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 (Ajustando as rotas no Back End) Note que nosso arquivo **app.js** já possui uma quantidade considerável de linhas de código. Manter um arquivo muito grande tende a comprometer a manutenabilidade do sistema a longo prazo. Por isso, vamos refatorar o Back End da seguinte forma.

- Crie uma nova subpasta de **backend** com o nome **rotas**. Ela fica **lado a lado** com a pasta **models.**

- Crie um novo **arquivo** chamado **clientes.js** na pasta **rotas**. Ele será responsável por abrigar as definições dos endpoints que lidam com requisições envolvendo clientes.

- No arquivo **clientes.js** vamos utilizar a funcionalidade de roteamento do Express. Para isso, importe o express e construa um objeto com o método Router, como na listagem 2.2.1.

Listagem 2.2.1

```
const express = require ("express");
const router = express.Router();
```

- A seguir, recorte todos os endpoints (app.get, app.post etc) do arquivo **app.js** e leve-os para o arquivo **clientes.js**. A seguir, substitua a palavra **app** por **router**. Veja a Listagem 2.2.2. Note que o **app.use não deve ser envolvido, ele permanece onde está.**

Listagem 2.2.2

```
const express = require ("express");
const router = express.Router();
router.post('/api/clientes', (req, res, next) => {
  const cliente = new Cliente({
    nome: req.body.nome,
    fone: req.body.fone,
    email: req.body.email
  })
  cliente.save().
    then(clienteInserido => {
      res.status(201).json({
        mensagem: 'Cliente inserido',
        id: clienteInserido._id
      })
    })
});
router.get('/api/clientes', (req, res, next) => {
  Cliente.find().then(documents => {
    console.log(documents)
    res.status(200).json({
      mensagem: "Tudo OK",
      clientes: documents
    });
  })
});
router.delete('/api/clientes/:id', (req, res, next) => {
  console.log("id: ", req.params.id);
  Cliente.deleteOne({ _id: req.params.id }).then((resultado) => {
    console.log(resultado);
    res.status(200).json({ mensagem: "Cliente removido" })
  });
});









router.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' })
    });
});

router.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!" })
  })
});
```

- Para que a funcionalidade de roteamento possa ser utilizada, ela precisa ser exportada do seu módulo. Adicione, portanto, o código da Listagem 2.2.3 ao final do arquivo **clientes.js**.

Listagem 2.2.3

```
module.exports = router;
```

- Leve também a importação do modelo Cliente do arquivo **app.js** para o arquivo **clientes.js**, como na Listagem 2.2.4. Note, contudo, que **o local relativo mudou**. Precisamos “sair” da pasta de rotas para entrar na pasta de modelos e acessar o arquivo cliente, de modelo.

Listagem 2.2.4

```
const express = require ("express");
const router = express.Router();
const Cliente = require('../models/cliente');
.
.
.
```

- O arquivo **app.js** pode importar aquilo que é exportado pelo arquivo em que definimos o roteador. Isso pode ser feito como exibe a Listagem 2.2.5.

Listagem 2.2.5

```
const express = require ('express');
```

```
const app = express();
const bodyParser = require ('body-parser');
const mongoose = require ('mongoose');
const clienteRoutes = require ('./rotas/clientes');
```

- Após isso, basta chamar o método **use de app**, entregando a ele o objeto clienteRoutes. Isso deve ser feito logo depois da função em que lidamos com os aspectos de CORS. Veja a Listagem 2.2.6. Estamos no arquivo **app.js**.

Listagem 2.2.6

```
const express = require ('express');
const app = express();
const bodyParser = require ('body-parser');
const mongoose = require ('mongoose');
const clienteRoutes = require ('./rotas/clientes');
mongoose.connect('mongodb+srv://user_maua:senha_maua@cluster0.ssm0w.mongodb.net/
pessoal-cliente?retryWrites=true&w=majority')
.then(() => {
  console.log ("Conexão OK")
}).catch((e) => {
  console.log("Conexão NOK: " + e)
});
app.use (bodyParser.json());

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();
});

app.use (clienteRoutes);

module.exports = app;
```

- Note que a string **/api/clientes**, no arquivo **clientes.js**, se repete para os endpoints. Podemos escrevê-la uma única vez no arquivo **app.js**, no momento em que configuramos o uso do roteador. A seguir, removemos a repetição do arquivo **clientes.js**. Veja como fica o arquivo **app.js** na Listagem 2.2.7.

Listagem 2.2.7

```
const express = require ('express');
```

```
const app = express();
const bodyParser = require ('body-parser');
const mongoose = require ('mongoose');
const clienteRoutes = require ('./rotas/clientes');


mongoose.connect('mongodb+srv://user_maua:senha_maua@cluster0.ssm0w.mongodb.net/
pessoal-cliente?retryWrites=true&w=majority')
.then(() => {
  console.log ("Conexão OK")
}).catch((e) => {
  console.log("Conexão NOK: " + e)
});
app.use (bodyParser.json());
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();
});

app.use ('/api/clientes', clienteRoutes);


module.exports = app;
```

- A Listagem 2.2.8 mostra como podemos remover os trechos repetidos do arquivo **rotas.js**.

Listagem 2.2.8

```
const express = require ("express");
const router = express.Router();
const Cliente = require('../models/cliente');

router.post('', (req, res, next) => {
  const cliente = new Cliente({
    nome: req.body.nome,
    fone: req.body.fone,
    email: req.body.email

  })
  cliente.save().
    then(clienteInserido => {
      res.status(201).json({
        mensagem: 'Cliente inserido',
        id: clienteInserido._id
      })
    })
});


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

router.delete('/:id', (req, res, next) => {
  console.log("id: ", req.params.id);
  Cliente.deleteOne({ _id: req.params.id }).then((resultado) => {
    console.log(resultado);
    res.status(200).json({ mensagem: "Cliente removido" })
  });
});

router.put("/: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' })
    });


});

router.get('/: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!" })
  })
});

module.exports = router;
```

2.3 (Ajustes no Front End: Levando o usuário para a listagem após uma inserção e uso de um componente de "loading"). Quando o usuário faz uma inserção de cliente, ele permanece na mesma tela. É natural esperar que ele seja levado para a tela de listagem após clicar no botão de inserção. Além disso, é bastante útil dar a ele um feedback visual que indica que a operação está sendo realizada caso ela demore um pouco.

- Para fazer a navegação automática, começaremos injetando o roteador do Angular (dessa vez, no lado do cliente, perceba) no serviço de manipulação de clientes (arquivo **clientes.service.ts**). Veja a Listagem 2.3.1.

Listagem 2.3.1

```
import { Cliente } from './cliente.model';
import { Injectable } from '@angular/core';
import { Subject } from 'rxjs';
import { HttpClient } from '@angular/common/http';
import { map } from 'rxjs/operators';
import { Router } from '@angular/router';
@Injectable({ providedIn: 'root' })
export class ClienteService {

  private clientes: Cliente[] = [];
  private listaClientesAtualizada = new Subject<Cliente[]>();

  constructor (private httpClient: HttpClient, private router: Router){

  }
```

- A seguir, nos métodos **adicionarCliente** e **atualizarCliente**, responsáveis pela inserção e atualização de clientes, vamos usar o método **navigate** do roteador para levar o usuário para a página que desejamos. Ele recebe um vetor de parâmetros. Utilizaremos somente um elemento no vetor, indicando que o usuário deve ser levado para a raiz da aplicação. Ela está mapeada para o componente de listagem, por isso ele será renderizado no router-outlet. Veja a Listagem 2.3.2.

Listagem 2.3.2

```
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]);
      this.router.navigate(['/'])
    }));
  }
```

Listagem 2.3.3

```
adicionarCliente(nome: string, fone: string, email: string) {
   const cliente: Cliente = {
     id: null,
     nome: nome,
     fone: fone,
     email: email,
   };
   this.httpClient.post<{mensagem: string, id: string}> ('http://localhost:3000/api/clientes',
cliente).subscribe(
     (dados) => {
       cliente.id = dados.id;
       this.clientes.push(cliente);
       this.listaClientesAtualizada.next([...this.clientes]);
       this.router.navigate(['/']);
     }
   )
 }
```

- Agora desejamos colocar um componente de progresso enquanto a navegação de uma página a outra acontece. Visite o Link 2.3.1 e veja a documentação do Angular Material. Veja o componente **Progress Spinner**.

Link 2.3.1

https://material.angular.io/components/progress-spinner/overview

- O primeiro passo para utilizá-lo é importar o módulo em que ele é definido. Faremos isso no arquivo **app.module.ts**. Veja a Listagem 2.3.4.

Listagem 2.3.4

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
import { HttpClientModule} from '@angular/common/http'
import { MatInputModule } from '@angular/material/input';
import { MatCardModule } from '@angular/material/card';
import { MatButtonModule } from '@angular/material/button';
import { MatToolbarModule } from '@angular/material/toolbar';
import { MatExpansionModule } from '@angular/material/expansion';
import { MatProgressSpinnerModule } from '@angular/material/progress-spinner'
import { AppComponent } from './app.component';
import { ClienteInserirComponent } from './clientes/cliente-inserir/cliente-inserir.component';
import { CabecalhoComponent } from './cabecalho/cabecalho.component';
import { ClienteListaComponent } from './clientes/cliente-lista/cliente-lista.component';

import { AppRoutingModule } from './app-routing.module';
@NgModule({
  declarations: [
    AppComponent,
    ClienteInserirComponent,
    CabecalhoComponent,
    ClienteListaComponent,
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule,
    BrowserAnimationsModule,
    MatInputModule,
    MatCardModule,
    MatButtonModule,
    MatToolbarModule,
    MatExpansionModule,
    MatProgressSpinnerModule,
    HttpClientModule,
  ],
  providers: [],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

- Desejamos que o componente seja exibido quando a busca por um cliente se inicia e que ele desapareça assim que ela terminar. No componente ClienteInserirComponent (arquivo **cliente-inserir.component.ts**) vamos adicionar uma variável booleana que indica se o componente deve ser exibido ou não. Ela começa com o valor **false**. O valor **true** é atribuído a ela quando a busca começa. Ela voltar a conter **false** quando a busca termina. Veja a Listagem 2.3.5.

Listagem 2.3.5

```
export class ClienteInserirComponent implements OnInit {

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

 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
     };
    });
   }
   else{
    this.modo = "criar";
    this.idCliente = null;
   }
  });
 }
 constructor(public clienteService: ClienteService, public route: ActivatedRoute) {}

 onSalvarCliente(form: NgForm) {
  if (form.invalid) {
   return;
  }
  this.estaCarregando = true;
  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();
 }
}
```

- Agora podemos utilizar o valor dessa variável para decidir se o componente de carregamento deve ser exibido ou não. Isso deve ser feito no arquivo **cliente-inserir.component.html**. Veja a Listagem 2.3.6. Note que a renderização do form também se torna condicional. Ele somente é exibido caso o componente de carregamento não seja.

Listagem 2.3.6

```
<mat-card>
 <mat-spinner *ngIf="estaCarregando"></mat-spinner>
 <form (submit)="onSalvarCliente(clienteForm)" *ngIf="!estaCarregando"
#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>
```

Nota: É possível que o componente não seja visto caso as operações de inserção e atualização executem muito rápido.

- Também podemos centralizá-lo aplicando uma regra CSS, que deve ser escrita no arquivo **cliente-inserir.component.css**. Veja a Listagem 2.3.7.

Listagem 2.3.7

```
mat-form-field,
input {
  width: 100%;
}

mat-spinner {
  margin: auto;
}
```

- Também podemos adicionar um desses ao componente de listagem de clientes. Vamos utilizar a mesma lógica. Abra o arquivo **cliente-lista.component.ts** e faça os ajustes da Listagem 2.3.8.

Listagem 2.3.8

```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { Cliente } from '../cliente.model';
import { ClienteService } from '../clientes.service';
import { Subscription, Observable } from 'rxjs';

@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;

  constructor(public clienteService: ClienteService) {}

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

  onDelete (id: string): void{
    this.clienteService.removerCliente(id);
  }
```

```
  ngOnDestroy(): void {
    this.clientesSubscription.unsubscribe();
  }
}
```

- O template (arquivo **cliente-lista.component.html**) fica conforme exibe a Listagem 2.3.9.

Listagem 2.3.9

```
<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>
    <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 && !
estaCarregando">
  Nenhum cliente cadastrado
</p>
```

- Também podemos centralizá-lo utilizando uma regra idêntica à anterior, desta vez escrita no arquivo **cliente-lista.component.css**. Veja a Listagem 2.3.10.

Listagem 2.3.10

```
:host {
  margin-top: 1rem;
  display: block;
}

mat-spinner {
  margin: auto;
}
```

2.4 (Adicionando um botão para upload de imagens) Para cada cliente cadastrado, vamos permitir que uma imagem seja associada. Para isso, cada um deles irá exibir um botão que permite que o usuário selecione uma foto que deseja utilizar.

- Começamos adicionando um botão no template do componente de inserção de clientes (arquivo **cliente-inserir.component.html**). Veja a Listagem 2.4.1.

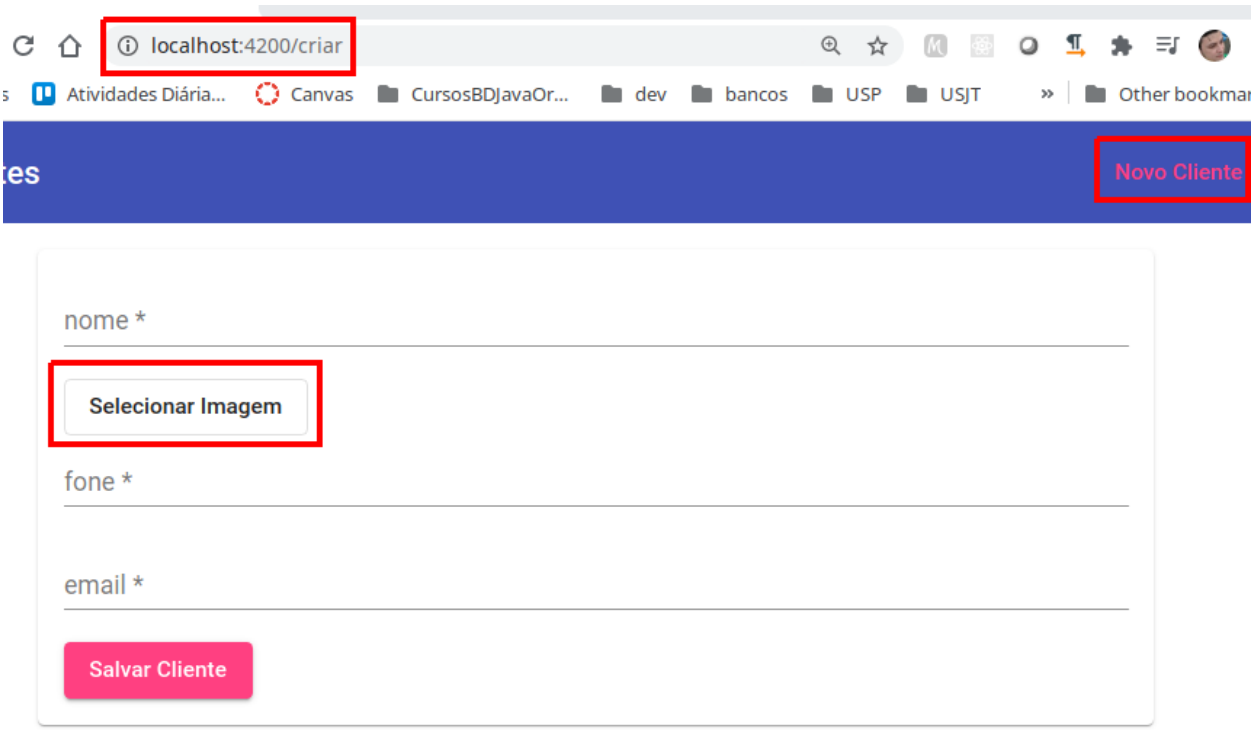
Listagem 2.4.1

```
<mat-card>
  <mat-spinner *ngIf="estaCarregando"></mat-spinner>
  <form (submit)="onSalvarCliente(clienteForm)" *ngIf="!estaCarregando"
#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>
    <div>
      <button mat-stroked-button type="button">Selecionar Imagem</button>
    </div>
    <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>
```

- Abra a aplicação e clique em **Novo Cliente** para verificar se o botão apareceu corretamente. O resultado deve ser parecido com o que exibe a Figura 2.4.1.

Figura 2.4.1



- Para adicionar a funcionalide de upload, vamos adicionar um elemento **input** com **type igual a file**. Veja a Listagem 2.4.2.

Listagem 2.4.2

```
<div>
    <button mat-stroked-button type="button">Selecionar Imagem</button>
    <input type="file">
    </div>
```

Visite novamente a página de inserção de clientes e veja que o component input que utilizamos exibirá um botão padrão cuja aparência não está de acordo com o Material Design. Desejamos somente a sua funcionalidade e desejamos que ela seja colocada em execução quando o botão que inserimos na página anteriormente for clicado. Para isso, vamos

- escrever uma regra CSS no arquivo **cliente-inserir.component.css** para esconder o input que acabamos de criar
- atribuir uma variável de referência de template a ele no arquivo **cliente-inserir.component.html** para que ele possa ser referenciado por outros elementos
- chamar o seu método click fazendo um event binding no botão que desejamos utilizar.

Veja as listagens 2.4.3 e 2.4.4.

Listagem 2.4.3

```
mat-form-field,
input {
  width: 100%;
}

mat-spinner {
  margin: auto;
}
input[type="file"] {
  visibility: hidden;
}
```

Listagem 2.4.4

```
<mat-card>
  <mat-spinner *ngIf="estaCarregando"></mat-spinner>
  <form (submit)="onSalvarCliente(clienteForm)" *ngIf="!estaCarregando"
#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>
    <div>
      <button mat-stroked-button type="button" (click)="selecionaArquivo.click()">Selecionar
Imagem</button>
      <input type="file" #selecionaArquivo>
    </div>
    <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>
```

2.5 (Lidando com Reactive Forms) O Angular possui dois tipos principais de forms: template driven forms e reactive forms. O primeiro é recomendado quando precisamos de forms simples, sem muitos controles e sem muita validação. É o que temos utilizado até então. Quando precisamos de um form mais detalhado, o segundo tipo (reactive forms) pode ser interessante.

- Para utilizar Reactive Forms, vamos começar ajustando o arquivo **app.module.ts**. O primeiro passo é **substituir** o módulo FormsModule pelo módulo ReactiveFormsModule. Veja a Listagem 2.5.1.

Listagem 2.5.1

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
//remover esse
//import { FormsModule } from '@angular/forms';
//inserir esse
import { ReactiveFormsModule } from '@angular/forms'
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
import { HttpClientModule} from '@angular/common/http'

import { MatInputModule } from '@angular/material/input';
import { MatCardModule } from '@angular/material/card';
import { MatButtonModule } from '@angular/material/button';
import { MatToolbarModule } from '@angular/material/toolbar';
import { MatExpansionModule } from '@angular/material/expansion';
import { MatProgressSpinnerModule } from '@angular/material/progress-spinner'

import { AppComponent } from './app.component';
import { ClienteInserirComponent } from './clientes/cliente-inserir/cliente-inserir.component';
import { CabecalhoComponent } from './cabecalho/cabecalho.component';
import { ClienteListaComponent } from './clientes/cliente-lista/cliente-lista.component';

import { AppRoutingModule } from './app-routing.module';

@NgModule({
  declarations: [
    AppComponent,
    ClienteInserirComponent,
    CabecalhoComponent,
    ClienteListaComponent,
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    ReactiveFormsModule,
    //FormsModule,
    BrowserAnimationsModule,
    MatInputModule,
    MatCardModule,
    MatButtonModule,
    MatToolbarModule,
    MatExpansionModule,
    MatProgressSpinnerModule,
    HttpClientModule,
  ],
```

```
  providers: [],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

- Note que a diretiva **ngForm percente ao módulo FormsModule**. Dado que o removemos de nossa aplicação, nosso **form de inserção de clientes**, que ainda a utiliza, deixou de funcionar.

- Para adaptar nosse template driven form para um reactive form, vamos fazer os seguintes ajustes:

- remover o uso da diretiva **ngForm** do elemento form
- remover o uso de **ngModel** dos elementos **input**
- remover as validações (**required** e **maxlength**) dos elementos **input**
- remover o argumento passado para o método **onSalvarCliente** no elemento form

Estamos no arquivo **cliente-inserir.component.html**. Veja a Listagem 2.5.2.

Listagem 2.5.2

```
<mat-card>
  <mat-spinner *ngIf="estaCarregando"></mat-spinner>
  <form (submit)="onSalvarCliente()" *ngIf="!estaCarregando" >
    <mat-form-field>
      <input
        type="text"
        matInput
        placeholder="nome"
        name="nome"/>
      <mat-error *ngIf="nome.invalid">Digite um nome válido</mat-error>
    </mat-form-field>
    <div>
      <button mat-stroked-button type="button" (click)="selecionaArquivo.click()">Selecionar
Imagem</button>
      <input type="file" #selecionaArquivo>
    </div>
    <mat-form-field>
      <input
        type="text"
        matInput
        placeholder="fone"
        name="fone"/>
      <mat-error *ngIf="fone.invalid">Digite um telefone válido</mat-error>
    </mat-form-field>
    <mat-form-field>
      <input
```

```
      type="text"
      matInput
      placeholder="email"
      name="email"
    />
    <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>
```

- Agora precisamos alterar a forma como o form é criado. A estrutura que o modela será criada com código Typescript. Para isso, abra o arquivo **cliente-inserir.component.ts** e declare um **formGroup**. Trata-se de um objeto que agrupa todos os controles de um form. Veja a Listagem 2.5.3.

Listagem 2.5.3

```
import { FormGroup, 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;
 public estaCarregando: boolean = false;
 form: FormGroup;
```

- A seguir, usando o método **ngOnInit**, vamos construir o objeto FormGroup. Ele recebe um objeto JSON com pares chave/valor quer descrevem seus controles (os controles de um form são os elementos que o usuário pode usar para inserir alguma coisa, como texto, imagem etc). Cada valor é do tipo **FormControl**, que também precisa ser importado de @angular/forms. FormControl recebe:

-valor inicial para o campo que representa
- objeto JSON que contém, entre outras coisas, validadores cujo funcionamento podemos especificar.

Note que o Angular já possui alguns validadores prontos para o uso, para os casos comuns. A Listagem 2.5.4 mostra a inicialização dos controles.

Listagem 2.5.4

```
import { FormControl, FormGroup, NgForm, Validators } from '@angular/forms';
ngOnInit(){
   this.form = new FormGroup({
      nome: new FormControl (null, {
        validators: [Validators.required, Validators.minLength(3)]
      }),
      fone: new FormControl (null, {
        validators: [Validators.required]
      }),
      email: new FormControl (null, {
        validators: [Validators.required, Validators.email]
      })
   })
   this.route.paramMap.subscribe((paramMap: ParamMap) => {
  ...
```

- O valor inicial igual a null para cada campo somente é válido caso não existam valores a serem exibidos neles. Quando o usuário está atualizando um cliente, ele já possui dados. Podemos atualizar os valores de cada campo usando o método **setValue**. Veja a Listagem 2.5.5.

Listagem 2.5.5

```
ngOnInit(){
   this.form = new FormGroup({
     nome: new FormControl (null, {
       validators: [Validators.required, Validators.minLength(3)]
     }),
     fone: new FormControl (null, {
       validators: [Validators.required]
     }),
     email: new FormControl (null, {
       validators: [Validators.required, Validators.email]
     })
   })
   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
        };
       this.form.setValue({
         nome: this.cliente.nome,
         fone: this.cliente.fone,
         email: this.cliente.email
       })
      });
     }
     else{
       this.modo = "criar";
       this.idCliente = null;
     }
   });
 }
```

- O método **onSalvarCliente** ainda recebe um objeto do tipo **NgForm**, o que já não faz sentido. Vamos remover o seu parâmetro e qualificar o acesso à variável chamada form com this, já que agora ela é uma variável de instância da classe. Também **substituimos o método resetForm pelo método reset**. Veja a Listagem 2.5.6.

Listagem 2.5.6

```
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
    );
  }
  else{
    this.clienteService.atualizarCliente(
      this.idCliente,
      this.form.value.nome,
      this.form.value.fone,
      this.form.value.email
    );
  }

  this.form.reset();
 }
```

- O próximo passo é vincular o objeto FormGroup ao template. Começamos fazendo isso por meio da diretiva FormGroup, aplicada ao elemento form, no template do componente. Veja a Listagem 2.5.7.

Listagem 2.5.7

```
<form [formGroup]="form" (submit)="onSalvarCliente()" *ngIf="!estaCarregando" >
```

- Cada objeto FormControl do FormGroup precisa ser vinculado ao seu respectivo elemento HTML explicitamente. Isso pode ser feito com a diretiva **formControlName**. Veja a Listagem 2.5.8. Estamos no arquivo **cliente-inserir.component.html**.

Listagem 2.5.8

```
<mat-card>
  <mat-spinner *ngIf="estaCarregando"></mat-spinner>
  <form [formGroup]="form" (submit)="onSalvarCliente()" *ngIf="!estaCarregando" >
    <mat-form-field>
      <input
        type="text"
        matInput
        placeholder="nome"
        formControlName="nome"
        name="nome"/>
      <mat-error *ngIf="nome.invalid">Digite um nome válido</mat-error>
    </mat-form-field>
    <div>
      <button mat-stroked-button type="button" (click)="selecionaArquivo.click()">Selecionar
Imagem</button>
      <input type="file" #selecionaArquivo>
    </div>
    <mat-form-field>
      <input
        type="text"
        matInput
        placeholder="fone"
        formControlName="fone"
        name="fone"/>
      <mat-error *ngIf="fone.invalid">Digite um telefone válido</mat-error>
    </mat-form-field>
    <mat-form-field>
      <input
        type="text"
        matInput
        placeholder="email"
        formControlName="email"
        name="email"
      />
      <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>
```

- Como removemos as variáveis de template ao trocar o tipo do form, os componentes **mat-error** já não podem acessá-las. Perceba que eles estão, inclusive, indicando erro neste momento. Ocorre que o objeto form possui um método chamado **get** que recebe o nome de um FormControl e devolve o objeto associado a ele. A seguir, podemos acessar as propriedades comuns de controles de form, tais quais **invalid**. Veja o ajuste da Listagem 2.5.9.

Listagem 2.5.9

```
<mat-card>
  <mat-spinner *ngIf="estaCarregando"></mat-spinner>
  <form [formGroup]="form" (submit)="onSalvarCliente()" *ngIf="!estaCarregando" >
    <mat-form-field>
      <input
        type="text"
        matInput
        placeholder="nome"
        formControlName="nome"
        name="nome"/>
      <mat-error *ngIf="form.get('nome').invalid">Digite um nome válido</mat-error>
    </mat-form-field>
    <div>
      <button mat-stroked-button type="button" (click)="selecionaArquivo.click()">Selecionar
Imagem</button>
      <input type="file" #selecionaArquivo>
    </div>
    <mat-form-field>
      <input
        type="text"
        matInput
        placeholder="fone"
        formControlName="fone"
        name="fone"/>
      <mat-error *ngIf="form.get('fone').invalid">Digite um telefone válido</mat-error>
    </mat-form-field>
    <mat-form-field>
      <input
        type="text"
        matInput
        placeholder="email"
        formControlName="email"
        name="email"
```

```
    />
    <mat-error *ngIf="form.get('email').invalid">Digite um nome válido</mat-error>
   </mat-form-field>
   <button color="accent" mat-raised-button>
    Salvar Cliente
   </button>
  </form>
</mat-card>
```

- Neste momento, o form deve estar funcionando corretamente novamente, embora a funcionalidade de inserção de fotos ainda não esteja funcionando.

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.