# Login pt.1 – Criando página para entrar

Nessa primeira parte é criado os componentes para efetuar o login. Criamos o sign-in dentro de uma pasta chamada **home.** Dentro da pasta home é criada o **modulo** para que possa importar em outras partes do projeto.

Cria-se a estrutura **html** de login. Faz a rota no **routing** para que seja o primeiro componente a ser carregado e depois faz o importe do modulo de Home no **app**.**module**.**ts.**

{

    path: '',

    component: SignInComponent

  },

Import em app.module.ts -> HomeModule,

# Login pt.2 – Validação de formulário

Primeiro é necessário importar no **modulo** do **home**(responsável por declarar **SignInComponent**) o **ReactiveFormsModule** e importar no **ngModule**.

imports:[

    ReactiveFormsModule,

CommonModule

Criar uma propriedade no TS do **Sign**-**in** do tipo **FormGroup** (que faz parte do ReactiveFormsModule) e faz um **data**-**binding** na tag do form com a propriedade [**formGroup**] que já faz parte da tag html e fazer a ligação com a propriedade criada no TS.

loginForm: FormGroup;

<form [formGroup]="loginForm" .... >

Agora é utilizado no construtor de formulário o **FormBuilder** sendo passado como injeção. Feito isso, implementamos o **OnInit** e criaremos o form.

Dizemos que o **loginForm** recebe o **formBuild** com o método **group.** Ele devolve um formGroup e passamos um objeto javascript com as propriedades **userName** e **password**. As propriedades passadas no group precisam ser associadas com o input de username e senha utilizando o **formControlName**. No array que elas recebem é passado um validador.

constructor(private formBuilder: FormBuilder,

ngOnInit(): void {

    this.loginForm = this.formBuilder.group({

      userName: ['', Validators.required],

      password: ['', Validators.required],

    });

  }

HTML

<input formControlName="userName" >

<input formControlName="password" >

Nessa parte do projeto utilizamos um componente criado em shared/componentes que traz uma mensagem caso os campos inputs não tenham sido preenchidos. Criamos também um modulo pra essa pasta de mensagem declarando o componente da VMessageComponent e também exportando. O modulo de mensagem é importado onde o componente será utilizado. No seu TS passamos um @**Input** para a propriedade que receberá um valor de fora.

<small class="text-danger d-block mt-2">{{text}}</small>

@Input() text: string = '';

Ele é importado logo abaixo do input de username e password recebendo seu valor através do **text**.

Para esse componente importado passamos um \***ngIf** para que possa ocultar a mensagem quando tiver digitado algum valor no campo input.

<appvmessage text="--" \*ngIf="loginForm.get('password').errors?.required"

Pegando o campo password e se tiver erro aplicar o required

Depois disso é necessário apenas fazer um **data**-**binding** com a propriedade **disable** para o **button** que recebe **loginForm**.**invalid**. Se tiver pelo menos um input invalido ele não deixa clicar.

<button [disabled]="loginForm.invalid" class="btn btn-primary btn-

block">Login</button>

# Login pt.3 - Enviando dados para login

Criando uma pasta e module Core em app e tudo que diz respeito a autenticação, filtros e etc colocar dentro do Core. É criado um **Service** dentro da pasta **Auth**. Para realizar requisições async é necessário do **HttpClient.** Ele é importado no construtor do **auth.service.**

Cria-se um método de autenticação que recebe como parâmetro userName e password (**os nomes das propriedades precisam ser as mesmas tanto para inputs de login e de service**). É criado também a const da api que fica concatenada com user/login e como é feita uma requisição do tipo **post** é necessário passar dois parâmetros com o mesmo userName e password.

constructor(private http: HttpClient...){}

  authenticate(userName: string, password: string){

    return this.http.post(API\_URL + '/user/login', {userName: userName, password:

password})

Agora no TS do componente que efetua o login importar o **service** que tem o método authenticate. Logo depois de NgOnInit criar um método de login que vai chamar o método do service de autenticação. Esse método login precisa ser chamado quando submeter o formulário e para isso é criado um binding de evento com o nome **submit** que recebe o método login.

constructor(private formBuilder: FormBuilder, private authService: AuthService

login(){

}

<form [formGroup]="loginForm" class="form mt-4" (submit)="login()">

Agora o método login precisa pegar o valor do input de userName e de password que são passados pra uma const. Da pra pegar usando a propriedade criada **loginForm** com o método get passando o name definido pelo **formControlName** e acessando seu valor.

const userName = this.loginForm.get('userName').value;

const password = this.loginForm.get('password').value;

Feito isso vamos utilizar o método do service injetado no construtor chamando-o e passando como parâmetro as duas const que retornam userName e password. Esse método devolve um observable e ai efetuamos o subscribe que recebe um call-back de sucesso e outro caso não consiga se autenticar. O segundo call-back mostra no console o erro e também pega a propriedade **loginForm** e aplica um reset.

login(){

    const userName = this.loginForm.get('userName').value;

    const password = this.loginForm.get('password').value;

    this.authService.authenticate(userName, password)

.subscribe(() => console.log('autenticado'),

      err => {

        console.log(err);

this.loginForm.reset();

      }

    )

  }

# Login pt.4 - Redirecionando para outra página

Para auxiliar a fazer a navegação depois do login é necessário importar o Router no construtor no TS responsável pelo login. Como boa prática e para ficar explicito de qual módulos o componente de login precisa para trabalhar, importar dentro do modulo responsável pelo componente o **AppRoutingModule**.

Agora logo após efetuar o subscribe no método utilizar o router para navegar para outra página. Utilizar o método **navigate** de **Router** passando uma estrutura em array onde o primeiro parâmetro é o endereço que quer ir e o segundo é o seguimento de rota que quer colocar nele.

constructor( … … …, route: Router) {}

login(){

    const userName = this.loginForm.get('userName').value;

    const password = this.loginForm.get('password').value;

    this.authService.authenticate(userName, password)

.subscribe(() => this.route.navigate(['user', userName]),

err => {

console.log(err);

this.loginForm.reset();

      }

    )

  }

# Login pt.5 – Utilizando o ViewChild para focar assim que errar login

Criaremos a propriedade do tipo **ElementRef** que é o tipo para manipular o DOM especificando ainda mais o tipo de elemento com **HTMLInputElement**. Usamos também o decoretor @**ViewChild** para que pegue o input do username. Passamos um identificador para que saiba quem é aplicando uma variável # de template no input.

@ViewChild('userNameInput') userNameInput: ElementRef<HTMLInputElement>;

<input formControlName="userName" type="text" class="form-

control" placeholder="user name" autofocus #userNameInput>

Feito isso aplicar no segundo call-back que a propriedade **userNameInput** utiliza o **nativeElement** para ter acesso ao DOM com o método **focus**(). Aplicar depois do loginForm.reset().

this.userNameInput.nativeElement.focus();

# Login pt.6 – Acessando o nome do usuário logado/Extraindo token

Quando o usuário efetua o login com sucesso ele recebe uma resposta, a web é assim. Nessa resposta vem no cabeçalho dela um (pode ser qualquer nome) **x**-**access**-**token** com um token. Se autenticar tem um token, se não tiver, não tem.

Quando efetuar o login precisa extrair essa informação do cabeçalho em x-access-token e guardar na aplicação.

Importar o operador **tap** do **rxjs**/**operators**. No método de autenticação da pasta **auth** depois do **post,** **aplicar** o **pipe.** Entre a execução da operação e o subscribe esse pipe vai executar um código arbitrário. O pipe permite colocar aplicações que quem fizer o subscribe essas operações vão ser aplicadas antes.

Faz o pipe, coloca o operador **tap** que vai ter uma resposta (**res**) e ela vai ser encaminhada para quem for fazer o subscribe. É criado um const **authToken**  que recebe **res.headers.get** com o nome do cabeçalho que quer e no nosso caso é o **x**-**access**-**token**.

import { tap } from 'rxjs/operators';

authenticate(userName: string, password: string){

    return this.http

      .post(API\_URL + '/user/login', {userName: userName, password: password,

{ observe: 'response' })

      .pipe(tap(res => {

        const authToken = res.headers.get('x-access-token');

console.log(authToken);

      }))

  }

Depois disso é necessário para acessar o **headers** na resposta é preciso passar a configuração para expor os headers e depois manipular. É passado um terceiro parâmetro no **post** um observe: ‘response’. Vai ter acesso ao cabeçalho e a tudo que tem direito da resposta.

# Login pt.7 – Armazenando token

Dentro de Core é criado a pasta Token com o service responsável por toda a lógica. Service é sempre um injectable.

Nesse service ele vai ter um método de **hasToken** que vai retornar verdadeiro ou falso se o token já existe, outro método com o nome **setToken** que recebe um parâmetro para guardar o token, outro para retornar o token **getToken,** e por último o **removeToken** para remover do localstorage.

No return do **hasToken** é passado dois exclamação para que ele retorne verdadeiro ou falso. Antes retornava string agora retorna um bool.

\*se for null o primeiro exclamação troca pra true e o segundo para false.

\*se for string que existe o primeiro troca para false e o segundo para true

const KEY = ‘authToken’;

hasToken(){

    return !!this.getToken();

  }

  setToken(token){

    window.localStorage.setItem(KEY, token)

  }

  getToken(){

    return window.localStorage.getItem(KEY);

  }

  removeToken(){

    window.localStorage.removeItem(KEY);

  }

No constructor do service de autenticação é injetado o service de token e é utilizado para receber a const **authToken**.

constructor(private http: HttpClient, private tokenService: UserService)

authenticate(userName: string, password: string){

     ..

..

        const authToken = res.headers.get('x-access-token');

        this.tokenService.setToken(authToken);

        console.log(authToken);

      }))

  }

# Login pt.8 – Colocando componente de cabeçalho na página

Criaremos um componente dentro de Core na pasta header, depois criar toda estrutura html do cabeçalho com o nome de usuário. Dentro do modulo do Core declarar e exportar esse componente para que possa ser utilizado fora. Como o header vai ser visível na aplicação toda ele é importado no componente principal, app-component. Fazer um hack para afastar o header dos outros conteúdos com um style.

# Login pt.9 – Exibindo os dados do usuário

Nesse momento é necessário pegar a informação do usuário logado que ficou guardada no localStorage e agora precisamos descriptografar o token, pegar o nome do usuário e exibir no componente header.

Para fazer isso utilizamos a ajuda do jwt-decode instalando no nosso projeto. Npm install jwt-decode@2.2.0.

Feito isso vai ser criado um serviço chamado **userService** e vai ter como finalidade **armazenar** o **token** com o **auxilio do tokenService** e retornar o usuário logado. É criado dentro de Core uma pasta **user** com o **service**.

No construtor do userService injetar o tokenService que vai servir de auxilio.

constructor(private tokenService: TokenService) {]

É criado os métodos **setToken** com o parâmetro do tipo string para receber o token e no escopo é chamado o **tokenService** para utilizar o método **setToken** que vai receber por parâmetro o token do user.

setToken(token: string){

    this.tokenService.setToken(token);

  }

Criaremos o método getUser.

Agora é necessário trocar no service de autenticação o AuthService que agora no construtor ele não depende mais de **tokenService** e sim de **userService** e na chamada trocar também para userService.

constructor(private http: HttpClient, private userService: UserService) { }

authenticate(userName: string, password: string){

     ..

      .pipe(tap(res => {

        const authToken = res.headers.get('x-access-token');

     -> this.userService.setToken(authToken);

        console.log(authToken);

      }))

  }

Implementando o usuário logado.

No userService quando efetuar o login vai setar o token e emitir uma informação para quem tiver se inscrevido na informação que será o cabeçalho componente. Utilizar o subject (você pode emitir um valor para ele e ao mesmo tempo se inscrever para saber o valor emitido). Criando uma propriedade privada instanciando um novo Subject.

private userSubject = new Subject<User>();

Criar uma interface na pasta user com as propriedades do token: id, name e email.

export interface User {

  id: number;

  name: string;

  email: string;

}

Agora precisamos descriptografar o token para extrair as informações. Fazemos o importe de tudo do jwt-decode.

import \* as jwt\_decode from 'jwt-decode';

Agora chamamos o método decodeAndNotify que vai ser um método auxiliar.

setToken(token: string){

    this.tokenService.setToken(token);

    this.decodeAndNotify();

  }

Criando o método: ele pega o token do método de **tokenService** que está auxiliando, agora ele pede auxilio ao jwt\_decode e passa para ele o token. Quando ele pegar o token vai decodificar o token e como tem certeza que as propriedades que estão lá são do tipo da interface criada vai fazer um casting garantindo que quando for decodificado vai promover a variável user do tipo User da interface.

Feito isso usamos nosso **userSubject** aplicando o método **next** que recebe o user.

private decodeAndNotify(){

    const token = this.tokenService.getToken();

    const user = jwt\_decode(token) as User;

    this.userSubject.next(user);

  }

Agora para salvar os dados no localStorage mesmo depois de fechar a página é passado no construtor do userService chamando o método decodeAndNotify se houver token. É chamado o método hasToken de tokenService que se tiver um token salvo no localStorage ele vai aplicar o decodeAndNotify com os dados do usuário.

constructor(private tokenService: TokenService) {

\*Entrou no sistema o serviço é criado e pergunta:

    this.tokenService.hasToken() &&  this.decodeAndNotify()

Tem token na aplicação? Sim. Então aplica esse método trazendo o user.

  }

Utilizando o operador AND && que só vai ser chamado se as duas condições forem verdadeiras. Entrou no sistema e o serviço

Agora para retornar no getUser o userSubject e quem chamar o getUser vai receber um observable e vai poder fazer o subscribe.

getUser(){

    return this.userSubject.asObservable();

  }

# Login pt.10 – Mostrando na tela o usuário

Agora no Header componente injetar no constructor o userService para retornar o usuário. No escopo do construtor usar o userService que retorna o método getUser.

constructor(userService: UserService) {

    userService.getUser();

  }

Criar a propriedade com o dólar $ no final como boa pratica dizendo que essa propriedade irá guardar um observable. Ele vai ser do tipo Observable com a interface User. Ele vai receber o resultado do getUser.

user$: Observable<User>;

  constructor(userService: UserService) {

    this.user$ = userService.getUser();

  }

Não pode ler a propriedade user$ direto no template de cabeçalho porque ele é um observable. Mas podemos fazer um subscribe e quando algum valor for emitido vai guardar dentro de user. Criamos uma nova propriedade do tipo **resUser: User** e dizemos que a propriedade **resUser** vai receber o valor emitido de **user** do **observable**.

user$: Observable<User>;

resUser: User;

  constructor(userService: UserService) {

    this.user$ = userService.getUser();

    this.user$.subscribe(user => this.resUser = user);

  }

Agora usamos esse resUser no template pegando a propriedade name dele e passamos o interrogação pois ele pode ou não existir.

<header class="fixed-top">

  <nav class="navbar navbar-light bg-white">

    <a class="navbar-brand">ALURAPIC</a>

    <div>

      <i class="fa fa-user-circle"></i>

      <a class="ml-2">{{ resUser?.name }}</a>

    </div>

  </nav>

</header>

Para que o usuário não suma depois de recarregar a página no userService vamos passar para a propriedade userSubject um **BehaviorSubject** e precisamos definir no seu construtor um valor padrão que vai emitir logo de cara. E esse valor é **null**.

O **BehaviorSubject** armazena a última emissão até que alguém apareça para consumi-la.

private userSubject = new BehaviorSubject<User>(null);

# Login p.11 – Simplificando código com o async

No TS do header responsável por trazer os dados do usuário para mostrar na tela é possível usar a propriedade do tipo observable direto no template passando um **pipe** **async**. Dessa forma não é mais necessário um **subscribe**.

user$: Observable<User>;

  constructor(userService: UserService) {

    this.user$ = userService.getUser();

}

Agora no próprio template, passamos a propriedade em dólar em parênteses no **ngIf** e aplicamos um **pipe** **async** e tratamos ele como **user**. Dessa forma conseguimos fazer a chamada dele normalmente com a propriedade name.

<div \*ngIf="(user$ | async) as user" >

      <i class="fa fa-user-circle"></i>

      <a class="ml-2">{{ user.name }}</a>

</div>

Feito isso podemos criar um template para que caso não esteja logado aparecer a mensagem pedindo que faça login utilizando o **ng-template** tendo o identificador #login.

<ng-template #login>

      <div class="navbar-text">

        <a>Please, login.</a>

      </div>

</ng-template>

Agora no nosso ngIf passamos o else, caso não tenha nenhum valor em name para aparecer na tela, aplicar o ng-template.

<div \*ngIf="(user$ | async) as user; else login" >

…

.

# Login pt.12 – Implementando o logout

No **service** responsável por saber tudo sobre o usuário que é o **userService** (onde pega o token, decompila etc) criar um método de logout sem parâmetros onde ele vai utilizar a ajuda do **tokenService** que foi injetado no construtor e chamar o método de **removeToken**. Feito isso é preciso passar um valor null para o subject que ai ele vai fazer a validação que se não tiver token ele não traz nenhum dado. Então chamamos ele e através do next passamos o valor null.

logout(){

    this.tokenService.removeToken();

    this.userSubject.next(null);

  }

Agora precisamos chamar esse método no nosso header onde mostra na tela o usuário. É necessário criar um template dentro da div que carrega o ngIf.

Agora no TS do nosso componente criamos outro método com o nome de **logout** que vai ser aplicado no template criado através do **event**-**binding**. Nesse método precisamos **chamar** o outro método logout que faz parte do service responsável por trazer os dados do usuário o userService, que já foi injetado no construtor. Feito isso, precisamos navegar para a rota de login assim que clicar no botão de logout e então é necessário injetar o **Router** e usar o método **navigate** passando uma string vazia.

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

    this.user$ = userService.getUser();

  }

  logout(){

    this.userService.logout();

    this.router.navigate(['']);

  }

<a (click)="logout()">(Logout)</a>

Chamando através do event-binding.

# Aplicando guarda de rotas

O guarda de rotas permite que você bloqueie o acesso de algumas rotas de acordo com a sua validação, por exemplo, se um usuário já está logado não é necessário mostrar a página de login.

Para aplicar, primeiro precisamos criar um método no service responsável por validar o login, o **userService** e retornar se o usuário está logado ou não e nesse método ele recebe a ajuda do **tokenService** que possui o método **hasToken.**

isLogged(){

    return this.tokenService.hasToken();

  }

Feito isso precisamos criar um arquivo **Injectable (.ts)** que vai ficar responsável por fazer essa validação, é criado na pasta onde é feita a autenticação, o **Auth**. Agora precisamos implementar uma interface nesse arquivo criado de guarda de rotas chamada de **CanActivated**. Aplicamos esse método CanActivated com dois parâmetros: o primeiro é o **ActivatedRouteSnapshot** e o segundo o **RouterStateSnapshot**.

Ele pode ter um tipo de retorno **boolean** OU do tipo **Observable**<**boolean**> OU uma **Promise**<**boolean**>.

export class AuthGuard implements CanActivate{

  constructor(){}

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean | Observable<boolean> | Promise<boolean>{

  }

Agora é necessário pegar o valor do usuário se está logado ou não injetando no construtor desse Injectable **AuthGuard** o service responsável por fazer a validação do login, o **userService**.

constructor(private userService: UserService)

Agora podemos chamar o método responsável que recebe se o usuário está logado no escopo do canActivated passando uma **condicional**. O retorno do método **isLogged** é um booleano então podemos fazer: se o método **isLogged** for true não deixa fazer a navegação para a página de login. Precisamos injetar o router que é responsável por fazer essa navegação.

Passamos no escopo da condicional que use o método **navigate** para pegar a rota e jogar o usuário no user/seu-usuário. Caso o isLogged seja falso ele retorna um **false**.

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

canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean | O

bservable<boolean> | Promise<boolean>{

    if(this.userService.isLogged()){

      this.router.navigate(['user', this.userService.getUserName()])

      return false;

    }

    return true;

  }

Agora precisa ser implementado o **getUserName** para trazer o nome do usuário para qual a página vai carregar, precisamos trazer esse nome em uma propriedade do nosso TS responsável pelo login. Para isso é criado uma propriedade **userName** do tipo string e o método para obter esse valor retornando o userName.

private userName: string;

 getUserName(){

    return this.userName;

 }

Agora no método responsável por decodificar o token e trazer as informações podemos usar a **const user** com os dados decodificados e jogar a propriedade de **name** na propriedade criada **userName**.

private decodeAndNotify(){

    const token = this.tokenService.getToken();

    const user = jwt\_decode(token) as User;

    this.userName = user.name;

    this.userSubject.next(user);

  }

Feito isso só precisamos chamar nosso Injectable criado no **routing**-**module** no **path** responsável por exibir a página de login. Depois de componente colocamos a propriedade **canActivate** que recebe o nome do nosso Injectable, [**AuthGuard**].

path: '',

    component: SignInComponent,

    canActivate: [AuthGuard]

# Interceptadores e envio de token

A ideia principal é: caso exista um botão de excluir apenas quem está logado tem o direito de excluir suas fotos e se a pessoa é um usuário que só está visualizando sem que tenha cadastro não pode ter acesso a esse artificio. Para que a execução possa ser feita é necessário enviar o token de acesso e depois é possível executar a solicitação de exclusão.

Criando interceptadores do **HttpClient.** É criado dentro de Core/Auth o arquivo request.interceptor que é um Injectable e para torna-lo é preciso implementar um **HttpInterceptor.** Agora é utilizado o método **intercept** que recebe como parâmetro o **req** do tipo **HttpRequest** e o **next** do tipo **HttpHandler**.

constructor(private tokenService: TokenService){}

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

  }

A ideia do intercept é que se você não interceptar nada ao final é necessário dar **return** utilizando o auxilio do parâmetro **next** utilizando o método **handle** que vai retornar o parâmetro **req.** Pois não houve mudança alguma. Mas é preciso modificar a request antes de retorna-la então no constructor é passado o **tokenService** que vai ajudar retornando se tiver token no localStorage o usuário vai estar logado. Então é passado a condicional que se estiver logado vai pegar o token utilizando o **getToken** do service e é obrigatório clonar essa requisição modificando essa requisição colocando o token no cabeçalho do **x**-**access**-**token.**

**Req** vai receber o resultado de **req.clone** que recebe um objeto javascript com a propriedade **setHeaders** sendo possível adicionar um token e na chave de setHeaders é colocado o ‘x-access-token’ com o valor da **const** que recebe o token.

intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

    if(this.tokenService.hasToken()){

      const token = this.tokenService.getToken();

      req = req.clone({

        setHeaders: {

          'x-access-token': token

        }

      })

    }

    return next.handle(req);

  }

Agora no modulo responsável (como estamos em **Auth** é o **CoreModulNe**) é preciso colocar esse intercept para que não seja aplicado o padrão do angular então adicionamos um **providers** e dentro do array colocamos um objeto que possui a propriedade **provide:** e passamos o **HTTP\_INTERCEPTORS** utilizando a classe criada do Intercept que é **RequesteInterceptor** e caso tenha mais de um interceptador utilizamos também o multi para delegar pra todos.

imports: [CommonModule, RouterModule],

  providers: [

    {

      provide: HTTP\_INTERCEPTORS,

      useClass: RequestInterceptor,

      multi: true

    }

  ]