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

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

- 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 (Tratando os estilos dos links) Após a adição do roteador Angular, temos dois links no cabeçalho ainda não estilizados. Vamos tratar deles agora.

- Caso ainda não tenha, crie um arquivo chamado **cabecalho.component.css** na pasta **src/app/cabecalho**.

- A seguir, vincule-o ao componente como mostra a Listagem 2.2.1. Estamos no arquivo **cabecalho.component.ts**.

Listagem 2.2.1

```
Component({
  selector: 'app-cabecalho',
  templateUrl: './cabecalho.component.html',
  styleUrls: ['./cabecalho.component.css']
})
export class CabecalhoComponent implements OnInit {

  constructor() { }

  ngOnInit(): void {
  }

}
```

- Vamos remover a sublinhado dos links e alterar a sua cor. Além disso, vamos tirar o estilo dos itens da lista (as bolinhas) e remover padding e margem. Veja a Listagem 2.2.2. Estamos no arquivo **cabecalho.component.css**.

Listagem 2.2.2

```
ul {
  list-style: none;
  padding: 0;
  margin: 0;
}
a {
  text-decoration: none;
  color: white;
}
```

- Podemos deixar o link **Clientes** à esquerda e as demais opções (por enquanto somente **Novo Cliente**) à direita adicionando um elemento **span** entre eles e ajustando o seu display com o valor **flex**. As listagens 2.2.3 e 2.2.4 mostram os ajustes nos arquivos **cabecalho.component.html** e **cabecalho.component.css** respectivamente.

Listagem 2.2.3

```
<mat-toolbar color="primary">
  <span><a routerLink="/">Clientes</a></span>
  <span class="separador"></span>
  <ul>
    <li><a routerLink="/criar">Novo Cliente</a></li>
  </ul>
</mat-toolbar>
```

Listagem 2.2.4

```
ul {
  list-style: none;
  padding: 0;
  margin: 0;
}
a {
  text-decoration: none;
  color: white;
}

.separador{
  flex: 1;
}
```

- Aplicamos também a diretiva **mat-button** ao botão de inserção de clientes, como na Listagem 2.2.5.

Listagem 2.2.5

```
<mat-toolbar color="primary">
  <span><a routerLink="/">Clientes</a></span>
  <span class="separador"></span>
  <ul>
    <li><a mat-button routerLink="/criar">Novo Cliente</a></li>
  </ul>
</mat-toolbar>
```

- Quando navegamos para a página de inserção de clientes, pode ser de interesse manter o botão de inserção em destaque. Podemos fazê-lo usando um recurso do próprio roteador Angular. Basta usar a propriedade **routerLinkActive**, como na Listagem 2.2.6.

Listagem 2.2.6

```
<mat-toolbar color="primary">
  <span><a routerLink="/">Clientes</a></span>
  <span class="separador"></span>
  <ul>
    <li><a mat-button routerLink="/criar" routerLinkActive="mat-accent">Novo
Cliente</a></li>
  </ul>
</mat-toolbar>
```

- Para que as cores fiquem condizentes, no arquivo **cliente-inserir.component.html**, troque a cor do botão para **accent**, como na Listagem 2.2.7.

Listagem 2.2.7

```
<mat-card>
  <form (submit)="onAdicionarCliente(clienteForm)" #clienteForm="ngForm">
    <mat-form-field>
      <input
        required
        minlength="4"
        type="text"
        matInput
        placeholder="nome"
        name="nome"
        ngModel
        #nome="ngModel"
      />
      <mat-error *ngIf="nome.invalid">Digite um nome válido</mat-error>
    </mat-form-field>
    <mat-form-field>
      <input
        required
        type="text"
        matInput
        placeholder="fone"
        name="fone"
        ngModel
        #fone="ngModel"
      />
      <mat-error *ngIf="fone.invalid">Digite um telefone válido</mat-error>
    </mat-form-field>
    <mat-form-field>
      <input
        required
        type="text"
```

```
      matInput
      placeholder="email"
      name="email"
      ngModel
      #email="ngModel"
    />
    <mat-error *ngIf="email.invalid">Digite um nome válido</mat-error>
  </mat-form-field>
  <button color="accent" mat-raised-button>
    Inserir Cliente
  </button>
 </form>
</mat-card>
```

2.3 (Lidando com a atualização de clientes) Nosso CRUD de clientes já permite a inserção de clientes, porém não é possível fazer atualizações e remoções. A seguir, trataremos das atualizações.

- Comece adicionando uma nova rota no arquivo **app-routing.module.ts**. A rota será **editar** e o componente a ser renderizado é o mesmo utilizado para a criação de novos clientes. Contudo, será necessário especificar o id do cliente cujos dados deverão ser exibidos e atualizados. Para isso, utilizaremos uma notação própria do Angular que permite a especificação de trechos que serão informados dinamicamente e que podem ser extraídos das rotas. Veja a Listagem 2.3.1.

Listagem 2.3.1

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

import { ClienteListaComponent } from './clientes/cliente-lista/cliente-lista.component';
import { ClienteInserirComponent } from './clientes/cliente-inserir/cliente-inserir.component';
const routes: Routes = [
  { path: '', component: ClienteListaComponent },
  {path: 'criar', component: ClienteInserirComponent},
  {path: 'editar/:idCliente', component: ClienteInserirComponent}
];
@NgModule({
  imports: [
    RouterModule.forRoot(routes)
  ],
  exports: [
    RouterModule
  ]

})
export class AppRoutingModule{

}
```

- Como estamos utilizando o mesmo componente para inserção e para atualização de clientes, precisamos de algum modo diferenciar qual operação está sendo utilizada. Para isso, vamos injetar no construtor do componente de inserção de cliente (arquivo **cliente-inserir.component.ts**) um objeto do tipo **ActivatedRoute**. Ele representa a rota que foi utilizada para que o componente fosse renderizado e, entre outras coisas, permite a obtenção dos parâmetros incluídos nela dinamicamente, caso eles existam. Veja a Listagem 2.3.2.

Listagem 2.3.2

```
import { Component } from '@angular/core';

import { NgForm } from '@angular/forms';
import { ActivatedRoute } from '@angular/router';
import { ClienteService } from '../clientes.service';
@Component({
  selector: 'app-cliente-inserir',
  templateUrl: './cliente-inserir.component.html',
  styleUrls: ['./cliente-inserir.component.css'],
})
export class ClienteInserirComponent {
  constructor(public clienteService: ClienteService, public route: ActivatedRoute) {}

  onAdicionarCliente(form: NgForm) {
    if (form.invalid) {
      return;
    }

    this.clienteService.adicionarCliente(
      form.value.nome,
      form.value.fone,
      form.value.email
    );
    form.resetForm();
  }
}
```

- Segundo o ciclo de vida de componentes Angular, o método **ngOnInit** da interface **OnInit** é executado automaticamente uma vez que um componente tenha sido instanciado pelo Angular. Vamos utilizá-lo para extrair informações da rota que foi injetada pelo Angular no construtor. Comece implementando a interface e definindo o método, como na Listagem 2.3.3.

Listagem 2.3.3

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

import { NgForm } from '@angular/forms';
import { ActivatedRoute } from '@angular/router';
import { ClienteService } from '../clientes.service';
@Component({
  selector: 'app-cliente-inserir',
  templateUrl: './cliente-inserir.component.html',
  styleUrls: ['./cliente-inserir.component.css'],
})
export class ClienteInserirComponent implements OnInit {

  ngOnInit(){

  }
  constructor(public clienteService: ClienteService, public route: ActivatedRoute) {}

  onAdicionarCliente(form: NgForm) {
    if (form.invalid) {
      return;
    }

    this.clienteService.adicionarCliente(
      form.value.nome,
      form.value.fone,
      form.value.email
    );
    form.resetForm();
  }
}
```

- A seguir, no método ngOnInit, tente pegar o parâmetro chamado **idCliente** (lembra que ele foi definido no objeto de rotas?). Note que a rota possui um mapa de parâmetros no qual podemos registrar uma função que entra em execução uma vez que a rota tenha sido ativada (acessada pelo usuário). Veja a Listagem 2.3.4.

Listagem 2.3.4

```
ngOnInit(){
   this.route.paramMap.subscribe((paramMap: ParamMap) => {
     if (paramMap.has("idCliente")){

     }
   });
 }
```

- A informação sobre qual operação está sendo realizada será armazenada em uma variável de instância. Ela, por padrão, armazena o texto “criar”. Se o parâmetro **idCliente**, existir no mapa de parâmetros da rota que fez com que o componente fosse renderizado, então seu valor passará a ser “editar”. Veja a Listagem 2.3.5.

Listagem 2.3.5

```
private modo: string = "criar";

 ngOnInit(){
   this.route.paramMap.subscribe((paramMap: ParamMap) => {
     if (paramMap.has("idCliente")){
       this.modo = "editar";
     }
     else{
       this.modo = "criar";
     }
   });
 }
```

- Caso o id de cliente exista, ele também será armazenado em uma variável da classe, assim podemos realizar a atualização. Veja a Listagem 2.3.6.

Listagem 2.3.6

```
private modo: string = "criar";
 private idCliente: string;

 ngOnInit(){
  this.route.paramMap.subscribe((paramMap: ParamMap) => {
    if (paramMap.has("idCliente")){
      this.modo = "editar";
      this.idCliente = paramMap.get("idCliente");
    }
    else{
      this.modo = "criar";
      this.idCliente = null;
    }
  });
 }
```

- Com o id do cliente a ser atualizado em mãos, precisamos obter seus dados na coleção de clientes para que eles possam ser exibidos e editados. Para isso, vamos criar um método na classe **ClienteService** (arquivo **clientes.service.ts**) que recebe o id do cliente e devolve uma cópia do objeto armazenado na coleção. Neste momento, estamos consultando a coleção existente no lado do cliente, não disparamos uma nova consulta ao servidor, embora também seja possível e talvez necessário em alguns cenários. Veja a Listagem 2.3.7.

Listagem 2.3.7

```
getCliente (idCliente: string){
   return {...this.clientes.find((cli) => cli.id === idCliente)};
 }
```

- Podemos agora, no componente **ClienteInserirComponent** (arquivo **cliente-inserir.component.ts**) chamar o método getCliente do serviço quando detectarmos que a operação sendo realizada é a operação de atualização de dados. Pode ser de interesse armazenar seus dados em uma variável de instância. Veja a Listagem 2.3.8.

Listagem 2.3.8

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

import { NgForm } from '@angular/forms';
import { ActivatedRoute, ParamMap } from '@angular/router';
import { Cliente } from '../cliente.model';
import { ClienteService } from '../clientes.service';
@Component({
  selector: 'app-cliente-inserir',
  templateUrl: './cliente-inserir.component.html',
  styleUrls: ['./cliente-inserir.component.css'],
})
export class ClienteInserirComponent implements OnInit {

  private modo: string = "criar";
  private idCliente: string;
  public cliente: Cliente;

  ngOnInit(){
    this.route.paramMap.subscribe((paramMap: ParamMap) => {
      if (paramMap.has("idCliente")){
        this.modo = "editar";
        this.idCliente = paramMap.get("idCliente");
        this.cliente = this.clienteService.getCliente(this.idCliente);
      }
      else{
        this.modo = "criar";
        this.idCliente = null;
      }
    });
  }
```

- A seguir, no arquivo **cliente-lista.component.html**, ajustamos o botão de atualização de clientes (ele passará a ser um link para que o roteador possa ser usado) para que ele acesse a rota de atualização que acabamos de configurar. Note que fazemos uma espécie de **property binding** na propriedade **routerLink** e entregamos para ela um vetor com os trechos (estáticos e dinâmicos) que compõem a rota que desejamos. Veja a Listagem 2.3.9.

Listagem 2.3.9

```
<mat-accordion *ngIf="clientes.length > 0">
  <mat-expansion-panel *ngFor="let cliente of clientes">
    <mat-expansion-panel-header>Nome: {{ cliente.nome }}</mat-expansion-panel-header>
    <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>
<p class="mat-body-1" style="text-align: center;" *ngIf="clientes.length <= 0">
  Nenhum cliente cadastrado
</p>
```

- No template do componente **ClienteInserirComponent** (arquivo **cliente-inserir.component.html**), utilizamos a diretiva **ngModel** para indicar ao Angular os campos que desejávamos que fossem incluídos no objeto JSON criado por ele uma vez que interceptasse a submissão do form. Faremos um **property binding** nesta mesma diretiva para dizer que ele deve exibir os valores existentes nas propriedades do objeto cliente que adicionamos ao componente. Ele existirá somente para os casos em que o componente for renderizado para atualização de dados, justamente o que precisamos. Veja a Listagem 2.3.10. Note o uso do operador **?**. Com ele, as propridades nome, fone e e-mail somente serão acessadas se cliente for diferente de null e diferente de undefined.

Listagem 2.3.10

```
<mat-card>
  <form (submit)="onAdicionarCliente(clienteForm)" #clienteForm="ngForm">
    <mat-form-field>
      <input
        required
        minlength="4"
        type="text"
        matInput
        placeholder="nome"
        name="nome"
        [ngModel]="cliente?.nome"
        #nome="ngModel"
      />
      <mat-error *ngIf="nome.invalid">Digite um nome válido</mat-error>
    </mat-form-field>
    <mat-form-field>
      <input
        required
        type="text"
        matInput
        placeholder="fone"
        name="fone"
        [ngModel]="cliente?.fone"
        #fone="ngModel"
      />
      <mat-error *ngIf="fone.invalid">Digite um telefone válido</mat-error>
    </mat-form-field>
    <mat-form-field>
      <input
        required
        type="text"
        matInput
        placeholder="email"
        name="email"
        [ngModel]="cliente?.email"
        #email="ngModel"
      />
      <mat-error *ngIf="email.invalid">Digite um nome válido</mat-error>
    </mat-form-field>
    <button color="accent" mat-raised-button>
      Inserir Cliente
    </button>
  </form>
</mat-card>
```

- Não se esqueça de testar a aplicação. Faça a inserção de um novo cliente, exiba a lista e depois clique no botão editar.

- A seguir, iremos implementar um método no serviço de manipulação de clientes (arquivo **clientes.service.ts**) para que seja possível enviar uma requisição ao servidor, solicitando que os dados de um cliente sejam atualizados. Sua definição é dada na Listagem 2.3.11. Ele recebe os dados de um cliente e faz uma requisição utilizando o método **PUT** do protocolo HTTP.

Listagem 2.3.11

```
atualizarCliente (id: string, nome: string, fone: string, email: string){
   const cliente: Cliente = { id, nome, fone, email};
   this.httpClient.put(`http://localhost:3000/api/clientes/${id}`, cliente)
   .subscribe((res => console.log (res)));
  }
```

- O método atualizarCliente que acabamos de criar precisa ser chamado no componente de inserção de clientes (arquivo **cliente-inserir.component.ts**). Veja a Listagem 2.3.12.

Listagem 2.3.12

```
onAdicionarCliente(form: NgForm) {
   if (form.invalid) {
     return;
   }
   if (this.modo === "criar"){
     this.clienteService.adicionarCliente(
       form.value.nome,
       form.value.fone,
       form.value.email
     );
   }
   else{
     this.clienteService.atualizarCliente(
       this.idCliente,
       form.value.nome,
       form.value.fone,
       form.value.email
     )
   }

   form.resetForm();
 }
```

- Talvez você queira alterar o nome do método onAdicionarCliente para algo como onSalvarCliente já que agora o que ele faz é mais geral do que simplesmente salvar. Também pode ser de interesse alterar o texto do botão de inserção. As alterações precisam ser feitaa nos arquivos **cliente-inserir.component.ts** e **cliente-inserir.component.html**. Veja as listagens 2.3.13 e 2.3.14.

Listagem 2.3.13

```
<mat-card>
  <form (submit)="onSalvarCliente(clienteForm)" #clienteForm="ngForm">
    <mat-form-field>
      <input
        required
        minlength="4"
        type="text"
        matInput
        placeholder="nome"
        name="nome"
        [ngModel]="cliente?.nome"
        #nome="ngModel"
      />
      <mat-error *ngIf="nome.invalid">Digite um nome válido</mat-error>
    </mat-form-field>
    <mat-form-field>
      <input
        required
        type="text"
        matInput
        placeholder="fone"
        name="fone"
        [ngModel]="cliente?.fone"
        #fone="ngModel"
      />
      <mat-error *ngIf="fone.invalid">Digite um telefone válido</mat-error>
    </mat-form-field>
    <mat-form-field>
      <input
        required
        type="text"
        matInput
        placeholder="email"
        name="email"
        [ngModel]="cliente?.email"
        #email="ngModel"
      />
      <mat-error *ngIf="email.invalid">Digite um nome válido</mat-error>
    </mat-form-field>
    <button color="accent" mat-raised-button>
      Salvar Cliente
    </button>
  </form>
</mat-card>
```

Listagem 2.3.14

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

   form.resetForm();
 }
```

- A aplicação Angular faz a requisição via PUT, porém não há endpoint apropriado para esse requisição em nosso Back End. É preciso implementar um endpoint que recebe os dados a serem atualizados e acessa o MongoDB para fazer as atualizações, como a Listagem 2.3.15 exibe. Estamos no arquivo **app.js**.

Listagem 2.3.15

```
app.put ("/api/clientes/:id", (req, res, next) => {
  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'})

});
```

- Note que a atualização ainda não funciona. Isso ocorre pois o método PUT do protocolo HTTP ainda não foi liberado no cabeçalho CORS. Adicione-o à lista como na Listagem 2.3.16, ainda no arquivo **app.js**.

Listagem 2.3.16

```
app.use ((req, res, next) => {
  res.setHeader('Access-Control-Allow-Origin', "*");
  res.setHeader('Access-Control-Allow-Headers', 'Origin, X-Requested-With, Content-Type,
Accept');
  res.setHeader('Access-Control-Allow-Methods', 'GET, POST, PATCH, PUT, DELETE,
OPTIONS');

  next();
});
```

- Uma vez que a aplicação Angular receba a resposta do servidor referente a uma atualização feita com sucesso, podemos atualizar a coleção mantida por ela localmente. Veja a Listagem 2.3.17. Estamos no arquivo **clientes.service.ts**.

Listagem 2.3.17

```
atualizarCliente (id: string, nome: string, fone: string, email: string){
   const cliente: Cliente = { id, nome, fone, email};
   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]);
   }));
  }
```

- Na página de edição de clientes, quando clicamos em atualizar no navegador, a página reaparece com os campos todos vazios. Note, contudo ,que o id do cliente cujos dados estavam sendo exibidos ainda está disponível na URL. Isso quer dizer que podemos buscar seus dados no servidor usando seu id e manter seus dados na tela. Começamos implementando um novo endpoint no Back End. Ele recebe uma requisição GET envolvendo o id do cliente. Veja a Listagem 2.3.18. Estamos no arquivo **app.js**.

Listagem 2.3.18

```
app.get('/api/clientes/:id', (req, res, next) => {
  Cliente.findById(req.params.id).then(cli => {
    if (cli){
      res.status(200).json(cli);
    }
    else
      res.status(404).json({mensagem: "Cliente não encontrado!"})
  })
});
```

- A seguir, vamos atualizar o método getCliente no arquivo **clientes.service.ts**. Ele fará uma requisição get para se comunicar com o endpoint que acabamos de criar. Porém, ele passará a devolver um Observable, já que a requisição é assíncrona. Veja a Listagem 2.3.19.

Listagem 2.3.19

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

- A seguir, chamamos o método subscribe no componente de inserção/atualização de clientes, no arquivo **cliente-inserir.component.ts**. Veja a Listagem 2.3.20.

Listagem 2.3.20

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

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.