1 Introdução

O desenvolvimento moderno de aplicações Web é comumente realizado utilizando-se diferentes tecnologias. Muitas vezes, sua combinação dá origem a algo que tem levado o nome de desenvolvimento “Full Stack”. Uma solução desenvolvida segundo esse paradigma possui, em geral, duas aplicações independentes que se comunicam por meio de uma interface bem definida. Uma delas oferece interfaces gráfica para o usuário e geralmente é chamada de aplicação Front End. A outra é responsável por disponibilizar as funcionalidades do sistema e leva o nome de aplicação Back End. Em geral, ela faz uso de um sistema independente que possui implementações eficientes para operações de acesso à memória secundária.

Além disso, seja no Front End ou no Back End, é comum o uso de diferentes *frameworks* que supostamente entregam um nível maior de abstração e promovem a produtividade dos desenvolvedores.

Nos dias atuais, uma das combinações mais utilizadas para esse fim tem a sigla “**MEAN**” associada, a qual deriva de “**MongoDB**”, “**Express**”, “**Angular**” e “**NodeJS**”.

Neste material desenvolveremos uma aplicação que faz uso da “pilha” MEAN.

2 Desenvolvimento

2.1 (Executando os servidores) Lembre-se de colocar os dois servidores em execução com

npm run start:server (Back End em NodeJS)
ng serve --open (para acesso à aplicação Angular)

- Execute cada um deles em um terminal separado e mantenha ambos em execução.

Nota: Lembre-se de acessar o serviço Atlas do MongoDB (se estiver utilizando ele, claro) e habilitar acesso para seu endereço IP, caso ainda não tenha feito ou caso tenha habilitado uma regra temporária que, neste momento, já pode ter expirado. Para isso, acesse o Link 2.1.1 e faça login na sua conta. A seguir, clique em **Network Access** à esquerda e clique em **Add IP Address**.

Link 2.1.1
https://www.mongodb.com/

2.2 (Atualização considerando as imagens) Quando o usuário faz uma atualização, a aplicação considera somente os campos textuais: nome, fone e e-mail. A figura ainda não faz parte dessa operação. Basta verificar o método **ngOnInit**. Repare que temos uma estrutura de seleção, destacada na Listagem 2.2.1, que verifica se a rota contém **idCliente**, o que quer dizer que o componente foi renderizado pois o usuário deseja fazer uma atualização. Note que, no momento, o objeto JSON usado para configurar os campos do form contém somente nome, fone e e-mail. Vamos adicionar também o valor de imagem, que é uma string. Estamos no arquivo **cliente-inserir.component.ts.**

Listagem 2.2.1

```
ngOnInit(){
  ...
   this.route.paramMap.subscribe((paramMap: ParamMap) => {
    if (paramMap.has("idCliente")){
      this.modo = "editar";
      this.idCliente = paramMap.get("idCliente");
      this.estaCarregando = true;
     this.clienteService.getCliente(this.idCliente).subscribe( dadosCli => {
       this.estaCarregando = false;
       this.cliente = {
        id: dadosCli._id,
        nome: dadosCli.nome,
        fone: dadosCli.fone,
        email: dadosCli.email,
        imagemURL: null
      };
      this.form.setValue({
        nome: this.cliente.nome,
        fone: this.cliente.fone,
        email: this.cliente.email,
        imagem: this.cliente.imagemURL
      })
     });
    }
    else{
      this.modo = "criar";
      this.idCliente = null;
    }
   });
 }
```

- A seguir, ajustamos a chamada ao método **atualizarCliente** feita pelo método **onSalvarCliente**, também no arquivo **cliente-inserir.component.ts**. Veja que ele ainda não envia a foto. Ocorre que o campo imagem do form pode ter dois tipos diferentes:

- **texto:** Quando o usuário navegou para a tela de edição e não alterou a foto clicando no botão Selecionar Imagem. Neste cenário, ele tem intenção de manter a foto atual.

- **imagem**: Quando o usuário navegou para a tela de edição e alterou a foto clicando no botão Selecionar Imagem. Neste cenário, ele tem intenção de atualizar a foto no Back End.

Sendo assim, o método **atualizarCliente** precisa ser capaz de lidar com os dois casos. A sua chamada, feita no método **onSalvarCliente**, aparece na Listagem 2.2.2.

Listagem 2.2.2

```
onSalvarCliente() {
   if (this.form.invalid) {
     return;
   }
   this.estaCarregando = true;
   if (this.modo === "criar"){
     this.clienteService.adicionarCliente(
       this.form.value.nome,
       this.form.value.fone,
       this.form.value.email,
       this.form.value.imagem
     );
   }
   else{
     this.clienteService.atualizarCliente(
       this.idCliente,
       this.form.value.nome,
       this.form.value.fone,
       this.form.value.email,
       this.form.value.imagem
     );
   }
   this.form.reset();
 }
```

- No arquivo **clientes.service.ts**, começamos adicionando um novo parâmetro à assinatura do método **atualizarCliente**. Ele pode ser do tipo **string ou File**, o que pode ser especificado utilizando-se o operador |. Veja a Listagem 2.2.3.

Listagem 2.2.3

```
atualizarCliente (id: string, nome: string, fone: string, email: string, imagem: File | string){
   const cliente: Cliente = { id, nome, fone, email, imagemURL: null};
   this.httpClient.put(`http://localhost:3000/api/clientes/${id}`, cliente)
   .subscribe((res => {
     const copia = [...this.clientes];
     const indice = copia.findIndex (cli => cli.id === cliente.id);
     copia[indice] = cliente;
     this.clientes = copia;
     this.listaClientesAtualizada.next([...this.clientes]);
     this.router.navigate(['/'])
   }));
  }
```

- O método **atualizarCliente** precisa decidir a forma como enviará os dados para o Back End. Isso dependerá do tipo do parâmetro imagem. Se ele for um arquivo, é preciso construir um objeto do tipo **FormData**. Caso contrário, enviamos um objeto JSON comum. Veja a Listagem 2.2.4. Note que esperamos obter, como parte da resposta, a URL da imagem, a qual ainda precisa ser tratada. É de se esperar que, neste momento, O VS Code indique um erro na linha em que tentamos utilizá-la. Ajustaremos isso em breve. Por isso, **atribua uma cadeia vazia a ela, por enquanto**.

Listagem 2.2.4

```
atualizarCliente (id: string, nome: string, fone: string, email: string, imagem: File | string){
   //const cliente: Cliente = { id, nome, fone, email, imagemURL: null};
   let clienteData: Cliente | FormData ;
   if (typeof(imagem) === 'object'){// é um arquivo, montar um form data
     clienteData = new FormData();
     clienteData.append("id", id);
     clienteData.append('nome', nome);
     clienteData.append('fone', fone);
     clienteData.append("email", email);
     clienteData.append('imagem', imagem, nome);//chave, foto e nome para o arquivo
   }else{
     //enviar JSON comum
     clienteData = {
       id: id,
       nome: nome,
       fone: fone,
       email: email,
       imagemURL: imagem
     }
   }
   console.log (typeof(clienteData));
   this.httpClient.put(`http://localhost:3000/api/clientes/${id}`, clienteData)
   .subscribe((res => {
     const copia = [...this.clientes];
     const indice = copia.findIndex (cli => cli.id === id);
     const cliente: Cliente = {
       id: id,
       nome: nome,
       fone: fone,
       email: email,
       imagemURL: ""
     }
     copia[indice] = cliente;
     this.clientes = copia;
     this.listaClientesAtualizada.next([...this.clientes]);
     this.router.navigate(['/'])
   }));
  }
```

- No **Back End**, no arquivo **backend/rotas/clientes.js**, precisamos fazer a refatoração necessária para que ocorra o tratamento da imagem enviada pela aplicação cliente. O primeiro passo é utilizar o pacote **multer** novamente, indicando que a requisição possui um único arquivo. A seguir exibimos, no console, o objeto recebido, a fim de entender a sua estrutura. Veja a Listagem 2.2.5.

Listagem 2.2.5

```
router.put(
  "/:id",
multer({ storage: armazenamento }).single('imagem'),
(req, res, next) => {
  console.log (req.file);
  const cliente = new Cliente({
    _id: req.params.id,
    nome: req.body.nome,
    fone: req.body.fone,
    email: req.body.email
  });
  Cliente.updateOne({ _id: req.params.id }, cliente)
    .then((resultado) => {
      //console.log(resultado)
      res.status(200).json({ mensagem: 'Atualização realizada com sucesso' })
    });
});
```

- Se tentarmos subter uma requisição sem selecionar um arquivo (portanto enviando uma string simples) nada acontecerá, pois o validador do form somente considera válidas requisições que possuam arquivos que sejam algum tipo de imagem. Por isso, abra o arquivo **mime-type.validator.js** e verifique o tipo recebido. Se for do tipo **string**, devolva um Observable que automaticamente devolve true. Um Observable pode ser construído explicitamente com o método **of** do pacote **rxjs**. Quando alimentado com **null**, o Observable construído devolve true. Veja a Listagem 2.2.6.

Listagem 2.2.6

```
import { AbstractControl } from "@angular/forms";
import { Observable, Observer, of } from "rxjs";

export const mimeTypeValidator = (
  control: AbstractControl
): Promise <{[key: string]: any}> | Observable <{[key: string]: any}> => {
  if (typeof(control.value) === 'string')
    return of(null);
  const arquivo = control.value as File;
  const leitor = new FileReader ();
  const observable = new Observable((
    observer: Observer <{[key: string]: any}>) => {
      leitor.addEventListener ('loadend', () => {
        const bytes = new Uint8Array (leitor.result as ArrayBuffer).subarray(0,4);
        let valido: boolean = false;
        let header = "";
        for (let i = 0; i < bytes.length; i++){
          header += bytes[i].toString(16);
        }
        switch (header) {
          case '89504e47':
          case 'ffd8ffe0':
          case 'ffd8ffe1':
          case 'ffd8ffe2':
          case 'ffd8ffe3':
          case 'ffd8ffe8':
            valido = true;
            break;
          default:
            valido = false;
        }
        observer.next(valido ? null : {mimeTypeInvalido: true});
        observer.complete();

      })
      leitor.readAsArrayBuffer(arquivo);
  });
  return observable;
}
```

- Note que a submissão ainda não ocorre. Ocorre que o método **getCliente** definido no arquivo **clientes.service.ts** – que é chamado quando clicamos em **editar** – não lida com a imagem previamente salva na base. Assim, o form fica com o campo da imagem não preenchido. Iremos, portanto, ajustá-lo para que passe a lidar com ela também. Veja a Figura 2.2.7.

Figura 2.2.7

```
getCliente (idCliente: string){
   //return {...this.clientes.find((cli) => cli.id === idCliente)};
   return this.httpClient.get<{_id: string, nome: string, fone: string, email: string, imagemURL: string}>(`http://localhost:3000/api/clientes/${idCliente}`);
  }
```

- No método **ngOnInit** definido no arquivo **cliente.inserir.component.ts**, que é utilizado também para atualizações de dados, passaremos a utilizar a imagem devolvida pelo Back End. Note que, neste momento, apenas atribuímos o valor **null**. Veja a Listagem 2.2.8.

Listagem 2.2.8

```
ngOnInit(){
  ...
   this.route.paramMap.subscribe((paramMap: ParamMap) => {
    if (paramMap.has("idCliente")){
      this.modo = "editar";
      this.idCliente = paramMap.get("idCliente");
      this.estaCarregando = true;
     this.clienteService.getCliente(this.idCliente).subscribe( dadosCli => {
       this.estaCarregando = false;
       this.cliente = {
        id: dadosCli._id,
        nome: dadosCli.nome,
        fone: dadosCli.fone,
        email: dadosCli.email,
        imagemURL: dadosCli.imagemURL
       };
       this.form.setValue({
        nome: this.cliente.nome,
        fone: this.cliente.fone,
        email: this.cliente.email,
        imagem: this.cliente.imagemURL
       })
     });
    }
    else{
      this.modo = "criar";
      this.idCliente = null;
    }
   });
 }
```

- Se tentarmos editar um cliente agora fazendo uma submissão sem alterar a foto, o terminal em que o Node está executando deve exibir **undefined**. Isso ocorre pois a requisição não contém um arquivo, somente o seu endereço.

- Agora podemos montar a URL correta para a imagem a fim de armazená-la na base. Caso uma string tenha sido enviada, vamos apenas mantê-la. Caso contrário, temos um novo arquivo. Neste caso, vamos montar a URL como feito no método post. Veja a Listagem 2.2.9.

Listagem 2.2.9

```
router.put(
  "/:id",
multer({ storage: armazenamento }).single('imagem'),
(req, res, next) => {
  console.log (req.file);
  let imagemURL = req.body.imagemURL;//tentamos pegar a URL já existente
  if (req.file) { //mas se for um arquivo, montamos uma nova
    const url = req.protocol + "://" + req.get("host");
    imagemURL = url + "/imagens/" + req.file.filename;
  }
  const cliente = new Cliente({
    _id: req.params.id,
    nome: req.body.nome,
    fone: req.body.fone,
    email: req.body.email,
    imagemURL: imagemURL
  });
  Cliente.updateOne({ _id: req.params.id }, cliente)
    .then((resultado) => {
      //console.log(resultado)
      res.status(200).json({ mensagem: 'Atualização realizada com sucesso' })
    });
});
```

2.3 (Paginação) Paginação é um recurso muito comum e bastante utilizado. Quando temos uma página que exibe uma coleção de itens, é muito comum não desejarmos que todos os itens sejam carregados de uma única vez, especialmente quando a coleção pode ser potencialmente grande. Uma estratégia bastante utilizada consiste em exibir os itens da coleção em **páginas**. A primeira página exibe, digamos, 5 itens. A segunda página exibe os próximos cinco. E assim por diante. Além disso, os itens de uma página são trazidos da base sob demanda: os itens de uma página são trazidos da base somente quando o usuário navega até ela.

- O Angular Material possui um componente chamado **Paginator**. Veja a sua documentação no Link 2.3.1.

Link 2.3.1
https://material.angular.io/components/paginator/overview

- O primeiro passo para utilizá-lo, é importar o seu módulo no módulo principal de nossa aplicação, definido no arquivo **app.module.ts.** Veja a Listagem 2.3.1.

Listagem 2.3.1

```
import { MatPaginatorModule } from '@angular/material/paginator';
@NgModule({
  declarations: [
    AppComponent,
    ClienteInserirComponent,
    CabecalhoComponent,
    ClienteListaComponent,
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    ReactiveFormsModule,
    BrowserAnimationsModule,
    MatInputModule,
    MatCardModule,
    MatButtonModule,
    MatToolbarModule,
    MatExpansionModule,
    MatPaginatorModule,
    MatProgressSpinnerModule,
    HttpClientModule,
  ],
  providers: [],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

- Vamos adicionar um componente paginator no arquivo que define a lista de clientes, o **cliente.lista.component.html**. Ele ficará logo abaixo do **mat-accordion**. Note que vinculamos a sua propriedade **length** a uma variável a ser definida no arquivo **.ts** do componente. As listagens 2.3.2 e 2.3.3 mostram o uso do paginator e a definição da variável, respectivamente.

Listagem 2.3.2

```
<mat-spinner *ngIf="estaCarregando"></mat-spinner>
<mat-accordion *ngIf="clientes.length > 0 && !estaCarregando">
  <mat-expansion-panel *ngFor="let cliente of clientes">
    <mat-expansion-panel-header>Nome: {{ cliente.nome }}</mat-expansion-panel-header>
    <div class="cliente-imagem">
      <img [src]="cliente.imagemURL" [alt]="cliente.nome">
    </div>
    <p>Fone: {{ cliente.fone }}</p>
    <hr />
    <p>Email: {{ cliente.email }}</p>
    <mat-action-row>
      <a mat-button color="primary" [routerLink]="['/editar', cliente.id]">EDITAR</a>
      <button mat-button color="warn" (click)="onDelete(cliente.id)">REMOVER</button>
    </mat-action-row>
  </mat-expansion-panel>
</mat-accordion>
<mat-paginator [length]="totalDeClientes"></mat-paginator>
<p class="mat-body-1" style="text-align: center;" *ngIf="clientes.length <= 0 && !
estaCarregando">
  Nenhum cliente cadastrado
</p>
```

Listagem 2.3.3

```
...
@Component({
  selector: 'app-cliente-lista',
  templateUrl: './cliente-lista.component.html',
  styleUrls: ['./cliente-lista.component.css'],
})
export class ClienteListaComponent implements OnInit, OnDestroy {
  clientes: Cliente[] = [];
  private clientesSubscription: Subscription;
  public estaCarregando = false;
  totalDeClientes: number = 10;
...
```

- Vamos também alterar o número de clientes a serem exibidos por página. Veja as Listagens 2.3.4 e 2.3.5.

Listagem 2.3.4

```
<mat-paginator [length]="totalDeClientes" [pageSize]="totalDeClientesPorPagina"></mat-paginator>
```

Listagem 2.3.5

```
export class ClienteListaComponent implements OnInit, OnDestroy {
  clientes: Cliente[] = [];
  private clientesSubscription: Subscription;
  public estaCarregando = false;
  totalDeClientes: number = 10;
  totalDeClientesPorPagina: number = 2;
```

- A aplicação também irá oferecer um menu que permite ao usuário selecionar quantos itens ele deseja ver por página. O padrão é 2, mas talvez ele queira mudar. Começamos definindo uma coleção com as opções, como na Listagem 2.3.6.

Listagem 2.3.6

```
export class ClienteListaComponent implements OnInit, OnDestroy {
  clientes: Cliente[] = [];
  private clientesSubscription: Subscription;
  public estaCarregando = false;
  totalDeClientes: number = 10;
  totalDeClientesPorPagina: number = 2;
  opcoesTotalDeClientesPorPagina = [2, 5, 10];
```

- Para exibir as opções, vamos vincular a coleção à propriedade **pageSizeOptions** do paginator, como mostra a Listagem 2.3.7.

Listagem 2.3.7

```
<mat-paginator
    [length]="totalDeClientes"
    [pageSize]="totalDeClientesPorPagina"
    [pageSizeOptions]="opcoesTotalDeClientesPorPagina"
></mat-paginator>
```

- Até o momento, temos somente a parte visual do paginator em funcionamento. Desejamos detectar eventos gerados por ele para executar ações. Para isso, podemos fazer um **event binding** em **page** e chamar um método que recebe como parâmetro o objeto gerado pelo paginator, que pode ser referenciado por **$event**. Veja a Listagem 2.3.8. Note que o método ainda não existe e será definido em breve.

Listagem 2.3.8

```
<mat-paginator
   [length]="totalDeClientes"
   [pageSize]="totalDeClientesPorPagina"
   [pageSizeOptions]="opcoesTotalDeClientesPorPagina"
   (page)="onPaginaAlterada($event)"
></mat-paginator>
```

- A Listagem 2.3.9 mostra a definção do método onPaginaAlterada. Ele recebe um objeto do tipo **PageEvent**. No momento estamos apenas exibindo este objeto no console a fim de conhecer a sua estrutura.

Listagem 2.3.9

```
import { PageEvent } from '@angular/material/paginator';

onPaginaAlterada (dadosPagina: PageEvent){
   console.log (dadosPagina);
}
```

- Interaja com o componente alterando o número de itens que ele mostra por página bem como alterando a página exibida. Averigue o conteúdo exibido no console do Chrome (CTRL + SHIFT + I, aba console).

- Defina também o seletor CSS da Listagem 2.3.10, no arquivo **cliente-lista.component.css**. Sua finalidade é definir margem entre a lista e o paginator.

Listagem 2.3.10

```
mat-paginator {
  margin-top: 1rem;
}
```

- O próximo passo é ajustar o método que busca todos os clientes no Back End. Trata-se de uma requisição GET do protocolo HTTP, o que quer dizer que podemos enviar dados como pares chave/valor na URL. No Back End, eles nos são entregues como um objeto JSON, na propriedade **query** do objeto **req**. Para entender seu funcionamento, adicione a linha destacada na Listagem 2.3.11 ao seu Back End (arquivo **backend/rotas/clientes.js**). A seguir, utilize o Link 2.3.1 em uma aba à parte de seu navegador para verificar o que ocorre no log do Node.

Listagem 2.3.11

```
router.get('', (req, res, next) => {
  console.log (req.query);
  Cliente.find().then(documents => {
    //console.log(documents)
    res.status(200).json({
      mensagem: "Tudo OK",
      clientes: documents
    });
  })
});
```

Link 2.3.1
http://localhost:3000/api/clientes?pagesize=4&page=1&qualquercoisa=oi

- O método **get** do Back End irá utilizar os parâmetros **pagesize** e **page** para especificar ao MongoDB exatamente quais dados devem ser traziados. Se cada página exibe **pagesize** itens e desejamos os itens da página **page**, então:

- Desejamos ignorar (skip) os primeiros **pagesize * (page – 1)** itens
- Desejamos limitar a busca a **page** itens.

Veja os ajustes na Listagem 2.3.12.

Listagem 2.3.12

```
router.get('', (req, res, next) => {
  //console.log (req.query);
  const pageSize = +req.query.pagesize;
  const page = +req.query.page;
  const consulta = Cliente.find();//só executa quando chamamos then
  if (pageSize && page){
    consulta
      .skip(pageSize * (page - 1))
      .limit(pageSize);
  }
  consulta.then(documents => {
    //console.log(documents)
    res.status(200).json({
      mensagem: "Tudo OK",
      clientes: documents
    });
  })
});
```

- Faça novos testes alterando o valor de pagesize e page na query, em seu navegador.

- O próximo passo é fazer com o que a aplicação Angular utilize o mecanismo de paginação. Para isso, basta que ela envie valores de pagesize e page ao fazer a busca pelos clientes. Essa busca é feita no método **getClientes**, definido no arquivo **clientes.service.ts**. Ele deve receber como parâmetro os valores desejados e encaixá-los como parâmetros da query que envia ao Back End. Veja a Listagem 2.3.13.

Listagem 2.3.13

```
getClientes(pagesize: number, page: number): void {
   const parametros = `?pagesize=${pagesize}&page=${page}`;
   this.httpClient.get <{mensagem: string, clientes: any}>('http://localhost:3000/api/clientes' +
parametros)
   .pipe(map((dados) => {
     return dados.clientes.map(cliente => {
       return {
         id: cliente._id,
         nome: cliente.nome,
         fone: cliente.fone,
         email: cliente.email,
         imagemURL: cliente.imagemURL
       }
     })
   }))
   .subscribe(
     (clientes) => {
       this.clientes = clientes;
       this.listaClientesAtualizada.next([...this.clientes]);
     }
   )
  }
```

- A seguir, precisamos ajustar o método **ngOnInit**, definido no arquivo **cliente-lista.component.ts**. Ele precisa enviar os valores para o método **getClientes**. Felizmente é ele quem os conhece, por ter acesso direto ao paginator. Veja a Listagem 2.3.14.

Listagem 2.3.14

```
paginaAtual: number = 1; //definir

ngOnInit(): void {
   this.estaCarregando = true;
   this.clienteService.getClientes(this.totalDeClientesPorPagina, this.paginaAtual);
   this.clientesSubscription = this.clienteService
     .getListaDeClientesAtualizadaObservable()
     .subscribe((clientes: Cliente[]) => {
       this.estaCarregando = false;
       this.clientes = clientes;
     });
  }
```

- Quando o paginator sofre alterações, precisamos realizar nova busca no Back End para atualizar os dados adequadamente. O método **onPaginaAlterada** entra em execução toda vez que um evento acontece. Neste momento podemos chamar o método getClientes novamente especificando os valores adequadamente. VamosVeja a Listagem 2.3.15.

Listagem 2.3.15

```
onPaginaAlterada (dadosPagina: PageEvent){
   //console.log (dadosPagina);
   this.paginaAtual = dadosPagina.pageIndex + 1; //no paginator a contagem começa de 0
   this.totalDeClientesPorPagina = dadosPagina.pageSize;
   this.clienteService.getClientes (this.totalDeClientesPorPagina, this.paginaAtual);

}
```

- Faça novos testes a partir da aplicação Angular alterando a página atual e a quantidade de itens por página.

- Vamos ajustar o componente de carregamento para que ele apareça momentamente quando ocorrer um novo evento emitido pelo paginator. Toda vez que isso acontece, as variáveis são alteradas e o método **getClientes** entra em execução novamente. Ele se encarrega de fazer com que o componente de carregamento desapareça. Veja a Listagem 2.3.16.

Listagem 2.3.16

```
onPaginaAlterada (dadosPagina: PageEvent){
   //console.log (dadosPagina);
   this.estaCarregando = true;
   this.paginaAtual = dadosPagina.pageIndex + 1; //no paginator a contagem começa de 0
   this.totalDeClientesPorPagina = dadosPagina.pageSize;
   this.clienteService.getClientes (this.totalDeClientesPorPagina, this.paginaAtual);

 }
```

- Desejamos que o paginator apareça somente se o accordion estiver sendo exibido pois se ele não estiver, quer dizer que não há clientes e, portanto, nenhuma razão para exibir um paginator. Para isso, basta aplicar uma diretiva estrutural ***ngIf** ao paginator envolvendo a quantidade de clientes. Veja a Listagem 2.3.17.

Listagem 2.3.17

```
<mat-spinner *ngIf="estaCarregando"></mat-spinner>
<mat-accordion *ngIf="clientes.length > 0 && !estaCarregando">
  <mat-expansion-panel *ngFor="let cliente of clientes">
    <mat-expansion-panel-header>Nome: {{ cliente.nome }}</mat-expansion-panel-header>
    <div class="cliente-imagem">
      <img [src]="cliente.imagemURL" [alt]="cliente.nome">
    </div>
    <p>Fone: {{ cliente.fone }}</p>
    <hr />
    <p>Email: {{ cliente.email }}</p>
    <mat-action-row>
      <a mat-button color="primary" [routerLink]="['/editar', cliente.id]">EDITAR</a>
      <button mat-button color="warn" (click)="onDelete(cliente.id)">REMOVER</button>
    </mat-action-row>
  </mat-expansion-panel>
</mat-accordion>
<mat-paginator
    *ngIf="clientes.length > 0"
    [length]="totalDeClientes"
    [pageSize]="totalDeClientesPorPagina"
    [pageSizeOptions]="opcoesTotalDeClientesPorPagina"
    (page)="onPaginaAlterada($event)"
></mat-paginator>
<p class="mat-body-1" style="text-align: center;" *ngIf="clientes.length <= 0 && !
estaCarregando">
  Nenhum cliente cadastrado
</p>
```

- Veja que há questões a serem resolvidas envolvendo a quantidade total de clientes e eventos como a remoção. Eles serão tratados na próxima aula.

Referências

Angular. 2020. Disponível em <https://angular.io>. Acesso em agosto de 2020.

Angular Material UI component library. 2020. Disponível em <https://material.angular.io>. Acesso em agosto de 2020

Express - Node.js web application framework. 2020. Disponível em <https://expressjs.com>. Acesso em agosto de 2020.

Node.js. 2020. Disponível em <https://nodejs.org/en/>. Acesso em agosto de 2020.

The most popular database for modern apps | MongoDB. 2020. Disponível em <https://www.mongodb.com>. Acesso em agosto de 2020.