# Início do projeto

Todo o projeto se encontra dentro da pasta **app.**

# Organização de componentes

Foi criado uma pasta chamada ‘photos’ e tudo que é referente a fotos fica dentro dessa pasta. O **photos.module** criado serve como encapsulamento pra que todo componente criado seja inserido nesse módulo, sendo assim, o código fica mais limpo e apenas é necessário fazer o importe do **PhotosModule** em **app.module.ts** para que o projeto possa ser compilado. Dessa forma não é necessário importar todo componente referente a foto no **app.module** (ficaria dezenas de linhas de importe) e sim fazê-los em **photos.module**.

@NgModule({

  declarations:

  [PhotoComponent, PhotosListComponent, PhotoFormComponent],

  exports:

  [PhotosListComponent],

  imports:

  [HttpClientModule, CommonModule]

})

# Componentes de renderização

Foi criado um componente para a estrutura do projeto com seu escopo e passando uma [ property binding ] para que seja feita uma Input.

<img class="img-thumbnail" [src]="url" [alt]="description">

Feito isso no ts desse componente é passado o **Input()** e sua propriedade ao lado sendo inicializada para que possa ser modificada/atribuída fora do componente.

@Input() url = '';

@Input() description = '';

Feito isso foi criado um **Service** para esse componente onde ele fica responsável por toda a lógica necessária para que busque de uma API os dados que precisamos. Passamos também ao construtor desse Service o **HttpClient** que será necessário para a renderização do nosso componente. E por fim, criamos um método de retorno.

const API = 'http://localhost:3000';

constructor(private http: HttpClient){}

listFromUser(userName: string){

    return this.http.get<Photo[]>(API + `/${userName}/photos`);

}

de retorno do método get é um Photo[ ], ou seja, uma classe que criamos com todas as propriedades que a nossa API está retornando.

export interface Photo{

  url: string;

  description: string;

...

}

O próximo passo é criar um outro componente que vai ser responsável por receber o método de **ListFromUser( )** e devolver o resultado disso para que possa ser renderizado na tela. Feito a criação desse novo componente, o seu ts recebe uma variável que vai guardar todas as informações retornadas.

response: Photo[] = [];

Passamos ao construtor desse componente o Service criado que está responsável pela lógica, para que possamos usar é necessário colocar o **private** na frente dele.

constructor(private photoService: PhotoService) { }

Agora podemos usar esse objeto instanciado assim que nosso módulo for carregado.

ngOnInit(): void{

    this.photoService.listFromUser('flavio')

    .subscribe(photos => response = photos)

  }

É chamado nosso método **ListFromUser()**, passamos o parâmetro que é solicitado e usamos nosso observable usando a variável **photos**(que irá guardar toda informação trazida da API) e guardamos essa informação na nossa variável de photos que foi definida no início.

Agora é necessário apenas passar o seletor do nosso componente com a estrutura de visualização para esse componente onde é responsável pela renderização. Junto com o ngFor para que possa percorrer por toda nossa variável do tipo array que foi criada no ts.

<ap-photo

\*ngFor="let photo of response" [url]="photo.url" [description]="photo.description">

</ap-photo>

# Dividindo componentes p.1

Nessa parte fazemos uma pequena alteração para que o componente possa ser carregado de acordo com a grid do bootstrap, vamos dividir nosso componente. **Photo-list** é o responsável por renderizar a página, tanto que ele estava com o html. Nessa parte iremos criar um novo componente dentro de **photo-list** que vai ser o responsável por trazer esses dados ao photo-list. Agora a estrutura está nesse novo componente criado.

<!--ol é equivalente a uma linha -row- -->

 <ol class="list-unstyled row">

  <!--cada li vai guardar 4/12 isso equivale a 3 li em cada linha -->

  <li \*ngFor="let photo of photosInput" class="col-4">

    <ap-photo

        [url]="photo.url"

        [description]="photo.description">

    </ap-photo>

  </li>

</ol>

No seu TS é criado a propriedade **photosInput** (que recebe o mesmo nome no momento do **ngFor**) que é atribuído o Input ao lado para que possa receber um valor fora do seu componente.

@Input() photosInput: Photo[] = [];

Agora fazemos a chamada do seletor desse novo componente no **Photo-list**, que é o responsável por renderizar a página de fotos e passamos uma **inbound-properties** [ ] da propriedade que pode receber um valor de fora. Ela recebe o valor da propriedade **photoList** que está recebendo todos os dados sendo retornados do back-end através do método criado no **Service**.

<app-photos [photosInput]="photosList"></app-photos>

# Divisão de componentes sendo carregados corretamente p.2

Alteramos a estrutura que contém o html da renderização de fotos, agora cada **li** vai retornar uma linha (row) aplicando o **ngFor**. E a **div** responsável por retornar as fotos também recebe um **ngFor**. Ela fica com a estrutura de devolver em cada linha três fotos utilizando o **col-4**.

<li \*ngFor="let cols of rows" class="row">

    <div \*ngFor="let itemPhoto of cols" class="col-4">

      <ap-photo

        [url]="photo.url"

        [description]="photo.description">

      </ap-photo>

    </div>

  </li>

No TS do componente é criado a propriedade **rows**, um **array** do tipo **any**. Ao final, essa propriedade vai receber no seu **ngonInit** o método criado para fazer a renderização das linhas.

O método criado recebe como parâmetro a estrutura Photo[] que contém todas as propriedades retornadas do back, no escopo do método é criado uma const **newRows** do tipo array. Criamos um laço de repetição em que enquanto o **index** for menor do que o **parâmetro** **do método** (que na chamada será a propriedade que recebe todos os dados de photos) ele vai continuar no laço e a sua iteração se dá por +=3. No escopo do laço, a constante criada recebe o método **push** que coloca dentro dela o resultado do **slice**.

groupColumns(photos: Photo[]){

    const newRows = [];

    for(let index = 0; index < photos.length; index+=3){

      newRows.push(photos.slice(index, index + 3));

    }

    return newRows;

  }

# Renderizando na tela utilizando OnChanges p.3

A inicialização do componente de photos é feita apenas uma única vez e na lógica do projeto o Input() que realiza o data-binding está vazio, então quando ele atualiza trazendo os dados a inicialização já foi feita. Ou seja, ngOnInit acaba não sendo útil nesse caso, vamos utilizar o Change, assim que mudar o valor, ele atualiza pra gente.

ngOnChanges(changes: SimpleChanges): void {

  }

OnChanges recebe como parâmetro todas as possíveis mudanças das inbounds properties do componente. O tipo do parâmetro é **SimpleChanges.**

@Input() photosInput: Photo[] = [];

***Inbound Properties***

No escopo do **ngOnChanges**, se houver uma mudança vai ser adicionado automaticamente uma propriedade do mesmo nome da inboud properties. Se não houver, não terá essa propriedade.

ngOnChanges(changes: SimpleChanges): void {

    if(changes.photosInput)

      this.rows = this.groupColumns(this.photosInput);

  }

# Utilizando o ActivatedRoute

No componente responsável pela lógica de renderização passar ao construtor o activatedRoute como privado e fazer o auto-import. Depois disso criar uma variável que irá receber a seguinte lógica:

constructor(private photoService: PhotoService, private activatedRoute: ActivatedRoute) { }

const userName = this.activatedRoute.snapshot.params.userName;

Essa variável é responsável por ir buscar no **app.routing** o parâmetro de acesso que foi definido lá na rota. Após o params é necessário utilizar o mesmo nome que foi definido.

 path: 'user/:userName',

Logo em seguida utilizar a variavel userName sendo o parâmetro passado na chamada do método **listFromUser( )**, sendo assim, fica acessível para fazer o roteamento manualmente.

 this.photoService.listFromUser(userName)

# Slice

Slice sempre recebe a posição inicial que você deseja considerar **slice(index**, ...) e a posição final não inclusiva **slice(...**, **index + 3).** Ele fatia o array.

for(let index = 0; index < photos.length; index+=3){

newRows.push(photos.slice(index, index + 3));

                                  0 (0, 1, 2) 3

3 (3, 4, 5) 6

}

*Ou seja, sempre vai pegar de 0 até 2. Logo em seguida a posição que deseja considerar passa a ser 3 e ai ele continua pegando de 3 até 6. Depois a posição inicia com 6 e continua pegando de 6 até 9...*

# Filtrando Fotos com pipe

A primeira coisa a ser feita é criar o input necessário de pesquisa. Feito isso, capturamos o valor que é digitado utilizando **two-way data binding**. É necessário importar FormsModule.

<input type="search" class="rounded" placeholder="Buscar..."

[(ngModel)]="buscar" name="buscar" id="buscar">

buscar: string = '';

Feito isso, é necessário criar nosso **pipe** para que seja aplicado o filtro no componente que traz o escopo das fotos.

O **pipe** é criado na pasta de acordo com o seu uso, no componente que será aplicado. Criamos o **filter-by-description.pipe.ts**, aplicamos seu decoretor @Pipe que recebe os parâmetros em objeto com o seu name. Esse **name** é utilizado no componente, quando aplicar o filtro.

@Pipe({ name: 'filterByDescription' })

Com a classe do pipe criada podemos usar o name dele e aplicar no nosso componente que vai levar em conta o que o usuário digitar no input para que realize o filtro nas fotos. Agora precisamos implementar a lógica.

<-Aplicando pipe em photos (filter recebe o valor digitado no input)->

<app-photos [photos]="responsePhotoList | filterByDescription: buscar"></app-photos>

Nossa classe pipe foi criada e agora precisa ser implementada o **PipeTransform** para que possa usar o método **transform( ).** O método transform recebe dois parâmetros, o primeiro é em quem você quer aplicar uma transformação, que no nosso caso, é o componente de **Photo** que tem todas as propriedades da API do back-end. O segundo parâmetro é a string utilizada na busca que no pipe representa por **descriptionQuery.**

transform(photos: Photo[], descriptionQuery: string) {

}

Depois disso, no escopo do **transform** dizemos que nossa **descriptionQuery** vai receber a mesma variável com .**trim()** retirando todo espaçamento e **toLowerCase()** transformando toda string digitada em caixa baixa.

Agora é feita uma validação de que: Se existir **descriptionQuery**, ou seja, se tiver algo digitado no input de busca vai filtrar e retornar as photos. Utiliza o parâmetro do transform do tipo **Photo[]** e aplica um método de **filter().** Para cada retorno de **photo =>** pegar a **description** que ela tem, jogar para caixa baixa e vê se o que digitou faz parte de **description**.

Se não fizer parte faz um retorno de photos sem nenhum filtro.

transform(photos: Photo[], descriptionQuery: string) {

    descriptionQuery = descriptionQuery.trim().toLowerCase();

    if(descriptionQuery){

      return photos.filter(photo => photo.description.toLowerCase().includes(descry

ptionQuery))

    }

    else {

      return photos;

    }

  }

# Aplicando Resolver

É criado atrelado na pasta do componente onde será usado, utilizando o **.resolver.ts.** Necessário usar o decoretor de **@Injectable (recebe providedIn: ‘root’)** pois será injetado no componente e sempre que for algo injetável precisa também passar no construtor o service que irá utilizar, nesse caso é o **PhotoService**.

Para se tornar um **resolver** precisa ser implementado um Resolve com o tipo de dado que o resolve vai devolver no final. E esse tipo é o mesmo que o método do PhotoService devolve que é um **Observable** do tipo **Photo[].**

@Injectable({providedIn: 'root'})

export class PhotoListResolver implements Resolve<Observable<Photo[]>>{

  //Importando o service com a lógica de retorno de photos

  constructor(private service: PhotoService){}

}

Agora é implementado o método **resolve**() que recebe como parâmetro o que está acontecendo com a rota naquele momento, o **ActivatedRouteSnapshot** e um state do tipo RouterStateSnapshot que não será utilizado.

Criamos uma variável com o userName que recebe o **route.params.userName** (esse **userName** é o que foi definido como parâmetro na rota). Agora ele retorna o **método** do **service** recebendo como parâmetro a const criada.

resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<Photo[]> {

    const userName = route.params.userName;

    return this.service.listFromUser(userName);

 }

Agora é necessário ir nas rotas do projeto e aplicar depois do path a propriedade resolve passando uma “variável” que recebe o nome do resolve criado. Feito isso, no **onInit** diz que a variável que recebe a lista de photos recebe o **activatedRoute.snapshot.data.photos** (permite pegar o valor do resolve no **rounting**, a propriedade criada)

path: 'user/:userName',

    component: PhotosListComponent,

    resolve: {

      photos: PhotoListResolver

    }

ngOnInit(): void{

    this.responsePhotoList = this.activatedRoute.snapshot.data['photos'];

  }

# Paginação

Criando a paginação: Ir ao **Service** responsável pelo método que captura a API do back-end e criar outro método passando o parâmetro de **page**.

listFromUserPaginated(userName: string, page: number){

    const resParams = new HttpParams().append('page', page.toString())

    return this.http.get<Photo[]>(API + `/${userName}/photos`, { params:

  resParams });

  }

Necessário criar uma **const** que vai receber a instancia de **HttpParams** sendo trazido do **HttpClient**. Logo depois usa o append para que possa colocar um valor dentro passando o **page**(a propriedade) e transformando em string.

No retorno do método adicionar um objeto que tem seu tipo como params e adicionar a const criada.

No Resolve do projeto (onde é feito todo carregamento da rota do componente antes de mostrar para o usuário) trocar o método utilizado para o que recebe os parâmetros de paginação e adicionar o parâmetro com valor 1, sendo a primeira página.

return this.service.listFromUserPaginated(userName, 1);

# Load Button

Criando um componente que para utilizar o botão de carregamento e também exibir uma mensagem dizendo que o carregamento acabou. No TS desse componente criar a propriedade de **hasMoreButton** do tipo **boolean** com inicialização **false**. Colocando o **Input( )** nessa propriedade para que ela receba um valor de fora caso alguém utilize esse componente.

@Input() hasMoreButton: boolean = false;

No componente **html** do botão utilizar a diretiva **ngIf** com a condição de que hasMoreButton for positiva exibe o botão para carregar mais e -se não-, utilizar um **else** para carregar a variável de template (uma variável que identifica o template sobre a tag **ng-template**).

<div class="text-center" \*ngIf="hasMoreButton; else messageTemplate">

  <button class="btn btn-primary">Load more</button>

</div>

<ng-template #messageTemplate>

  <p class="text-center text-muted">No more data to load</p>

</ng-template>

No TS do componente que renderiza as fotos na tela -**photos.list.component**- criar uma propriedade hasMoreList do tipo boolean e com inicialização em true. Essa propriedade irá alterar o valor de **hasMoreButton**(que utiliza o input).

hasMoreList: boolean = true;

Na chamada do selector do botão é passado o [**hasMoreButton**] que recebe o **hasMoreList** onde é realizado um **data-binding**. O hasMoreButton está recebendo o valor que é passado em hasMoreList.

<app-load-button [hasMoreButton]="hasMoreList"></app-load-button>

Depois é necessário criar a propriedade userName e passar ao **ngOnInit** a propriedade de **userName** ao **activatedRoute**.

this.userName = this.activatedRoute.snapshot.params.userName;

1: Agora importamos nosso Service responsável pelo método de paginação que traz as fotos do back-end no construtor. Logo depois criamos o método de **load()** e utilizamos o método **listFromUserPaginated** com os parâmetros necessários de **userName** e **page**. Temos nossa propriedade userName criada e só nos falta a propriedade de **page** que irá começar com seu valor em 1.

2: Depois de passar os parâmetros necessários ao nosso método (o parâmetro de currentPage recebe ++ para cada chamada ele acrescentar +1 a propriedade. Assim sendo ela começa em 1 e quando houver outra chamada passa a ter 2...3...4...).

3: Fazemos o subscribe retornando os dados do back na variável local **photos** e depois dizemos que **responsePhotoList** (variável global do componente) está recebendo uma nova referencia e acrescentando photos. Automaticamente o Angular entende que o valor mudou e atualiza em toda chamada da propriedade.

1 currentPage: number = 1;

2 load(){

    this.photoService.listFromUserPaginated(this.userName, ++this.current

Page)

3   .subscribe(photos => {

      this.responsePhotoList = this.responsePhotoList.concat(photos);

      if(!photos.length)

        this.hasMoreList = false;

    })

  }

Por fim, é realizado uma condição de que: Se **!photos.length** (o valor inicia em true mas com a exclamação passa a ser not-true, ou seja, se **photos.length** for false ele executa) vai alterar o valor da propriedade hasMoreList para false. Com isso hasMoreButton passa a ser falso também por conta do data-binding.

if(!photos.length)

        this.hasMoreList = false;

com o subscribe photos sempre retorna um valor true (1) enquanto houver dados no back-end, a partir do momento que todos os dados são carregados ele passa a ser false (0).

Agora chamamos nosso método **load( )** no selector do Load Button.

# Criando Diretiva – Manipulando DOM

A diretiva foi criada em **Shared** pois dependendo do funcionamento pode ser aplicada em diversos componentes. Basta criar o arquivo com **.directive.ts** depois é necessário usar o **Declaretor** de **Directive**, passando o seu seletor em **colchetes** [] pois para a utilização é necessário apenas passar o nome do seletor no local de uso.

@Directive({

  selector: '[HoverDirective]'

})

A diretiva criada tem utilização em alterar o DOM no momento que o mouse entra e quando sai de um determinado elemento. Passamos ao construtor **ElementRef** que é responsável por alterar o DOM e também o **Renderer** que faz com que você não precise diretamente ir no DOM fazer essa alteração, pode utilizar esse método.

constructor(private el: ElementRef, private render: Renderer2){}

Tanto para o método de entrada e o de saída é passado o @**HostListener** especificando qual evento será aplicado.

@HostListener('mouseover')

  mouseEntre(){

  }

  @HostListener('mouseleave')

  mouseFora(){

  }

No método dos eventos, utilizar o render passando o **setStyle** que recebe como parâmetros o elemento nativo que faz a **alteração** do DOM, depois o que ele irá **aplicar** e por último o **valor** do que será aplicado.

@Directive({

  selector: '[HoverDirective]'

})

export class HoverDirective {

  constructor(private el: ElementRef, private render: Renderer2){}

  @HostListener('mouseover')

  mouseEntre(){

    this.render.setStyle(this.el.nativeElement, "backgroundColor", "red")

  }

  @HostListener('mouseleave')

  mouseFora(){

    this.render.setStyle(this.el.nativeElement, "backgroundColor", "transparent")

  }

}

Feito isso é necessário passar na declaração do módulo o a diretiva criada, depois está pronta para a utilização.

# Aplicando rotas filhas

Nesse caso o intuito de aplicar uma rota filha é que no componente principal assim que carregado a página, mostrar uma divisão entre uma foto e o componente de login e de registro. Para fazer isso é necessário criar um componente que no caso foi de home.component e no template fazer a divisão com a foto e os outros componentes. Na outra parte é colocado o **router**-**outlet** onde sera carregado os componentes filhos.

<div class="row">

  <div class="col-md-6 mb-2">

    <img class="img-fluid d-none d-sm-

block" src="../../assets/img/home.jpg" alt="">

  </div>

  <div class="col-md-6">

    <router-outlet></router-outlet>

  </div>

</div>

Agora no **app-routing** no path responsável pelo primeiro carregamento é passado o **HomeComponent** para ser carregado e na propriedade **children** é passado um array que recebe como objeto os paths seguintes, as rotas filhas. É passado um novo **path** com uma string em branco que vai ser o primeiro componente carregado ao lado da imagem e depois declarando qual o componente. Logo depois, o próximo componente junto com a string de acordo com o componente.

 {

    path: '',

    component: HomeComponent,

    canActivate: [AuthGuard],

    children: [

      {

        path: '',

        component: SignInComponent,

      },

      {

        path: 'signup',

        component: SignUpComponent,

      },

    ]

  },

Agora vamos adicionar o **focus** para o input desejado quando fizer a navegação entre componentes no home. Para que isso possa ser executado vamos implementar junto ao **OnInit** o **AfterViewInit** que vai executar 1x depois de ser carregado.

ngAfterViewInit(): void {

  }

Agora no escopo colocamos a estrutura utilizando o método que verifica se o usuário está em um navegador ou mobile. **SE** a plataforma for um **Browser** o focus é carregado e implementado no momento de redirecionamento. Lembrando que foi preciso utilizar o @**ViewChild** que pega a variável de template para colocar na propriedade criada do tipo **ElementRef<HTMLInputElement>** para manipular o DOM. O método que identifica se o usuário está no Browser ou não foi criado dentro de **Core** e precisa ser injetado.

<input formControlName="email" . . . autofocus #inputEmail>

constructor(. . . , private platformDetectorService: PlataformDetectorService)

ngAfterViewInit(): void {

    if(this.platformDetectorService.isPlatformBrowser()){

      this.userNameInput.nativeElement.focus();

    }

  }

# Fazendo Build do projeto

É necessário parar o projeto e executar **ng** **build** –**prod**. Esse código vai replicar varias técnicas de otimização para que no final possa gerar o arquivo do projeto js no final o menor possível.

É criado o arquivo **dist** onde ele vai ser disponibilizado para que o projeto suba em algum lugar.

Para colocar o # hash na rota é necessário ir no **app**-**routing** e no importe, no método **forRoot** passar um segundo parâmetro em objeto com a propriedade **useHash: true.**

imports: [RouterModule.forRoot(routes, {useHash: true})],

# Implementando o Lazy Loading

Ajustando as rotas fazendo com que possa ver o carregamento Lazy loading de forma mais clara. No **routing** é criado um novo **path** em branco e depois um **pathMatch = “full”** para que considere exatamente igual a rota depois da barra e por ultimo um **redirectTo** chamando a rota **home**.

{

    path: '',

    pathMatch: 'full',

    redirectTo: 'home'

},

{

    path: 'home',

    component: HomeComponent,

    canActivate: [AuthGuard],

    children: [

      {

        path: '',

        component: SignInComponent,

      },

      {

        path: 'signup',

        component: SignUpComponent,

      },

    ]

  },

A feature que desejamos fazer o carregamento preguiçoso é o component home e para isso criaremos o próprio arquivo de rotas dele. Copiaremos rodo código do **AppRouting** e deixaremos apenas as rotas do componente home. Só que na rota, no lugar de **home** é colocado uma string vazia. Essa rota vai ser dependente de uma rota criada no pai e no pai vai ser criado a rota e dizer que ele é dependente dessas aqui. Mas não pode ser montando com **forRoot**, precisa ser montado com **forChild** passando o arquivo de rotas que foi criado.

const routes: Routes = [

  {

    path: '',

    component: HomeComponent,

    canActivate: [AuthGuard],

    children: [

      {

        path: '',

        component: SignInComponent,

      },

      {

        path: 'signup',

        component: SignUpComponent,

      },

    ]

  },

];

NgModule({

  imports: [RouterModule.forChild(routes)],

  exports: [RouterModule]

})

export class HomeRoutingModule { }

Agora é apagado todas as rotas que fazem parte do **HomeRouting (HomeComponent, SignInComponent e SingUpComponent )** no componente de rota principal. Uma coisa importante é que se quer carregar um módulo preguiçosamente ele não pode fazer parte do **app**.**module**.**ts.** Então é necessário tirá-lo. Agora no modulo do componente que será feito o carregamento preguiçoso é necessário importar o **HomeRoutinModule.** No arquivo de rotas principal é criado o **path** com a string ‘**home’** e depois a propriedade **loadChildren** que vai receber uma arrow-function que chama um importee dentro é passado o **caminho do modulo do componente** que nesse caso é o **HomeModule** que **depois** é criado uma propriedade **m** e vai chamar o **m.HomeModule.**

{

    path: 'home',

    loadChildren: () => import('./home/home.module').then(m => m.HomeModule)

},

# Refinando os services para utilizar apenas onde é chamado

Como forma de refinar ainda mais o projeto vamos tirar o **provideIn** ‘**root’** para que ele fique disponível apenas nos componentes necessários. Da forma padrão como é feita o **service** fica visível para o projeto inteiro então nos dois services de sign-up (**SignUpService** e **UserNotTaken**) vamos tirar o provideIn root dos dois e no TS do component de **SignUp** vamos injetar em providers: [] depois de templateUrl o **UserNotTaken.** O outro service é importado no modulo do componente.