**How to build an Angular (v. 9.1.4) project with .Net Core 3.1 Web API**

0) For the API the ref. DatingApp.API):

(ref. in GitHub: <https://github.com/aleph0mc/DatingAppSol/tree/master/DatingApp.API>)

**Packages** used for DatingApp.API:

AutoMapper

AutoMapper.Extensions.Microsoft.DependencyInjection

CloudinaryDotNet

Microsoft.EntityFrameworkCore.Sqlite

Microsoft.EntityFrameworkCore

Microsoft.EntityFrameworkCore.Design

Microsoft.AspNetCore.Mvc.NewtonsoftJson

Microsoft.AspNetCore.Authentication.JwtBearer

Microsoft.EntityFrameworkCore.Proxies

Microsoft.AspNetCore.Identity.EntityFrameworkCore; **(when IdentityServer4 is used)**

From API to create DB model first:

- create e model (class) ex. Values

- create a data context

- use SQlite (SqL Serve would be the same) - import nuget lib

- add connection in appsettings.json

- add services in Startup.css

- in Package Manager Console go to API folder the command line:

**dotnet ef migrations add MigrationName** (to add a new migration)

**dotnet ef database update** (to create the database, in this case a SQLite DB, under the root folder).

To update DB Tools**: dotnet tool install --global dotnet-ef --version 3.1.5**

In case of issues on updating the tool remove the previous version using the command

**dotnet tool uninstall -g dotnet-ef**

and then run the installa gain.

To import some raw data in SQLite db, we can add some a Seed.cs under the data folder with a UserData.json containing the data, then we modify the Program.cs, the list is as follows:

. . . .

public static void Main(string[] args)

{

//CreateHostBuilder(args).Build().Run();

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

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

{

var services = scope.ServiceProvider;

try

{

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

//With this method no need to use "dotnet ef migrations"

//this command takes care of migration in case not applied beforehand.

context.Database.Migrate();

//Add dummy data

Seed.SeedUsers(context);

}

catch (System.Exception ex)

{

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

logger.LogError(ex, "An error occurred during migration");

throw;

}

host.Run();

}

}

public static IHostBuilder CreateHostBuilder(string[] args) =>

Host.CreateDefaultBuilder(args)

.ConfigureWebHostDefaults(webBuilder =>

{

webBuilder.UseStartup<Startup>();

});

Using VSCode as IDE it’s suggested to install the following extensions:

- c#

- c# Extensions

- NuGet Package Manager

- Angular Snippets (Version 9) **by John Papa**

- Angular Files

- Angular Language Service

- Auto Rename Tag

- Bracket Pair Colorizer2

- Debugger for Chrome (for Chrome users)

- Material Icon Theme

- Prettier – Code Formatter

- TSLint

- Angular2-Switcher

1) Create an Angular app: (ref. <https://cli.angular.io/>)

(ref. in GitHub: <https://github.com/aleph0mc/DatingAppSol/tree/master/DatingApp-SPA>)

npm install -g @angular/cli # Install Angular framework

ng new my-dream-app # Create an Angular app in the folder called my-dream-app (**IMPORTANT: do not use dots in names**)

cd my-dream-app # Go to folder

ng serve # start the application

TO UPDATE ANGULAR VERSION RUN

npm install @angular/platform-browser –save

ng update @angular/core @angular/cli  
ng update @angular/material **===> only if Angular Material is used**

2) package.json contains all the dependencies for the project

3) File **app.module.ts** contains @NgModule directive which is used to bootstrap all the angular components [AppComponent,......]

4) Every Angular Component is decorated with @Component directive, it is a TypeScript (or JavaScript) class but with Angular features:

Ex.

selector: 'app-root', //child selector

templateUrl: './app.component.html', //html template

styleUrls: ['./app.component.css'] //stylesheet included (if any)

Basically Component runs under a sort of MVC idea.

5) The **main.ts** file contains info for the Angular project:

platformBrowserDynamic because we use a browser to view data

6) The index.html contains the start page and contains the selector for app-root which is the first page shown for our application.

7) the file **angular.json** is the configuration for the Angular project, it has the directive that point to the index file and to main.ts:

"index": "src/index.html",

"main": "src/main.ts",

8) To add a new component we go inside the src/app folder (in VS2019 we can use the Package Manager Console) and run the command:

**ng generate component component-name**

(ex. ng generate component value, to create a component called 'value' - convention: all lower case) or it's possible to use an addon in Visual Studio 2019 from the menu "Add | Add New Item" and select Angular Component. The file "value.component.spec.ts" is just used for testing.

9) In app.module.ts the new component is added automatically in the module declaration, if not it must be added manually:

Ex.

....

**import { HttpClientModule } from '@angular/common/http';**

....

**import { ValueComponent } from './value/value.component';**

....

@NgModule({

declarations: [

AppComponent,

**ValueComponent**

],

imports: [

BrowserModule,

**HttpClientModule**

],

providers: [],

bootstrap: [AppComponent]

})

The imports section contain all the modules (libraries) that we use in our project (ex. we need to use the HttpClientModule to call the API, hence we need to import it).

10) In value.component.ts in order to use the HttpClientModule, we need to declare it in the constructor of the class:

....

import { HttpClient } from '@angular/common/http'; // It is usually performed automatically with intellisense, if not must be added manually.

....

export class ValueComponent implements OnInit {

constructor(private http: HttpClient) { }

....

The constructor is good to inject or initialize services (ex. http) but not good to perform calls, for that better use the method called ngOnInit.

11) The selector for the value component is app-value (check value.component.ts), now go to the app.component.html and replace the code with the following

<div>

<h1> Welcome to App</h1>

</div>

<br />

<app-value></app-value>

While in value.component.html we can use and \*ngFor a structural directive in Angular to see the values, the html code is as following

<p **\*ngFor="let val of values"**>

{{val.id}} ---- {{val.name}}

</p>

So the value component is a child component of app component.

12) The css frameworks to use are **Bootstrap** (<https://getbootstrap.com>) and **Font-Awesome,** under the SPA directory we run the command line to import them in our app:

**"npm install bootstrap font-awesome"**

We can even use the NuGet windows for the project.

Now Bootstrap relies on JQuery but we don't want it in our Angular project. There is a way to use Bootstrap relying on TypeScript, therefore we can check the file angular.json and the option marked with "styles", we can then add the Bootstrap css there but the order when the css are compiled is not kept by the compiler (we cannot control the cascade order), so better choose another way, we open the file src/styles.css and import the Bootstrap and Font-Awesome css:

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

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

13) For the layout we can use the Bootstrap Jumbotron and review it a little bit. We create a new component for navigation

**ng generate component nav**

the component has been automatically added in app.module.ts, if not, we need to add it manually.

14) The we copy the code for the Jumbotron in the example, just copy the Outer Html for the tag <nav…> and then paste the code in the nav.component.html, we need some adjustments. Then we need to add the nav component tag in the app.component.html:

<app-nav></app-nav>

....

Then we go in the nav.component.html and we change a little bit (see the code) to adapt to our needs.

15) We turn the form into an Angular form with to import the form module in app.module.ts

. . . .

**import { FormsModule } from '@angular/forms';**

. . . .

],

imports: [

. . . .,

**FormsModule**

],

Then we change the code in nav.component.html in the following way, the

<form **#loginForm="ngForm"** class="form-inline my-2 my-lg-0" **(ngSubmit)="login()">**

<input class="form-control mr-sm-2" type="text" name="username" placeholder="Username"

**required** **[(ngModel)]="model.username"**> 🡺 “**model**” must match the name defined in the nav.component.ts

<input class="form-control mr-sm-2" type="password" name="password" placeholder="Password"

**required** **[(ngModel)]="model.password"**>

<button class="btn btn-success my-2 my-sm-0" type="submit">Login</button>

</form>

The parts in bold are very important to identify a template form (different from the reactive form, see below).

16) Angular Services: a service is a mean to avoid duplication (ex. Retrieving data), it’s possible to create a service to be injected (@Inject decorator) in any component which requires that service, in the following way:

- create in app folder a new folder to hold all the services: \_services

- In \_services folder under the console we can run the following command:

**ng generate service service-name (ex.** ng generate service auth)

This will create a class called ‘AuthService’. Now we need to add this service in **app.module.ts** under the providers array:

. . . .

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

. . . .

],

providers: [

**AuthService**

],

. . . .

Then we inject the service in the constructor of our component ex.

constructor(private http: HttpClient) . . . .

Now HttpClient is used to call a service on the server and returns a response token, in the login case we need to call the login method in our API, basically the class auth.service.ts will contain a method similar to this

login(model: any) {

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

.pipe(

map((response: any) => {

const user = response;

if(user) {

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

}

})

);

}

Post method can accept a third parameter as a header but we don’t need it in this case. The response from the server is returned as an object.

17) Now in order to make use of the AuthService we need to inject it into our component in the similar way as the HttpClient, ex. in the nav.component.ts we have

. . . .

constructor(private authService: AuthService) { }

. . . .

then to use it

. . . .

login() {

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

console.log('logged in successfully');

}, error => { // error is optional

console.log(error);

}, () => { // complete is optional

console.log('serve call completed');

});

console.log(this.model);

}

. . . .

We can add another two simple methods to test the login and logout actions such as:

. . . .

loggedIn() {

const token = localStorage.getItem('token'); //The token is saved in the auth.service.ts

return !!token;

}

logout() {

localStorage.removeItem('token');

console.log('logged out');

}

. . . .

And in the nav.component.html we can use the \*ngIf Angular directive to show/hide the form or the dropdown in the following way:

. . . .

<!-- BOOTSTRAP DROPDOWN SHOWN WHEN LOGGED IN -->

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

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

Welcome User

</a>

<div class="dropdown-menu">

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

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

<a class="dropdown-item" href="#"><i class="fa fa-sign-out"></i> Logout</a>

</div>

</div>

<!---------------------------------------------------------------->

<!--SHOWN WHEN USER NEEDS TO LOG IN-->

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

<input class="form-control mr-sm-2" type="text" name="username" placeholder="Username"

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

<input class="form-control mr-sm-2" type="password" name="password" placeholder="Password"

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

<button [disabled]="!loginForm.valid" class="btn btn-success my-2 my-sm-0" type="submit">Login</button>

</form>

. . . .

18) Then we add a home component and a register component in src/app folder:

**ng generate component home**

**ng generate component register**

(if not added automatically, do not forget to add these components in app.module.ts)

Then we can add the home component in app.component.html, while the register component will be a child of the home component

App.component.html will look like this

<app-nav></app-nav>

<app-home></app-home>

. . . .

Now we add the register component as a child component into home component, the communication from child to parent is performed using an @Input directive in register.component.ts ex.

. . . .

**@Input() registerModeFromHome: boolean; //value from parent**

. . . .

In home.component.ts we can difene a variable called registerMode

. . . .

registerMode: boolean = false;

. . . .

And in home.component.html we have a tag to the register component with a directive in [] brackets

. . . .

<app-register **[registerModeFromHome]="registerMode"**></app-register>

. . . .

In bold the directive for Angular to pass the value to the child component.

Viceversa, when we need to pass a value from child to parent we need to use the directive into register.component.ts and the event inside the method

. . . .

**@Output() cancelRegister = new EventEmitter(); //value to parent**

. . . .

cancel() {

console.log('CANCELLED');

**this.cancelRegister.emit(false);**

}

. . . .

The emit(..) accept any parameter, in this case a Boolean.

While in the home.component.html to get the value from the child we need to add a directive in () brackets

. . . .

<app-register [registerModeFromHome]="registerMode" **(cancelRegister)="cancelregisterMode($event)"**></app-register>

. . . .

Then in [home.component.ts](http://home.component.ts) we need to create the method **cancelregisterMode($event)**

. . . .

cancelregisterMode(registerModeFromregister: boolean) {

this.registerMode = registerModeFromregister;

}

. . . .

Now we want to add the method when the register button is pressed in our application to send data to the API, and we add it as a service, therefore we add the register method in the auth.service.ts

. . . .

register(model: any) {

return this.http.post(this.baseUrl + 'register/', model);

}

. . . .

19) For the error management we need to create an error interceptor, so we add inside the **\_service** folder a new file called **error.interceptor.ts** which has to be injectable (@Injectable), in order to catch any error. The code is as follows:

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

import { HttpInterceptor, HttpErrorResponse, HTTP\_INTERCEPTORS } from '@angular/common/http';

import { catchError } from 'rxjs/operators';

import { throwError } from 'rxjs';

@Injectable()

export class ErrorInterceptor implements HttpInterceptor {

intercept(

req: import('@angular/common/http').HttpRequest<any>,

next: import('@angular/common/http').HttpHandler):

import('rxjs').Observable<import('@angular/common/http').HttpEvent<any>> {

return next.handle(req)

.pipe(

catchError(errorResponse => {

if (errorResponse.status === 401)

return throwError(errorResponse.statustext); // throw the error to the component

if (errorResponse instanceof HttpErrorResponse) {

const applicationError = errorResponse.headers.get('Application-Error'); //the name must match what is returned from the API

if (applicationError)

return throwError(applicationError);

const serverError = errorResponse.error;

let modalStateErrors = '';

//to deal with the modewl state

if (serverError.errors && typeof serverError.errors === 'object') {

for (const key in serverError.errors) {

if (serverError.errors[key]) {

modalStateErrors += serverError.errors[key] + '\n';

}

}

}

return throwError(modalStateErrors || serverError || 'Server Error');

}

})

);

}

}

//We need to export as a provider in order to catch the errors in our components

//This must be added in app.module.ts under the providers array

export const ErrorInterceptorProvider = {

provide: HTTP\_INTERCEPTORS, //we catch HTTP errors

useClass: ErrorInterceptor, //the class we created above

multi: true //to allow multiple type of interceptors

}

It is important to add the **ErrorInterceptorProvider** in the app.module.ts under the providers array in order to make it work.

. . . .

import { **ErrorInterceptorProvider** } from './\_services/error.interceptor';

. . . .

],

providers: [

AuthService,

**ErrorInterceptorProvider**

],

. . . .

20) In order to have some notifications we can use a third party library called Alertify (<https://alertifyjs.com/>), to do that we need to import the library, so we go into the SPA root folder and we run the following command line

**npm install alertifyjs**

This package comes with some css that we need to add to our styles.css

. . . .

@import url('../node\_modules/alertifyjs/build/css/alertify.min.css');

@import url('../node\_modules/alertifyjs/build/css/themes/bootstrap.min.css');

. . . .

Our main css is Bootstrap hence with Alertify we have to import the theme relative to Bootstrap.

Now to use the library we create a service as a wrapper, in the \_services folder we run the command line

**ng generate service alertify**

Now it can happen that there might be missing some definition files (.d.ts) for a given lib, a workaround to solve this issue is to create under the /scr folder a new typescript file called **typings.d.ts** where we can add a declaration similar to the following

**declare module 'alertify'**

Then we open the TypeScript config file called **tsconfig.json** (updated: **tsconf.app.json**) and add the node

. . . .

,

"typeRoots": [

**"src/typings.d.ts"**

]

. . . .

Or if the node already exists just add the file in bold.

Then we can create the methods in the wrapper class such as

. . . .

import \* as alertify from 'alertifyjs'; //imports the alertify library into typescript

. . . .

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

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

if (e) {

okCallback();

} else { }

});

}

message(message: string) {

alertify.message(message);

}

. . . .

To use the service it’s enough to add it in the constructor of a component, such as

. . . .

constructor(private authService: AuthService, **private alertify: AlertifyService**) { }

. . . .

And use it such as in ex.

. . . .

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

. . . .

In case there is a **warning** such as the following

**WARNING in …\alertify.service.ts depends on 'alertifyjs'. CommonJS or AMD dependencies can cause optimization bailouts.**

It is possible to remove it by adding a directive in the **angular.json** file as following

"build": {

"builder": "@angular-devkit/build-angular:[browser](https://angular.io/api/animations/browser)",

"options": {

"**allowedCommonJsDependencies**": [

"**alertifyjs**"

],

...

}

...

},

21) Let us now use a third party lib to manage the token that we use for authentication every time the user sends a request to the server in order to be identified, and we use the **@auth0/angular2-jwt** (<https://github.com/auth0/angular2-jwt>), in the root SPA folder using the command line we run

**npm install @auth0/angular-jwt**

Then we need to use this lib in **auth-service.ts** ex.

. . . .

loggedIn() {

const token = localStorage.getItem('item'); //Gets the token from the local storage

return !this.**jwtHelper.isTokenExpired**(token); //must return true if token has not yet expired

}

. . . .

Now ex. we can use the token to get the username rather than query the server for it is in the local storage, to do the we do in the following way in the nav.component.ts

. . . .

constructor(public authService: AuthService, . . . )

. . . .

In nav.component.html wehave (**titlecase** after the pipe makes the name to start with the capital letter)

. . . .

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

. . . .

That is why authService is declared as public in the constructor. The issue is that in this way when the page is refreshed the value is lost; hence, we need to find a way to keep the value and we need to change a bit the **app.component.ts**, the top component, and when first the application is loaded we can set our global variable there in the following way

. . . .

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

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

. . . .

export class AppComponent **implements OnInit {**

**//title = 'AngularAppDemoSPA';**

jwtHelper = new JwtHelperService()

constructor(**private authService: AuthService**) { }

**ngOnInit**(): void {

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

if (token) {

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

}

}

}

. . . .

22) To give our app a kick we can use **Ngx Bootstrap**, a third party lib to help **Bootstrap** to get rid of the required JQuery library, therefore **Ngx Bootstrap** will keep all Bootstrap components (ex. dropdown) just using **Angular** without importing JQuery and this is very important (<https://valor-software.com/ngx-bootstrap>). In SPA root folder we run the command line

**ng add ngx-bootstrap --component dropdowns**

Once imported we need to add the Ngx component (in this case a dropdown) inside the **app.module.ts file**

**. . . .**

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

. . . .

],

imports: [

. . . .

BrowserAnimationsModule,

**BsDropdownModule.forRoot**()

],

. . .

And in the nav.component.html we can use the component as dropdown

. . . .

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

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

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

</a>

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

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

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

<a class="dropdown-item" href="#"><i class="fa fa-sign-out"></i> Logout</a>

</div>

</div>

. . . .

To better set the look & feel check the na.componant.css file.

23) To add a bit of color we can use **Bootswatch** (<https://bootswatch.com/>) which provides some themes for **Bootstrap**, under the SPA root folder we run the command line

**npm install bootswatch**

after the impor styles.css must look like this

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

**@import url('../node\_modules/bootswatch/dist/cerulean/bootstrap.min.css');**

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

@import url('../node\_modules/alertifyjs/build/css/alertify.min.css');

@import url('../node\_modules/alertifyjs/build/css/themes/bootstrap.min.css');

In bold the Bootswatch css for the theme we chose.

24) We can now speak a bit of Angular routing, but first we create new components and under /src/app folder we run the command line

**ng generate component member-list**

**ng generate component lists**

**ng generate component messages**

As usual these components should be automatically added to **app.modules.ts**, if not they must be added manually.

Now under the **app** folder we create a new TypeScript file called **routes.ts** which can look like this

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

export const appRoutes: Routes = [ //The order in the list is important, the wildcard must be last

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

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

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

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

{ path: '\*\*', redirectTo: 'home', pathMatch: 'full' } //wildcard to redirect on wrong paths

];

Then we need to add this routing file to the **app.module.ts** in the import section where we specify the export constant created above

. . . .

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

import { appRoutes } from './routes';

. . . .

],

imports: [

. . . .,

**RouterModule.forRoot(appRoutes)**

],

. . . .

Then we can add router links to out html file, in this case **nav.component.html**, the snippet is as follows

. . . .

<a class="navbar-brand" **[routerLink]="['/home']"**>AngularDemoApp</a>

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

<li class="nav-item" **routerLinkActive="active"**>

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

</li>

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

<a class="nav-link" [routerLink]="['/lists']">Lists</a>

</li>

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

<a class="nav-link" [routerLink]="['/messages']">Messages</a>

</li>

. . . .

Basically we need to add a **routerLink** attribute to our tag where we specify the route (defined **routes.ts**) and to specify the css class as active we add another attribute **routerLinkActive="active"**, the class “active” matches the name defined in Bootstrap in this case.

Finally in order to use these routes we need to go to the file **app.componant.html** and add a router outlet, basically we now have

<app-nav></app-nav>

<app-home></app-home>

And we need to replace the **app-home** tag with a **router-outlet** tag, the result is

<app-nav></app-nav>

**<router-outlet></router-outlet>**

Now we need to fix the navigation as when the user logs out must be redirected to the login screen, in addition the user can still access the paths such as members, lists, etc. which must be forbidden, therefore we need to add some more changes in order to make the navigation work properly.

First, in out **nav.component.ts** we use an Angular service called **Router** that we initialize in the constructor

. . . .

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

. . . .

constructor(. . . ., **private router: Router**) { }

. . . .

and in the login and logout methods we have

. . . .

login() {

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

. . . .

//we could have redirect the user here but we can use the complete even as well to redirect

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

}, error => { // error is optional

. . . .

}, () => { // complete is optional, but we use this to redirect to the members page after the login

**this.router.navigate(['/members']);** //redirects the user to the members page

});

. . . .

}

. . . .

logout() {

. . . .

**this.router.navigate(['/home']);** //redirects the user to the home page

}

. . . .

Secondly, we need to fix the issue with the paths as the users can still navigate to the members page or others when not logged in (ex. <http://localhost:4200/lists>) and to do so we need to introduce the Angular Guard, in order to protect the pages that require an authentication. Inside scr/app we create a new folder called \_guards and insiede this folder we run the command line

**ng generate guard auth --skipTests** (or **ng g guard auth --skipTests** ( last parameter is to avoid to generate the test file)

Then we change the **auth.guard.ts** to get the following

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

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

import { Observable } from 'rxjs';

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

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

@Injectable({

providedIn: 'root'

})

export class AuthGuard implements CanActivate {

constructor(private authService: AuthService, private router: Router, private alertify: AlertifyService) { }

//canActivate(

// next: ActivatedRouteSnapshot,

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

// return true;

//}

//In our case we just need this information

canActivate(): boolean {

if (this.authService.loggedIn()) //if loggedin ok go ahead

return true;

//otherwise show a message and redirect to home page

this.alertify.error('You shall not pass!!!!');

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

return false;

}

}

Now we need to add this **guard** to the **routes.ts** file for the component we want to protect

. . . .

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

. . . .

similar for others we want to control.

This way can be a tedious for it requires to apply the guard to each route for our app, a better way is to protect multiple routes with a single guard, using dummy routes. Therefore we change **routes.ts** in the following way

. . . .

export const appRoutes: Routes = [ //The order in the list is important, the wildcard must be last

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

{

path: '', //This path is added to the following, if it was ex. 'dummy' the members path would be: ...4200/dummymembers

runGuardsAndResolvers: 'always',

canActivate: [AuthGuard],

children: [

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

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

{ path: 'lists', component: ListsComponent }

]

},

{ path: '\*\*', redirectTo: 'home', pathMatch: 'full' } //wildcard to redirect on wrong paths

];

. . . .

The issue we still have it that we do not have any route set for the path <http://localhost:4200> there we need to adjust a bit the previous in the following way, before the home path we can add an empty path to HomeComponent and we set the empty path in the wildcard path

. . . .

{ path: '', component: HomeComponent },

. . . .

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

. . . .

We’ll see further down how to use resolvers to get data to pass to querystring.

25) In our app we can generate various interfaces to specify our models, so we create a new folder called **\_models** under src\app then inside that folder we create some interfaces running the following command line

**ng g interface user** (g stands for generate)

**ng g interface photo**

**ng g interface pagination**

This creates the user.ts, photo.ts and pagination.ts files. Then check the code in GitHub for those interfaces.

Then we create a service to get users from API, for the code check the GitHub repository. Under \_services folder, we run

**ng g service user**

**IMPORTANT: Remember to add this service to app.module.ts under providers**

Now in order to have our API url globally we can save it in the file **environment.ts** under src\environments folder, we have

. . . .

export const environment = {

. . . .,

**apiUrl: 'http://localhost:50800/api/'**

};

. . . .

The API address can be different of course. When we want to use it we can just set

. . . .

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

. . . .

**apiUrl = environment.apiUrl;**

. . . .

There is a production environment, **environment.prod.ts**, as well but we are not interests in that for now. We can update the path even in **auth.service.ts**, and rather than baseUrl we rename it as apiUrl

. . . .

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

. . . .

apiUrl = **environment.apiUrl** + 'auth/'; // was baseUrl = 'http://localhost:50800/api/auth/'

. . . .

To make the things faster we add the code of the **user.service.ts** below (**IMPORTANT: this file will be changed further down**)



Then we can use this service inside the **member-list.component.ts** in the following way

. . . .

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

. . . .

constructor(private userService: UserService . . . ) { }

. . . .

loadUsers() {

this.userService.getUsers().subscribe(users => {

this.users = users;

}, error => {

this.aletify.error(error);

})

}

. . . .

And call it

. . . .

ngOnInit(): void {

this.loadUsers();

}

. . . .

And changing a little bit **member-list.component.ts** we can show some user’s data

<div class="container">

<div class="row">

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

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

</div>

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

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

</div>

</div>

</div>

26) We saw above (**user.service.ts**) the importance of the JWT token to be send to the server in order to pass tha authentication, that way is a bit tedious for it has to replicated in each component where an authorization must be sent to the server. We provide now a way to do that automatically.

We need to change the app.module.ts adding the JwtModule

. . . .

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

. . . .

//This function is used to send the JWT token automatically without using httpHeaders

export function tokenGetter() {

return localStorage.getItem('token');

}

. . . .

],

imports: [

. . . .,

**JwtModule.forRoot({**

**config: {**

**tokenGetter: tokenGetter, //The function defined above**

**whitelistedDomains: ['localhost:50800'], //Our domain**

**blacklistedRoutes: ['localhost:50800/api/auth'], //these are the routes where the token is not required**

**}**

})

],

. . . .

Now in the **user.service.ts** we can comment out the part relative to the headers for the JWT token for it is now sent automatically according with the configuration in **app.module.ts**, therefore

. . . .

//const httpOptions = {

// headers: new HttpHeaders({

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

// })

//}

. . . .

And the methods getUsers and getUser can become

. . . .

getUsers(): Observable<User[]> { //returns an array of User

**return this.http.get<User[]>(this.apiUrl + 'users');** //The observable returned by the get method is an array of User

}

getUser(id): Observable<User> {

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

}

. . . .

No need anymore HttpHeaders.

27) We can observe now that in order to get the user we need the **id** anytime we call the method getUser, but we can provide a different and better way using **route resolvers**.

We can create a resolver for the member-detail component, under src\app we create a new folder **\_resolvers** and inside a file called **member-detail.resolver.ts**, a resolver must be injectable (@Injactable) and the code is as follow



Then we need to add the route resolver to the **app.component.ts** in the providers array

. . . .

import { MemberDetailResolver } from './\_resolvers/member-detail.resolver';

. . . .

],

providers: [

. . . .,

**MemberDetailResolver**

],

. . . .

and to **route.ts**

. . . .

children: [

. . . .

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

. . . .,

]

. . . .

And in the member-detail.component.ts we can use the resolver

. . . .

ngOnInit(): void {

//rather than this

//this.loadUser();

//we can use this

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

**this.user = data['user'];** **//name in data['...'] must match the name given in routes.ts for the 'resolve' attribute**

});

. . . .

28) In a reactive form (see below) we can add some in order to prevent to change the page when the form is dirty. We then can create a new guard. In the folder **\_guards** under src\app we can run the following command line

**ng g guard prevent-unsaved-changes**

(**IMPORTANT**: make sure this **guard** is added in the **providers** list under **app.modules.ts** file)

this will generate a new file called **prevent-unsaved-changes.guard.ts**, the event we can consider is called **canDeactivate** and the code can be as follows



All the possible cases of guards can be described the following template-code



Next, we need to add the guard to the **routes.ts** file

. . . .

{

path: 'members/edit', component: MemberEditComponent, resolve: { user: MemberEditResolver },

**canDeactivate: [PreventUnsavedChangesGuard]**

},

. . . .

Another check is when a user tries to close the form window on the browser using the **HostListener**, when the form is dirty then a message can be shown to warn the user for unsaved changes, in member-edit.component.ts the code to add is as follows

. . . .

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

. . . .

**@HostListener**('window:beforeunload', ['$event']) //access browser events

unloadNotification($event: any) {

if (**this.editForm.dirty**) {

$event.returnValue = true;

}

}

. . . .

29) There two types of angular forms, **template forms** and **reactive forms**.

**a)** Template Forms:

Ex. **nav.component.html**

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

<input class="form-control mr-sm-2" type="text" **name="username"** placeholder="Username" required **[(ngModel)]="model.username"**>

<input class="form-control mr-sm-2" type="password" **name="password"** placeholder="Password" required **[(ngModel)]="model.password"**>

<button **[disabled]="!loginForm.valid"** class="btn btn-success my-2 my-sm-0" type="submit">Login</button>

</form>

The model is defined in **nav.component.ts** and the name model in the HTML must be the same defined in the class

. . . .

**model: any = {};**

. . . .

**login()** {

console.log(this.model);

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

//console.log('Logged in successfully');

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

}, error => { //erros

//console.log(error);

this.alertify.error(error);

}, () => { //complete

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

});

}

. . . .

**b)** Reactive Forms:

Ex. **register.compomnent.html**



And the **register.component.ts**



30) Any to any component communication using **BehaviorSubject** which is a type of **Observable**.

We saw above parent-child and child-parent communication, in order to send some data from one component to another. Now we can add a service to update the photo in the **navbar** when the user updates photo in the member edit page.

In **auth.service.ts** we add the following code

. . . .

import { BehaviorSubject } from 'rxjs'; //USED FOR COMMUNICATION IN ANY TO ANY COMPONENT

. . . .

photoUrl = new BehaviorSubject<string>('../../assets/user.png'); //AN INITIAL VALUE IS SPECIFIED

currentPhotoUrl = this.photoUrl.asObservable();

. . . .

changeMemberPhoto(photoUrl: string) {

this.photoUrl.next(photoUrl); //This is the new photoUrl

}

. . . .

login(model: any) {

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

.pipe(

map((response: any) => {

const user = response;

if (user) {

. . . .

**this.changeMemberPhoto(this.currentUser.photoUrl);** //This updates the photoUrl

}

})

)

}

. . . .

Now, we need to subscribe to this service where we want apply the changes, in this case the new photoUrl, specifically in our context in the navbar, therefore in **nav.component.ts** we have the following code

. . . .

**photoUrl: string;**

. . . .

ngOnInit(): void {

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

}

. . . .

And in the template **nav.componemnt.html** we have to use the **photoUrl** variable define above in the image tag

. . . .

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

. . . .

The pipe (|) means that when the **photoUrl** is null the we can use the default photo from the assets folder.

Then we need to update the **app.component.ts** so that when the application is loaded the **photoUrl** is updated with the correct one

. . . .

ngOnInit(): void {

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

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

if (user) {

this.authService.currentUser = user;

**this.authService.changeMemberPhoto(user.photoUrl); //Updates the current photoUrl...**

}

. . . .

Finally we need to subscribe to that service in the **member-edit.component.ts** in order to set the correct photoUrl

. . . .

**photoUrl: string;**

. . .

ngOnInit(): void {

. . . .

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

}

. . . .

And in the template **member-edit.component.html** we need to update the image tag

. . . .

**<img class="card-img-top img-thumbnail" src="{{photoUrl || '../../../assets/user.png'}}" alt="{{user.knownAs}}" />**

. . . .

Basically the idea is that wherever we update the photoUrl we can use

. . . .

**this.authService.changeMemberPhoto(user.photoUrl)**

. . . .

So that each component that subscribes to that service will be uodated.

And to update the localStorage: we first set the new photoUrl for the current user then we update the localStorage

. . .

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

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

. . . .

30) Sometimes we need to reference a template element and we can use the hashtag. Ex in **member-detail.component.html** considering

. . . .

<tabset class="member-tabset" **#memberTabs**>

. . . .

Then we can reference that template element in member-detail.component.ts using the ViewChild

. . .

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

. . . .

Then we can add a method to change the tab programmatically

  selectTab(tabId: number) {

    this.memberTabs.tabs[tabId].active = true;

  }

Finally we can add the click event on the button we want to perform the change tab action (**member-detail.component.html**) to select the Messages tab, index starts from 0, therefore the tab number is 3.

<button class="btn btn-success w-100**" (click)="selectTab(3)">**Message</button>

31) It is possible to pass query parameters in a link in the following way

<button class="btn btn-primary" [routerLink]="['/members/', userToImport.id]"

          [queryParams]="{tab: 3}"><i class="fa fa-envelope"></i></button>

Then we can retrieve the value of the param in the **ngOnInit** method of the **xxxxx**.**component.ts** using some code similar to the following

    this.route.**queryParams**.subscribe((params) => {

      const selectedTab = params['tab'];

      this.memberTabs.tabs[selectedTab > 0 ? selectedTab : 0].active = true;

    });

32) When there is a click event on row table

<tr \*ngFor="let message of messages" [routerLink]="['/members',

          messageContainer == 'Outbox' ? message.recipientId : message.senderId]" [queryParams]="{tab: 3}">

and inside the row there is a button

<button class="btn btn-danger" (click)="deleteMesage(message?.id, $event)">Delete</button>

Then in order to propagate the click on the row as well there are two options:

a) Add another click event on the button

<button class="btn btn-danger" (click)= "$event.stopPropagation" (click)="deleteMesage(message?.id)">Delete</button>

b) Specify the event on the button

<button class="btn btn-danger" (click)="deleteMesage(message?.id, $event)">Delete</button>

And the pass the $event to the method in **xxxxx.component.ts**

  deleteMesage(id: number, **event: any**) {

**event.stopPropagation();** // avoid to propagate the click to the tr tag (see html)

. . . .

}

33) Tap operator (**member-messages.component.ts**)

. . . .

      .pipe( // allows to perform some ops before actually calling the method in subscribe

        tap((messages: Message[]) => { // tap operator from RxJS allows to perform the required ops before subscribe

          . . . .

Code goes here

        })

      )

      .subscribe(. . . .);

32) Deploy Angular

In **angular.json** we can choose the output path for the **ng build** the commend which allows to build our Angular solution, in particular we want to change the output path, therefore

. . . .

      "prefix": "app",

      "architect": {

        "build": {

          "builder": "@angular-devkit/build-angular:browser",

          "options": {

**"outputPath": "dist/DatingApp-SPA",**

            "index": "src/index.html",

            "main": "src/main.ts",

            "polyfills": "src/polyfills.ts",

            "tsConfig": "tsconfig.app.json",

. . . .

The current value is **dist/DatingApp-SPA**, but we want to change it to **../DatingApp.API/wwwroot**

      "architect": {

        "build": {

          "builder": "@angular-devkit/build-angular:browser",

          "options": {

**"outputPath": "../DatingApp.API/wwwroot",**

Then from the command line inside the SPA root folder we can run

**ng build --prod**

to build our optimized solution.

IMPORTANT: do not forget to update the **environment.prod.ts** with value that are in **environment.ts** file.

Now we need to inform the API to deliver those staic files inside wwwroor and to do that, we need to apply a small change in **startup.cs** file, under Configuree method

. . . .

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

//Here the order is important

app.UseRouting();

//For dev purposes - temporary all are allowed

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

//Following two lines required for production only

//to run Angular under Kestrel server /DatingApp.API/wwwroot

**app.UseDefaultFiles();**

**app.UseStaticFiles();**

**. . . . .**

**App.UseEndpoints(. . . .);**

}

. . . .

At this point, under this basic configuration there is an issue with the routes, for the API doesn’t know anything about Angular routing, therefore we need to introduce some additional changes in order to make the API routing working in sync with Angular routing.

Therefor we need to add a new controller, so in the controller folder we add a new MVC controller and we can write the following code



Finally we need to take care of this new controller in startup.cs, adding a new endpoint in the following way

. . .

app.UseEndpoints(endpoints =>

{

endpoints.MapControllers();

**endpoints.MapFallbackToController("Index", "Fallback");**

});

. . .

Now everything should be working fine, the only issue is the size of the files downloaded from the server which are still quite big, as they continue to use JIT (just in time) compilation, so we need to perform some optimization in order to drastically reduce their size.

It comes to our help the concept of AOT (ahead of time) compilation, meaning that files are precompiled before being delivered to the client, which can be performed adding an option to the command **ng build** we just saw above, in this case we have to change some files, the **environment.prod.ts** (under the **environments** folder) in the following way, specifying the apiUrl

export const environment = {

  production: true,

**apiUrl: 'api/**'

};

In **angular.json** we can find some parameters to optimize the build as follows

              "optimization": true,

              "outputHashing": "all",

              . . . . . . .

              "vendorChunk": false,

              "buildOptimizer": true,

The **buildOptimizer** option thought when set to true might affect some css (as it happens with Alertify library) therefore in some cases there might be a need to set it to ***false*** with the result of having the file size a big larger, but that in general in not a big deal.

Now we can manage the migration using different databases, let us consider SqlServer, the steps to follow are

1) Add the database provider [Microsoft.EntityFrameworkCore.SqlServer](https://www.nuget.org/packages/Microsoft.EntityFrameworkCore.SqlServer) via NuGet

2) configure **appsettings** for development (SQLite) and production (SqlServer) added the desired connections strings

Dev

. . . .

"Connectionstrings": {

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

},

. . . .

Prod

. . . .

"Connectionstrings": {

"DefaultConnection": "Server=ASUSMLK\\SS2017DEV; Database=datingapp; User Id=sa; Password=sa"

},

. . . .

Then in **Startup.cs** we need to add a configuration method based on convention in the following way

. . . .

// The following 2 methods are convension based for .Net Core,

// therefore it is important to keep the names exactly as they are

public void **ConfigureDevelopmentServices**(IServiceCollection services)

{

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

ConfigureServices(services);

}

public void **ConfigureProductionServices**(IServiceCollection services)

{

services.AddDbContext<DataContext>(x => x.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

ConfigureServices(services);

}

// This method gets called by the runtime. Use this method to add services to the container.

public void ConfigureServices(IServiceCollection services)

{

. . . . . . .

}

. . . .

Finally there might be an issue with the **Annotation** in the migration files, therefore we might need to add an annotation for SqlServer in each migration file in the following way

. . . .

protected override void Up(MigrationBuilder migrationBuilder)

{

migrationBuilder.CreateTable(

name: "Values",

columns: table => new

{

Id = table.Column<int>(nullable: false)

**.Annotation("SqlServer:ValueGenerationStrategy", SqlServerValueGenerationStrategy.IdentityColumn)**

**.Annotation("Sqlite:Autoincrement", true)**,

. . . . . .

},

. . . . . . .

);

}

. . . .

Another way to perform a migration to another DB is just to remove or rename the **migrations** folder then perform the database migration already shown at the beginning of this document.

33) Lazy loading can be useful when we prefer to let DB engine to manage the Include in LINQ queries, we need to add a package **Microsoft.EntityFrameworkCore.Proxies** then we need to perform the following changes in our API project, in Startup.cs under the method ConfigureDevelopmentServices and ConfigureProductionServices we need to specify

public void Configure. . . . Services(IServiceCollection services)

{

services.AddDbContext<DataContext>(x =>

{

x.UseLazyLoadingProxies();

x. . . .(Configuration.GetConnectionString("DefaultConnection"));

});

ConfigureServices(services);

}

Then in the DatingRepository we can remove all the **.Include(…)** in LINQ queries, finally in our models where there is reference to a foreign entity (navigation property) we need to make it **virtual**, ex. In the **User.cs**

. . . .

public **virtual** ICollection<Photo> Photos { get; set; }

. . . .

and so forth for the others navigation properties.

34) Before publishing we need to configure Azure.

i) Login Azure portal and then select **Resource Group**



Then press Add to create a new Resource Group



Then press the Review & Create button.

After checking data is ok



press the Create button at the bottom of the screen



Then in the Notification area press Go To Resource Group



ii) At this point we need to creare the resources. We can press the button **Create Resources** which will bring to the Market place where we can look for an offer that can allow us to have a WebApp, an AppServicve and a Database, therefeore we can click **Database** on the right menu, we can select the option Web App + SQL



Then press the **Create** button.

iii) We have to fill the data



Important to notice it that the **App name** must be unique in **.azurewebsites.net** therefore you might need to choose a different name.

iv) On the same screen we need to create an App Service plan which is how much money to spend on the application, of course we don’t want to spend anything then we need to create a new service plan for free, therefore we click on the service plan and then we create a new one



We have



Then we click on the pricing tear and we select a free plan (under Dev/Test).

v) Next, we need to configure the database, we click on Sql Database to configure it



For the pricing tear we can find an offer for free going to basic option and 

Now most of the config is in place



Now we can press the Create button at the bottom of the screen and wait some minutes until the resources have been created. The resource group finally created should look like the following



vi) To finish off our config we can click on the icon for App Service and check if Url and other config is ok. The address for the web app should be something like <https://dateservice.azurewebsites.net/> then we can click on the Configuration



If we click on the **defaultConnection** we can check the DB connection string, in this place we can store our keys as the app can read the config here. For the sake of completeness we capitalize the name for defaultConnection to meka it the same as defined in our code, so DefaultConnetcion



Now we can press **OK** button and then press the **Save** button to confirm our changes.

vii) We can go back to the Resource Group and then we click on the SQL server icon as we need to set up the firewall settings then click on Show firewall settings



Then we have



Then we need to copy the Client Ip Address in the Rule form as shown above, this is required to authorize our client to access the database. Then click **Save** button. At this point we can even manage the DB from our local machine or go to the **Query Editor** in Azure to check what is in our DB. Finally, as we deploy on Azure and we have already defined a connection string there, we can remove the ConnectionStrings attribute in **appsettings.json**.

~~"Connectionstrings": {~~

~~"DefaultConnection": "Server=???????;Database=datingapp;User Id=?????;Password=?????;MultipleActiveResultSets=true"~~

~~},~~

That’s all, now we’re good to go deploying our app on Azure.

34) **Paging**

We can add paging and passa parameters in a querystring …**?pageNumber=18&pageSize=5** in the following way:

* **Backend** (Web API)
  + We first create a class called **PagedList** and we can save it in the **Helpers** folder



* + Then we need to add a pagination header, then we create a class called **PaginetionHeader** which is used to send pagination info to client, in the following way



* + Then, if not already present, we can create an **Extentions** class to append the pagination to the response, in the following way  
    
  + Then we create a class to pass the parameters for pagination, in this case to pass messages  
    
  + In the repository then we need to pass the new parameter for pagination, therefore something like  
    
  + Then in the controller  
    
* **Frontend** (Angular)
  + We create an interface inside our **\_models** folder that we can call **pagination**, in the following way
  + Then the function to get the messages must be as follows  
    
  + Then in the resolver, if any, we have  
    
  + We need to add the pagination component to the message list, in order to allow user to perform pagination, use an ngx-bootstrap pagination component. We add the import the component in **app.modules.ts**

. . . .

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

. . . .

  imports: [

    . . . .

    PaginationModule.forRoot(),

. . . .

  ],

* + Then we need to add this component inside the html page where the message list is shown  
    
  + We need to manage the parameters inside the component typescript file, therefore we add  
    . . .   
     pagination: Pagination;  
    . . . .
* this.route.data.subscribe((data) => {
* this.messages = data.messages;
* this.pagination = data.pagination;
* });

. . .

 pageChanged(event: any): void {

    this.pagination.currentPage = event.page;

    this.loadMessages();

  }

.Finally in the component we need a method to get the messages as follows  
