-----\*\*\*\*\* PROJET REALISE AVEC VISUAL STUDIO CODE \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*------------------

------------Glossaire-------

pipe : appel de fonctions en chaine – chaque fonction est appelée avec le résultat de la précédente

**activatedeRoute**🡪 ***dynamique***, renvoie un observable de la route

il est détruit quand la route est détruite

**activatedRouteSnapshot**🡪***statique*** chaque appel crée une instance de route lancée lors du ***OnInit*** donc si changement la route statique reste la même

* Utilisé pour logger
* Pour accéder au paramètres de la route
* Provider🡪 dans app.module le section providers :[] représente les services injectés lors de la compilation

------------------------------

launchsetting.json --> http seulement

-Lancer **dotnetcore: dotnet watch run** --- watch en option

-creer .net visualstudio code :

en ligne de commande : **dotnet new 'nomprojet'**

package pour entity DbContext ----> voir DattingApp.api.csproj

<PackageReference Include="Microsoft.EntityFrameworkCore" Version="3.1.3"/>

<PackageReference Include="Microsoft.EntityFrameworkCore.Sqlite" Version="3.1.3"/>

---> créer

-les classes pour les tables

-le dataContext pour communiquer entre projet et base via entity

-option dans dataContext la **connectionString**

Dans **appsettings.json** "ConnectionStrings":{

"DefaultConnection":"Data Source=datingapp.db"

Dans Startup ---> injection public void **ConfigureServices**(IServiceCollection services)

{

services.AddDbContext<DataContext>(X=>X.UseSqlite(Configuration.GetConnectionString("DefaultConnection")));

services.AddControllers();

}

**copie du projet Angular pour transmettre --> ne pas inclure node\_module ====> les fichiers sont rechargés avec le package.json**

entity code first avec **migration**

I) installer Microsoft.EntityFrameworkCore.Design nuget---

en ligne de commande : **dotntet tool install --global** si non installé

si installé ---->

**dotnet ef -h** ---> pour trouver l'option à entrer pour creer la bdd depuis le code

puis dans le rep du projet :--

Création de la base :

-**dotnet ef migrations** add InitialCreate

**Dans une api les controleurs dérivent de la classe ControllerBase -🡪 pas de vue associée au projet**

**Dans une appli mvc les controleurs dérivent de la classe Controller 🡪 vues**

**Angular**

Angular 2 = Angular

Valor software🡪 ++ pour angular

POUR LES RECHERCHES : TAPER GENRE🡪 ng2-composant

ou npm install angular 8 composant voulu

POUR AFFICHER LE CONTENU D’UNE CLASSE

🡪 dans visual studio code 🡪btn-dr

🡪peek

🡪peek definition

**Installation de Angular client:**

dans le repertoire du projet

**npm install -g @angular/cli**

dans le repertoire parent du projet

créer le projet angular :

**ng new DattingApp-SPA**

Outils à installer dans visual :

- Angular snippets --> ajoute des extraits de code

- Angular files ---> ajout de fonctionnalités éditeur

- Angular language ---->

* - Rename tag ---> permet de modifier les balises html plus facilement
* - Braket pair colorised2 ---> config des couleurs de visual
* - Debugger for chrome --
* - Material icon theme --> change les icones de fichier dans VSC
* - Pretier code formatter --> améliore l'apparence du code
* - TS lint ??
* - Angular 2 switched --> raccourci clavier pour bascule html css

**Exemple page simple développée sur visual studio code**

Affichage : Données de la table value (id,name) avec **DattingApp-SPA** Angular

Données : Table value créée dans l’api **DattingApp.api** utilisant Entity code first

Ajout d’une page

Angular-->nlle page : bn dr--> dans app--> generate componenent (nom minuscule)

* création d'un rep avec .css .html .ts

Pour fonctionnement en client

**Dans app.modules.ts**

* En tête 🡪 référencer le chemin

import { HttpClientModule } from '@angular/common/Http';

* Ajouter HttpClientModule

Imports [

HttpClientModule

]

**ANGULAR**

Comme c’est un projet page unique lancement auto au démarrage, **dans value.component.ts**, ajouter une requête dans le constructeur :

1. Importer HttpClient et l’ ajouter comme instance du constructeur
2. import {HttpClient} from '@angular/common/http';

export class ValueComponent implements OnInit {

  values : any;

  constructor(private http:HttpClient) { }

  ngOnInit() {

    this.getValues();

  }

  getValues(){

    this.http.get('http://localhost:5000/api/values').subscribe(response=>{

      this.values = response;

    },error => {console.log(error);});

  }

}

**.NET CORE**

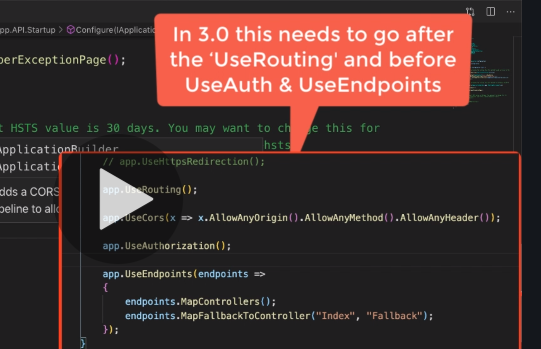
Il faut enfin ajouter un service pour pouvoir communiquer du port 5000 à 4200 malgré les règles de sécurité dans l’app serveur Api: **startup.cs**

Dans **ConfigureServices** :

  services.AddCors();//ajout service pour le passage d'un port à un autre

            // ce service est configuré dans void Configure

2)dans Configure :



1. Afficher dans valueComponent.html

<p \*ngFor="let value of values">

  {{value.id}},{{value.name}}

</p>

1. Mise en page:

**Install bootstrap font-awesome** en ligne de commande

Importer bootstrap dans le fichier CSS

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

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

**copie du projet Angular pour transmettre --> ne pas inclure node\_module ====> les fichiers sont rechargés avec le package.json**

Cacher les ifno sensibles

1er solution :

Gitignore – ajouter le appsetting.json qui contient la clé de token

2eme solution :

SecretLocalSrotage

Dans DattingApp.api.csproj :

<UserSecretsId>79a3edd0-2092-40a2-a04d-dcb46d5ca9ed</UserSecretsId>

En ligne de commande :

dotnet user-secrets set "AppSettings:Token" "Super clé hyper secrete"

pour contrôler en ligne de commande :

dotnet user-secrets list

*🡪 AppSettings:Token = Super clé hyper secrete*

Ensuite on peut enlever la ligne dans Appsettings:

"AppSettings":{

    "Token":"Super clé hyper secrete"

  },

PROBLEME : NE FONCTIONNE QU’EN MODE DEVELOPPEMENT

Token authentification :

Token Json 🡪 Standards RFC7519



Informations publiques

Algo codage

Type de codage

Zone cryptée

Cryptage en tête

Crypotage Public

Permet au programme de vérifier l’authentification d’un utilisateur sans interroger le serveur

Token :

En tête

Coprs

Signature : Infos de login à contrôler

* A la première connexion envoi mdp et nom au serveur
* Le serveur hache et contrôle la concordance entre mdp envoyé et contenu de la base
* Le serveur envoi un jeton au client
* Le client sauvegarde le jeton en local
* Ensuite les requêtes se font ensuite en envoyant le jeton

Autentification

Modules à ajouter pour l’autentification :

* Microsoft.IdentityModel.Tokens
* System.IdentityModel.Tokens.Jwt

Autentification par logiciel :

**Module à ajouter :**

* Microsoft.AspNetCore.Authentication.JwtBearer

Dans startup.cs

* Configurer le service
* services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme).AddJwtBearer(options=>{
* options.TokenValidationParameters = new TokenValidationParameters
* {
* ValidateIssuerSigningKey = true,
* IssuerSigningKey = new SymmetricSecurityKey(Encoding.ASCII.GetBytes(Configuration
* .GetSection("AppSettings:Token").Value)),
* ValidateIssuer = false,
* ValidateAudience = false
* };
* });
* Configurer
* app.UseRouting();
* app.UseAuthentication();
* app.UseAuthorization();

**ANGULAR**

Page accueil avec login dans Angular :

* Dans app 🡪 *new component* ---- répertoire **nav**
* Copie modele jumbotron depuis Bootstrap 🡪<nav class= ‘navbar’…….
* Modifications selon affichage désiré
* Dans le <Form> contenant le login on crée une variable pour binder la form
* Chaque input est associé à un binder [(ngModel)]= ‘’ et le submit [(ngSubmit)]= ‘function()’

Après… les services :

Pour ne pas répéter le même code plusieurs fois, on crée un service.

Pour le log : **auth.service**

@Injectable({

  providedIn: 'root'

})

export class AuthService {

  baseUrl  = 'http://localhost:5000/api/auth/';

// tslint:disable-next-line: typedef-whitespace

constructor(private Http : HttpClient) { }

// tslint:disable-next-line: whitespace

// tslint:disable-next-line: typedef-whitespace

login(model:any){

      // pipe empile chk fonction avec le resultat de la fonction précédente

  return this.Http.post(this.baseUrl + 'login', model).pipe(

      // on passe user venant du serveur en paramètre

    map((response: any) => {

      const user = response;

      if (user){

        // le token est enregistré en local pour les connexions futures

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

      }

    })

  // tslint:disable-next-line: semicolon

  )

}

}

Puis l’**injection** du service :

Dans nav.component.ts

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

import { AuthService } from '../\_services/auth.service';

@Component({

  selector: 'app-nav',

  templateUrl: './nav.component.html',

  styleUrls: ['./nav.component.css']

})

export class NavComponent implements OnInit {

  // pour récupérer les valeurs depuis navcomponent.html

  model: any ={};

  constructor(private authService : AuthService) { }

  ngOnInit() {

  }

  login(){

    //il faut souscrire au service avec le this.model

      //subscribe(observer=>)

    this.authService.login(this.model).subscribe(next =>{

      console.log('loggin OK !!!')} , error => {

        console.log('error');

      });

  }

Passage de paramètres du parent vers l’enfant

Dans **home.compnent**, on récupère les valeurs depuis l’api **DattingApp**

**register.component.html (app-register)** est l’enfant appelé dans **home.component.html**

Dans **homeController.html** on binde la variable de **register.component.ts**

<app-register [valuesFromHome] = "values"></app-register>

Dans l’enfant (**register.component.ts**), créer la variable en Input

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

export class RegisterComponent implements OnInit {

  @Input() valuesFromHome;

  model: any ={};

  constructor() { }

les valeurs sont ensuite utilisables dans le **register.component.html** du fait du **@Input**

exemple de liste récupérée depuis l’api via home.component:

 <div class="for-group">

    <label for="favorite-value">Votre valeur préférée</label>

    <select name="" id="favorite-value" class="form-control">

        <!--\*\*\*\*\*\*\*\*[value]-> pour option select-option-value -->

        <!--\*\*\*\*\*\*\* "value"-> valeur de la boucle \*ngFor -->

        <!--\*\*\*\*\*\*\*\* values -> passé à app-regiser avec :

              \*\*<app-register [valuesFromHome] = "values"></app-register> -->

      <option \*ngFor="let value of valuesFromHome" [value]="value">{{value.name}}</option>

    </select>

  </div>

Passage de paramètres de l’enfant vers le parent

Ici on envoie un **event** en passant par un @**Output** dans **l’enfant**

On veut renvoyer un cancel venant de **register.component.html** vers le parent

  @Output() cancerRegister = new EventEmitter();

Qui a été importé

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

dans la fonction cancel on déclenche l’event

  cancel(){

    //envoi de l'event avec la valeur false pour valider le cancel

    this.cancelRegister.emit(false);

    console.log('cancelled');

  }

Enfin dans le [**home.component.html**](http://home.component.html)on lie le cancel à l’enfant **register.component**

        <app-register [valuesFromHome] = "values" (cancelRegister)="cancelRegisterMode($event)"></app-register>

Il faut finalement créer la méthode dans le **home.component.ts** pour passer la valeur de l’enfant vers le parent en l’occurrence

**Récapitulatif**:

* Parent🡪 enfant : Déclarer @Input dans le fichier **enfant .ts**

Utiliser la variable importée du parent dans le **.Html** : *{{value.name}}*

* Enfant🡪 parent : Déclarer @Output entant qu’**EventEmitter** dans le fichier **enfant .ts**
  + - ***Exemple pour un Cancel envoyé depuis l’enfant***
* @Output() cancelRegister = new EventEmitter();

Dans la fonction qui emet l’**Event**

this.cancelRegister.emit(false);

Dans le **parent** **.Html** appel de l’enfant et récupération de l’**Event**:

<app-register [valuesFromHome] = "values" (cancelRegister)="cancelRegisterMode($event)"></app-register>

Dans le **parent .ts** déclaration de la fonction pour récupérer l’**Event**

  cancelRegisterMode(registerMode: boolean){

    this.registerMode = registerMode;

  }

**Module js qui apporte des compléments visuels :**

alertifyjs dans le projet Angular

***npm install alertifysj***

Ensuite créer un **reperoire typing .d.ts** dans **src**

Dans **typing .d.ts** ajouter la ligne

declare module 'alertyfyjs'

Dans **tsconfig.json ajouter :**

"typeRoots": [

      "node\_modules/@types",

      "src/typings.d.ts"

Créer un service ex : **alertify.ts** dans lequel on déclare les fonctions pour les messages

export class AlertifyService {

  constructor() { }

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

    alertify.confirm(message, (e: any) => {

      if (e){

        okCallBack();

      }

        else{}

    });

  }

    // les boites de dialogue type Alertify

  success(message: string){

    alertify.success(message);

  }

  error(message: string){

    alertify.error(message);

  }

  warning(message: string){

    alertify.warning(message);

  }

  message(message: string){

    alertify.message(message);

  }

}

Finalement utiliser les boites de message créées

Exemple du login :

 login(){

    // il faut souscrire au service avec le this.model

      // subscribe(observer=>)

    this.authService.login(this.model).subscribe(next => {

      this.alertify.success('loggin OK !!!'); } , error => {

        this.alertify.error(error);

      });

  }

**Auth0 angular jwt  jwtHelperService** :

Cela permet de vérifier le token envoyé par le client pour la sécurité car une fois émis un token ne peut être révoqué avant son expiration

Ainsi, un token corrompu peut continuer à permettre des échanges car aucune vérification n’est faite par rapport à la BDD serveur

De plus plutôt que de vérifier si le token existe on vérifie s’il est expiré.

Dans le net choisir la version 2 pour Angular y copier le **npm** pour charger le package

# installation with npm

npm install @auth0/angular-jwt

dans auth.services créer une instance de l’objet **JwtHelperService**

export class AuthService {

  baseUrl  = 'http://localhost:5000/api/auth/';

  jwtHelper = new JwtHelperService();

l’import doit être fait manuellement :

import {JwtHelperService} from '@auth0/angular-jwt';

BON JE COMPRENDS MAL LE TRUC….

Ajouter une fonction **loggedIn()** dans le service de gestion des autorisations **auth.service**

 loggedIn() {

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

    return !this.jwtHelper.isTokenExpired(token);

  }

Evidemment cette fonction est appelée lors du login par l’appel depuis la page html via le fichier .ts

**Nav.component.html Nav.component.ts**

  loggedIn() {

    return this.authService.loggedIn();

  }

NGX Bootstrap**:**

Bootstrap adapté pour Angular sans **Jquery**

Installation :

npm install ngx-bootstrap --save

après il faut importer les modules necessaires dans app.modules

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

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

…

…

*imports: [*

      BrowserModule,

      HttpClientModule,

      FormsModule,

      BrowserAnimationsModule,

      BsDropdownModule.forRoot()

Ensuite se reporter à la façon pour l’utiliser sur la doc en ligne

Pour les thèmes : bootstrapwatch

Dans help🡪 methode pour installer

**Routage** :

Créer les composants pour les pages

Dans app créer un ficher ts

**Routes.ts**

Ce fichier contient :

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

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

import {MemeberListComponent} from './memeber-list/memeber-list.component';

import {MessagesComponent} from './messages/message.component';

import {ListsComponent} from './lists/lists.component';

// tableau de routes

export const appRoutes: Routes = [

    {path : 'home', component: HomeComponent},

    {path : 'members', component: MemeberListComponent},

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

    {path : 'lists', component: ListsComponent},

    // pathMatch: 'full' pour correspondance totale

    {path : '\*\*', redirectTo: 'home', pathMatch: 'full'}

]

Dans **nav.component.html** on déclare une instance dans le constructeur

 constructor(

    public authService: AuthService,

    private alertify: AlertifyService,

    private router: Router) { }

ensuite on appelle la fonction pour router aux endroits souhaités.

this.router.navigate(['/home']);

**Route guard** :

Certaines routes ne doivent pas être accessibles depuis la barre du navigateur par un user non identifié

Créer un repertoire

**\_gards**

En ligne de commande dans ce repertoire taper :

ng g guard auth

dans **auth.guards.ts** on simplifie la fonction car on ne désire que savoir si le user est loggé.

export class AuthGuard implements CanActivate {

          // \*\*\*\*\*\*\*\*\*\* appel de base avec ng g ruard auth \*/

  // canActivate(

  //   next: ActivatedRouteSnapshot,

  //   state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {

          // on n'a besoin que de savoir si l'user est connecté donc un boolean

  canActivate(): boolean{

    return true;

  }

On développe le code de CanActivate

 canActivate(): boolean{

    //loggé ? ok

    if (this.authService.loggedIn()){

      return true;

    }

    // non loggé message erreur

    this.alertify.error('dégage de là gros pédé');

    // redirection vers accueil

    this.router.navigate(['/home']);

    return false;

  }

Dans **routes.ts** on appelle canActivate avec en paramètre un tableau de guards ???

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

    {path : 'messages', component: MessagesComponent, canActivate: [AuthGuard]},

    {path : 'lists', component: ListsComponent, canActivate: [AuthGuard]},

**Les dumyroutes** :

Routes qui contiennent plusieurs routes enfant

  {path : '', component: HomeComponent},

*Vide pour permettre de démarrer plusieurs pages avec le même login*

    {   path : '',

        runGuardsAndResolvers: 'always',

        canActivate: [AuthGuard],

        children:[

            {path : 'members', component: MemeberListComponent},

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

            {path : 'lists', component: ListsComponent}

        ]

    },       // canActivate [tableau de guards]

**.NET CORE**

**Mise à jour de la BDD après modifs** :

But : compléter la classe User et ajouter la classe Photo.

Dans le même temps, on compéte les tables avec des données.

On ajoute les champs désirés à User et une collection qui réfère à la classe Photo

public ICollection<Photo> photos { get; set; }

et les liaisons avec User dans Photo

* **IsMain** : *indique si la photo est la photo de profil*

// photo de profil

        public bool IsMain { get; set; }

        //liaisons entre photo et user pour permettre la suppression en cascade

        public User User { get; set; }

        public int UserId { get; set; }

Ces liaisons permettent de réaliser la suppression en cascade en cas de suppression d’un user

        modelBuilder.Entity("DattingApp.api.Models.Photo", b =>

                {

                    b.HasOne("DattingApp.api.Models.User", "User")

                        .WithMany("Photos")

                        .HasForeignKey("UserId")

                        .OnDelete(DeleteBehavior.Cascade)

                        .IsRequired();

                });

* Pour supprimer la Bdd

***Dotnet ef database drop***

* Pour supprimer les migrations précedentes :

**dotnet ef migrations remove**

* Pour créer une nouvelle migration

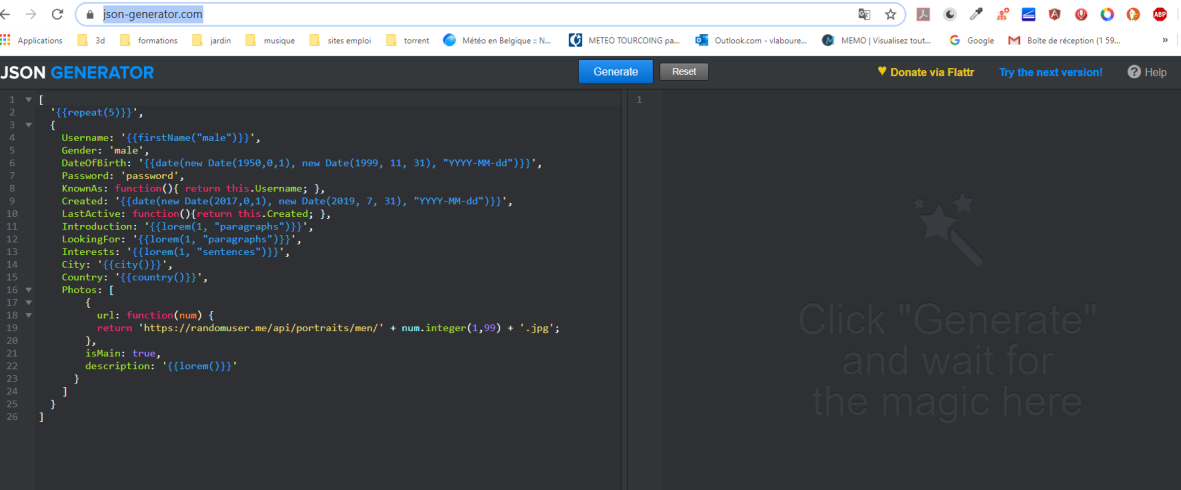
**dotnet ef migrations add *« nom\_de\_la\_migration »***

* Pour enregistrer dans la base

**dotnet ef database update**

Dans **Data** un fichier **UserSeedDatata.json** contient les données à entrer sous forme **json**

App pour créer les données :



Données à entrer

===========

Snippet for JSON Generator site at https://www.json-generator.com/

===========

[

'{{repeat(5)}}',

{

Username: '{{firstName("female")}}',

Gender: 'female',

DateOfBirth: '{{date(new Date(1950,0,1), new Date(1999, 11, 31), "YYYY-MM-dd")}}',

Password: 'password',

KnownAs: function(){ return this.Username; },

Created: '{{date(new Date(2017,0,1), new Date(2017, 7, 31), "YYYY-MM-dd")}}',

LastActive: function(){return this.Created; },

Introduction: '{{lorem(1, "paragraphs")}}',

LookingFor: '{{lorem(1, "paragraphs")}}',

Interests: '{{lorem(1, "sentences")}}',

City: '{{city()}}',

Country: '{{country()}}',

Photos: [

{

url: function(num) {

return 'https://randomuser.me/api/portraits/women/' + num.integer(1,99) + '.jpg';

},

isMain: true,

description: '{{lorem()}}'

}

]

}

]

Les données sont ajoutées dans le **UserSeedDatata.json** après génération.

Ensuite on crée dans **Data** une classe **Seed**

On y recopie la méthode **CreatePassword** car il faut créer les passwords en plus des données générées par le générateur de json

Contenu de la classe :

using System.Collections.Generic;

using System.Linq;

using DattingApp.api.Models;

using Newtonsoft.Json;

namespace DattingApp.api.Data

{

    public class Seed

    {

        public static void SeedUsers(DataContext context)

        {

            if(!context.Users.Any())

            {

                // lecture dans le fichier json

                var userData = System.IO.File.ReadAllText("Data/userSeedData.json");

                // déserialisation du fichier dans users

                var users = JsonConvert.DeserializeObject<List<User>>(userData);

                foreach (var user in users)

                {

                    // ajout dans l'objet

                    byte[] passwordHash, passwordSalt;

                    CreatePassword("password", out passwordHash, out passwordSalt);

                    user.PasswordHash = passwordHash;

                    user.PasswordSalt = passwordSalt;

                    context.Users.Add(user);

                }

                // ecriture dans la base

               context.SaveChanges();

            }

        }

private static void CreatePassword(string password, out byte[] passwordHash,  out byte[] passwordSalt)

        {

            //creation du hash -- using pour gérer le try catch idisposable

            using(var hmac = new System.Security.Cryptography.HMACSHA512()){

                //crée la clé associée au password crypté

                passwordSalt = hmac.Key;

                //transorme chq caractère de password en crypté

                passwordHash = hmac.ComputeHash(System.Text.Encoding.UTF8.GetBytes(password)

            }

        }

    }

}

L’appel pourrait se faire dans **Startup.cs**, mais microsoft recommande de l’injecter dans le **Programm.cs**

    public class Program

    {

        public static void Main(string[] args)

        {

            // changer CreateHostBuilder(args).Build().run :

            var host = CreateHostBuilder(args).Build();

            // using pour liberer après création

                                // CreateScope pour l'injection de dépendance

            using(var scope = host.Services.CreateScope())

            {

                //-------- le but est d'injecter un service pour entrer les données

                //-------- de userSeedData en passant par la methode statique

                //-------- SeedUsers de Seed

                //ServiceProvider permet l'injection de service depuis le main

                var services = scope.ServiceProvider;

                try

                {

                    //appel du service pour injection des données dans la base

                    var context = services.GetRequiredService<DataContext>();

                    // appel de Migrate pour modif database

                    context.Database.Migrate();

                    // appel de la méthode statique pour insertion

                    Seed.SeedUsers(context);

                }

                catch(Exception ex)

                {

                    var logger = services.GetRequiredService<ILogger<Program>>();

                    logger.LogError(ex, "An error occured during the migration process");

                }

            }

            host.Run();

        }

Pour entrer les données, il suffit de lancer l’API

**Création Repository/UserController** :

Afin d’échanger les données entre les bases et les Html, il faut installer un package Json

Ctrl+maj+p 🡪 ajouter un package :

Microsoft.AspDotNetCore.NewtonSoft.Json

Comme avec les autorisations, on crée **l’IDattingRepository** et le **DattingRepository**

Méthodes instanciées :

 void Add<T>(T entity) where T: class;

         void Delete<T>(T entity) where T: class;

         Task<bool>SaveAll();

         Task<IEnumerable<User>> GetUsers();

         Task<User> GetUser(int id);

Ne pas oublier d’ajouter dans startup l’injection du service

            services.AddScoped<IDattingRepository,DattingRepository>();

On crée aussi les DTO

**UserForDetailDto 🡪** pour le GetUser(int id)

**UserForListDto 🡪** pour le GetUsers

**AutoMapping** :

Commencer par installer la package pour l’automapping

AutoMapper.Extensions.Microsoft.DependencyInjection

Important 🡪 dans **startup** injecter le service

Dans helper création d’une classe dérivant de Profile

    public class AutoMApperProfiles : Profile

    {

        public AutoMApperProfiles()

        {

            // création de mapp entre les classes

            CreateMap<User,UserForDetailDto>()

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

                            src.Photos.FirstOrDefault(p => p.IsMain).Url))

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

                            src.DateOfBirth.CalcAge()));

            CreateMap<User,UserForListDto>()

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

                            src.Photos.FirstOrDefault(p => p.IsMain).Url))

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

                            src.DateOfBirth.CalcAge()));

            CreateMap<Photo,PhotoForDetailDto>();

        }

    }

On mappe le user avec le DTO les données affichées seront celles du DTO alimentées par le map

Dans le même temps on récupère dans User l’Url de la photo et on calcule l’age du User

* Url :

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

                           src.Photos.FirstOrDefault(p => p.IsMain).Url))

dest🡪destination

opt🡪 conditions pour selection

* Age :

On utilise une méthode statique créée dans helpers

public static int CalcAge(this DateTime theDateTime)// this DateTime theDateTime-->appel à travers objet

            // dans AutoMapperProfiles : =>src.DateOfBirth.CalcAge() --- src pointe sur User

        {

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

// si l'année de naissance + age > date actuelle anniversaire pas encore arrivé on decrémente

            if (theDateTime.AddYears(age) > DateTime.Today)

                age--;

            return age;

        }

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

Enfin, dans UsersController le map est appelé pour afficher le résultat des requêtes get vers l’api

 [HttpGet]

        public async Task<IActionResult> GetUsers()

        {

            var users = await \_repo.GetUsers();

            // users avec le mapping

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

            return Ok(usersToReturn);

        }

        [HttpGet("{id}")]

        public async Task<IActionResult> GetUser(int id)

        {

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

            // utilisation de automapp

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

            return Ok(userToReturn);

        }

**ANGULAR**

**Présentation de l’interface User en utilisant les interfaces**

* On crée une ***interface*** **User.ts** et une **Photo.ts** qui contiennent les données à afficher
* On crée un ***service*** qui récupére les donées depuis l’api **user.service.ts**

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

import { environment } from 'src/environments/environment';

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

import { Observable } from 'rxjs';

import { User } from '../\_models/User';

/\*\*

 \* création d'un header pour autorisations pour les requêtes get

 \*/

const httpOptions = {

  headers: new HttpHeaders({

    // tslint:disable-next-line: object-literal-key-quotes

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

  })

};

@Injectable({

  providedIn: 'root'

})

export class UserService {

  baseUrl = environment.apiUrl;

constructor(private http: HttpClient) { }

  getUsers(): Observable<User[]>{

    // il faut typer le retrun <User[]>car get retourne un object et pas un user

    return this.http.get<User[]>(environment.apiUrl + 'users', httpOptions);

  }

  getUser(id): Observable<User>{

    // il faut typer le retrun <User[]>car get retourne un object et pas un user

    return this.http.get<User>(environment.apiUrl + 'users/' + id, httpOptions);

  }

}

Les fonctions utilisées retournent un observable de user

* Enfin dans le composant ou on veut afficher **member-list.component** (pour lister tous les users), on appelle ce service

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

import { User } from '../\_models/User';

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

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

@Component({

  selector: 'app-memeber-list',

  templateUrl: './memeber-list.component.html',

  styleUrls: ['./memeber-list.component.css']

})

export class MemeberListComponent implements OnInit {

  users: User[];

  constructor(private userService: UserService, private alertifyService: AlertifyService ) { }

  ngOnInit() {

    this.loadUSers();

  }

    //appel de l'observable getUsers

  loadUSers(){

    this.userService.getUsers().subscribe((users: User[]) =>{

      this.users = users;

    }, error =>{

      this.alertifyService.error(error);

    })

  }

}

* Boucle pour afficher dans **member-list.component.html**

<div class="container">

  <div class="row">

    <div class="col-lg-2 col-md-3 col-sm-6">

      <p \*ngFor="let user of users">{{user.knownAs}} </p>

    </div>

  </div>

</div>

**Page détail user**

* Création d’un composant **member-detail**

Dans **member-detail.ts**, on crée une fonction pour charger le user

export class MemberDetailComponent implements OnInit {

  user: User;

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

private route: ActivatedRoute) { }

  ngOnInit() {

    this.loadUser();

  }

  loadUser(){                       // route.snapshot.params[`id`]--> récupère le paramètre de l'@

    this.userService.getUser(this.route.snapshot.params[`id`]).subscribe((user: User) =>{

      this.user = user;

    }, error => {

      this.alertify.error(error);

    });

  }

}

* route : ActivatedRoute 🡪 route dans la barre de navigation
* route.snapshot.params[`id`])🡪 renvoie un **observable**

il faut donc **souscrire** en passant par une fonction **fléchée**

le (user: User) est optionnel, on peut juste faire subscribe(user=>{…. })

* Pour le test dans le **member-detail.html** afficher le user.knownAs
* Créer une route dans **routes.ts**

 {path : 'members/:id', component : MemberDetailComponent}

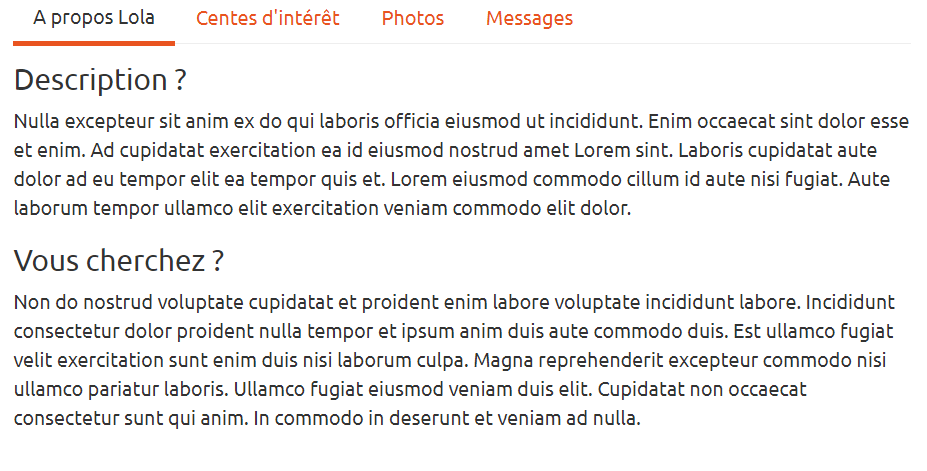
* Enfin, créer l’event dans la card pour le click sur user
* [routerLink]="['/members/',user.id]"

Création de la carte membre

class="member-tabset"

* Réalistation d’un layout

class="tab-panel"

C:\Program Files (x86)\Microsoft Office\MEDIA\CAGCAT10\j0183290.wmf

Ville

Age

Last active

creation

Utilisation de bootstrap ngx (bootstrap sans jqquery)

     <div class="col-sm-8">

        <div class="tab-panel">

          <tabset class="member-tabset">

            <tab heading="A propos {{user?.knownAs}}">

              <h4>Description ?</h4>

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

              <h4>Vous cherchez ?</h4>

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

            </tab>

            <tab heading="Centes d'intérêt">

              <h4>Intérêt</h4>

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

            </tab>

            <tab heading="Photos">

              <p>photos a mettre</p>

            </tab>

            <tab heading="Messages">

              <p>Messages ici</p>

            </tab>

          </tabset>

        </div>

      </div>

**Route resolver**

Le problème est que les datas sont chargées après l’affichage de la page, donc au premier chargement les données sont absentes.

Pour pallier on peut utiliser le ? mais une meilleure solution est d’utiliser un service injecté avant l’affichage pour récupérer les datas.

* Création d’un repertoire \_resolver
  + Dans le répertoire création de **member-detail-resolver.ts**

C’est la classe qui sera injectée par le compilateur pour récupérer les data du user à la place du ***loadUser***de **member-detail.component.ts**

@Injectable()

export class MemberDetailResolver implements Resolve<User>{

   constructor(private userService: UserService, private router: Router,

               private alertify: AlertifyService){}

   resolve(route: ActivatedRouteSnapshot): Observable<User>{

      return this.userService.getUser(route.params['id']).pipe(

         catchError(error => {

            this.alertify.error('Erreur lors de la récupération des datas');

            this.router.navigate(['/members']);

            return of(null); // return of--> return observable of(null)

         })

      );

   }

}

* Dans **app.module.ts** ajout du resolver créé comme provider

 providers: [

      ErrorInterceptorProvider,

      AuthService,

      AlertifyService,

      UserService,

      MemberDetailResolver,

   ],

* Dans **routes.ts** modification de la route vers ***members/id :***

{path : 'members/:id', component : MemberDetailComponent, resolve: {user: MemberDetailResolver}}

* Dans **member-detail.component.ts** modifier le ***ngOnInit*** :

  ngOnInit() {

    // this.loadUser();--> inutile car resolver

        // refére à route path: member/:id ... , resolver ...

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

      this.user = data[`user`];

    });

  }

  // loadUser(){                       // route.snapshot.params[`id`]--> récupère le paramètre de l'@

  //   this.userService.getUser(+this.route.snapshot.params[`id`]).subscribe((user: User) =>{

  //     this.user = user;

  //   }, error => {

  //     this.alertify.error(error);

  //   });

  // }

}

* Finalement dans le **member-detail.component.html** on peut supprimer les ? après les user

**Galerie de photo**

Galerie de photo open source

Code : ***npm install ngx-gallery***

**Ngx-gallery** a besoin de définition - D’un tableau d’options

- D’un tableau d’images

* Dans **app.module**

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

* + Dans imports

  imports: [

      /\*\*\*\*\*\* utilier .froRoot permet de créer un singleton

       \* \*\*\*\* pour n'appliquer le service que sur la route en cours

       \*/

      BrowserModule,

      HttpClientModule,

      NgxGalleryModule,

* Dans **member.component.ts** on crée deux variables pour le composant ngx-gallery:

export class MemberDetailComponent implements OnInit {

  @ViewChild('memberTabs', {static: true}) memberTabs: TabsetComponent;

  user: User;

  galleryOptions: NgxGalleryOptions[];

  galleryImages: NgxGalleryImage[];

  this.galleryOptions = [

Tableau d’options de ngx-gallery

      {

          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,

Fonction pour récupérer les images

        medium: photo.url,

        big: photo.url,

        description: photo.description

      });

    }

    return imageUrls;

  }

* Finalement dans **member.component.html** on insere le module ngx-gallery dans l’onglet photo

  <tab heading="Centes d'intérêt">

              <h4>Intérêt</h4>

              <p>{{user.interests}}</p>

            </tab>

            <tab heading="Photos">

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

            </tab>

warning-icon-24 - Maman Cat

Ne pas oublier de définir une classe pour la mise en forme du composant

**.class="ngx-gallery"**

**Edition du profil**

Comme le composant ressemble à **memberDetail** copier le composant, le renommer

* Le ***ngOnInit*** est idem
* On ajoute une fonction pour sauvegarder

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

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

import { User } from 'src/app/\_models/User';

import { AlertifyService } from 'src/app/\_services/alertify.service';

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

@Component({

  selector: 'app-member-edit',

Variable référant au dom pour accéder aux propriétés du Form

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

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

})

export class MemberEditComponent implements OnInit {

  // variable pour accéder au dom et modifier le editForm pour faire disparaître le message d'avertissement

  // Vous avez fait des changements sans valider, vos modifications seront perdues

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

  user: User;

  // le navigateur et non l'appli envoie un message si quitter après modif sans enregistrer

  @HostListener('window:beforeunload',['$event'])

  unloadNotification($event: any){

    if (this.editForm.dirty){

Event envoyé au navigateur avant de fermer l’onglet (si changement)

      $event.returnValue = true;

    }

  }

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

  ngOnInit() {

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

      this.user = data['user'];

    });

  }

  updateUser(){

    this.alertify.success('modifiactions enregistrées');

    console.log(this.user);

    this.editForm.reset(this.user);

  }

}

**Le member-edit.component.html** comporte

* 2 onglets
* Un bouton btn-block

<div class="container mt-4">

En liaison avec le Form (ngSubmit) si un changement dans le formulaire, le texte apparaît et disparaît après sau vegarde

En liaison avec le Form (ngSubmit) si un changement dans le formulaire, le texte apparaît et disparaît après sauvegarde

  <div class="row">

    <div class="col-sm-4">

      <h1>Votre profil</h1>

    </div>

    <div \*ngIf="editForm.dirty" class="col-sm-8">

      <div class="alert alert-info">

        <strong>Attention </strong>Vous avez fait des changements sans valider, vos  modifications seront perdues</div>

  </div>

  </div>

  <div class="row">

    <div class="col-sm-4">

      <div class="card">

        <img src="{{user.photoUrl}}" alt="{{user.knownAs}}" class="card-img-top img-thumbnail">

        <div class="card-body">

          <div>

            <strong>Location</strong>

            <p>{{user.city}}, {{user.country}}</p>

          </div>

          <div>

            <strong>Age</strong>

            <p>{{user.age}}</p>

          </div>

          <div>

            <strong>Last active</strong>

            <p>{{user.lastActive}}</p>

          </div>

          <div>

            <strong>Create</strong>

            <p>{{user.created}}</p>

          </div>

        </div>

        <div class="card-footer">

          <!-- button-block-> toute la largeur -->

          <button [disabled]="!editForm.dirty" form="editForm" class="btn btn-success btn-block">Sauver</button>

        </div>

      </div>

    </div>

    <!-- ici on fait un container avec des onglets tabmodule -->

      <div class="col-sm-8">

        <div class="tab-panel">

          <tabset class="member-tabset">

            <tab heading="Editer le profile">

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

                <h4>Description ?</h4>

                <textarea name="introduction" rows="6" class="form-control"

                 [(ngModel)]="user.introduction"></textarea>

                <h4>Vous cherchez ?</h4>

                <textarea name="lookingFor" rows="6" class="form-control"

                 [(ngModel)]="user.lookingFor">

[(ngModel)]🡪permet d’afficher et de modifier. Input et textarea

                </textarea>

                 <h4>Centres d'interêt?</h4>

                <textarea name="interests" rows="6" class="form-control"

                  [(ngModel)]="user.interests">

                </textarea>

                  <div class="form-inline mt-2">

                    <label for="city">Ville</label>

                    <input type="text" class="form-control" name="city" [(ngModel)]="user.city">

                    <label for="city">Pays</label>

                    <input type="text" class="form-control" name="country" [(ngModel)]="user.country">

                  </div>

                  <div class="form-inline mt-2">

                    <label for="city">Nom</label>

                    <input type="text" class="form-control" name="userName" [(ngModel)]="user.userName">

                    <label for="city">Age</label>

                    <input type="text" class="form-control" name="age" [(ngModel)]="user.age">

                  </div>

              </form>

            </tab>

            <tab heading="Editer les photos">

              <p>photo a venir</p>

            </tab>

            </tabset>

        </div>

      </div>

    </div>

  </div>

* Dans **app.module.ts** on ajoute la déclaration du composant

MemberEditComponent

* Dans routes.ts on ajoute la route

{path : 'members/edit', component: MemberEditComponent,

                    resolve: {user: MemberEditResolver}},

* Pour éviter de perdre les modifications, on crée un service dans guards\_

**Prevent\_unsaved-changes.ts**

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

import { MemberDetailComponent } from '../members/member-detail/member-detail.component';

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

import { MemberEditComponent } from '../members/member-edit/member-edit.component';

@Injectable()

export class PreventUnsavedChanges implements CanDeactivate<MemberEditComponent>{

    canDeactivate(component: MemberEditComponent){

        if (component.editForm.dirty){

            return confirm('Etes vous sur de vouloir quitter ? tous les changements seront perdus');

        }

        return true;

    }

}

On le déclare dans le app.module

providers: [

      ErrorInterceptorProvider,

      AuthService,

      AlertifyService,

      UserService,

      MemberDetailResolver,

      MemberListResolver,

      MemberEditResolver,

      PreventUnsavedChanges,

      {provide: HAMMER\_GESTURE\_CONFIG, useClass: CustomHammerConfig}

   ],

   bootstrap: [

Injection du service

on l’injecte dans routes.ts

            {path : 'members/edit', component: MemberEditComponent,

                    resolve: {user: MemberEditResolver}, canDeactivate: [PreventUnsavedChanges]},

            {path : 'lists', component: ListsComponent},

**Sauvegarde des modifications de profil** :

**.NET CORE**

* Création d’un DTO pour les données à enregistrer dans la base
* Création de la fonction update dans le **UserControler**

***UpdateUser :***

 [HttpPut("{id}")]

        public async Task<IActionResult> UpdateUser(int id, UserForUpdateDto userForUpdateDto)

Verification id/id du token

        {

            // vérif si le token correspond à l'id reçu dans la requête

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

                return Unauthorized();

Données user en BDD

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

            \_mapper.Map(userForUpdateDto, userFromRepo);

            if(await \_repo.SaveAll())

                return NoContent();

AutoMap pour mapper les données depuis Angular vers la BDD via User

            throw new Exception($"Erreur sauvegarde id {id}");

        }

* Dans AutoMappreProfil.cs ajouter le mapping entre UserForUpdateDto et User

      CreateMap<UserForUpdateDto,User>();

**ANGULAR :**

* Dans **user.service.ts** on implémente la fonction ***updateUser***

 updateUser(id: number,user: User){

    return this.http.put(this.baseUrl + 'users/' + id, user);

  }

* Dans **member-edit.component** appel de ***updateUser***

Pour les paramètre à passer à la fonction on a besoin de UserService (accès au user) et AuthService(accès au user courant par l’id du token)

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

                                             private authService: AuthService) { }

* Fonction update

  updateUser(){

    this.userService.updateUser(this.authService.decodedToken.nameid, this.user).subscribe(next => {

      this.alertify.success('mise à jour réussie');

      this.editForm.reset(this.user);

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

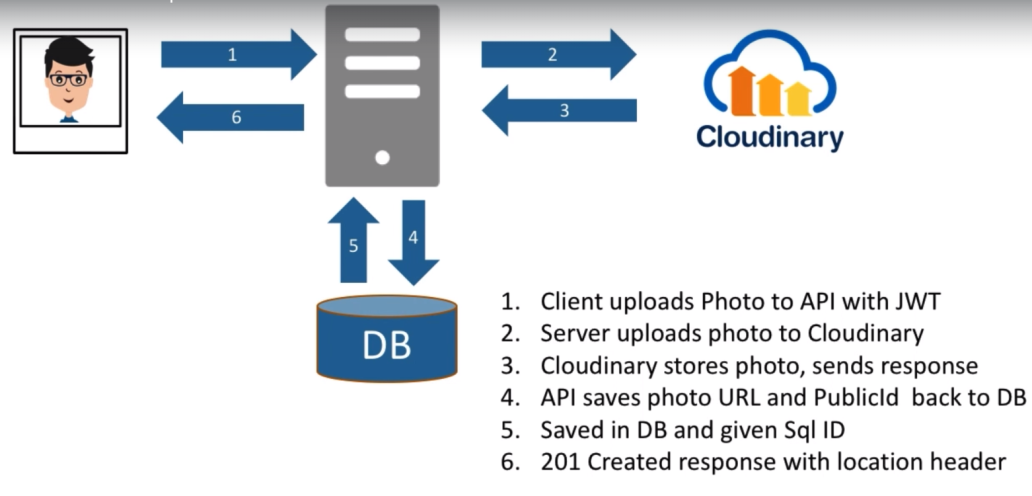
  }

**Gestion de la galerie de photo**

Utlisation du stockage sur le cloud



Fonctionnement :



* Côté serveur datting.api, on configure le service pour echanger avec **Cloudinary**
* Côté client **ng2-file-upload** pour la gestion de l’upload

**.NET CORE**

A la fin du fichier **appsettings.json** ajouter les lignes suivantes :

  "CloudinarySettings":{

    "CloudName":"djff7mylm",

    "ApiKey":"785929736947663",

    "ApiSecret":"N5bbx91xMtvYiX9qggcMz5J7oUg"

  }

Les valeurs notés sont récupérées du compte créé sur le site cloudinary

Dans helpers création classe **CloudinarySettings**

namespace DattingApp.api.helpers

{

    public class CloudinarySettings

    {

        public string CloudName { get; set; }

        public string ApiKey { get; set; }

        public string ApiSecret { get; set; }

    }

}

* Dans **startup.cs** on alimente le CloudinarySettings avec les données de appsettings.json

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

et comme il y a une pour accéder à l’emplacement dans le cloud, on ajoute une propriété dans **photo.cs**

    public string PublicId { get; set; }

forcement il faut refaire une migration

***dotnet ef migrations Add AddPublicId***

***dotnet ef database update***

* puis on crée un controller et un Dto **PhtotoForCreationDto**

    public class UserForUpdateDto

    {

        public string Introduction { get; set; }

        public string LookingFor { get; set; }

        public string Interests { get; set; }

        public string City { get; set; }

        public string Country  { get; set; }

    }

* Dans photoController
* Propriétés :
  + Repository pour les requêtes
  + Mapper pour mapper Photo et le Dto
  + cloudinaryConfig pour les options de photo
* Instance de classe :
  + Cloudinary pour accéder à Cloudinary (cloud)
* Une variable locale dans le constructeur comme paramètre pour le Cloudinary
  + Acount acc = new Account()
* public Cloudinary(Account account);
* Dans le constructeur on initialise les propriétés
* La variable Account avec les valeurs de ***clounidarySettings*** de **appSettings.json**

 public PhotoController(IDattingRepository repo, IMapper mapper

                           , IOptions<CloudinarySettings> cloudinaryConfig)

        {

            \_cloudinaryConfig = cloudinaryConfig;

            \_mapper = mapper;

            \_repo = repo;

                // on renseigne les données de CloudinarySettings de appsettings.json

            Account acc = new Account(

                \_cloudinaryConfig.Value.CloudName,

                \_cloudinaryConfig.Value.ApiKey,

                \_cloudinaryConfig.Value.ApiSecret

            );

            \_cloudinary = new Cloudinary(acc);

        }

Rôle des méthodes pour l’insertion de photo dans le contrôleur

* [Httppost]—AddPhotoForUser
* [httpGet] – GetPhoto
* AppPhotoDorUser :

En plus du [Authorize] on vérifie que le token est valide avant d’uploader

Même code que pour le UpdateUser

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

                return Unauthorized();

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

classes pour accés au Cloudinary :

ImageUploadResult 🡪 contient l’objet envoyé avec au Cloudinary

ImageUploadParams🡪 paramètres de l’image et image

Transformation🡪 transformations appliquées à l’image dans ImageUplloadParams

Partie du post qui envoie les données au cloud

      [HttpPost]

      public async Task<IActionResult> AddPhotoForUser(int userId,

              [FromForm]PhotoForCreationDto photoForCreationDto)

      {

            //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

           //même code que dans userController

            // vérif si le token correspond à l'id reçu dans la requête

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

              return Unauthorized();

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

         var file = photoForCreationDto.File;

            //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

                                // methode de cloudinary contenant l'image du cloud avec parametres

         var uploadResult = new ImageUploadResult();

         if (file.Length >0)

         {

                //lecture de fichier image

            using(var stream = file.OpenReadStream())

            {               // methode de cloudinary pour les paramètres de l'image

              var uploadParams = new ImageUploadParams()

              {           // methode de cloudinary pour les fichiers

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

               Transformation = new Transformation().Width(500).Height(500).Crop("fill").Gravity("face")

                   };

                // dans la variable d'image uploadResult on charge les paramètres d'affichage et l'image

                    // avec les paramètres : nom chemin, taille...

                            //chargement de la photo sur le cloud

                uploadResult = \_cloudinary.Upload(uploadParams);

             };

         }

Partie du HttpPost qui récupère l’image du cloud avec PublicId

  // on transmet au DTO les valeurs récupérées depuis le coudinary

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

            // publicId fourni par le cloudinary

            photoForCreationDto.PublicId = uploadResult.PublicId;

            // enfin on mappe photo / photoForUpdate

            //mapp avec type de retour<> et passage de parmètre

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

            userFromRepo.Photos.Add(photo);

            // si une seule photo

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

                photo.IsMain = true;

            if (await \_repo.SaveAll())

            {

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

               return CreatedAtRoute("GetPhoto", new {userId = userId, id = photo.Id},photoToReturn);

            }

GetPhoto methode du controller qui utilise une méthode du repository qu’il faut ajouter

            return BadRequest("Impossible d'ajouter la photo");

Méthode pour récupérer la photo depuis le cloud

 [HttpGet ("{id}", Name = "GetPhoto")]

        public async Task<IActionResult> GetPhoto(int id)

        {

            var photoFromRepo = await \_repo.GetPhoto(id);

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

            return Ok(photo);

        }

Cette methode est ajoutée au IDattingRepository et DattingRepository

        Task<Photo> GetPhoto(int id);

DattingRepository

        public async Task<Photo> GetPhoto(int id)

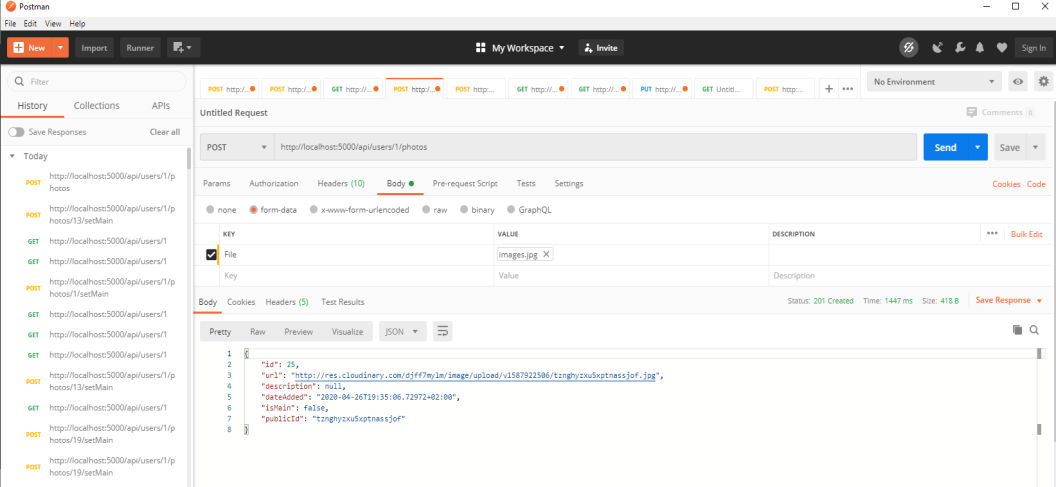
        {

            var photo = await \_context.Photos.FirstOrDefaultAsync(p => p.Id == id);

            return photo;

        }

Test avec postman



Post avec user id=1 🡪 (lola)

Le header contient le Bearer

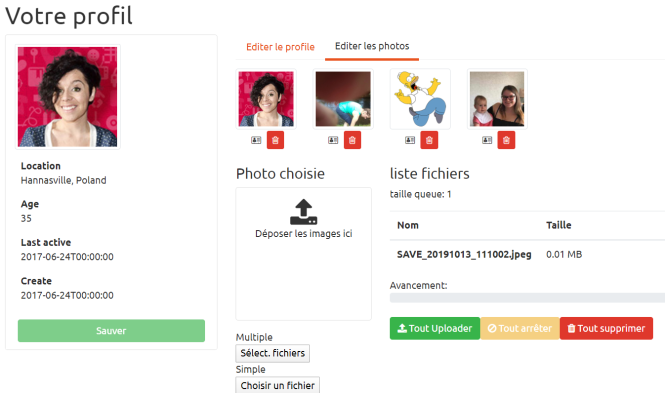
**Body** : form-data

Key : File 🡪 File utilisé dans l’api dans le DTO ***PhotoForCreationDto***

Type : File

**ANGULAR :**

**Ajout de l’upload dans l’onglet éditer**

****

* Dans **member** on crée un nouveau composant ***photo-editor.component***
* Ajouter le app-photo-editor dans les déclarations de **app.module**

Ce composant est un enfant de **member-edit.component**.

Il faut donc une variable en input dans **photo-editor.component**🡪 on utilise un tableau de photos

export class PhotoEditorComponent implements OnInit {

  @Input() photos: Photo[];

  constructor() { }

  ngOnInit() {

  }

Et dans **member-edit.component.html**, le composant est appelé avec en argument un tableau alimenté avec le tableau de photos de User

 <tab heading="Editer les photos">

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

            </tab>

Le photo-editor.component quant à lui contient une boucle qui affiche les photos du tableau avec 2 boutons :

🡪photo de profil

🡪supprimer

<div class="row">

  <div class="col-sm-2" \*ngFor="let photo of photos">

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

    <div class="text-center">

      <button type="button" class="btn btn-sm"><i class="fa fa-address-card-o"></i></button>

      <button type="button" class="btn btn-sm btn-danger"><i class="fa fa-trash-o"></i></button>

    </div>

  </div>

</div>

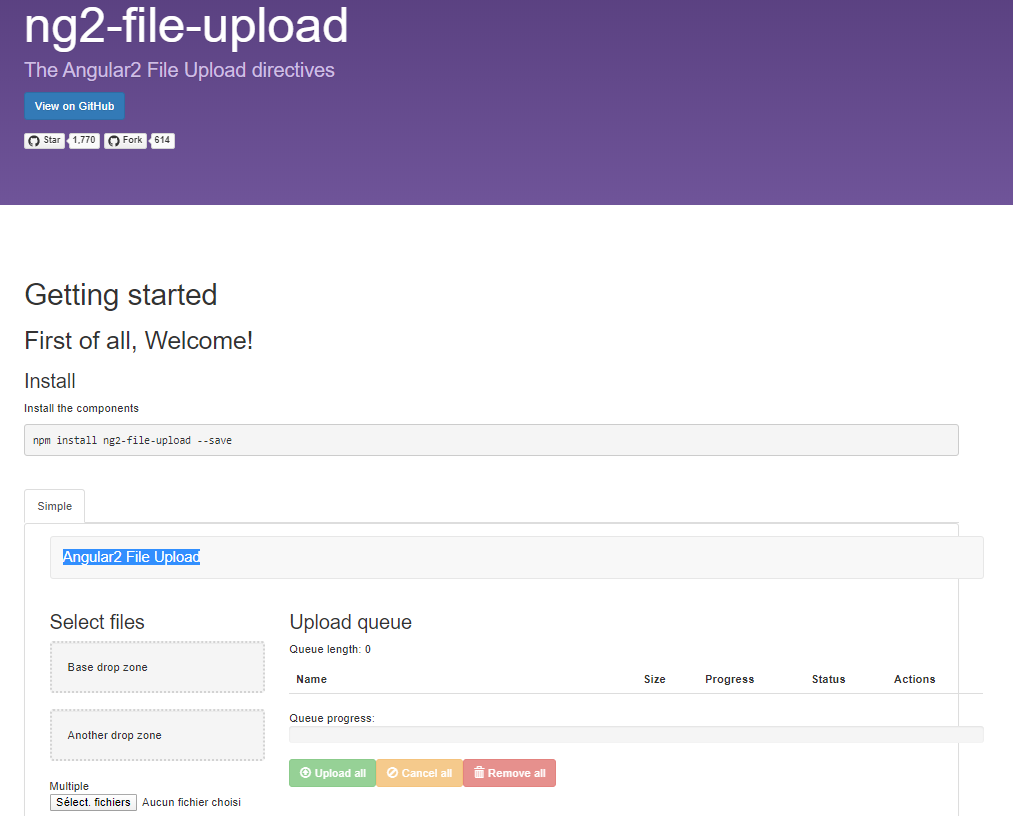
En dessous des photos, on veut pouvoir ajouter des photos

**Fonction Upload**

On utilise Angular2 file upload ou

ng2-file-upload

Sur google taper : ng2-file-upload pour aller sur le site de Valor Software



Pour faire simple, on copie la quasi totalité du html de la démo que l’on arrangera

Puis dans TypeScript on copie la partie upload

export class PhotoEditorComponent implements OnInit {

  @Input() photos: Photo[];

  uploader: FileUploader;

  hasBaseDropZoneOver = false;

  baseUrl = environment.apiUrl;

  ngOnInit() {

  }

  public fileOverBase(e: any): void {

    this.hasBaseDropZoneOver = e;

  }

Ensuite on crée une fonction pour initialiser le ***uploader***

initializeUploader(){

    this.uploader = new FileUploader({

      url : this.baseUrl + 'users/' +  this.authService.decodedToken.nameId + '/photos',

      authToken : 'Bearer ' + localStorage.getItem('token'),

      isHTML5: true,

      allowedFileType: ['image'],

      removeAfterUpload: true,

      autoUpload: false,

      maxFileSize: 10 \* 1024 \* 1024

    });

En têtes nécessaires :

import { environment } from 'src/environments/environment';

import { AuthService } from 'src/app/\_services/auth.service';

* Récupération de l’adresse 🡪

*url : this.baseUrl + 'users/' + this.authService.decodedToken.nameId + '/photos'*

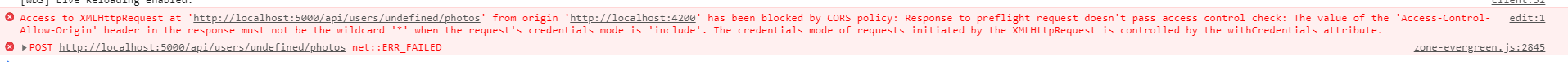
* Jeton 🡪

*authToken : 'Bearer ' + localStorage.getItem('token')*

*Adresse pour liste des methodes et propriétés de fileuploader*

<https://github.com/valor-software/ng2-file-upload/blob/development/src/file-upload/file-uploader.class.ts#L293>

A la fin on tombe sur une erreur :



warning-icon-24 - Maman Cat

Pour permettre de requettes d’un domaine à un autre on a utilisé le cross origin dans le startup.cs de l’api **DattingApp.api**

  app.UseCors(x=>x.AllowAnyOrigin().AllowAnyMethod().AllowAnyHeader());

il faudrait ajouter .***AllowCredential*** dans le startup mais cela autoriserait les **cookies** mais le token est utilisé pour éviter les cookies

ou ***.WithOrigins(‘http://localhost :4200’)***

* Solution : on la fonction ***onAfterAddingFile*** à ***FileUploader***

C’est l’utilisation de methode statique d’extension qu’on ajoute à la création du uploader

initializeUploader(){

    this.uploader = new FileUploader({

      url : this.baseUrl + 'users/' +  this.authService.decodedToken.nameId + '/photos',

      authToken : 'Bearer ' + localStorage.getItem('token'),

      isHTML5: true,

      allowedFileType: ['image'],

      removeAfterUpload: true,

      autoUpload: false,

      maxFileSize: 10 \* 1024 \* 1024

    });

   // création d'un méthode d'extension

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

  }

* Enfin dès qu’on upload une photo, il faut l’afficher dans l’edition du profil

    // après chargement dans le cloud et envoi du PublicId à la base par le cloud

    // on charge les données depuis

    this.uploader.onSuccessItem = (item, response, status, headers) => {

      if (response){

        const res: Photo = JSON.parse(response);

        const photo = {

          id: res.id,

          dateAdded: res.dateAdded,

          url: res.url,

          description: res.description,

          isMain: res.isMain

        };

        this.photos.push(photo);

      }

    };

***OnSuccessItem*** event déclenché quand le serveur à renvoyé un header confirmant la réussite

On récupère les données de la réponse renvoyée par le serveur « ***response*** »

On parse le string http pour créer un objet Photo

const res: Photo = JSON.parse(response);

On ajoute la nouvelle photo au tableau de photos du **photo-editor.component**

  @Input() photos: Photo[];

        this.photos.push(photo);

**Gestion de la main photo**

**.NET CORE**

Pour updater une application **restful**, il faut utiliser **push** ou **patch** mais pour des changements mineurs comme passer un booléen de vrai à faux, non peut utliser un **post**

  [HttpPost ("{id}/setMain")]

On commence par vérifier le l’id du jeton et l’id de la requête avec ClaimTypes.NameIdentifier

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

                return Unauthorized();

On vérifie ensuite que la photo est bien dans la gallerie du user

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

            if(!userFromRepo.Photos.Any(p =>p.Id == id))

                return Unauthorized();

Pour cela il faut créer une nouvelle méthode pour trouver la photo main, dans IDattingRepository/DattingRepository

        public async Task<Photo> GetMainPhotoForUser(int userId)

        {

            return await \_context.Photos.Where(u=> u.UserId == userId).

                    FirstOrDefaultAsync(p=>p.IsMain == true);

        }

enfin on passe la photo IsMain à false pour la photo de profil et on change le IsMain de la photo choisie

code complet :

Méthode post complète

        [HttpPost ("{id}/setMain")]

        public async Task<IActionResult>  SetMainPhoto(int userId, int id)

        {

            //même code que dans userController

            // vérif si le token correspond à l'id reçu dans la requête

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

                return Unauthorized();

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

            if(!userFromRepo.Photos.Any(p =>p.Id == id))

                return Unauthorized();

            // récupération de la photo choisie

            var photoFromRepo = await \_repo.GetPhoto(id);

            if(photoFromRepo.IsMain)

                return BadRequest("Photo déjà en profil");

            // récupération de la photo de profil et changement de IsMain

            var currentMainPhoto = await \_repo.GetMainPhotoForUser(userId);

            currentMainPhoto.IsMain = false;

            photoFromRepo.IsMain = true;

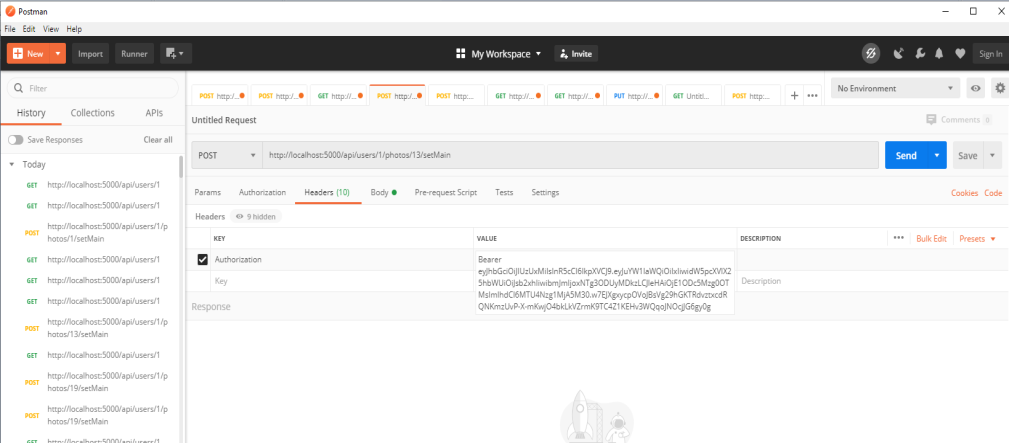
            if(await \_repo.SaveAll())

                return NoContent();

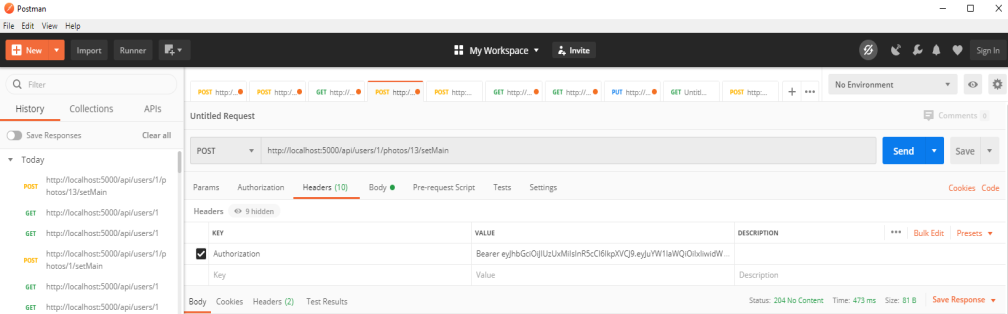
            return BadRequest("Erreur lors du chargment de la photo de profil");

        }

* Test avec postman



Envoi du post avec le jeton du user (lola🡪 id : 1)



204 = post effectué sans problème

**ANGULAR :**

Dans service on ajoute une fonction ***setMainPhoto***

setMainPhoto(userId: number, id: number){

    return this.http.post(this.baseUrl + 'users/' + userId + '/photos/' + id + '/setMain', {});

  }

On envoie la requête :

* this.baseUrl 🡪http//localhost :5000/
* users/userId/photos/idPhoto/setmain

Il faut terminer par deux acollader car le post attend un objet.

On utilise cette méthode dans **photo-editor.component.ts**

  setMainPhoto(photo: Photo){

    this.userServices.setMainPhoto(this.authService.decodedToken.nameid , photo.id).subscribe(() => {

      // on change le main dans la partie angular

      // le but est d'afficher l'image sans interroger le serveur

      // reste à rafraichir le parent--> member-edit.component

      console.log('photo passée en main');

    }, error => { this.alertifyService.error('erreur lors de la tentative de changement de main photo')})

  }

Après dans **photo-editor.component.html** associer le bouton main et la fonction

      <button type="button" class="btn btn-sm mr-1"

                (click)="setMainPhoto(photo)"

                [disabled]="photo.isMain"

                [ngClass]="photo.isMain?'btn-success active':'btn-secondary'">Main

      </button>

Changement de la classe bootstrap en fonction de l’état de la photo

Les changements ne sont pas répercutés dans le parent.

Pour permettre la transmission de **photo-editor** à **member-edit** on utilise le ***@output***

* création de l’eventEmitter dans photo-editor

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

* dans setMainPhoto, récupération de la photo main, passage de isMain à false et iMain de la photo choisie à true
  + pour cela on utilise une variable **Photo** pour stocker la photo main initialement

 currentMain: Photo;

  setMainPhoto(photo: Photo){

    this.userServices.setMainPhoto(this.authService.decodedToken.nameid , photo.id).subscribe(() => {

      // on change le main dans la partie angular

      // le but est d'afficher l'image sans interroger le serveur

      // reste à rafraichir le parent--> member-edit.component

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

Envoi de la nouvelle photo au parent

      this.currentMain.isMain = false;

      photo.isMain = true;

      this.getMemeberPhotoChange.emit(photo.url);

    }, error => { this.alertifyService.error('erreur lors de la tentative de changement de main photo')})

  }

* Dans **member-edit.component.html** on injecte l’output de l’enfant

 <tab heading="Editer les photos">

                <!-- on lie output getMemberPhotoChange de l'enfant photo-editor

                  à une méthode a ajouter dans le member-edit.component.ts  -->

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

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

</app-photo-editor>

Injection de la photo au parent

 </tab>

* Finalement la methode ***updateMainPhoto()*** est implémentée dans **member-edit.component.ts**

  updateMainPhoto(photoUrl){

    this.user.photoUrl = photoUrl;

  }

**Affichage de la photo dans la navbar**

Pour afficher la photo :

* on ne désire pas aller interroger le serveur pour récupérer l’image de profil
* on de désire pas ontégrer la photo dans le token -🡪 il faudrait d’ailleurs génerer un nouveau token pour stocker la nouvelle photo à chaque changement de photo

Solution retenue :

* stockage de l’url de la photo dans le token de login et sauvegarde dans un localStorage(‘user’)

**.NET CORE**

* On commence par maper le userForList et le user dans une variable ***user*** dans ***AuthController.cs***

var user = \_mapper.Map<UserForListDto>(userFromRepo);

* Cette variable est ajoutée au token

token = tokenHandler.WriteToken(token),user

* Dans le AuthRepository.cs il faut modifier la requêtte sur la table ***User***

 var user = await \_context.Users.Include(p => p.Photos).FirstOrDefaultAsync(x=>x.UserName==userName);

Une fois l’url de la main photo récupérée, il faut passer sur Angular pour l’affichage

**ANGULAR :**

Dans app.component, il faut récupérer le user avec l’url de sa photo de profil pour l’affichage dans le navmenu et stocker ce user.

Le stockage se fait dans **Auth.service.ts**

Variable pour stoker le currentUser

decodedToken: any;

currentUser: User;

Dans le login on enregistre le user dans le ***localStorage***

On initialise ***currentUser*** (exporté avec la classe) pour l’utiliser dans l’affichage **nav.component.html**

On ajoute le ***user*** dans le ***localStorage*** dans le login de auth.component.ts

On initialise la varable ***currentUser***

 map((response: any) => {

        const user = response;

        if (user){

                  // le token est enregistré en local pour les connexions futures

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

          localStorage.setItem('user', JSON.stringify(user.user));

          // user ---> token dans le http envoyé par le serveur

          this.currentUser = user.user;

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

        }

Dans le app.component pour le cas ou le user est déjà loggé, on ajoute au ngOnInit la récupération des données user en plus des données de token

 ngOnInit(){

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

    const user: User = JSON.parse(localStorage.getItem('user'));

    if (token){

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

    }

    if (user){

      this.authService.currentUser = user;

    }

  }

Finalement on insère la photo dans **nav.component.html**

 <span>

    <img src="{{authService.currentUser.photoUrl}}" alt="">

</span>

Avec un ajout au css pour l’affichage :

img{

    max-height: 50px;

    border:2px solid wheat;

    display: inline;

    margin-right: 10px;

}

Finalement comme ils ne sont plus utilisés, on supprime les données du localStorage dans le ***logout*** dans **nav.component.ts**

 logOut(){

    localStorage.removeItem('token');

    localStorage.removeItem('user');

    this.authService.decodedToken = null;

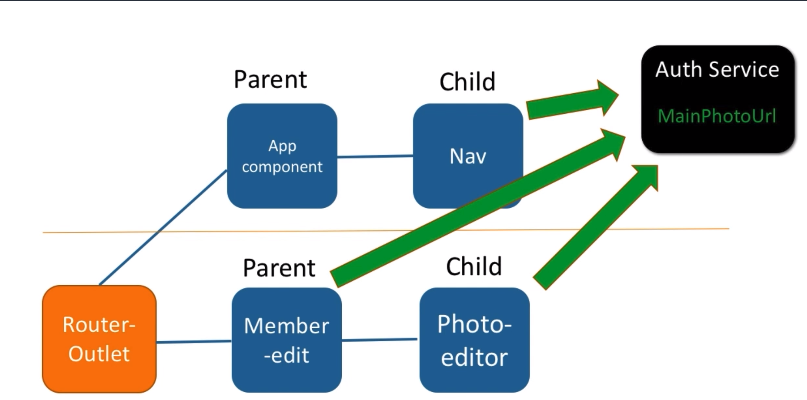
    this.authService.currentUser = null;

    console.log('logged out');

    this.router.navigate(['/home']);

  }

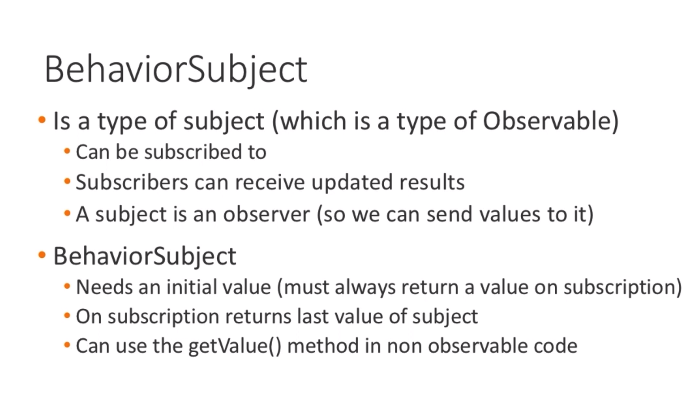
**Communication entre composants sans relation**



On crée une ***propriété*** ***observable*** dans un Service et les composants ***souscrivent***

Comportements des behaviorSubjects à la fois observables et observateurs

* Initialisé avec une valeur
* Doit retourner une valeur au souscripteur qui l’invoque



* Afficher une photo de profil vide par defaut

Le travail se passe dans login dans **auth.service.ts**

* On crée un behaviorSubject exportable ***photoUrl***
* Un ***observable*** ***currentPhotoUrl*** pour soucrire à la photo courrante
* Pour cela on a besoin d’importer behaviorSubject de rxjs

Variables exportables :

 photoUrl = new BehaviorSubject<string>('../../assets/user.png');

  currentPhotoUrl = this.photoUrl.asObservable();

on utilise une methode pour changer la photo de profil et charger la photo du user

 photoUrl = new BehaviorSubject<string>('../../assets/user.png');

  currentPhotoUrl = this.photoUrl.asObservable();

On crée une méthode pour changer la photo

changeMemberPhoto(photoUrl: string){

  this.photoUrl.next(photoUrl);

}

dans le login on appelle la méthode créée

 this.changeMemberPhoto(this.currentUser.photoUrl);

* Pour charger la photo par defaut du **menu** dans le **nav.component.ts** et le render accessible dans le html, on crée une variable string que l’on passe à l’observable 🡪 currentPhotoUrl

  ngOnInit() {

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

  }

Et du fait on change l’appel dans **nav.component.html**

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

        <span>

          <img src="{{photoUrl}}" alt="">

        </span>

Avec une variable exportable pour accès à la photo du user

  photoUrl: string;

* On changera aussi la photo dans **member.edit.component.ts**

  ngOnInit() {

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

  }

Et du fait dans **member.edit.component.html**:

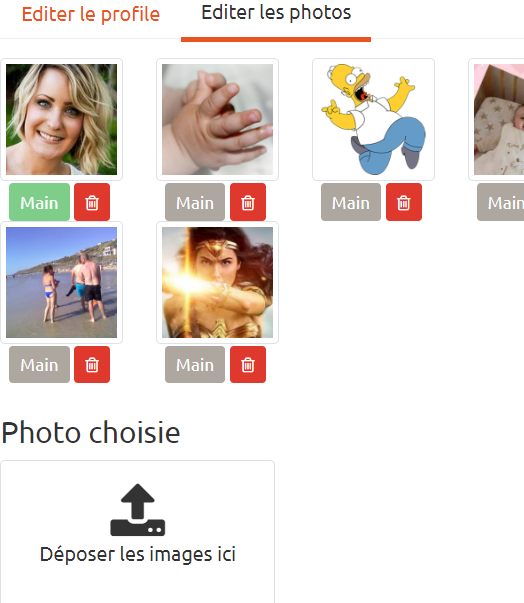
<div class="card">

        <img src="{{photoUrl}}" alt="{{user.knownAs}}" class="card-img-top img-thumbnail">

Avec une variable exportable pour accès à la photo du user

  photoUrl: string;

* Il faut aussi effectuer le changement dans le **photoEditor.component**

****

Il faut que le changement se fasse dans

* nav.component
* edit.component
* photoEditor.component
* app.component (pour affichage au lancement)

dans la méthode de gestion de changement de photo ***setMainPhoto***

   this.authService.changeMemberPhoto(photo.url);

      // this.getMemeberPhotoChange.emit(photo.url); changé avec le behaviorSubject

Et pour afficher la photo si le user est enregistré dans **app.component.ts** :

 if (user){

      this.authService.currentUser = user;

      this.authService.changeMemberPhoto(user.photoUrl);

    }

Reste un problème : si on rafraichit les changements sont annulés.

Il faut donc changer le ***AuthService.currentUser.photoUrl*** ainsi que le ***localStorage*** lors du click sur main dans **photoEditor.component.ts**

Dans la méthode de changement de photo :

    this.authService.changeMemberPhoto(photo.url);

      // pour le changement de photo et le rafraichissement

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

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

**Conclusion utilisation any any**:

On souscrit au service quand on instancie un composant :

* nav.component🡪 lancement de la page
* member-edit.component🡪 lancement de la page d’édition

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