this is being done within a Child method passing a “photo.url” string value

src>app>members>photo-editor>photo-editor.component.ts

import {Component, EventEmitter, Input, OnInit, Output} from '@angular/core';

export class PhotoEditorComponent implements OnInit {

  @Input () photos: Photo [];

--Output properties emit events. The event being emitted is the “photoUrl” which is a property of Photo = type string (used to change the Url value in the User object)

  @Output () getMemberPhotoChange = new EventEmitter<string> ();

  currentMain: Photo;

setMainPhoto (photo: Photo) {

    this. userService

      . setMainPhoto (this. authService. decodedToken. nameid, photo.id)

      .subscribe (

        () => {

          this. currentMain = this. photos. filter (p => p. isMain === true) [0];

          this. currentMain. isMain = false;

          photo. isMain = true;

--Takes the single “photo” the Client has clicked on and populates "photoUrl" of Photo

--as the param value that is passed as an Emit event that updates “getMemberPhotoChange”

          this. getMemberPhotoChange. Emit (photo.url)

3. In the Parent’s template we update the child selector <app-photo-editor> which is housing the communication between the Parent <-> Child relationship. “getMemberPhotoChange” is basically populating the param ($event)

src>app>members>member-edit>member-edit.component.html

<tab heading="Edit Photos">

<app-photo-editor [photos]="user. photos"

--The photUrl string is being emitted from the @Output property “getMemberPhotoChange” of the child. The “updateMainPhoto” is a method of the Parent which will make changes to its “user” property

       (getMemberPhotoChange)="updateMainPhoto($event)">

</app-photo-editor>

</tab>

4. The output “getMemberPhotoChange” represents an $event which is a string that is being passed to the “updateMainPhoto (photoUrl string)” method of the parent.

The Parent component src>app>members>member-edit>member-edit. component

export class MemberEditComponent implements OnInit {

user: User;

updateMainPhoto (photoUrl: string) {

    this. user. photoUrl = photoUrl;

  }

### Any-To-Any component communication

This allows two components to communicate that do not have a direct relationship. This happens in the Service b/c when a component injects a Service. The other component has access to the Service’s methods and variables (not necessary to be a parent or a child). There are 5 steps

1. Make Member variables in the Service

“CurrentPhotoUrl” basically turns the BehaviorSubject “photoUrl” into an Observable. Now any subscriber can access the “photoUrl” property by subscribing to “currentPhotoUrl”.

2. Make a Service method to update its variables

app>\_service>auth. service

import {BehaviorSubject} from "rxjs";

-- BehaviorSubjects’ require a default value. Using a ‘BS’ b/c we want this default value to be updated by other components. ‘BS’ can also be turned into Observables. This means that you can have a property that others can subscribed to.

--Notice these two var’s are working together. One sets a default property, while the other gives access to this property for any component that subscribes

export class AuthService {

  photoUrl = new BehaviorSubject<string>("../../assets/user.png"); <--holds value

  currentPhotoUrl = this. photoUrl. asObservable (); <--access point

  decodedToken: any;

  currentUser: User;

-- The method calls the BehaviorSubject’s internal next (method) that just assigns the next value which, is being passed to it from the param. So, whatever you pass becomes the new value for photoUrl: BehaviorSubject<string>

changeMemberPhoto (photoUrl: string) {

    this. photoUrl. next (photoUrl);

  }

3. Execute the Service method that updates Service variables

When a user Logins in. A response is sent back that has a token and a User object which, are set to local storage. The member variables of this AuthService are also set (decodedToken: any; and currentUser: User;). The this. currentUser. photoUrl of the logged in User is passed to the “changeMemberPhoto (). So, the ‘BS’ photoUrl is now assigned with the logged in User’s picture (current. User. photoUrl) Log-in -> observable request -> Api -> observable response -> update variable of the Service. At this point only the Service has the updated data from login. Any component who wants it, has to subscribe

app>\_service>auth. service

Login (model: any): Observable<void> {

    return this. http. Post (this. baseUrl + "login", model)

      . pipe (

        Map ((response: any) => {

          const user = response;

          if (user) {

            //user. token = response. key

            localStorage.setItem("token", user. token);

            //user. user = response. key

            localStorage.setItem("user", JSON.stringify(user.user))

            this. decodedToken = this. jwtHelper. decodeToken (user. token);

            this. currentUser = user.user;

            this. changeMemberPhoto (this. currentUser. photoUrl);

          }

        }) //--map transform one-at-a-time

      ); //--pipe transforms to Observable

The other component you want to have the data

4. Make variable to hold data from Service

5. Assign the value of local variable by subscribing to the Service’s observable property -ngOnInit

app>nav. nav. component

export class NavComponent implements OnInit {

  model: any = {};

  photoUrl: string;

ngOnInit (): void {

    this. authService. currentPhotoUrl. subscribe (p => this. photoUrl = p);

  }

Now you can use the variables of this component in its template

app>nav. nav. Component. html

<img src="{{photoUrl}}" alt="Current User Image">

6. Set property to be assigned when the application loads (app. component)

Problem: When other components subscribe (). There is an order. I.E. They subscribe before my Login () observable stream has finished so, they get the default value of the ‘BS’ not the value I want. So, Navbar photo shows default value from the stream not, the updated value that should be updated when the user logs in.

Solution: Force the order by executing the results in the application root on start of the application. We do this by calling our Service methods that update the service variables in (app. component. ts) the ngOnInit

We are still populating Service variables from the User login details so, the values from Login have not changed. Just the order so, we are forcing Login () and what it does to complete before another component may subscribe.

app>app. component. ts

ngOnInit () {

    var token = localStorage. getItem ("token”);

    var user = JSON. Parse (localStorage. getItem("user”));

    if (token) {

//initialize the var of authService with browser token string value

      this. authService.

decodedToken = this. jwtHelper. decodeToken (token);

}

//authService var "decodedToken" now has token data

    if (user) {

      this. authService. currentUser = user;

--On app load, change the default value of ‘BS photoUrl’ with the User info passed on log-in. Notice this data is being pushed to local storage and now it is also updating the member variables of this Service as well

      this. authService. changeMemberPhoto (user. photoUrl);

    }

Secondary problem: Multiple components who want the data (everyone not subscribed needs to subscribe)

Problem: At this point a photo is being displayed in the Navbar correctly but, it is not in synch with any member-edit component. Basically, two different pictures are being displayed b/c they are getting the data from 2 different sources

Solution: Repeat steps 4 & 5 for this component

4. Make variable to hold data from service

5. Assign the value of local variable by subscribing to the Service’s observable property -ngOnInit

app>member>member-edit>member-edit. component. ts

export class MemberEditComponent implements OnInit {

  user: User;

  photoUrl: string;

ngOnInit (): void {

    this. route. data. Subscribe (data => {

      this.user = data['user'];

    })

    this. authService. currentPhotoUrl. Subscribe (p => this. photoUrl = p);

  }

Now you can use the variables of this component in its template

app>member>member-edit>member-edit. component. html

<div class="card">

                <img class="card-img-top img-thumbnail" src="{{photoUrl}}"

alt="{{user. knownAs}}">

                <div class="card-body">

Secondary Problem: Components that have Child -> Parent relationships

Problem: At this point a photo is being displayed in the Navbar + member-edit but, it is not in synch photo-editor (child of member-edit) or, changes tied to isMain. The child component is not subscribed but, should be for the data to be the same across the 3 locations where they should all be the same.

Solution: ~~Repeat steps 4 & 5 for this component~~. Instead of emitting an output property (child -> Parent) we are going to call the method of the Service that updates Service’s variable value for all.

Previously photo-editor (child of member-edit) had been emitting a string value that was passed to the @Output property (to update the Parents main Photo) -> Child’s method “setMainPhoto” was executed (User clicking on stuff). Instead, just update the Service variable that everyone is subscribing to.

Photo-Editor is still getting all of its “Photos” from the parent but, instead the below method is using Services subscription variable -> changes member-edit & Navbar

1. Pass this. Photo. Url (this component’s member variable) to the Service method that updates the Services var.

The Url is now updated in Service variable “photoUrl” & observable that pushes this value out

app>members>photo-editor>photo-editor. ts

setMainPhoto (photo: Photo) {

    //This takes mbrId, PhotoId and sets this single Photo's isMain (turns off the other).  returns nothing

    this. userService

      . setMainPhoto (this. authService. decodedToken. nameid, photo.id)

      . subscribe (

        () => {

          this. currentMain = this. photos. Filter (p => p. isMain === true) [0];

          this. currentMain. isMain = false;

          photo. isMain = true;

         /\* this. getMemberPhotoChange. Emit (photo.url) //being replaced by below\*/

         this. authService. changeMemberPhoto (photo.url);

Secondary Problem: Persisting changes

Problem: Though we have made all of this work. It is not being persisted so, if the user refreshes the page. None of the above changes will stay.

Solution: We need to persist the updates to our local browser storage.

1. Update the Service’s “currentUser” variable

At current Service’s “currentUser” has the values from when user logged in. So, it’s a replica of what is in photo-editor component except. The single change to the Url. So, we need to update this single property in the AuthService which has the User. Photo-editor only has access to Photo not User

2. Update local storage with “setItem” which is replacing local storage ‘user’ with our updated AuthService variable “currentUser” that has the changes we want to persist

setMainPhoto (photo: Photo) {

//This takes mbrId, PhotoId and sets a single Photo's isMain (turns off the other).  returns nothing. userService -> photosController

    this.userService

      . setMainPhoto (this. authService. decodedToken. nameid, photo.id)

      . subscribe (

        () => {

          this. currentMain = this. photos. filter(p => p.isMain === true)[0];

          this. currentMain. isMain = false;

          photo. isMain = true;

          /\* this. getMemberPhotoChange. emit(photo.url) //being replaced by below\*/

          this. authService. changeMemberPhoto (photo.url);

          this. authService. currentUser. photoUrl = photo.url;

--“currentUser” is an object and local storage is in string form

          localStorage.setItem("user", JSON. stringify(this. authService. currentUser));

When we refresh a page, the app kinda gets restarted which means. The root application (app. component. ts) that everything else is pulling from needs to be correctly updated. So, remember that we had previously made changes that would pull the user values from localstorage on start-up. Because this was completed earlier. We are now good when the Client refreshes the page for this functoinality

# Terms

## angular snippets a-, ng-, fx b-

## forof operator – Shortcut for making a for-loop in a component

galleryImages: NgxGalleryImage [];

this. galleryImages = this. getImages ();

  getImages () {

    const imageUrls = [];

    for (const photo of this.user. Photos) { <--For loop

       imageUrls.push({

         small: photo.url,

         medium: photo.url,

       });

    }

    return imageUrls;

  }

## View encapsulation – Parent .css does not affect the child elements even if, using the same class names

If you want the parent-child to respond to the .cs file in the parent … add an additional decorator

-> ({selector, templateUrl, styleUrls}

-> encapsulation: ViewEncapsulation.

Emulated - Default (normal way)

None - No styling encapsulation - apply globally

ShadowDom

Alternately, you could just drop your css into styles.css and make it global

## Callback

In JS functions are objects so, a method can take other methods (functions) as arguments. Callbacks are used to make sure one method does not execute till the 1st method completes…Then the other method is “called back” for execution. This is helpful when making http requests b/c you may not want the 2nd function to start till after you have gotten back the response.

app>members> photo-editor>photo. editor. component. ts

deletePhoto (id: number) {

    //confirm (display message, IF YES callback () => {}) else exit method

    this. alertify. confirm('Are you sure you want to delete this photo?', () => {

      this. userService

        . deletePhoto (this. authService. decodedToken. nameid, id)

        . subscribe (

// If successful response, do this

          () => {

            this. photos. Splice (this. photos. findIndex (p => p.id === id), 1);

            this. alertify. success ('Photo has been deleted');

},

          error => {

            this.alertify.error('Failed to delete the photo');

          }

    ); //subscribe is part of callback+ assumes Req is completed and returns something

  }); //This callback is called if Yes click is completed

}

Yes = success -> callback … service called -> If something returned = success () => {} or, error => {}

## Slice vs Splice

Slice () – Does not change the original array. It returns the sliced items as a new array

var array= [1,2,3,4,5]

console. Log (array. Slice (2));

// shows [3, 4, 5], returned selected element(s).

console. Log (array. slice (-2));

// shows [4, 5], returned selected element(s).

console. log (array);

// shows [1, 2, 3, 4, 5], original array remains intact.

Splice (start-index, deleteCount) – Changes the original array + deletes. Returns removed items

// Find this index and delete 1 object in Photos [] starting from this index

            this. photos. Splice (this. photos. findIndex(p => p.id === id), 1);

            this. alertify. success ('Photo has been deleted');

## Partial Class:

**Partial<…>** Surrounding a class makes all fields including the required fields optional. In this scenario we only want to implement a single property of the BsDatepickerConfig class.

type Partial<T> = { [P in keyof T]?: T[P]; }

import {BsDatepickerConfig} from 'ngx-bootstrap/datepicker';

export class RegisterComponent implements OnInit {

  model: any = {};

  registerForm: FormGroup;

  bsConfig: Partial<BsDatepickerConfig>;

ngOnInit (): void {

    this. bsConfig = {containerClass: "theme-red"}

## Object. assign ()

This is a JS method that, copies enumerable values (in this case, all the values from a Reactive Form) from one or more sources. Takes all that stuff and puts it into a target object. Then returns that target object.

If (this. registerForm. valid) {

      this.user =

Object. assign (

{}, <-- Target

this. registerForm. value) <-- Source of values being cloned to “Target”

    }

--Empty {} is being assigned all values from the “source”. The var “user” is receiving all of these values from our Reactive Form to populate its User object

## Photo / img

Sometimes a photo cannot be displayed from the html component. If this is the case, it is good to have a default photo local to the angular application. Below is an example of using an || or statement to deal with a “null” phot value. Dealing with photo’s that have null values

<img class="card-img-top"

[src]=”user. photoUrl || '../../../assets/user.png' “

[alt]="user. knownAs"

>

-or-

<img src="{{photoUrl || '../../assets/user.png'}}"

alt="Current User Image"

>

## Extends vs. Implements syntax

Java does not support multiple inheritance. So, you can only inherit/extend from a single base class.

However, you may implement unlimited number of Interfaces. The ordering does matter so, the inherited class (class being extended) must always be listed 1st. Then the Interfaces are listed and separated by comma.

class MyClass: Foo\_extended, IClass1, IClass2

class MyClass: IClass1, IClass2

## IQueryable vs IEnumerable

Is this collection getting data from the DB or, does the data already exist in memory?

**IQueryable** – Generally you would use this with a variable because, the query is not yet executed by the Db->Client. The query to the Db is deferred/waiting for an execution function like ToList (), ToArray (), ToDictionary (), Count() to execute the waiting query to perform the output.

In this scenario we are making a DB request for all Users but, do not want all users returned. So, the app execution is deferred/waiting for the Db execution until the sorting has completed. This allows for all the sorting/filtering work to be done by the Db, not the application. Use a “Queryable” collection as the type = Delay execution till I call an execution function

// Line from Repo method IIncludableQueryable<out TEntity, out TProperty>

// Get Users + u. Photos from the context using EF Core but, do not execute, just hold

//

            var users = \_context. Users. Include (u => u. Photos);

// This is a separate PagedList<T> class and its CreateAsync method

// users is passed as source but, it is still not executed

//

public static async Task<PagedList<T>> CreateAsync

        (IQueryable<T> source, int pageNumber, int pageSize)

        {

            var count = await source. CountAsync (); <--Executing function called

            var items = await source. Skip ((pageNumber - 1) \* pageSize)

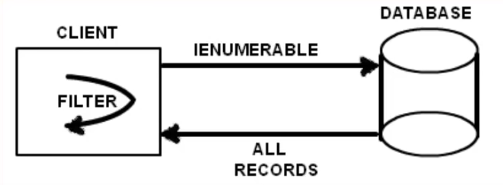
                . Take (pageSize). ToListAsync (); <--Executing function called

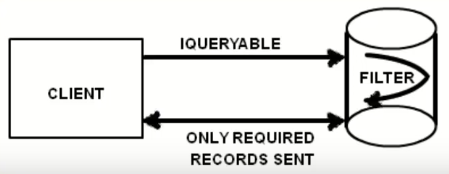
Now only a few records will be passed

to create new object below

            return new PagedList<T> (items, count, pageNumber, pageSize);

        }





**IEnumerable** – Iterate over the application’s In-Memory collection

Notice that, the mapping is interacting with a Dto. The Dto is not part of the Db, it lives in our application and its memory (Dto is a class that shadows the Db but, does not exist as a table in the Db). Therefore, there is no Db to interact with, only our application and its memory. Since there is not Db data that is needed in this scenario, we are simply working with an in-memory collection (the Dto). The entire transaction is happening in our application with no communication with the Db.

[HttpGet]

        public async Task<IActionResult> GetUsers([FromQuery]UserParams userParams)

        {

            var users = await \_repo. GetUsers (userParams);

            var usersToReturn = \_mapper. Map <IEnumerable<UserForListDto>>(users);

users = is what was referenced in the IQueryable example above. Since this interaction is Db based, “users” IQueryable is being used to defer the Db execution till all filtering methods have completed on the Db side and before an execution method is called in our application causing the DB to return its results post query.

## Extension Class

You make these for the purpose of adding functionality to classes. For example, the DateTime class does not already have a method that can calculate age. Datetime does not have a CalculateAge method that takes a given Date as a param input then, subtract it from the DateTime.Now (Date1.Now in years – Date2 passed).

So, we can augment DateTime (add a method to it that does not yet exist) by adding our own custom method

### How to extend a method?

1. Take the class you want to extend as a parameter and put “this” keyword before it. Notice that, this method is being housed in its own class called Extensions. cs which make sense because, here is where we are keeping all of the extension methods

Api>Helpers>Extensions. cs

public static int CalculateAge (this DateTime theDateTime)

        {

2. Define the logic you want to perform with this passed parameter. Notice that this method is static. This means that you can just chain this method to the class. Class. Method

var age = DateTime. Today. Year - theDateTime.Year;

            if (DateTime. Now. DayOfYear < theDateTime.DayOfYear)

            {

                age--;            //cannot combine calc + return, must be on sep lines

                return age; //Else will not accept the --

            }

            else return age;

3. Use the extension

In this scenario “DateofBirth” = public DateTime DateOfBirth {get; set;} which is a property of a User object. Because “DateOfBirth’s” type is a DateTime object we can use our extension method. In this example, we are solving for the “Age” property of the Dto that does not exist in our User object. The User is the source and we are targeting the value of its DateOfBirth property which, by chaining the CalculateAge extension (passing DateTime dobFromUserObject). Our extensions logic is calculating age which returning an int as the age to the destination. The destination is dest. Age = Age property is being set for our Dto

Api>Helpers> AutoMapperProfiles

// <source, destination) --Take in a Source and return a Destination

//

CreateMap<User, UserForDetailedDto> ()

                . ForMember (dest => dest. PhotoUrl, opt => opt. MapFrom (

                    u => u. Photos. Where (p => p. IsMain != false).FirstOrDefault().Url)

                ). ForMember (dest => dest. Age, opt => opt. MapFrom (

                    u => u. DateOfBirth. CalculateAge ()

                ));

## Observable

One way to think of an Observable is that of an array whose items populate asynchronously over time. Since the Request to the Server is generally made and Response received in a “Service”. This is where they are most commonly used. Observables are basically a function that can give value/s over time a/synchronously. They are lazy and need. subscribe () for their results to be called. They have 3 potential values observer.

next (next value in stream, then do this) () => {…},

error (if an error has occurred, do this) error => {…},

complete (all values delivered, then do this) () => {…}

Observables are perfect for working with Http because, you must wait for data to arrive back over Http network call and the results of the response may be staggered in some way. To have control over each item individually, you need to have access to rxjs operators which are made by using the pipe method “Pipeable”.

### Observable operators – using “pipe”

You cannot compose observable operators (rxjs operators – Pipeable operators) without using a pipe. Think of a pipe like a physical plumbing pipe that receives water from a source. Now we have this stream of flowing water, this water is no longer at the source it is contained in our pipe. What this means is that we now have control over every drop (each item received from source) in our own pipe container.

In a pipe, we can take each item and transform it using operators (I.E. map, filter). So, when an observable item is taken from source, we can grab the item -> compose a new observable -> and apply an operator to that item so that it can be transformed. For example, $someSource. Pipe (map (x => x\*2), filter (x => x%3 ==0)). In the below we are taking the Response and not necessarily transforming it but, using each item in the Response stream to populate our variables. For example “paginatedResult. result” is just an array of User objects being populated one at a time from the Response. Whereas the “paginatedResult. pagination” is a single object that is being populated by the Response header.

getUsers(page?, itemsPerpage?): Observable<PaginatedResult<User[]>> {

    const paginatedResult: PaginatedResult<User[]> = new PaginatedResult<User[]>();

    let params = new HttpParams ();

    if (page != null && itemsPerpage != null) {

      params = params. append ("pageNumber", page);

      params = params. append ("pageSize", itemsPerpage);

    }

    return this.http.get<User[]>(this.baseUrl + "users", { observe: "response", params })

      .pipe (

        map (response => {

          paginatedResult. result = response. body;

          if (response. headers. get ("Pagination") != null) {

            paginatedResult.pagination = JSON.parse(response.headers.get("Pagination"));

          }

          return paginatedResult;

        }

        )

      );

  }

## Serialization

When you pass information across a network using Http. The values must exist as text so, they can be universally sent and received. Serializing = Turn into text, De-Serialize = Convert text to object. JSON is text base so, JSON can be converted either way.

Angular:

--Example: Turn string -> Object

JSON. parse (response. headers. get("Pagination"));

.NET:

--Example: Turn Object -> String

var camelCaseFormatter = new JsonSerializerSettings ();

   camelCaseFormatter.ContractResolver = new CamelCasePropertyNamesContractResolver ();

   response. Headers. Add

("Pagination", JsonConvert.SerializeObject(paginationHeader, camelCaseFormatter));

## Angular Routing

***Routes*** - View to display when a user clicks a link or pastes a URL into the browser address bar(“path”). A separate component is kept “routes.ts” that has, all rout configurations {path: “members/:id”, component: xyz}. So, when the client enters the Url path, a component is loaded and its accompanying template are added to the Spa.

**<router-outlet>** Exists on the root app template and is an outlet that routes are plugged into. When Client selects a new path. That path is plugged into this outlet and the template for that path is displayed in the html where this outlet exists.

**Parameterized routes** – This is a route that requires a parameter to be passed to it. The route itself exists in “routes.ts” like {path: “members/:id”, component: xyz}. You must pass this parameter in an html template like this path [routerLink]="['/members', user.id]". In this way, a value is passed to the path parameter

**Routeguards** - Protects routes from unauthorized users. implements CanActivate which has a single method canActivate () This is the logic to determine how the route path can be accessed ... You put a route guard in the path object of a route -> {path: "members", component: MemberListComponent, canActivate: [AuthGuard]}. These are not components, they are guards. ng g guard auth

**Rout Resolvers** - When the Client requests a route. The rout is activated and the Url path can be accessed prior to component being called. At the component level we would normally DI ActivatedRoute and access using snapshots (we have access to the complete path and its parts). A resolver is a separate class that implements Resolve<T> and has a single method “resolve” (route: ActivatedRouteSnapshot) that is an observable. This param takes in the Url path when the Client requested the route. Makes the snapshot of that route path immutable so, the resolver has, grabbed the Url path. **Purpose**: use a resolver to get data ready before the component is initialized. Grabs data from the route path and uses its implemented route () -> stores the Url path info -> to make Service call (it now has Observable<T>) -> Matching component can now subscribe and get this data (from the Api call) before the component is loaded. In this project, resolvers are used to identify errors with the Api call on the Client-side, display any errors to the Client and re-direct them to another path. And this is all happening (being resolved) before the component for the path is loaded by appending the router’s path.

{path: "members", component: MemberListComponent, resolve: {users: MemberListResolver}},

## Event Emitter

When you want to emit an object or property from one class and make in available to another. The event is happening in one class. This event outputs something (string, number object etc.) in the one class. The other class takes this event (what the other class produced) and uses it as a parameter which feeds this other classes method.

1. xyz = new EventEmitter<String> class 1 @Output () xyz = new EventEmitter<T>

2. Some method (class 1) kicks off and has a value you want to give to another class

3. this. xyz. emit (pass value here) class 1 – The Event Emitter is assigned this value

4. In the html (class 2) pass the event <elem (xyz) = method($event) >

Class 2 method receives Class 1 value as a parameter, notice it’s an $event

Only happens when class 1 actually emits an event so, its event driven

Class 2 method must accept the same Type as the event. The types must match

5. Class 2 method when executed, has the event as a param

method (event: any) {this. a = event. property} method (url: string) {this. a = url}

# Angular Services

Service = Centralizing the API req/resp to a single location vs. duplicating code in every class. Use services to create methods that communicate with API as a stream

Service Methods = Used to send Http Req (get, post, put, delete) -> get back a stream as the Response

1. Make a folder "\_services" to hold all of your services spa>src>app>\_services

a) R-click on the app folder -> “Open in Integrated Terminal”

b) mkdir \_services

2. ng g service auth -> component: auth.service.ts export class: AuthService

Components are auto-injectable but Services are not that is why “services” need @Injectable decorator

>src>app>\_services>auth.service.ts

@Injectable({

  providedIn: 'root' -- Which module is providing this service? The root module is so

}) this will be made available throughout the entire

src>app

export class AuthService { -- You can inject this instance to another class as DI

3. The service should get auto-registered in spa>app>**app. module. ts**

providers: [AuthService, ErrorInterceptorProvider], --src>app is a root level module

4. Import {Observable} from 'rxjs';

import {map} from "rxjs/operators";

Services use Observables because they use HttpClient to communicate with REST API's

In order to do something with a Resp (from server). You need to use rxjs operators

You pass these operators(filter/map/reduce) through a pipe method so, you can chain rxjs operators to the request.

The. pipe returns Observable<void> ...remember, Observables are lazy so, they don't return anything. You must subscribe to them to elicit the data

export class AuthService {

  baseUrl: string = environment. apiUrl + "auth/";

  jwtHelper = new JwtHelperService ();

  decodedToken: any;

  constructor (private http: HttpClient) {}

login (model: any): Observable<void> { -- interprets the body as JSON and

returns it as an Observable in JSON

    return this. http. Post (this. baseUrl + "login", model) -- return this.http.post("Url", obj)

      . pipe ( pipe= transform the response to your request as Observable

        . map ((response: any) => {= transform incoming values on-at-a-time

          const user = response; -- incoming resp is a bearer (“token”: "token string")

          if (user) {

            localStorage.setItem("token", user. token);

//decode (v) token string and hold as var

            this. decodedToken = this. jwtHelper. decodeToken (user. token);

//to see the fields of jwt decode method in browser

            console. log (this. decodedToken);

/\* console. Log ("user: " + this. decodedToken. unique\_name); > fields comes from AuthController ...claims + tokenDescriptor

             this. decodedToken. exp, this. decodedToken. iat, this. decodedToken. nbf,

this. decodedToken. nameid\*/

          }

        }) //--map transform one-at-a-time

      ); //--pipe transform to Observable

  }

--localstorage = store (k, v) in a web browser.

--So, data is persisted when you close the browser or, refresh the page.

--The data is still there (persisted). (-) You can only store strings, store as json

import { map } from "rxjs/operators";

### rxjs Operators Examples

These are examples of how you might use rxjs operators in a pipe to transform an observable

const filterOutWithEvens = filter (x => x % 2)

const doubleByValue = x => map (value => value \* x);

const sumValue = reduce ((acc, next) => acc + next, 0);

const source$ = Observable. Range (0, 10)

source$. pipe (

filterOutWithEvens,

doubleByValue (2),

sumValue)

.subscribe (console.log); // 50

## Base Url

Base Url = Http address to connect to the Api. It is just a property that you want to use throughout the entire app

Create base Url property for entire spa

src>environments>environment.ts (for non-production environment ...for now)

src>environments>environment.prod.ts (production ...for later)

src>environments>environment.ts

export const environment = {

  production: false,

  apiUrl: "http://localhost:5000/api/"

};

--This gives us an option to switch the api source across the entire application

When we run in production. We can just update the env.prod file with the final location.

Just change Url for production api. Therefore, we don’t have to change any of the "services" we have already created.

Since the base Url “apiUrl” is in src>environment. You import the object to use its properties. Here is an example of how you would access the Api Url in this “service”

src>app>\_services>user.services.ts

import {environment} from "../../environments/environment";

export class UserService {

  //apiUrl: "http://localhost:5000/api/"

  baseUrl: string = environment.apiUrl;

## Manually adding headers to Requests being sent to the Api

Here is what we have done so far:

1. Injected HttpClient into the constructor

2. Make methods that return an observable. One of the param options allows for headers that we can just pass into the parameter of the method

Example: This service methods is calling the Api’s endpoint to get back something of a certain type. In this case we are expecting a collection of “user objects” (actually user.Interface <-> UserDTO ). In the method parameter, we are passing the (endpointUrl, headers)

3. Manually create a new header

a) Create a variable below imports section to hold Header info to pass with the req

b) Create a new HttpHeader (K:V) this must be a legit header name

purpose: After login a user has populated their username/pass that are being stored in the browser's local storage. We want to grab that authentication info which is the saved token. Remember that the server passed the token as a Jwt Bearer (token : string) which had user details and a server signing key. We want to pass this info back to the server with this request so that we are authenticated with a valid token

c) Add the headers to the method issuing the request just add at end of method params

src>app>\_services>user. services. ts

import {Injectable} from '@angular/core';

import {environment} from”. /. /environments/environment";

import {HttpClient, HttpHeaders} from '@angular/common/http';

import {Observable} from 'rxjs';

import {User} from”. /\_models/user";

const httpOptions = {

  headers: new HttpHeaders ({

    "Authorization": "Bearer " + localStorage. getItem ("token")

  })

}

@Injectable ({

  providedIn: 'root'

})

export class UserService {

  //apiUrl: "http://localhost:5000/api/"

  baseUrl: string = environment. apiUrl;

  constructor (private http: HttpClient) {}

  getUsers (): Observable<User[]> {

    return this.http.get <User []> (this. baseUrl + "users", httpOptions);

  }

  getUser(id): Observable<User []> {

    return this.http.get<User []> (this. baseUrl + "users/" + id, httpOptions);

  }

}

# Angular Service methods - Observables

## GET

1. Angular “getUser ()” has no params and is a stream so, it is using HttpClient to call a GET method

2. stream = Observable<> = Something is sent via http and something is **return**ed as a response code with potentially something or nothing inside of it. The observable is doing the pitching and the catching

3. To get what the observable stream has (return from the api). Another method needs to subscribe

Example: The api when called is returning a collection of User objects that it is passing through the 200-response code. These User objects are received through the stream as Observable<User []>

Notice: return this.http.get<User []> Api returns a collection of User objects so, the stream must have a matching observable<User []>

export class UserService {

  //apiUrl: "http://localhost:5000/api/"

  baseUrl: string = environment. apiUrl;

  constructor (private http: HttpClient) {}

  getUsers (): Observable<User []>

{

    return this.http.get<User []>

(this. baseUrl + "users");

// Get api/users/

// ----

[HttpGet]

public async Task<IActionResult> GetUsers ()

{

var users = await \_repo. GetUsers ();

var usersToReturn =

\_mapper. Map<IEnumerable<UserForListDto>> (users);

return Ok(usersToReturn);

## GET w/ Id

The observable is passing an “id” as a parameter which is being sent in the path to the api. Whatever component subscribes to this Service must pass “id: any”. That subscribe () will kick off this service and pass the Url path to the Api. The Api grabs the “id” -> runs its method and returns the UserDto. This Dto matches up with app>models>user.ts () which is a replica of what a User object or, its Dto can be.

Notice: The observable get <User> is expecting a User object (or acceptable Dto) as a response. Which matches what the Api is returning as an object inside of the OK (object) response

spa>app>\_service>userService. Ts

//baseUrl: "http://localhost:5000/api/"

getUser(id): Observable<User>

{

    return this.http.get<User>

(this. baseUrl + "users/" + id);

}

//Get api/users/{id}

        //

        [HttpGet("{id}")]

        public async Task<IActionResult> GetUser (int id)

        {

            var user = await \_repo. GetUser(id);

            var userToReturn = \_mapper. Map <UserForDetailedDto>(user);

            return Ok(userToReturn);

        }

## POST w/ variables

1. Angular “setMainPhoto (p1, p2)” takes in (2) params and is a stream so, it is using HttpClient to call a POST method. When using the Http stream to call a post method, a body is required (empty body is ok). Notice in the Api method that no objects are part of this parameter…aka…no body is expected. The values will be caught from the Url path

2. stream = Observable<> = Something is sent via http and something is **return**ed as a response code with potentially something or nothing inside of it. The observable is doing the pitching and the catching

3. To get what the observable stream has (return from the api). Another method needs to subscribe

Example: The api when called is returning nothing but a 204-No-Content-response code. This response is received through the stream as Observable<Object>. The purpose of this call in this case is not to get something back. It is to pass needed information (userId, photoId) so that when this path is called in the Api. This particular photo will be set as “main”. Any other Photo in the User’s collection of photos will have a Photos.IsMain = false.

export class UserService {

  //apiUrl: "http://localhost:5000/api/"

  baseUrl: string = environment. apiUrl;

  constructor (private http: HttpClient) {}

setMainPhoto (userId: number, photoId: number):

Observable<Object> {

return this.http.post (this. baseUrl +

"users/" + userId +

"/photos/" + photoId +

"/setmain", {});

}

//POST

api/users/5/photos/{photoId}/setMain

[HttpPost("{photoId}/setMain")]

        public async Task<IActionResult>

setMainPhoto (int userId, int photoId)

{

// If we are able to save these updates

//

if (await \_repo. SaveAll ())

return NoContent ();

else return BadRequest ("Could not set this photo to main");

## POST with no body

In the below scenario a service method was made to interact with the api. It sends an empty object body and returns nothing. However, the information it sets in the Url path give the api what it needs to change the IsMain property of a single photoId of a collection of photos tied to a User.

We call this service to interact with the api on our behalf in this component.ts. we pass the needed params (the “photo” from our component and the “userId” stored from login). In order to activate the service, we must “subscribe ()” to it. If our subscription is successful, we use the next => -or- () => since nothing is being returned

src>app>\_services>userService

setMainPhoto

(userId: number, photoId: number)

: Observable<Object> {

return this.http.post

(this. baseUrl +

"users/" +

userId + "/photos/" +

photoId + "/setmain",

{}

);

}

--The userId will be the result of a previous api call to a different service “authService”. Whereby, the user logged-in with username/pass -> sent to api -> returned token -> held as variable “decodedToken” that has all the token info to include namedid = userId

src>app>members>photo-editor>photo-editor. component

//Pass the 2 params from this component needed to call this service method, then subscribe to its stream

setMainPhoto (photo: Photo) {

    this. userService

      . setMainPhoto (

this. authService. decodedToken. nameid,

photo.id). subscribe (

      ()  => {console.log(“success”);},

error => {this. alertify. error(error)}

);

--The “photo” is received from this component’s template. The \*ngFor within a div gives us access to all photos for this and is what is populating “photo” -> photo.Id.

## DELETE

The observable returns Observable<Object> b/c it is expecting a response code. The two params being passed are what the api requires to be set into its Url path. The Api retrieves the two params which are sent as a stream in the Url path -> the Controller method is executed and returns a response code. Whatever component subscribes to this observable must pass the two params and can expect a 200-response code or 400-response code

src>app>\_services>userService

  //apiUrl: "http://localhost:5000/api/"

deletePhoto (userId: number, photoId: number)

{

    return this. http.

Delete (this. baseUrl + "users/" +

userId + "/photos/" +

photoId)

}

// DELETE

api/users/{userId}/photos/{photoId}

        //

        // ---

[HttpDelete("{photoId}")]

public async Task<IActionResult>

DeletePhoto (int userId, int photoId)

        {

if (photoFromRepo. PublicId == null)

            {

   \_repo. Delete (photoFromRepo);

            }

if (await \_repo. SaveAll ())

            {

   return Ok (); --200 is the correct response for a successful delete

            }

   else return BadRequest ("Failed to delete the Photo");

# Angular Subscribe ()

Subscription is how you connect the service observable method (interaction with api) to your component. Because this is a stream, you must subscribe to this method in order to get the data from the stream (observable<Type>)

# Angular routing

## How to set up

***Routes*** tell the Router which view to display when a user clicks a link or pastes a URL into the browser address bar

### Register RouterModule

Registering the “RouterModule” at the app root level "app. module". This allows the entire app to use this/any registered modules + auto-registers components to roll up to this app-root

src>app>app. module.ts

import {NgModule} from '@angular/core';

@NgModule({ --Registering the “RouterModule” in **@NgModule** metadata initializes the

  declarations: [], router and starts it listening for browser location changes.

imports: [ -- configures “RouterModule” with the (routes) in one step

    RouterModule. forRoot(appRoutes) --configure this module with any routes we made from angular

routing “appRoutes. ts” (our custom routes)

  ],

## How to set route paths?

Define Routes= [] as an array of objects {}. Each object is a Url paths that map to your -> components

\*\*Order is important.

echo > routes.ts Makes a new file that will contains routes for our application

src>app>routes. ts

import {Routes} from '@angular/router';

export const appRoutes: Routes = []

-or- ng g module app-routes –module=app Registers in app. module. ts as an import + makes it a module

src>app>routes.ts

export const appRoutes: Routes = [

{path: "", component: HomeComponent},

{path: "members", component: MemberListComponent},

{path: "\*\*", redirectTo: "", pathMatch: "full"}

]

]

### <router-outlet>

<router-outlet> is an outlet that routes are plugged into. Having an outlet + routes = Deep links exist and you can navigate to the page by appending the Url. Ideally, users should be able to click a link to navigate rather than pasting a route URL into the address bar

src>app.component.html

<app-nav></app-nav>

<router-outlet></router-outlet>

### parameterized id for Route Path

This is a path that requires a param to be passed. This is how it is set-up in routes

src>app>routes.ts

{path: "members/:id", component: MemberDeailComponent},

This is how we would pass the route from a component template

src>app>members>member-card>member-card.component.html

<ul class="…">

            <li class="…"><button class="…"

[routerLink]="['/members', user.id]">

<i class="…"></i></button></li> --notice that user. property Info is being grabbed from this class and passed with the route

### snapshot. params (Import ActivatedRoute)

When the route is requested (see above) on-click of the button. host/users/# The rout is activated and can be access prior to component being called. We need to have logic in this component to grab the data via a method on ngOnInit (). A method is called on init that grabs the route id and passes it to the Service to assign a specific User object from the Api

src>app>member>member-detail>member-detail. component. ts

import {ActivatedRoute} from '@angular/router';

export class MemberDeailComponent implements OnInit {

  user: User;

  constructor (private userService: UserService, private alertify: AlertifyService,

    private route: ActivatedRoute)} <-- This is what turns on snapshot. params

  ngOnInit () {

    this. loadUser (); <--When component called, execute this method

  }

  //using rout snapshots to get params

  loadUser () {

--We pass the id from route as the param to this service method

    this. userService. getUser(+this. route. snapshot. params['id'])

      . subscribe (

        (user: User) => {this.user = user;},

        error => {this. alertify. error(error);}

      )

  }

}

-- A snapshot is used to grab the params that are bring passed

-- + is used to convert this param from string to a number

-- Service method requires this param as an id: number. Once the param

Is in place. Subscribe to the observable which returns a specific user based on the

param that we passed to it and user: User; is initialized on init of this component

## How to use [routerLink] in html templates for <a> navigation

Nav component has <a> nested in <li> as a drop-down menu for navigation. The <a> has routing to direct people to the appropriate URL [routerLink]="['/routePath']" routerLinkActive="router-link-active"

--Adding [routerLink] to the nav.html -a router… active = light-up on click/hover

<ul class="navbar-nav mr-auto" \*ngIf="loggedIn ()">

<li class="nav-item" routerLinkActive="router-link-active">

    <a class="nav-link" [routerLink]="['/members']">Matches </a>

  </li>

--<a> on click will take you to the Url http://host/members

--Change app.component.html structure to accommodate routing -> router-outlet structure

src>app>app.component.html

<app-nav></app-nav>

<router-outlet></router-outlet>

Here you can see that only the nav bar is being displayed on the site but, in our routing (Routes path), we have any non-specific path “\*\*” to default to our home component. This means that both “app-nav” and “home” will be displayed when the application starts

The way to navigate around the rest of the site is to click the <a> which will have “routerLinks” with rout paths that change the Url and take you to other components.

## Using methods to Navigate to new components

When a customer logs in. The login () is called. When this happens, we want the routing to change

When a customer logs out. The logout () is called. When this happens, we want the routing to change

To use routing for a method inside of a class. We need to inject Routing for that class

Inject routing as a service -      private router: Router

Make routing change when method called this. router. navigate(['/members']);}

Subscribe to a service’s observable (api call) …call routing from inside the. Subscribe () (w/ subscribe param)

src>app>nav. component. ts

import {Router} from '@angular/router';

export class NavComponent implements OnInit {

  model: any = {};

 constructor (public authService: AuthService, private alertify: AlertifyService,

    private router: Router) {} <----inject routing

  ngOnInit (): void {

  }

  Login (): void {

    this. authService. Login (this. model)

      . subscribe (

        next => { //next=Req is successful + Resp has data from api

          this. alertify. success("Logged in successfully");

        },

        error => {this. alertify. error (error);},

--route to this Url after all successful req/resp ----

        () => {this. router. navigate(['/members']);} <-- Subscribe param used

   Effect routing change

--Do this after all next have been called ----------

      );

  }

//alternate use complete (): void after success req/resp

1. Effect routing from an observable method by subscribing to it

Subscribing to a service api call. Here are the subscription parameters:

next = Req is successful, response from api has token (our request has our log in details)

error = data stream error that we are grabbing/passing and returning to browser via alertify

complete= () => {…after all next's are called, …do this, …returns void}

2. Effect routing from regular method

### this. router. navigate ([“…/path”]);

make sure you have injected routing as a service

src>app>nav. component. ts

import {Router} from '@angular/router';

export class NavComponent implements OnInit {

  model: any = {};

 constructor (public authService: AuthService, private alertify: AlertifyService,

    private router: Router) {} <----inject routing

  ngOnInit (): void {

  }

Logout (): void {

    localStorage. removeItem("token");

    this. alertify. message ("logged out");

    this. router. navigate(['/members']); <-- call routing to change when this method called

  }

\*Router change in methods = this. router. navigate(['…']) in html = [routerLink]="['/lists']"

# Routeguards

Routeguards protect routes from unauthorized users. implements CanActivate ... can this route

-Make Folder + guard class: be activated (t/f)

src>app mdir \_guards Make a dir to hold auth guards for routes

cd \_guards ng g guard auth --skipTests Make guard with no tests(spects.ts)

-which interfaces to use...space to select

-select "CanActivate" return

auth. guard. ts is kind of like a separate component. I stripped off most return options since this is just a bool to

see if, a user is logged in or not. If not logged in, they can only access the home page. 1st, make the guards class -> Inject the guard into the constructor of routes. ts to use this protection.

1. Make the route guard + implement CanActivate method

src>app>\_guards>auth.guard.ts

import {Injectable} from '@angular/core';

import {CanActivate, Router} from '@angular/router';

@Injectable ({

  providedIn: 'root'

})

Protection for url path routes.  Has a single method (t/f) to verify if this path can be accessed or not?

//

export class AuthGuard implements CanActivate {

  constructor (private authService: AuthService,

    private alertify: AlertifyService, private router: Router) {}

  canActivate (): boolean { //Method call of canActivate true = user can proceed

//Call service's method to check if tokens exist for login

    if (this. authService. Loggedin ())

      return true;                   //if true this means user has logged in

    else {

      this. alertify. error ("Area:  Access denied");

    }

  }

}

Example using a guard in the appRoute configuration class routes.ts to affect a single route routes.ts

2. Set the guard to a specific route

src>app>routes.ts

import {AuthGuard} from”. /\_guards/auth. guard";

export const appRoutes: Routes = [ --can only access this route if guard = true

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

\*canActivate: [] Is a type that can hold an array of guards

Holds an array of dependency-injection tokens used to look up CanActivate () handlers, to see if, the rout is T or F.

## Using guards to protect multiple routes

Routes are treated as separate objects {path: “”, component}

Here we want to take a rout and give it a bunch of children {path: “…”, guard stuff, children: [\_\_{c1}, {c2} \_\_] }

This accomplishes nesting multiple routes as children inside an object rout

src>app>routes.ts

import {Routes} from '@angular/router';

import {HomeComponent} from "./home/home. component";

import {MemberListComponent} from "./member-list/member-list.component";

import {MessagesComponent} from ". /messages/messages. component";

import {ListComponent} from "./list/list. component";

import {AuthGuard} from "./\_guards/auth. guard";

export const appRoutes: Routes = [

    {path: "", component: HomeComponent},

    {

        path: "“, //localhost:4200 + “…” path + add Url path

//always run guards for these children. verify if this path can be accessed or not

runGuardsAndResolvers: "always”,

        canActivate: [AuthGuard], //This single guard (one of many)

        children: [

            {path: "members", component: MemberListComponent},

            {path: "messages", component: MessagesComponent},

            {path: "lists", component: ListComponent}

        ]

    },

    {path: "\*\*", redirectTo: "", pathMatch: "full"}

]

## Route Resolvers

Route Resolvers – get data from an activated route that allows data to be loaded before a component is called.

Purpose: use a resolver to get data ready before the component is initialized. Grabs data from the route path and uses its implemented route () -> store the Url path info -> make Service call (it now has Observable<User>) -> Matching component can now subscribe and get this data before the component is loaded

1. make a folder to hold all resolvers mkdir \_resolvers

2. make resolver file echo > member-detail.resolver.ts must build from scratch

a) Make injectable

b) implement resolve<Type you want to grab from route> comes with ActivatedRouteSnapshot

3. Implement the resolve () method. method takes in route using ActivatedRouteSnapshot as a parameter and returns an Observable.

-ActivatedRouteSnapshot =immutable object. This a param of resolve () that must be passed to me. It represents a Url route <http://local:4200/members/4> fixed value

--ActivatedRoute = fluid route from the component.ts method used to grab live route from its component.html [routerLink]="['/…', user.id]" on-click. This Url value could be different each and every time based on where the Client clicks. This will not be used in resolve (). It is used in the component.ts method to “observe” the Url route in real-time coming from its component.html

Building resolve ()

Passes data | params | Url to the service method w/o subscribe, it auto subscribes for us. Basically, just calling the service and passing the “id” as a route.snapshot.params from a snapshot in time, which is what the “ActivatedRouteSnapshot” does for us

Option 1: Get route param directly from routes.ts Url using ActivatedRouteSnapshot

Pass param from the route itself to the Service

src>app>\_resolvers>member-detail.resolver.ts

import {Injectable} from "@angular/core";

import {User} from "../\_models/user";

import {Resolve, ActivatedRouteSnapshot, Router} from "@angular/router";

import {UserService} from "../\_services/user.service";

import {AlertifyService} from "../\_services/alertify.service";

import {Observable, of} from "rxjs";

import {catchError} from "rxjs/operators";

@Injectable ()

export class MemberDetailResolver implements Resolve<User> {

    constructor (private userService: UserService,

        private router: Router,  <-- Turns on routing

private alertify: AlertifyService) {}

--method: Pass a snapshot of a given route

    Resolve (route: ActivatedRouteSnapshot) <-- Route + Component info snapshot in time

i.e. route.snapshot.params | Url | data

: Observable<User> { <-- Returns

        this.userService.getUser(route.params['id']) <--from the routes.ts

         .pipe(              //pipe is used here just to deal with potentail error

            catchError(

               error => {

                  this.alertify.error("Problem retrieving data");

                  this.router.navigate(["/members"]) <-- back to page they came from

return of(null); <--Return an observable “of”

                    })

            )

    }

Building resolve ()

Option 2: re-use resolve () pass routes.ts Url using ActivatedRouteSnapshot only to make sure the params are consistent for use of this method

Get “id” a different way by calling it from AuthService which has a decodedToken property which has “nameid” of the token

src>app>\_resolvers>member-edit.resolver.ts

export class MemberEditResolver implements Resolve<User> {

    constructor (private userService: UserService, private authService: AuthService,

        private router: Router, private alertify: AlertifyService) {}

//ActivatedRouteSnapshot= Immutable Url passed from routes.ts

    resolve(route: ActivatedRouteSnapshot): Observable<User> {

        return this.userService.getUser(this.authService.decodedToken.nameid)

            .pipe(              //pipe is used here just to deal with potentail error

                catchError(

                    error => {

                        this.alertify.error("Problem retreiving data");

                        this.router.navigate(["/members"])

                        return of(null);

                    })

            )

    }

4. Add this to a specific route so, the route knows that it (the route) is responsible for fetching the data, not the component. We do this by adding the Route Resolver resolve () to the Route path

src>app>routes.ts

--This step connects a specific route -> (component -> resolver)

import {MemberDetailResolver} from './\_resolvers/member-detail.resolver';

import {MemberListResolver} from './\_resolvers/member-list.resolver';

export const appRoutes: Routes = [

    {path: "", component: HomeComponent},

    {

        path: "", //localhost:4200 +blank path to add onto this

//always run guards for these children. verify if this path can be accessed or not

        runGuardsAndResolvers: "always",

        canActivate: [AuthGuard], //This single guard (one of many)

        children: [

{path: "members", component: MemberListComponent,

resolve: {users: MemberListResolver}}, <-- Route resolver added here, ‘users’=access

path is passed to the resolver as a param

            {path: "members/:id", component: MemberDetailComponent,

resolve: {user: MemberDetailResolver}}, <--This is the ActivatedRouteSnapshot being used

from MemberDetailResolver. This means when (routing is hit) -> the component is called -> which calls our Resolver -> which calls its resolve () which, now has the captured “url/:id”. This resolve () uses the “id” to pass as a param -> service api call which returns a user object as an observable -> To the component that is tied to this path … we just need to go to that component and subscribe to this Observable via ActivatedRoute

5. Register the Route Resolver to the list of providers (guards, services, etc.)

src>app.module.ts

import {MemberDetailResolver} from './\_resolvers/member-detail.resolver';

@NgModule({

  declarations: [

    MemberDetailComponent],

imports: [Modules\_here],

providers: [MemberDetailResolver

  ],

6. Add a method …to the component that needs the data prior, to this component being initialized.

Not a named method but, put in ngOnInit. Uses the ActivatedRoute tied to this component from routes.ts. Basically, we are calling “route” (the snapshot of user-click -> calls a route -> which we capture w/ our route resolver in routes.ts) We have access to this route because the two (the route + our resolver) are tied together in routes.ts. Hence we are calling that “route”

This “route” gives us access to route. snapshot. params | url | data. In this case, the resolve () returns an observable object so, we would use route .data

Since resolve () returns an Observable<User> we would need to subscribe to this stream -> initialize our local user:User with “user” (the reference from our route ) so that, it is available to our component

src>app>members>member-detail>member-detail.component.ts

import {Component, OnInit} from '@angular/core';

import {User} from 'src/app/\_models/user';

import {UserService} from 'src/app/\_services/user.service';

import {AlertifyService} from 'src/app/\_services/alertify.service';

import {ActivatedRoute} from '@angular/router';

@Component({

  selector: 'app-member-detail',

  templateUrl: './member-detail.component.html',

  styleUrls: ['./member-detail.component.css']

})

export class MemberDeailComponent implements OnInit {

  user: User;

  constructor (private userService: UserService, private alertify: AlertifyService,

private route: ActivatedRoute) {} <--Gives you access to the Url route from routes.ts

--The data is received from the route itself

  ngOnInit() {

    this.route.data

.subscribe(data =>

{this.user = data['user']}) <--‘user’ is name from routes.ts resolve: {objName}

  } call this route which is an observable stream

   call its data that is being passed

subscribe to this data

} assign local var to stream[‘nameOf\_obj\_passed’]

## Route Guards + CanDeactivate ()

These are used to pause users from navigating away from a page. These are tied to a route (Url page) that execute when a user tries to go to another Url location

Example: If user is saving data on a page but, tries to go to another page. The data will be lost so, before going to that other page. This Route Guard and its method CanDeactivate is executed. In this example we are using a “confirm ()” pop up box to make sure the user has a chance to save before navigating to a different location.

1. Create the guard

2. Register the guard in app.module with the other services

src>app.module.ts

import {PreventUnsavedChanges} from "./\_guards/prevent-unsaved-changes.guard";

providers: [

    PreventUnsavedChanges

  ],

3. Add the guard to the routes.ts … attach the guard to the path you are trying to protect

src>app.routes.ts

import {PreventUnsavedChanges} from './\_guards/prevent-unsaved-changes.guard';

{path: "member/edit", component: MemberEditComponent,

            resolve: {user: MemberEditResolver}, canDeactivate: [PreventUnsavedChanges]},

            {path: "messages", component: MessagesComponent},

## DOM event @HostListener

In this scenario the host of the Angular app is the browser. We are listening to possible events that will happen with the browser. <https://www.w3schools.com/jsref/dom_obj_event.asp> beforeunload the event occurs before the document is about to be unloaded (close the browser window)

Scenario: From our component we want to stop users from closing the window before they have saved their changes. In the below solution, a browser auto-initiated box will appear similar to a pop-up box

Route guards and resolvers are tied to a specific route but … you cannot use them to stop a user from closing a browser tab (beforeunload $event). Because, this is outside of the route

@HostListener = Listen for browser events

export class MemberEditComponent implements OnInit {

  user: User;

  @ViewChild("editForm", { static: true }) editForm: NgForm;

  /\*  static - True to resolve query results before change detection runs,

false to resolve after change detection. Defaults to false.\*/

  @HostListener ("window:beforeunload", ["$event"])

  unloadNotification($event: any) {

    if (this.editForm.dirty) {

//onbeforeunload event =

before the document is about to be unloaded (close the browser window) return a browser pop-up to make sure they really want to close the page. you must assign/attach the event on the window object, and use the returnValue property

      $event.returnValue = true;

    }

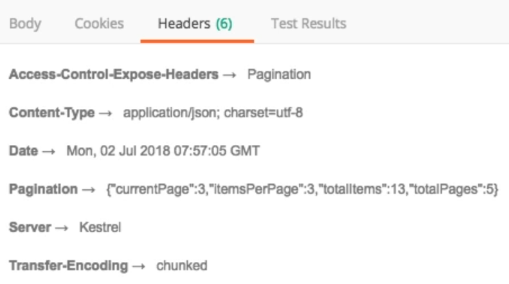
  }

# Angular Paging – SPA

Scenario: At this point, we have an API controller method that is able to receive query parameters through a GET method by accessing the UsersController. Once the request is sent, the Api Controller returns a response code, response body and response header. We have only tested this in Postman and have no way to actually send from the SPA as of yet.







## Set-up Paging

### 1. Create an interface to capture/match Header Response information

### 2. Create a Class to capture/match Response body. The response body at current is a JSON object of Uers

app>models>pagination. ts

export interface Pagination { <-- To capture/match incoming Header response

    currentPage: number;

    itemsPerPage: number;

    totalItems: number;

    totalPages: number;

}

export class PaginatedResult <T> { <--To capture/match incoming response body. This

    result: T; can be User, User [] etc. whatever response is

    pagination: Pagination; expected from a Service call.

} “result” = hold response body here

“pagination” = object to hold response headers

src>app>\_services>user. service .ts

getUsers (): Observable<User[]> {return this.http.get<User[]> (this.baseUrl + "users");}

### 3. Update the service

At current, the service is using a GET request for the UsersController and returning a Json multiple User objects as the body. We now need to:

a) send query parameters b) Receive response header + body data

The GET method has overloads to accomplish both of our new goals. See overloads below

(method) HttpClient. get<User []>(url: string, options: { <--Options to send with Req

  headers?: HttpHeaders | {

      [header: string]: string | string[];

  };

  observe: "response"; <-- This is defaulted to “body” but, we can ask to

“observe the response” so, we can access body + header

  params?: HttpParams | { <-- Send params to Api using a params object

      [param: string]: string | string[];

  };

Service Goals

The service is responsible for the Observable (request + response stream via HttpClient). The observable can send a request to the Api w/ our paging query params, sent as HttpParams () params. The observable stream can receive a response back from the Api. This Response from the Api Controller method has Header as HttpResponse.header, and Body as HttpResponse.body params.

#### 1. Object to capture Response Response: body = Users [] and header = Json object as K-V

getUsers (page?, itemsPerpage?): Observable<PaginatedResult<User[]>> {

    const paginatedresult: PaginatedResult<User []> = new PaginatedResult<User []> ();

The PaginatedResult takes in type<T>. Here, we are passing <User []> so, this means that “paginatedresult” is a new object that accepts an array of Users. We need this to capture the response body. This object also, has an undefined Pagination member object which is empty and will be used to capture the response headers.

#### 2. Parameters that we want to send to the Api with the request

let params = new HttpParams ();

    if (page != null && itemsPerpage != null) {

      params = params. append ("pageNumber", page);

      params = params. append ("pageSize", itemsPerpage);

    }

Since the parameters of getUser (optional1?, optional2?) are optional, we need to conditionally add them to the HttpParams (). If there are no params, we will pass this value as null. Else, we can send these Request parameters as K-V pair. From the overload mentioned above the key must be a string and the value is always received as a string. So, once all appends are made to the HttpParams () object. This finished product can be passed in the request

#### 3. Sending up the Request

return this.http.get<User[]>(this.baseUrl + "users", { observe: "response", params })

The HttpClient will make the request and receive a response back. When this Service method is subscribed to, it will return that result. Notice that the above is only for the HttpRequest. **Our request includes: url: string, options:**

a) Type = get and we expect an array of Users to be returned back from the response

b) url: string = this.baseUrl + “users”

c) options = HttpParams() which, are optional from whoever wants to use this method.

#### 4. How to deal with the response

When a Response is received back via Http from a server its default is to “observe the body”. We have told the request that we want to include options: “to include the full response”. In order to transform this response to isolate the response. header and response. body. We need to make the observable pipeable (transform to use rxjs operators). So, “pipe” the stream of the response and “map” each item coming through that we can affect one-at-a-time.

return this.http.get<User[]>(this.baseUrl + "users", { observe: "response", params })

      .pipe(

        map(response => {

          paginatedResult. result = response. body; --Take body and hold in this var

          if (response. headers. get("Pagination") != null)

{

            paginatedResult. pagination = JSON. parse (response. headers. get

("Pagination"));

          } --take response header (“..” matches extension method that returned header as a K-V, this is the name of the key, header. get = return value of this key) and hold in this var

          return paginatedResult; --return the fully formed object that has both header +

body

        } //end response mapping

        ) //end map

      ); //end pipe

# AuthO JwtModule

<https://github.com/auth0/angular2-jwt>

This works in conjunction with JwtHelperService import which helps us manage our tokens. Used to decode the token and see if it is expired or not. In this library you also get an HttpInterceptor which attaches the web token to the client request. In our application we are using a token for validation and a request every-time we interact with the api except

-Login () which sends username, password and gets back a new token

-Register () which send new user information

In all other activities we are requesting interaction with the api and need token info

Usage Injection - How set-up AuthO

src>app.module

import {JwtModule} from "@auth0/angular-jwt";

import {HttpClientModule} from "@angular/common/http";

export function tokenGetter () {

return localStorage.getItem("your\_token\_name\_here");

}

@NgModule({

bootstrap: [AppComponent],

imports: [

// ...

HttpClientModule,

JwtModule.forRoot({

config: {

tokenGetter: tokenGetter, --This is the method we just made above as an export

allowedDomains: ["example.com", “localhost:5000”],

disallowedRoutes: ["http://example.com/examplebadroute/", “needs full path/api/auth”],

--disallowed is optional

},

}),

],

})

export class AppModule {}

Previously (services -> manually add header) we had manually added a variable to hold a new HttpHeaders … the header was passing our token info and putting at the end of the request params….in “service” as we sent requests to the api. We no longer need this http option. Because

**This library relies on the URL interface which is not supported in IE11.** To solve the IE11 compatibility, you can add a polyfill.

* run  npm i --save url-polyfill
* add  import 'url-polyfill'; to  polyfills.ts   in your project

# Dotnet Stuff

## Add Coors policy

CORS - Browser security feature that protects you from accessing data from an unknown origin.

Our API is a foreign site (localhost:5000) being Requested by angular (localhost:4200)

A cors policy is needed else an ERROR - "No Access-Control-Allow-Origin" header is coming from request

app.UseRouting();

            app.UseAuthentication();

            app.UseAuthorization();

            app.UseCors(req => req.AllowAnyOrigin().AllowAnyMethod().AllowAnyHeader());

## Global exception handling

properties -> launchSetting.json "Development" -or- switch to "Production"

you want to use global exception handling to limit try/catch blocks in each class

public void Configure (IApplicationBuilder app, IWebHostEnvironment env)

{

if (env.IsDevelopment ()) Exception Handler adds middleware to the

{ pipeline. That catches the exceptions

app.UseDeveloperExceptionPage (); logs it-> re-executes the request in an

} alternate pipeline...so, no unhandled, this

else is handling ...I.E equivalent of global try/catch

{

app.UseExceptionHandler(builder => This needs to be configured and returns IApplicationBuilder

} Set all exceptions as 500, output our custom Extensions class method "AddApplicationError"

passes our error to the response. Return s status 500 w/ message in the headers

# Route Paths – Controller

1. [HttpXxx(“{xyz}”)] param values MUST MATCH Method (int xyz)

2. [HttpXxx ("\_\_name\_\_")] Appends this to the route’s path path/\_\_name\_\_

3. [HttpXxx (Name = "\_\_name\_\_")] Names this route to be accessed internally path/

## User Controller Examples

### Base route

[Authorize]

    [Route("api/[Controller]")] http://host/api/users

    [ApiController]

    public class UsersController: ControllerBase

    {

-- All methods of this controller take the given route

### GET Route + Path

// Get  api/users/

        // ----

        [HttpGet]

        public async Task<IActionResult> GetUsers ()

        {

### Get Route w/ Id + Path

//Get  api/Users/{id}

        //

        [HttpGet("{id}")]

        public async Task<IActionResult> GetUser (int id)

        {

### PUT Route w/ Id + Path

//Put  api/Users/{id}

        //

[HttpPut("{id}")]

public async Task<IActionResult> UpdateUser (int id, UserForUpdateDto userForUpdateDto)

## Photo Controller Examples

### XII.1.i Base route

[Authorize]

    [Route("api/users/{userId}/photos")] http://host/api/users/5/photos

    public class PhotosController: ControllerBase

    {

-- All methods of this controller take the given route

### GET Route w/ Id + Path

// GET api/users/5/photos/{id}

        // GetPhoto is the name of this route

        // ---

[HttpGet("{id}", Name = "GetPhoto")]

        public async Task<IActionResult> GetPhoto (int id)

        {

### POST Route + Path

// POST api/users/5/photos

        //

        // ---

        [HttpPost]

        public async Task<IActionResult> AddPhotoForUser

        (int userId, [FromForm] PhotoForCreationDto photoForCreationDto)

        {

### POST w/ Id & + added Path

//POST api/users/5/photos/{id}/setMain

        //

        // ---

        [HttpPost("{photoId}/setMain")]

        public async Task<IActionResult> setMainPhoto (int userId, int photoId)

\*\* Route path “{photoId}” MUST MATCH the param variable

## DELETE

The observable returns Observable<Object> b/c it is expecting a response code. The two params being passed are what the api requires to be set into its Url path. The Api retrieves the two params which are sent as a stream in the Url path -> the Controller method is executed and returns a response code. Whatever component subscribes to this observable must pass the two params and can expect a 200-response code or 400-response code

src>app>\_services>userService

  //apiUrl: "http://localhost:5000/api/"

deletePhoto (userId: number, photoId: number)

{

    return this. http.

Delete (this. baseUrl + "users/" +

userId + "/photos/" +

photoId)

}

// DELETE

api/users/{userId}/photos/{photoId}

        //

        // ---

[HttpDelete("{photoId}")]

public async Task<IActionResult>

DeletePhoto (int userId, int photoId)

        {

if (photoFromRepo. PublicId == null)

            {

   \_repo. Delete (photoFromRepo);

            }

if (await \_repo. SaveAll ())

            {

   return Ok (); --200 is the correct response for a successful delete

            }

   else return BadRequest ("Failed to delete the Photo");

## Auth Controller Examples

### Base Route

[Route("api/[controller]")] http://host/api/auth

    [ApiController]

    public class AuthController: ControllerBase

    {

-- All methods of this controller take the given route

### POST Route + Path

// POST  api/auth/register

        // ----

[HttpPost("register")]

        public async Task<IActionResult> Register (UserForRegisterDto userForRegisterDto)

        {

// POST  api/auth/login

        // ----

        [HttpPost("login")]

        public async Task<IActionResult> login (UserFromLoginDto userFromLoginDto)

        {

# Response Codes -Controller

For simple changes. It is not necessary to use a full-blown PUT or Patch. Van accomplishes with POST

## Success Codes

### 200 OK () – Returns an object response (or, empty or string)

Object that produces a Microsoft. AspNetCore. Http. StatusCodes. Status200OK response.

// Anonymous object passed that we can customize

            //

            return Ok (

                new

                {

                    token = tokenHandler. WriteToken (token),

                    user

                }

            );

### 201 Created – Returns an object response (or, empty or string)

return StatusCode (201);

## 201 CreatedAtRoute ()

This is used when you want to return a 201 response with the created object info and the location of the created object. The newly created location info gets passed in the header and the body has the newly created object details. It takes 3 params in order to create.

ControllerBase. CreatedAtRoute (string routeName, object routeValues, object value)

Api>Controllers>AuthController.cs

1. [HttpGet("{id}", Name = "GetPhoto")] --Route to be called after this object is created

return CreatedAtRoute ("GetUser",

2. The object routeValues = Path + required info in path. So here you cannot call getUser () … after the object is created w/o an Id param for a specific user

                new

                {

                    Controller = "Users",

                    id = createdUser. Id

                },

                  userToReturn);

3. object value= the returned object info. We are using a DTO b/c we do not want to return sensitive user data like password/salt etc.…

2nd Example: Notice that the Controller does not need info b/c the named route is coming from the same Controller vs. the example above. Where the named route is coming from a different Controller.

var phototoReturn = \_mapper. Map <PhotoForReturnDto>(photo);

                // Show location header of created resource

                // string routeName = Name [httpGet{"{id}"}, Nme = "GetPhoto"]

                // object routeValues= new object with values (params) that are required

                // by this route to call the controller method

                // object value = The object being created and returned

                return CreatedAtRoute ("GetPhoto",

                    new {userId = userId, id = photo. Id}, phototoReturn);

### 204 NoContent() – Returns a no content, empty object response

Microsoft.AspNetCore. Http.StatusCodes. Status204NoContent response.

// If we are able to save these updates

            //

            if (await \_repo. SaveAll ()) return NoContent();

            else return BadRequest("Could not set this photo to main");

## Failure Codes

### 400 BadRequest () – Returns an error object response (or, empty, or string)

ControllerBase.BadRequest (object error) Microsoft.AspNetCore.Http.StatusCodes.Status400BadRequest response.

//Is this passed photo the main photo for this user already?

            //

      if (photoFromRepo.IsMain) return BadRequest ("This is already the main photo");

### 401 unauthorized – Returns an unauthorized object response(or, empty, or string)

UnauthorizedObjectResult ControllerBase.Unauthorized(object value)

//Does this photo exist in this User's photo collection (yes/No)

            //

            if (!user.Photos.Any(p => p.Id == photoId))

                return Unauthorized("No matched photos");

# System.Security.Crytography

These are Objects you can create that can scramble inputs like passwords

using Crypto object method like computeHash can encode passwords to a hash of bytes.

These Crypto objects have this secret hash output and also provide a key.

With this key you can replicate the hash given another Crypto object

The unique key is called a "Salt”, use this key to reverse-engineer. Just create another Crypto object and pass it the previous key. Now It's output will be the same as your original Crypto object

/// <summary>

        /// Convenience method to verify If a given password matches the encryted password in the DB

        /// </summary>

        /// <param name="passWord">User supplied Login password</param>

        /// <param name="passwordHash">Encripted password from the DB</param>

        /// <param name="passwordSalt"> Previous Key given from Security. Cryptography object that generates same hash, given same password...from same key</param>

        /// <returns>True = Encrypted passwords match</returns>

        /// ----

        private bool VerifyPasswordHash(string passWord,

byte[] passwordHash, byte[] passwordSalt)

        {

            using (var hmac = new System.Security.Cryptography.HMACSHA512(passwordSalt))

            {

                var computedHash = hmac.ComputeHash(System.Text.Encoding

.UTF8.GetBytes(passWord));

                for (int i = 0; i < computedHash.Length; i++)

                {

                    if (computedHash[i] != passwordHash[i])

                        return false;

                }

                return true;

            }

        }

## Passwords

Passwords in Db are stored as byte [] password hash + byte [] salts(key) ... not visible text

Even if two passwords are the same, they will have a different Hash values because, a new security crypto object (unique key) is created making a new key

That new key will output a completely different hash specific to that object. Each and every time a user registers

--New User with password flow

Controller --> Register method--> authRepo method (encrypts the password + track + save) --> DB

authRepo method (returns created object) --> Controller’s Register method (returns status code 201)

# Registering Services in the Startup class

By registering Services, you make them injectable throughout the App

public void ConfigureServices(IServiceCollection services)

{

services.AddSingleton<I\_Repo, Impl\_Repo>(); = Single instance created on 1st request

This instance is shared across all new requests. Problem when you have concurrent requests

services.AddTransient<I\_Repo, Impl\_Repo>(); = New instance created of repo for each http request

Good for stateless services

services.AddScoped<I\_Repo, Impl\_Repo>(); = Created only once per Req w/i the same scope

Uses this single instance when other calls made as part of this Request

The user's initial Request is grouped as a scope so, same instance is used throughout

Good for an Authentication request because they are tied to a single session

# Repositories – CRUD operations

Task<…> Means wrap this in an asynchronous Task that can return a value

Works with DataContext: DbContext to communicate with DB as an intermediary that our Controller does not directly deal with DB CRUD operations

References this DataContext and its DBSets (replicas of our model objects) to perform CRUD operations

\*\*Don’t forget the imports!!!

using System.Collections. Generic;

using System.Linq;

using System.Threading. Tasks;

using DatingApp.API.Models; --Wherever you Context is located

using Microsoft.EntityFrameworkCore;

## Examples

### Returns object of specific value

Specific User -> All photos of this User’s collection -> Specific photo that has a single Boolean value

Notice Task<…> perform this operation asynchronously and return Photo Object

public async Task<Photo> GetMainPhotoForUser (int userId)

        {

            return await \_context. Photos.Where (p => p. UserId == userId)

                . FirstOrDefaultAsync (p => p. IsMain);

        }

### Generic Tracking

Uses generics to take in any object to start DB tracking

        public void Add<T> (T entity) where T: class

        {

            \_context. Add(entity);

        }

Uses generics to start tracking this object prior to being deleted

public void Delete<T> (T entity) where T: class

        {

            \_context. Remove (entity);

        }

Calls EF Core change tracker. It can only effect changes that have been tracked

True = Tracked changes have been saved to the DB False = No changes have occurred

        public async Task<bool> SaveAll ()

        {

            //SaveChanges () returns an int of how many changes saved to the DB

            //

            return await \_context. SaveChangesAsync () > 0;

        }

# API Controllers

Task<…> Means wrap this in an asynchronous Task that can return a value

IActionResult = Http Response

async Task= A task that doesn't block other Req’s in thread while waiting for returned results

Task<User> = Just a User obj that is wrapped in a Task for Threading (uses async await pattern)

using Microsoft.AspNetCore.Mvc; =We are using MVC

[Route("api/[Controller]")]

    [ApiController] =We are enforcing attribute routing | remove =std

    public class UsersController: ControllerBase

    {

:ControllerBase = MVC (C) with NO Views : Controller = MVC (C) Allows Views

[ApiController] = Does a lot of work for us. The [] annotations that tell the server

That the param is an object vs. null is no longer required. Therefore, we do not need

the below Model State validation

        [HttpPost("register")]

        public async Task<IActionResult> Register([FromBody] ObjType varNm)

        {

If (! ModelState.IsValid()) return BadRequest (ModelState) -- no longer needed

## [HttpPut]

Api>controllers>UsersController.cs

//Put api/Users/{id}

        //

        [HttpPut("{id}")]

   public async Task<IActionResult> UpdateUser(int id, UserForUpdateDto userForUpdateDto)

        {

// Check if the current User is the one that passed the token to the server

// Trying to match passed id to what is in their token ... see authController line 79

            // User = check the passed token and get info from it

            //

            if (id != int.Parse(User.FindFirst(ClaimTypes.NameIdentifier).Value))

                return Unauthorized ();

// Call the repo method to return a single user from the repo <-> DB based on Id

            //

            var userFromRepo = await \_repo.GetUser (id);

// Map(theSource\_to Map\_from  ,  Destination)

// Dto only has some data but, we want to update this limited data to a full User object

            // The mapper takes Dto's 5 fields and updates them into the user object

            //

            \_mapper.Map(userForUpdateDto, userFromRepo);

//When GetUser(id) was called on userFromRepo.  EF Core registered a change to this object

//SaveAll saves all tracked changes to the DB.  The updated User object is now saved

            //

            if (await \_repo.SaveAll())

                return NoContent();     //produces an empty response

            else throw new Exception($"Updating user {id} failed to save");

        }

# Entity Framework Core

## Entity Framework commands

dotnet ef -h dotnet ef migrations -h

dotnet ef migrations add <migrationName> Adds updates to existing migration file <update name>

dotnet ef database update Update these changes to the DB

dotnet new -h listing of all available projects to be built

dotnet new webapi -n DatingApp.API ->Creates new project. NetCore as type web API called …

dotnet --info Display all your sdk's + info

## When Class Fields || Properties are changed in you App

\*Assume that new properties have been added or deleted

--> The DB entities are now different than the Models in our application classes

--> Required: Make updates in EF Core to account for these changes

Option 1: Normal flow

dotnet ef migrations add <someNameOfMigration> Adds Migration

dotnet ef database update Commits to the DB

Option 2: Oops I need to adjust something 1st

dotnet ef migrations add <someNameOfMigration> Adds Migration

ef migrations remove Removes recent most migration

--Make any fixes you need 1st, add another migration

dotnet ef database update Commits to the DB

"dotnet ef migrations list" shows all existing migrations

Option 3: Oops I already committed to the DB

Once you commit to the DB, you cannot undo the commit so, you will have to wipe the DB which is ok in development because you are just using dummy data

dotnet ef database drop gets rid of our DB and all tables + data

--> Migrations are still intact but, the actual Db is gone

dotnet ef migrations remove removes most recent migration (back to previous migration)

dotnet ef database update Recreates the schema (DB) up to the existing migrations

--> You are now back to the previous committed migration

## EF Core Foreign Keys

Navigation Property = Manually define the relationship by placing an instance as a member field

Manually map this relationship to the other class = object field + id field

public class User

{

//Property Navigation

public ICollection<Photo> Photos {get; set;}

}

public class Photo

{

//Property navigation resolution to solve for cascading deletes

        public User Users {get; set;}

        public int UserId {get; set;}

}

-If the relationship is not mapped (obj + id) = Restricted delete (onDelete: ReferentialAction.Restrict)

EF Core will still make the relationship but, will hold the FK as nullable. What this means is that, if we delete one. The other is orphaned. They float around and still exist regardless of which (User or connected photo) is deleted

-When relationship has a Navigation property + mapping

EF Core will create a cascading delete so, when a User is deleted, all photos will be deleted. onDelete: ReferentialAction.Restrict); vs. onDelete: ReferentialAction.Cascade);

If using SQLite you need to add these NuGet packages to your csproj file

Microsoft.EntityFrameworkCore

Microsoft.EntityFrameworkCore.Design

Microsoft.EntityFrameworkCore.Sqlite

## Repository Pattern

The Repo\_Interface is the Layer between Repo and Controllers that gets injected into the Controller.

Repo exposes its methods for the (C) to use so, the (C) does not directly talk with the DB

private readonly IMapper \_mapper;

        private readonly IDatingRepository \_repo;

        public UsersController(IDatingRepository repo, IMapper mapper)

        {

            \_mapper = mapper;

            \_repo = repo;

        }

Since the Repo interface is injected into the (C), the controller’s logic does not need to change. All changes about talking to the DB happen in the in the Repository.

This is called “Separation of concerns” so, (C) only deals with (Http Request-Response)

Repo deals solely with talking to DB. You register Repo as a service in the Startup file so, it can be injected anywhere in the App.

\_context = Middleware (Program <-> DB) The repo Implementation will communicate with the DB

### Tracking changes to the Db

Before entities are saved to the DB, all actions need to be tracked (all changes in the entities state)

Else it cannot be persisted(saved) … state needs to be added … in order to be saved

Some of the tracking is done automatically for us. For example, whenever we get an entity from the DbContext (Values, Users, Photos). By calling these, tracking happens automatically. Below we are calling the repo’s GetUser method that calls \_DbContext. User … as you can see the entity (User) of the context is being called which starts the auto tracking by entity Framework for us.

var user = await repo. GetUser (userId);

    user. LastActive = DateTime.Now;

    await repo. SaveAll ();

### Examples of tracking entity state -> Save to DB

await \_context. User. AddAsynch (userObject); --ADD command Tracks changes to the domain. model

await \_context. SaveChangesAsynch (); -- SAVECHANGES () command to persist any tracked changes

without the Asych part, an Exception would be thrown vs. Null

FirstOrDefaultAsynch (x => x. Id == id) returns null if not found

## Seeding data using JSON---- https://www.json-generator.com/

file.json goes into the data folder (Repo’s live here) app>data

make new class Seed.cs

Method -->Requires Json serialization text->object obj->DB

Seeding only works w/ empty DB "dotnet ef database drop"

public class Seed

    {

        public async static void SeedUsers(DataContext context)

        {

            //Check if DB is empty ... if so, lets seed the data

            //

            if (!context.Users.Any())

            {

                //Read the .json file in Data folder, hold as text

                //

                var userData = System.IO.File.ReadAllText("Data/UserSeedData.json");

                //Use Newtonsoft to convert Json <-> .Net

                //Read json data and convert into .NET User objects

                //Hold Users as a List ... pass text as a param

                //

                var users = JsonConvert.DeserializeObject<List<User>>(userData);

                foreach (var u in users)

                {

                    //Hord code the passwords for dummy data

                    byte[] passwordHash; byte[] passwordSalt;

                    CreatePasswordHash("1234", out passwordHash, out passwordSalt);

                    u.PasswordHash = passwordHash;

                    u.PasswordSalt = passwordSalt;

                    await context.Users.AddAsync(u);

                }

                await context.SaveChangesAsync();

            }

        }

In Program.cs call static Seed.SeedUsers(context) to save dummy data to the DB

ASP.NET core 2.2 vs. 3.0 ---Difference in the way JSON serialization happens and returns data

previously Newtonsoft.json was used to serialize obj responses to text -> defaults to System.Text.Json

3.0 requires that you register Microsoft.AspNetCore.Mvc.NewtonsoftJson w/ NuGet to register in csproj

<PackageReference Include="Microsoft.AspNetCore.Mvc.NewtonsoftJson" Version="3.1.7"/>

NewstonsotJson also needs to be registered as a service in Startup.cs

// Called at runtime. Shell to add + hold services for DI.

        public void ConfigureServices(IServiceCollection services)

        {

            services.AddControllers().AddNewtonsoftJson(

                opt =>

                {

                    opt.SerializerSettings.ReferenceLoopHandling =

                    Newtonsoft.Json.ReferenceLoopHandling.Ignore;

                }

            );

# DTO’s

## Flatten Nav Properties “To User ” example:

Used to map our classes (coming from DB as entities) into more simple objects

Keep Dto’s in a separate folder

When you make a Dto class, end its name with Dto nameDto.cs

public class Photo

    {

 public int Id {get; set;}

 public string Url {get; set;}

 public string Description {get; set;}

 public DateTime DateAdded {get; set;}

 public bool IsMain {get; set;}

//Property navigation resolution to solve for cascading deletes

        public User Users {get; set;}

        public int UserId {get; set;}

    }

public class PhotosForDetailedDto

{

public int Id {get; set;}

public string Url {get; set;}

public string Description {get; set;}

public DateTime DateAdded {get; set;}

public bool IsMain {get; set;}

//Only difference is that DTO does not include the Property navigation.

--This DTO is used to flatten out the data. Client should not see details of the User who the Photo belongs to

}

1. Dto’s are generally caught -or- sent by the (Controller) as an object –

2. They represent info we want “from User” -or- Specific data to “send to User”

3. Dto -> Entity needs to be converted … otherwise we would be unable to … save to the DB

## Server-Side Validation Dto

Data Annotations done on the model that is interacting with the Client. In this scenario, the Dto is being sent to the Client so, the validation needs to be on the Dto. Because, the Client is being sent this Dto not the underlying Model.cs class

public class UserForRegisterDto

    {

[Required]

[MinLength (4, ErrorMessage = "Must be at least 4 characters")]

        public string Username {get; set;}

[Required]

[StringLength (50, MinimumLength = 4, ErrorMessage = "Must be between 4-50 characters")]

        public string Password {get; set;}

    }

## Less/Different fields Dto “From User” example:

In this example, a client is uploading a Photo. The fields that they interact with are a little different than our Photos.cs model’s entity object. The goal is for the Client to only use the fields needed on their side. While the app grabs those fields and behind the scenes does everything else required. To save their interaction into the Photo.cs entity class which is what our app needs to save to the DB

public class Photo

    {

public int Id {get; set;}

public string Url {get; set;}

public string Description {get; set;}

public DateTime DateAdded {get; set;}

public bool IsMain {get; set;}

public string PublicId {get; set;}

//Property navigation resolution to solve for cascading deletes

public User Users {get; set;}

public int UserId {get; set;}

    }

public class PhotoForCreationDto

    {

public string Url {get; set;}

// The photo being uploaded via http req. as a file

public IFormFile File {get; set;}

public string Description {get; set;}

public DateTime DateAdded {get; set;}

public string PublicId {get; set;}

### Why we need Dto in this scenario

1. dto does not need Photo.Id – When Client sends up their Photo, it has not been saved to the DB so, there is no Photo.Id, yet. We cannot send a field to a client that does not yet exist

2. dto extra field/Prop “IFormFile File” – This is a photo/image that will not be saved on our DB so, we do not need to hold this file in our model/entity. It is true the Client is sending this “photo/img” with the request to the Controller but, Cloudinary (3rd party photo storage) will save this photo not our app. Therefore, we need to make this field available, to the Client (via Dto) but, it cannot exist on DB (resource hog) …Cannot exist as a field/Prop in Photo. cs class model/entity

## Adjusting DTO’s to match

When we change fields coming from our application. We need to change the Dto’s to match

Scenario: We have changed the fields in our register form from 2 -> 7 fields. The underlying User object has not changed but, the Dto facilitating the transfer of data between app <-> api now needs to match. We are now returning 7 fields

1. Make sure the fields in the Dto match the app. models interface class of angular app

a) Add missing fields. These would be extra fields that now exist on the register form that previously were not included on the Dto. So, we add these as Properties to the Dto

2. Make sure there is a mapper that accounts for the relationship so, the Dto can be saved to the DB

// <source, destination) --Take in a Source and return a Destination

            //

            CreateMap<UserForRegisterDto, User>();

Old: Previously we were using just the username and password to generate a new User by passing a slim User Object and a password which were sent to Register (). However, now that we have 7 vs. 2 fields of information. We need to pass more data

// POST api/auth/register

        // ----

        [HttpPost("register")]

        public async Task<IActionResult> Register (UserForRegisterDto userForRegisterDto)

        {

            //If userName has already been taken ... return BadRequest

            //

            if (await \_repo. UserExists (userForRegisterDto. Username))

                return BadRequest ("Username already exists");

            else

            {   //Create User with encrpted password & return status coode

                //

                var userToCreate = new User {UserName = userForRegisterDto. Username};

                User createdUser = await \_repo. Register (userToCreate, userForRegisterDto. Password);

                return StatusCode (201);

            }

        }

New: See CreatedAtRoute (). Basically, this allows you to include all the info after the object is created and send full info about the location of this new object as a 201 response.

# Using an AutoMapper---- map Entities/models <-> DTO's

1. Use Nuget to install AutoMapper. This gets installed in the csproj file

<PackageReference Include="AutoMapper.Extensions.Microsoft.DependencyInjection" Version="8.0.1"/>

1. Register AutoMapper into the Services container. This allows you to inject AutoMapper into the app

public void ConfigureServices(IServiceCollection services)

        {

--//target of AutoMapper= Repo...translation of entities to DTO (shaped data)

services.AddAutoMapper(typeof(DatingRepository).Assembly);

3. Inject the AutoMapper in the Controller ... The Controller deals w/ shaped data

OUT - Shaped data to client as Resp

IN - Client only interacts/sees what we want them to

NOT exact fields as they exist in DB

using AutoMapper;

Controller>UserController

[Route("api/[Controller]")]

    [ApiController]

    public class UsersController: ControllerBase

    {

        private readonly IMapper \_mapper;

        private readonly IDatingRepository \_repo;

        public UsersController (IDatingRepository repo, IMapper mapper)

        {

            \_mapper = mapper;

            \_repo = repo;

        }

## 4. Returning Mapped values from the DTO … from Controller

Controller>UserController

//Get api/Users/{id}

 //

[HttpGet("{id}")]

        public async Task<IActionResult> GetUser (int id)

        {

            var user = await \_repo.GetUser (id);

--Transform User object --> DTO < Destination\_Out\_theDTO > (input\_source\_IN)

--Pass the DTO as a return input for the response code 200

            var userToReturn = \_mapper.Map<UserForDetailedDto>(user);

            return Ok(userToReturn);

        }

## 5. Tell AutoMapper about the mappings we need it to support

Separate folder called "Helpers" this folder is stuff that assists other classes I.E. Extensions for global error handling -or- Mapper for creating a relationship between the entity/Model <-> DTO

A helper class is needed to extend Profiles. this class's purpose is to understand the source (IN) and the destination (OUT) of what is being mapped. That’s why we are extending "Profiles"

namespace DatingApp.API.Helpers

{

    /// <summary>

    /// This AutoMapper defines what can be mapped as a source & output

    /// The entity class is the source

    /// The DTO is the destination/output

    /// </summary>

    public class AutoMapperProfiles: Profile

    {

// Destination mappings are done in the constructor

        //

        public AutoMapperProfiles ()

        {

-- We now have two DTO mapping that are supporting the User entity / class

--Take in a Source and return a Destination

---Source, Destination

CreateMap<User, UserForListDto> ();

CreateMap<User, UserForDetailedDto> ();

## DTO fixes -output to Client

**How to fix common Navigational Property issues (part 1):**

**Passing DTO’s with DTO member fields**

At current the User...DTO is returning the full user Object inside of the Photo.cs since Photo has

Navigational Property of (User + UserId ...to establish FK) which return a lot of data we don't want

In our Controller method (step 4) we made adjustments so that, the DTO would be returned vs. the entity object.

We need to make sure that the DTO is only pulling data from the PhotoDTO vs. the entity object. Since the PhotoDTO does not have mapping properties (obj + id). The data returned will be flattened. Otherwise full User details will be displayed showing passwords, salts etc.…

Notice that one DTO has an instance of the other DTO, not the Photos.cs object. Now only these DTO properties will be passed, not the full entity object details

The Controller is now only passing the data we want to show from our DTO’s

public class UserForDetailedDto

    {

public string PhotoUrl {get; set;}

 public ICollection<PhotosForDetailedDto> Photos {get; set;}

}

public class PhotosForDetailedDto

    {

public int Id {get; set;}

public string Url {get; set;}

public string Description {get; set;}

public DateTime DateAdded {get; set;}

public bool IsMain {get; set;}

    }

**How to fix common Navigational Property issues (part 2):**

How to configure AutoMapping relationships using .ForMember() Helpers>AutoMapperProfile

Remember that our Mapping relationship is between User object and UserDto

problem1: At current the UserDto that is being sent as the response from the Controller. Is not showing the "PhotoUrl" property = null

public class User

    {

        //Property Navigation

        public ICollection<Photo> Photos {get; set;}

--User.cs -> Photo.cs -> a Photo property

public class UserForDetailedDto

    {

public string PhotoUrl {get; set;}

 public ICollection<PhotosForDetailedDto> Photos {get; set;}

Reason: The User entity object has an ICollection<Photo> so, it has access to Photo class. However, User obj does not have a field that exists in DTO "PhotoUrl" (only exists in DTO). The User obj cannot pass info it does not have.

solve: Configure the mapping, between User entity and Dto to assign the “PhotoUrl” property from Photos entity -> PhotosDto.

I.E. find a way to get the "PhotoUrl" prop from Photo.cs instance that's inside of User object.

var Url = \_context.Users.Include(u=> u.Photos).FirstOrDefault(p => p.IsMain).url

Mapper is between two classes so, does not make sense to make call from DB like above

The solution needs to be between the DTO grabbing data from the Entity. This code needs to happen inside of the AutoMapper.

### Using. ForMember ()

to solve for specific Property fields which Transforms a single member = Pass single property from the entity -> dto

ForMember (destMbr, mbrOptions) dest=dto mbrOption= use opts to get User obj details

--add a configuration to the existing mapping

CreateMap<User, UserForListDto>()

                .ForMember(dest => dest.PhotoUrl,     //destination member = dto.prop

                    opt => opt.MapFrom(                     //opt to map src =User obj

                        src => src.Photos.FirstOrDefault(    //All Photos for this User

                            p => p.IsMain == true            //Return 1st where IsMain

                        ).Url                        //Of all those a single will return

                    )                             //For that one, give me url prop field

                )

problem2: DateTime DOB property in User obj is null when it is passed to Dto as int Age.

public DateTime DateOfBirth {get; set;}

public int Age {get; set;}

reason: can’t pass User's DateTime DOB property -> Dto's int Age property. They are different types so, the AutoMapper cannot convert w/o more instructions

solution: We want this property to take CurrDate (Yr) - DOB(Yr) = Age

Convert User's DateTime DOB through a custom "extension method" that can take a DateTime as an argument. Do the above calculation and return an int. When an int is returned, the AutoMapper can now pass this int value to the DTO int Age value. Basically, we need to map another individual property.

-->append existing CreateMap<src, dest>() .ForMbr(...) w/ another .ForMbr(...) to get this field and pass the updated value. Since this mapping is already solving user->userDto we make additions to this existing CreateMap relationship

Making custom extension method ... add new method to existing extension class Helpers>Extensions

new method "CalculateAge" will extend DateTime ... extension methods should be static. The parameter will take a (this DateTime dt) which means that this method can accept the DateTime as a chain vs. direct param

public static class Extensions

    {

public static int CalculateAge(this DateTime theDateTime) -- extends this DateTime

        {

       var age = DateTime.Today.Year - theDateTime.Year; --return int Yr but,has dt info

       if (DateTime.Now.DayOfYear < theDateTime.DayOfYear) --if dob>cur doy not bday yet

       {

           age--;            //cannot combine calc + return, must be on sep lines

           return age; //Else will not calc the age= age-1

       }

       else return age;

        }

## AutoMapper Direction

Api>Helpers>AutoMapperProfiles.cs

        public AutoMapperProfiles ()

        {

Ex.1 – Take in a User -> output a Dto

            // <source, destination) --Take in a Source and return a Destination

            //

            CreateMap<User, UserForListDto> ()

--How to use in a Controller method (pushes eligible model fields into Dto)

--Each called model is being converted to a Dto and stored as IEnumerable<Dto>

--Send objects with the response code

var users = await \_repo.GetUsers();

var usersToReturn = \_mapper.Map<IEnumerable<UserForListDto>>(users);

return Ok(usersToReturn);

Ex.2 – Take in Dto -> output a User

// <source, destination) --Take in a Source and return a Destination

            //

            CreateMap<UserForUpdateDto, User> ();

--How to use in a Controller method (pushes Dto data to the model/entity for save)

  var userFromRepo = await \_repo.GetUser(id);

\_mapper.Map(userForUpdateDto, userFromRepo);

### DTO’s exist in the API

Whether we are pushing Dto to client (shaped data object from DB) -or-

Whether we are receiving only a few fields from the Client (must make Dto to catch)

The Dto is made in the Api -and- the models exist in Agular as Interfaces

# Token Authentication

JWT = Json Web Tokens (industry standard) https://jwt.io/

Token are used to authenticate the client against the API server.

Users cannot make up their own token and send to the server. Even if they have the correct username b/c the server has its own signature. Each token must be exact to verify against the server

The server uses a crypto object with a key to ensure that the token is valid...The token cannot be manipulated as it will not pass check. The token doesn't go to DB to validate, it calls the token itself

JwTokens are passed back from client to the server to authenticate the user.

Since JWT are signed with a secure key, no DB call to server is needed to check if authenticated

Because, they are sending request with the Token we have already provided, this ensures authentication for a given period of time

Tokens have self-contained properties (Credentials, Claims to the application + more)

JWT is a text file in this format "Token" : "Token string" (k, v)

1.Header ("algoUsed”: encryted algorithm “type”: "JWT")

What kind of security is used?

2.Payload (info stored inside the token) \*\*careful - easily decoded

Body that has some info about response

3.Secret (Used to Hash 1+2) stored on the server and never revealed to the client

This is a signature. Basically, an encryted string that comes from server. Only this exact string can be sent back to verify each token.

The request to the server must come with this exact signature + user log-in credentials

Secret being passed needs to match ... else not a valid token … not authorized

## Identity Claims in the Controller method

Claims are how we build and approve the identity of a User. Claims we are making about their identity

-->Login credentials are received in a method of the Controller

--> The username and password are verified against the DB

-->Since UserName + Password are correct we start building the Claims

var claims = new []

            {

                new Claim (ClaimTypes.NameIdentifier, userFromRepo.Id.ToString() ),

                new Claim (ClaimTypes.Name, userFromRepo.UserName)

            };

Authentication Middleware you will need - Once you have authorization set up in your Controller

1. using Microsoft.IdentityModel.Tokens this Controller
2. using System.IdentityModel.Tokens.Jwt this Controller
3. using Microsoft.AspNetCore.Authentication.JwtBearer Startup

## Adding Authentication as a service

The application needs to know which Type of Authorization you are using. So, the authentication needs to be registered as a service in “Startup.cs” to be used throughout the application

        // Called at runtime. Shell to add + hold services for DI.

        public void ConfigureServices(IServiceCollection services)

        {

services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)

                .AddJwtBearer(options =>

                {

                    options.TokenValidationParameters = new TokenValidationParameters

                    {

                        //--options to validate against our JWT auth

                        ValidateIssuerSigningKey = true,

                        IssuerSigningKey = new SymmetricSecurityKey(Encoding.ASCII

                            .GetBytes(Configuration.GetSection("AppSettings:Token").Value)),

                        ValidateIssuer = false,

                        ValidateAudience = false

                    };

                });

The Http pipeline also needs to be set up in the “Startup.cs”

// This is middleware to interact w/ Req on its journey to deliver a Resp

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

        {

. . .

app.UseRouting();

            app.UseAuthentication(); --Can this Request(User) be authorized

            app.UseAuthorization(); --If so, what can they do

-- This must happen before app.UseEndpoint / UseMvc(). Because, we want to short circuit the http pipeline and protect the Controller and its methods from unauthorized users

app.UseCors(req => req.AllowAnyOrigin().AllowAnyMethod().AllowAnyHeader());

app.UseEndpoints(endpoints => // Endpoint API's rout incomming Req to Controllers

            {

                endpoints.MapControllers();  // Controllers map to the endpoints

            });

## Testing Authentication in Postman

Security Annotations that can be put on a Controller class or, individual method

[Authorize]  = class attribute  = Each request requires authorization Token

[AllowAnonymous]  = method attribute =  No authorization token required

Postman … Paste in the request url

-> Log in by sending “post” with username + password in the body {"username": "Lola", "password": "1234"}

-> Grab the token from the response body "token":” …”

-> Make a new “get” request

-> Go to Headers

-> Add a new Key value “Authorization”

-> Input the value “Bearer stringValueOftheToken”

## (1) Hiding the token key from being pushed to GitHub

The normal way would be to stage the changes and publish them to GitHub ...secrets are pushed

= Source control -> stage (+) -> "message" -> commit (enter)

How to hide a file using app>. gitIgnore file

--> add the file -or- file type you want to omit from being pushed to Git

--> git rm appsettings.json –cached Removes from staging and keeps any changes as local. Does not go with push request

.vscode --This is the .gitIgnore file

bin

obj

\*.db

//appsettings.json --This file is holding the secret string that our app is encrypting …

{ --This is the appsettings.json file

  "AppSettings": {

    "Token": "…secret value here…” <-- This should not be on a public server like GitHub

  },

  "ConnectionStrings": {

    "defaultConnection": "Data Source =datingapp.db"

  },  "Logging": {

## (2) Hiding the token key from being pushed to GitHub

Use this option for DEV mode only. In production you must use ENV Variables in production

1. dotnet user-secrets init Adds secrets GUID to <PropertyGroup> in csproj

<PropertyGroup>

    <TargetFramework>netcoreapp3.1</TargetFramework>

    <UserSecretsId>d316194c-5298-416c-84f9-1611dc803cc2</UserSecretsId>

  </PropertyGroup>

("K" "V") - from appsettings.json set as a secret locally

1. dotnet user-secrets set "AppSettings:Token" "secret string value"

dotnet user-secrets list Retrieves token value from local

1. Delete the string value from “AppSettings": {“Token": "..."} from appsettings.json because, it is now being saved as a secret locally on your computer
2. Delete the user-secret dotnet user-secrets clear

------------------

# Errors

## Safe Navigation operator? Elvis

error: component is loaded before the data is received (page loads but error in console log)

Problem: Error is caused b/c component is saying, I don't have this component so, I will put an error in the browser's console log -

solution: Use optional operator/Elvis to avoid seeing errors in the console in the template where data is being displayed (mbr-details.html). Make the property optional {{user? .knownAs}}

# Action Filters

## What is an ActionFilter?

IAsyncActionFilter is an Interface that is implemented on your own custom class, used as a cross-cutting pattern. Basically, instead of executing code inside a specific method, you would use the filter itself on the outside of [a single method, an entire class (applies to all class methods) or the entire application (all application methods)].

public class LogUserActivity: IAsyncActionFilter

    {

Below Controller “UsersController” cannot be accessed without calling the Filter “LogUserActivity”. Every time this Controller class is used to call one of its methods … our filter is called and its single method which is not showing above will be executed.

This is separation of concerns. The Controller still does its one job (deal with req and resp) and our filter, a completely separate class has code that can be updated or changed without effecting the function or code of the Controller.

[ServiceFilter (typeof (LogUserActivity))] //Any access to this controller

//requires ActionFilter

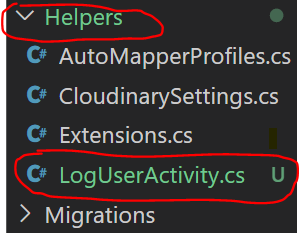
    [Route("api/[Controller]")]

    [ApiController]

    public class UsersController: ControllerBase

    {

## How does IAsyncActionFilter work?



Filters live outside of the main folder structure (Data, Models, Controllers) because, they can be applied across all classes and methods. Action Filters isolate a method execution like <IActionResult> and surround this execution. The <IActionResult > includes our executed code before the response is sent

All though it looks like we need to pass two params to make this method work. We are not actually the one doing the passing. When we put [ServiceFilter] on the Controller class, this means that the Controller’s method (dealing with req & response) is being targeted and its Action (req + resp) is the parameter being passed by the application. In other words, when the Controller method executes, it is auto-passing its method to our filter and auto-passing itself as the context param.

public class LogUserActivity: IAsyncActionFilter

    {

        public async Task OnActionExecutionAsync (ActionExecutingContext context,

ActionExecutionDelegate next)

        {

context = Target this method and surround it

next = After the method is done, do our code next automatically -> then pass resp.

Before the response is sent, the code in our filter executes and is send with the <IActionResult> as part of the completed response.

## How to use IAsyncActionFilter

Remember that we are surrounding a method and doing something before the response is being sent so, we have access to the request being sent and all updates that happen in the executing code.

In the example below, we just want to update a single field when one of these methods gets executed. Problem: The methods themselves (UsersController) are using the repository pattern to interact with the DB to give or update user information which, we do not want to interrupt. So, we need to get what we need from the request itself or, data tied to this request.

Solution: Get what you need from the Request itself after the method has bound the model. In other words. Leverage the fact that, the request has already interacted with the Db (targeted method execution) and has the info you can use. So, the request being sent already has params about a single or, multiple users. The request has already cleared authorization so, it has the token information (Claims details about a specific authorized user). Your tools = the method execution, the request and Claims Principle object. Properties.

## IAsyncActionFilter code Example

next() = Method representing executed target method HttpContext= request getter

public async Task OnActionExecutionAsync (ActionExecutingContext context,

ActionExecutionDelegate next)

        {

--You need a way hold the executing method and say what should happen after it is executed. This var means before the response is sent, I want to hold the method result and modify it a bit before a response is sent.

var resultContext = await next ();

// I need the Id of the User that sent the request

//

            int userId = int. Parse (resultContext. HttpContext. User

            . FindFirst (ClaimTypes.NameIdentifier). Value);

// I need a repo to return full User by ID so, we can change a prop value

resultContext. HttpContext = get the request tied to this method

// requestService = Get or set a service instance from StartUp to use a service

// GetService<> = get a particular service and return the instance so we can use it

RequestServices. GetService<…> = DI an IRepo that we can use

            var repo = resultContext. HttpContext. RequestServices. GetService

<IDatingRepository> ();

// Re-using the existing repo that was called by the request

var user = await repo. GetUser (userId);

// Now that we have the correct User object, we can set its property

            user. LastActive = DateTime.Now;

//Use the existing repo to save our changes to the Db

            await repo. SaveAll ();

        }

## IAsyncActionFilter set-up

1. Create a class that implements IAsyncActionFilter

public class LogUserActivity: IAsyncActionFilter

    {

2. Implement the methods of the Interface

public async Task OnActionExecutionAsync (ActionExecutingContext context,

ActionExecutionDelegate next)

        {

3. So that your program can make use of this Filter. You need to make it available as a Service that classes may access as an [Attribute] similar to Dependency Injection

// Add scoped = new instance per request

        public void ConfigureServices (IServiceCollection services)

        {

services. AddScoped <LogUserActivity>();

4. Add the Filter Attribute where you want to filter (on a class example)

    [ServiceFilter (typeof (LogUserActivity))] //any method of this class will

    [Authorize] auto-execute the filter’s method

    [Route("api/[Controller]")]

    [ApiController]

    public class UsersController: ControllerBase

    {

# Paging

## What is Paging + Why we need it

Data -> Api (paging) -> Spa (filtered data)

Paging has to do with breaking up received data into smaller manageable pieces. Whereby, the returned data is broken up into Pages. I.E. If 20 objects are output, you may NOT want to display all 20 on a single page to the Client. Paging gives flexibility to only return 4 objects at a time. Which can be broken up across 5 pages 4x5=20. This way, the data has flexibility to be requested in smaller groups (pageSize) as to not overload the network on Client requests. Also, Paging allows you to hold these object groups (pageSize) per request together on a Page. So, to the Client they are just clicking to the next page to see the data. But, actually, a new request is being made to pull a piece of the larger data and display this to them.

## Client Paging Request

Paging is a GET request for data

Get requests do not have bodies so, the information required to know what page and page size. Needs to be passed as a request/route parameter [FromQuery]. Therefore, this info needs to be applied in the Url string, we are getting additional data we need “from the query” string.

Host +Controller ? query string parameters

<http://localhost:5000/api/users?pageNumber=2&pageSize=3>

## Paging Process

### Client sends request

1. Request has query parameter information in the Url request string (host + controller? query string params)

http://localhost:5000/api/users?pageNumber=2&pagesize=3

[HttpGet]

        public async Task<IActionResult> GetUsers([FromQuery]UserParams userParams)

        {

The passed params must be a completely separate object because they are not a Db structure. Their only purpose is to filter objects that, we want to retrieve from the Db. So, on the Api application side we have a way to catch and store the param string values being passed with the GET request [“From” the “Query” string being passed]

### Controller receives the request with query parameter information [FromQuery] UserParams userParams

1. UserParams = class that is specifically made to match the data we want passed from the User. Therefore, this class must have the properties PageNumber and PageSize to match the ? params in the request string

A. PageNumber property must have a default in case no information is passed

public class UserParams

    {

public int PageNumber {get; set;} = 1;

B. PageSize must have a default and logic so that, we can adjust the pageSize

// Max users per page

        //

        private int pageSize = 10;

        public int PageSize

        {

            get {return pageSize;} //50 //50

            set {pageSize = (value > MaxPageSize) ? MaxPageSize : value; }

        }

### Controller calls Repo for Data

When a Controller method use a Repository Pattern. It delegates Db CRUD operations to another class whose responsibility is to deal with Db transactions. The repository pattern uses an Interface and an implementation so, both should include the userParams.

Req (w/ params) -> Controller (pass params) -> Repo (Use params)

//Controller

//

var users = await \_repo. GetUsers (userParams);

// Repo Method

//

public async Task<PagedList<User>> GetUsers (UserParams userParams)

        {

The Repo now has the parameters originally passed by the Client. The reason we need these is, to control how many + specifically which objects should be returned from the Api.

### Repo Helper Class

Ultimately, we want to return a List of specified objects with 2 additional parameters. These are passed to perform logic and calculate which (from the total of all Users) User objects should be returned as a List.

Problem: The User/Photo etc. objects do not have fields to accept these params. Also, these params have nothing to do with those objects so, it makes no sense to change our existing objects or, their Db structure. Hence, we cannot add these params (pageSize and pageNumber) to every object.

Since no Class exists that can combine:

1) Accepting a collection of User objects

2) Filter these objects for a defined count and

3) Already has the two params (pageSize and pageNumber). A new class needs to be created as a return type for the repo method.

Solution: The new class PagedList<T> (the Repo Helper class) will serve as a return type for our Repo method. The Repo will grab User objects from the Db as normal. The repo is still only responsible for CRUD operations and returning a finished product to the Controller.

However, the Repo method will now use this PagedList<T> (Helper class) as its return type. Because the Helper class can now be responsible for filtering the results from the Repo before sending to the Controller. Plus, since this is a completely separate class, we can add properties (the params being passed). In this way the Controller still has the same responsibilities (Req + Response). The Repo still has the same responsibilities (communicate w/ Db) but, has separated the concern of filtering + adding new fields (that do not exist in the DB) to this Helper class. That is why we need a completely separate class to help the Repo.

### Repo Helper Class Method

The “Helper Class” is an object that will be returned by the Repo. Ideally, this object will be a filtered List<User> that are able to take the userParams. However, this class needs to receive the Data from the repo 1st in order to filter. It is easier to receive Repo object data as a method parameter. So, it makes sense to have a “Method” be responsible for taking the params passed by the Repo to create this “Helper class” instance. Once the instance is created, it can be returned from the repo to the Controller.

### What we are solving for:

TotalCount = How many total records are being passed from the Repo. This uses. Count () to figure out how many total pages will there be if, we change the page size. I.E count =30, if pageSize=5 --> 6 totalPage’s with 5 per page.

PageSize = How many records will be displayed per page? This is what the Client will pass as a query param in the Url string which will initialize this property. However, default values are needed in case no value is passed. A limit is also needed in case, the Client passes an unreasonable number. We need logic to keep this property value within reason.

CurrentPage = Of the pages available to the Client, which of these pages are they requesting data for? This is what the Client will pass as a query param in the Url string which will initialize this property. However, default values are needed in case no value is passed

Which items to return? When we receive data from the Repo, the repo becomes the source. It represents all records in the Db. However, we only want the records associated with a particular page. This is calculated by. Skip () all the pages prior to this one and. Take () the “PageSize” of this page --> ToListAsync () which executes the query. We now have all the “items” (specific objects filtered) … what to do with them?

How to pass a List and initialize object?

PagedList<T>: List<T> This means that this class is inheriting from List<T>. This class is a List with our own custom properties. What this also means is that, we can pass collections directly to this object since it is already a List. So, a single object would be this. Add(T), this. IndexOf(T), this. Remove(T) … The class is itself a List so, you can use methods of List<T> just put. this before the method to use. In our case, we have a multiple object’s that we want to add --> this. AddRange (items we got from our method)

Static method to initialize?

// Repo Method

        public async Task<PagedList<User>> GetUsers (UserParams userParams)

        {

            var users = \_context. Users. Include (u => u. Photos);

            return await PagedList<User>

. CreateAsync (users, userParams. PageNumber, userParams. PageSize);

        }

The “Helper Class Method” is static so, an instance is not needed. We can directly use a Class reference. In other words, we can just return a PagedList<T> and throw this method on the end. This method can be used to:

1) Receive the params from repo (source + user params)

2) Initialize the its class object

3) Pass the List as part of the Class initialization

#### Example Repo Class + Method

/// <summary>

    /// This class inherits List so; it is basically a List<T> that has 4 properties

    /// Use these properties to determine the total size of the List object

    /// </summary>

    /// <typeparam name="T"></typeparam>

    public class PagedList<T>: List<T>

    {

        public int CurrentPage {get; set;}

        public int TotalPages {get; set;}

        public int PageSize {get; set;}

        public int TotalCount {get; set;}

        public PagedList (List<T> items, int count, int pageNumber, int pageSize)

        {

            TotalCount = count;

            PageSize = pageSize;

            CurrentPage = pageNumber;

            TotalPages = (int)Math.Ceiling(count / (double)pageSize);

            this. AddRange (items);

        }

        /// <summary>

        ///

        /// </summary>

        /// <param name="source">Object data passed by the Repo</param>

        /// <param name="pageNumber">Passed from Controller-> Repo-> here</param>

        /// <param name="pageSize">Passed from Controller-> Repo-> here</param>

        /// <returns>A fully created PageList<T> object</returns>

        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);

        }

    }

## Extension Methods

### What is an extension method?

An extension method is a static method that you chain onto a class type instance. The purpose is to give that instance additional functionality. Below is how to create these methods:

1. Create a folder api>Helpers to hold a class that will keep all of your extension methods
   1. You need a separate folder to keep organized since, all classes in this folder are outside of the normal application flow. The folder is to hold your custom classes that will serve as helpers, to the existing framework (Data, Models, Controllers, DTOs, Migrations).
   2. Classes in this folder examples:
      1. AutoMapperProfiles: Profile – This class is used for mapping between your Dto’s and models. It is a helper class that allows you to created mappings and define what should be mapped as a source vs. an output. This allows you to transform data from Dto -> Model -or- Model -> Dto.
      2. LogUserActivity: IAsyncActionFilter – This class is used for filtering when a request is received. Filtering allows you to grab the request and surround any or all methods so that, before or after execution you can perform additional logic.
      3. Custom classes that do not exist in the DB. When dealing with Request or Responses from a Controller. The data may need an intermediary structure that does not exist in the Db (not a mode or Dto). In this “Helpers folder” is where you would keep such classes. Examples:
         1. UserParams class – A custom class for paging that only takes page# and pageSize. It does not make sense to change your Db structure for this small paging item so, this class is made just to hold in coming params [FromQuery] on in incoming request.
         2. PagedList Class – A custom class for paging that combines the UserParams + Users output from a DB query. It does not make sense to affect the Db structure for all Users for information that will only be used to output header info. So, this class is a custom output for the controller that can be used to gather needed info to pass as an input for a header response.
         3. Extension class for extension methods – A custom class that hold methods that can be chained to object type instances
2. Create a class to hold your static extension methods – Extensions. cs
3. Create a static method that accepts a parameter of “this” + the class type instance you want to extend

public static void AddPagination (this HttpResponse response, int currentPage,

                        int itemsPerPage, int totalItems, int totalPages)

        {

### Example for paging extension method

In this example, an extension method is being chained to the HttpResponse of the Controller. The purpose is to add stuff to the response header. The Controller has a separate object type “PagedList” that has all of this info but, it also has a list of User objects that we do not want. So, a separate class “PaginationHeader” must be created to only hold these 4 values only.

When a header is sent, it is a K-V pair. Instead of chaining each value as a String to complete the value. We have opted to make the “PaginationHeader” object to hold each value as a field/property. This way the key can be the name “Pagination” and the value can be passed as a single object.

{

            var paginationHeader = new PaginationHeader (currentPage, itemsPerPage, totalItems, totalPages);

            var camelCaseFormatter = new JsonSerializerSettings ();

  camelCaseFormatter.ContractResolver = new CamelCasePropertyNamesContractResolver ();

            response. Headers. Add ("Pagination", JsonConvert.SerializeObject(paginationHeader, camelCaseFormatter));

            // Expose the header is needed else cors error

            //

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

}

## Serialization

In the above example we have used an extension method to add (“Pagination”, paginationHeader) which is being added to the HttpResponse response. Headers. Add (Key, value). However, there are two problems.

Problem 1: Http is sent as text and we are passing “paginationHeader” which is an object. So, in order for our header data to be passed we need to convert the object to text. AKA serialize the data into text

Solution 1: Invoke JsonConvert. SerializeObject (object to be serialized). This will turn object into JSON format which can be passed as text

Problem 2: Angular is case specific so, the information needs to come across as all lowercase or, camelCase

Solution 2: JsonConvert. SerializeObject () is an overloaded method so, there is an option to format the passed object as camelCase (object, JsonSerializerSettings settings ) but, it requires that we configure the settings.

1. Create a new JsonSerializerSettings ()

2. Chain the ContractResolver to the above object and assign as camelCase

var paginationHeader = new PaginationHeader (currentPage, itemsPerPage, totalItems, totalPages);

     var camelCaseFormatter = new JsonSerializerSettings ();

     camelCaseFormatter.ContractResolver = new CamelCasePropertyNamesContractResolver ();

            response. Headers. Add ("Pagination", JsonConvert.SerializeObject(paginationHeader, camelCaseFormatter));

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

The header is now ready to be sent as text to the client over Http as part of the Response.

# CSS

## Radio Button

In this example we are working with an un-styled radio button. Notice:

1. All contained in a <div> so all are running horizontal

2. <label> and <input> run horizontal

3. <input> are inside of the <label> only that we can assign bootstrap classes to the input

Think of these are 3 separate elements to style label 1,2, 3 

<div>

  <label class="">I am a: </label>

--value = pass this value --formControlName= match FormControl field name

  <label class="">

    <input type="radio" value="male" formControlName="gender" />Male

  </label>

  <label class="">

    <input class="" type="radio" value="female" formControlName="gender" />Female

  </label>

</div>

Put some space between the 1st of the 3 in-line elements 

<label class="mr-5">I am a: </label>

-or-

<label class="" style="margin-right: 30px;">I am a: </label>

Put some space between the radio circle and its text 

\*Notice this is done within the <input> that has the text

<input class="mr-3" type="radio" value="male" formControlName="gender" />Male

Put some space between <label> holding radio1 and radio2 

<label class="mr-3">

    <input class="mr-3" type="radio" value="male" formControlName="gender" />Male

  </label>

--Just add some margin-right to the 3rd <input>label to finish making it look good