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 (Lidando com a foto do lado do cliente) No momento, quando o usuário clica no botão para upload de foto, fazemos com que o método click do input de tipo file seja executado. Contudo, não manipulamos o arquivo escolhido pelo usuário. Para fazê-lo, vamos fazer um **event binding** utilizando o evento **change** no input de tipo file. Ele é detectado toda vez que o usuário **altera** o arquivo selecionado (daí seu nome).

- No arquivo **cliente-inserir.component.html**, faça o event binding como mostra a Listagem 2.2.1. Note que impolementaremos o método vinculado ao evento em breve, por isso, é de se esperar que o VS Code (dependendo das extensões que você tiver instalado) indique algum erro). Lembre-se que **$event** é um símbolo especial do Angular. Ele é do tipo **Event** e representa o evento que ocorreu que dá acesso a dados de interesse. Neste caso, ao arquivo selecionado, por exemplo.

Listagem 2.2.1

```
...
<div>
    <button mat-stroked-button type="button" (click)="selecionaArquivo.click()">Selecionar
Imagem</button>
    <input type="file" (change)="onImagemSelecionada($event)" #selecionaArquivo>
  </div>
...
```

- A seguir, no arquivo **cliente-inserir.component.ts**, implementamos o método **onImagemSelecionada**, cujo uso acabamos de especificar no template do componente. Note que o objeto recebido é, de fato, do tipo **Event**. Observe que Event não requer uma instrução import. Isso ocorre pois trata-se de um tipo nativo de Javascript e, assim, o compilador Typescript já nos dá acesso a ele por padrão. Veja a Listagem 2.2.2. Trata-se de um método comum. Como qualquer outro, ele deve ficar no corpo da classe, entre seus símbolos **{** e **}**.

Listagem 2.2.2

```
onImagemSelecionada (event: Event){


 }
```

- A partir de **event**, podemos acessar uma propriedade chamada **target** (que é o elemento Input). Por sua vez, target nos dá acesso a uma propriedade chamada **files**, que é uma lista de arquivos selecionados pelo usuário. Para poder acessar a propriedades files, precisamos explicar que event.target é um HTMLInputElement com um casting explícito. Como temos um único arquivo, pegaremos o elemento na posição zero da lista. Veja a Listagem 2.2.3.

Listagem 2.2.3

```
onImagemSelecionada (event: Event){
   const arquivo = (event.target as HTMLInputElement).files[0];
}
```

- Precisamos de um novo controle para o form, que será responsável por lidar com a imagem. Por enquanto, temos controles somente para nome, fone e e-mail. No método **ngOnInit**, adicione o novo controle, que é destacado pela Listagem 2.2.4. Ele começa com null. Logo atribuiremos a imagem a ele. Adicione também um validador indicando que trata-se de um valor obrigatório. Estamos no arquivo **cliente-inserir.component.ts**.

Listagem 2.2.4

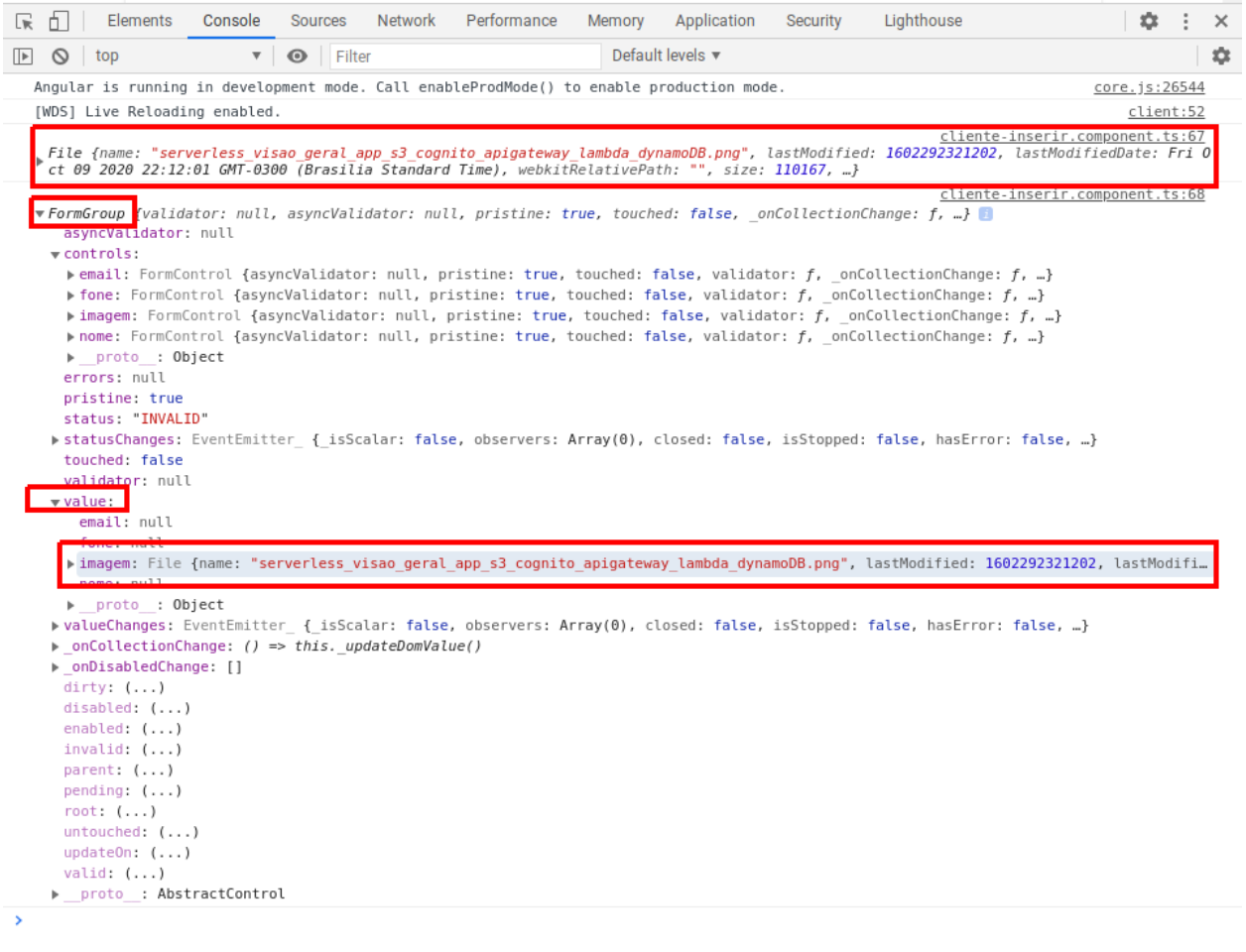
```
ngOnInit(){
   this.form = new FormGroup({
     nome: new FormControl (null, {
       validators: [Validators.required, Validators.minLength(3)]
     }),
     fone: new FormControl (null, {
       validators: [Validators.required]
     }),
     email: new FormControl (null, {
       validators: [Validators.required, Validators.email]
     }),
     imagem: new FormControl(null, {
       validators: [Validators.required]
     })
   })
...
```

- No método onImagemSelecionada, vamos associar a imagem ao novo controle do form. O método **patchValue** recebe um objeto JSON em que as chaves são os nomes dos controles e os valores associados são as expressões a serem vinculadas aos controles. A seguir, utilizamos o método **updateValueAndValidity** para que o form seja atualizado e as validações sejam realizadas. Exiba o arquivo e o form no console para entender o que aconteceu. No console, expanda o form e a sua propriedade **value**, como na Figura 2.2.1. Lembre-se de usar CTRL+SHIFT+I no Chrome para abrir o Chrome Dev Tools. Clique, então, na aba console. Veja a Listagem 2.2.5.

Listagem 2.2.5

```
onImagemSelecionada (event: Event){
   const arquivo = (event.target as HTMLInputElement).files[0];
   this.form.patchValue({'imagem': arquivo});
   this.form.get('imagem').updateValueAndValidity();
   console.log(arquivo);
   console.log(this.form);
  }
```

Figura 2.2.1



- Podemos exibir um preview da foto para o usuário. Para isso, vamos começar adicionando um div ao form. A seu atributo alt, podemos associar o nome do cliente, por exemplo. Observe o uso de um property binding com [ ]. Estamos no arquivo **cliente-inserir.component.html**. Veja a Listagem 2.2.6.

Listagem 2.2.6

```
...
</mat-form-field>
   <div>
     <button mat-stroked-button type="button" (click)="selecionaArquivo.click()">Selecionar
Imagem</button>
     <input type="file" (change)="onImagemSelecionada($event)" #selecionaArquivo>
   </div>
   <div class="imagem-preview">
     <img src="" [alt]="form.value.nome">
   </div>
   <mat-form-field>
...
```

- A seguir, precisamos de uma URL por meio da qual possamos acessar a imagem. Podemos produzi-la no método **onImagemSelecionada**, no arquivo **cliente-inserir.component.ts**. Usaremos um **FileReader**, próprio da API do Javascript. Seu método **readAsDataURL** recebe um arquivo e devolve uma URL que permite acessá-lo. Antes de chamá-lo, contudo, vinculamos uma arrow function à sua propriedade **onload**. Ela deve atribuir a propriedade **reader.result** a uma string que declararemos como variável de instância (**previewImagem**) da classe que define o componente. Note que também removemos as chamadas ao método log de console. Veja a Listagem 2.2.7.

Listagem 2.2.7

```
…
  private modo: string = "criar";
  private idCliente: string;
  public cliente: Cliente;
  public estaCarregando: boolean = false;
  form: FormGroup;
  previewImagem: string;
...

onImagemSelecionada (event: Event){
    const arquivo = (event.target as HTMLInputElement).files[0];
    this.form.patchValue({'imagem': arquivo});
    this.form.get('imagem').updateValueAndValidity();
    const reader = new FileReader();
    reader.onload = () => {
      this.previewImagem = reader.result as string;
    }
    reader.readAsDataURL(arquivo);
  }
```

- Para exibir a imagem, basta fazer um property binding envolvendo a propriedade **src** do elemento **img** e a variável **previewImagem**, no arquivo **cliente-inserir.component.html**. Além disso, faremos com que a renderização da div que contém o elemento img seja condicional. Ela somente será exibida caso exista uma foto. Veja a Listagem 2.2.8.

Listagem 2.2.8

```
<div class="imagem-preview">
     <img [src]="previewImagem" *ngIf="previewImagem && previewImagem !== ""
[alt]="form.value.nome">
    </div>
```

- A seguir, definimos a classe CSS **imagem-preview**. Iremos limitar a altura da div que contém a imagem e dar a ela uma margem. Além disso, faremos com que o elemento img ocupe 100% da div a que ele pertence. Veja a Listagem 2.2.9. Estamos no arquivo **cliente-inserir.component.css.**

Listagem 2.2.9

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

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

.imagem-preview {
  height: 5rem;
  margin: 1rem 0;
}

.imagem-preview  img {
  height: 100%;
}
```

- Faça um teste. Na págia principal da aplicação, clique em Novo Cliente e, então, clique em Selecionar Imagem. Escolha uma foto qualquer. Neste momento, o preview deve funcionar.

- Idealmente, a aplicação deve restringir o tipo de arquivo que o usuário seleciona. Desejamos que ele possa selecionar somente arquivos que realmente são fotos. Para isso, vamos escrever um validador personalizado. Clique com o direito na pasta **cliente-inserir** e crie um arquivo chamado **mime-type.validator.ts**.

- Vamos escrever uma função que devolve um validador assíncrono, com as seguintes características.

- Recebe como argumento um do tipo **AbstractControl**. Ele representa o controle de um form a ser validado.
- Por operar de maneira assíncrona, devolve uma Promise ou um Observable. Seu tipo genérico é um JSON com uma lista de tipo any, com uma chave do tipo string (que ficará mais claro em breve).
- Extrai o arquivo do controle.
- Constrói um leitor de arquivos.
- Constrói um objeto Observable que recebe uma arrow function. Ela recebe um Observer cujo tipo genérico é igual ao do Observable e da Promise.
- A função registra uma função que entra em execução uma vez que o evento **loadend** do leitor acontecer.
- Executa readAsArrayBuffer para fazer com que a leitura aconteça. O arquivo será colocado em memória como uma sequência de bytes.
- Uma vez que o evento aconteça, os dados serão colocados em um vetor de inteiros de 8 bits “unsigned”. Ou seja, um vetor de bytes.
- O MIME type se encontra nos primeiros quatro bytes. Por isso, usamos **subarray** com os índices 0 e 4.
- Para fazer a validação, iteramos sobre o vetor, convertendo cada posição para uma string em hexadecimal.
- O valor obtido permite determinar se o arquivo é, de fato, uma foto. Os valores que usamos estão documentados no Link 2.2.1.

Link 2.2.1
https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types

- Se o tipo for válido, devolvemos null. Caso contrário, devolvemos um objeto JSON com uma chave (com nome arbitrário, porém de acordo com o significado da validação) associada ao valor true. Veja a Listagem 2.2.10.

Listagem 2.2.10

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

export const mimeTypeValidator = (
  control: AbstractControl
): Promise <{[key: string]: any}> | Observable <{[key: string]: any}> => {
  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;
}
```

- Agora podemos vincular o validador ao form. No arquivo **cliente-inserir.component.ts**, importe a função que acabamos de criar e vincule-a à propriedade **asyncValidtors** do controle associado à imagem, como na Listagem 2.2.11.

Listagem 2.2.11

```
import { mimeTypeValidator } from './mime-type.validator';
…
imagem: new FormControl(null, {
      validators: [Validators.required],
     asyncValidators: [mimeTypeValidator]
})
...
```

- Também desejamos garantir que a **div** que contém o **img** no form somente seja exibida caso o controle tenha sido validado. Podemos adicionar um teste à sua diretiva estrutura ngIf. Veja a Listagem 2.2.12. Estamos no arquivo **cliente-inserir.component.html**.

Listagem 2.2.12

```
<div class="imagem-preview">
    <img [src]="previewImagem" *ngIf="previewImagem && previewImagem !== '' &&
form.get('imagem').valid" [alt]="form.value.nome">
  </div>
```

- Faça um novo teste. Tente selecionar arquivos de diferentes tipos (fotos, textuais etc) e veja o resultado.

2.3 (Lidando com a foto no servidor) As submissões do form levam consigo uma foto potencialmente selecionada pelo usuário. No Back End, precisamos explicar como se dá o tratamento dela, como ela será armazenada em meio persistente.

- Comece abrindo um novo terminal no VS Code para instalar o pacote **multer** com

npm install --save multer

Segundo a sua documentação, encontrada no Link 2.3.1, ele adiciona objetos **file** e/ou **files** à request. Deles podemos obter o(s) arquivo(s) enviados.

Link 2.3.1
https://www.npmjs.com/package/multer

- Abra o arquivo **backend/rotas/clientes.js**. Comece importando o **multer** e configurando o seu diretório de armazenamento como mostra a Listagem 2.3.1.

Listagem 2.3.1

```
const express = require ("express");
const multer = require ("multer");
const router = express.Router();
const Cliente = require('../models/cliente');
const armazenamento = multer.diskStorage({
  //requisição, arquivo extraido e uma função a ser
  //executada, capaz de indicar um erro ou devolver
  //o diretório em que as fotos ficarão
  destination: (req, file, callback) => {
    callback(null, "backend/imagens")
  }
})
```

- Crie uma pasta chamada **imagens** como subpasta de backend.

- A seguir, vamos especificar o padrão que desejamos para os nomes dos arquivos. Vamos, por exemplo, substituir espaços em branco por traços e converter para caixa baixa todas as letras. Veja a Listagem 2.3.2.

Listagem 2.3.2

```
const armazenamento = multer.diskStorage({
  //requisição, arquivo extraido e uma função a ser
  //executada, capaz de indicar um erro ou devolver
  //o diretório em que as fotos ficarão
  destination: (req, file, callback) => {
    callback(null, "backend/imagens")
  },
  filename: (req, file, callback) =>{
    const nome = file.originalname.toLowerCase().split(" ").join("-");
  }
})
```

- Precisamos especificar, também, a extensão do arquivo. Para isso, vamos criar um mapeamento de MIME Type para extensões desejadas. Veja a Listagem 2.3.3.

Listagem 2.3.3

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

const MIME_TYPE_EXTENSAO_MAPA = {
  'image/png': 'png',
  'image/jpeg': 'jpg',
  'image/jpg': 'jpg',
  'image/bmp': 'bmp'
}
...
```

- A seguir, podemos chamar a função callback de filename especificando o nome completo do arquivo. A fim de garantir nomes únicos, iremos concatenar, também, a data/hora atual. Veja a Listagem 2.3.4.

Listagem 2.3.4

```
const armazenamento = multer.diskStorage({
  //requisição, arquivo extraido e uma função a ser
  //executada, capaz de indicar um erro ou devolver
  //o diretório em que as fotos ficarão
  destination: (req, file, callback) => {
    callback(null, "backend/imagens")
  },
  filename: (req, file, callback) =>{
    const nome = file.originalname.toLowerCase().split(" ").join("-");
    const extensao = MIME_TYPE_EXTENSAO_MAPA[file.mimetype];
    callback(null, `${nome}-${Date.now()}.${extensao}`);
  }
})
```

- A seguir, iremos validar o tipo do arquivo recebido. Caso não seja um tipo esperado (uma foto), construiremos um objeto Error que será entregue à função callback de **destination** em seu primeiro parâmetro. Veja a Listagem 2.3.5.

Listagem 2.3.5

```
destination: (req, file, callback) => {
   let e = MIME_TYPE_EXTENSAO_MAPA[file.mimetype] ? null : new Error ('Mime Type Invalido');
    callback(e, "backend/imagens")
  },
```

- O próximo passo é utilizar o objeto armazenamento. Ele fará parte da rota **post**, já que ela é responsável por atender requisições de inserção de clientes. Chamamos **multer** entregando a ela o objeto armazenamento e, a seguir, chamamos **single** indicando que esperamos um único arquivo. O argumento entregue a single é o nome da propriedade a que o arquivo estará associado. Veja a Listagem 2.3.6.

Listagem 2.3.6

```
router.post('',multer({storage: armazenamento}).single('imagem'), (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
      })
    })
});
```

2.4 (Lidando com a foto do lado do cliente (no serviço)) Já temos um form capaz de lidar com fotos e um Back End capaz de recebê-las. Contudo, o service, responsável pela comunicação entre cliente e servidor ainda não sabe da existência da foto possivelmente existente no form. Vamos ajustá-lo para tratá-la também.

- Abra o arquivo **clientes.service.ts**. Precisamos ajustar o método **adicionarCliente**. Ele deixar de enviar um objeto JSON que representa um Cliente para o servidor e passa a enviar um **FormData**. Iremos adicionar os campos de interesse ao FormData, incluindo a foto. O método também precisa receber o arquivo, pois ainda não tem acesso a ele. Veja os ajustes na Listagem 2.4.1.

Listagem 2.4.1

```
 adicionarCliente(nome: string, fone: string, email: string, imagem: File) {
   /*const cliente: Cliente = {
      id: null,
      nome: nome,
      fone: fone,
      email: email,
    };*/
    const dadosCliente = new FormData();
    dadosCliente.append("nome", nome);
    dadosCliente.append('fone', fone);
    dadosCliente.append('email', email);
    dadosCliente.append('imagem', imagem);

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

- Como adicionamos um parâmetro ao método adicionarCliente, precisamos ajustar seu cliente para que o utilize adequadamente, entregando mais um argumento: o arquivo. Quem chama o método adicionarCliente é o componente **ClienteInserirComponent**. Abra o arquivo **cliente-inserir.component.ts** e ajuste seu método **onSalvarCliente**, como exibe a Listagem 2.4.2.

Listagem 2.4.2

```
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.reset();
 }
```

- Faça um novo teste e verifique se o arquivo foi salvo na pasta de imagens do Back End.

- Após armazenar a foto em disco, precisamos guardar o seu endereço no MongoDB para que no futuro ela possa ser recuperada e devolvida para o cliente. O primeiro passo é alterar o modelo, definido no arquivo **backend/models/cliente.js**. Precisamos incluir uma propriedade para representar o endereço da imagem. Veja a Listagem 2.4.3.

Listagem 2.4.3

```
//importando o pacote
const mongoose = require ('mongoose');

//definindo o "schema"
//note a semelhança com recursos de bases relacionais
const clienteSchema = mongoose.Schema ({
  nome: {type: String, required: true},
  fone: {type: String, required: false, default: '00000000'},
  email: {type: String, required: true},
  imagemURL: {type: String, required: true}

});

//criamos o modelo associado ao nome Cliente e exportamos
//tornando acessível para outros módulos da aplicação
module.exports = mongoose.model('Cliente', clienteSchema);
```

- A seguir, montamos o endereço da foto. O primeiro passo é descobrir o protocolo (HTTP ou HTTPS) e o endereço do host. Ambas as informações estão disponíveis no objeto **req,** que representa a requisição. Com essas informações em mãos, montamos o endereço desejado incluindo o nome do arquivo que também se encontra na requisição. Note que as fotos estão originalmente armazenadas em **backend/imagens**. Iremos, contudo, fazer com que os clientes possam acessá-las diretamente de imagens, escondendo a existência da pasta backend. Trata-se de algo que requer configuração ainda a ser feita. Veja a Listagem 2.4.4.

Listagem 2.4.4

```
router.post('',multer({storage: armazenamento}).single('imagem'), (req, res, next) => {
  const imagemURL = `${req.protocol}://${req.get('host')}`
  const cliente = new Cliente({
    nome: req.body.nome,
    fone: req.body.fone,
    email: req.body.email,
    imagemURL: `${imagemURL}/imagens/${req.file.filename}`
...
```

- Passaremos a devolver o objeto cliente completo a fim de que a aplicação Angular seja capaz de recuperar a foto. Veja a Listagem 2.4.5.

Listagem 2.4.5

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

- A aplicação Angular precisa ser ajustada para tratar o novo resultado, que agora é um objeto que contém um cliente e não mais somente um id. Comece ajustando o modelo no arquivo **cliente.model.ts**, como mostra a Listagem 2.4.6.

Listagem 2.4.6

```
export interface Cliente {
  id: string;
  nome: string;
  fone: string;
  email: string;
  imagemURL: string;
}
```

- Abra o arquivo **clientes.service.ts** e faça o ajuste da Listagem 2.4.7, no método **adicionarCliente**.

Listagem 2.4.7

```
adicionarCliente(nome: string, fone: string, email: string, imagem: File) {
   /*const cliente: Cliente = {
     id: null,
     nome: nome,
     fone: fone,
     email: email,
   };*/
   const dadosCliente = new FormData();
   dadosCliente.append("nome", nome);
   dadosCliente.append('fone', fone);
   dadosCliente.append('email', email);
   dadosCliente.append('imagem', imagem);

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

- Note que o método **atualizarCliente** presente no arquivo **clientes.service.ts** também apresenta erro, pois tenta construir um objeto Cliente sem a propriedade imagemURL. Por agora, vamos apenas colocar o valor null. Veja a Listagem 2.4.8.

Listagem 2.4.8

```
atualizarCliente (id: string, nome: string, fone: string, email: 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 mesmo ajuste é necessário no arquivo **cliente-inserir.component.ts**, no método **ngOnInit**. Veja a Listagem 2.4.9.

Listagem 2.4.9

```
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]
     }),
     imagem: new FormControl(null, {
       validators: [Validators.required],
       asyncValidators: [mimeTypeValidator]
     })
   })
   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
       })
      });
     }
     else{
       this.modo = "criar";
       this.idCliente = null;
     }
   });
 }
```

- No arquivo **clientes.service.ts**, precisamos ajustar o método **getClientes** para que ele faça uso da propriedade imagemURL também. A Listagem 2.4.10 mostra a alteração.

Listagem 2.4.10

```
getClientes(): void {
   this.httpClient.get <{mensagem: string, clientes: any}>('http://localhost:3000/api/clientes')
   .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]);
     }
   )
  }
```

- Os clientes já armazenados na base não têm a URL para a imagem. Para realizar novos testes, pode ser uma boa ideia removê-los da base. Isso pode ser feito diretamente no MongoDB. Acesse o Link 2.4.1, clique em **Collections** , clique sobre a sua coleção e, então, clique na lata de lixo à frente do nome dela. As figuras 2.4.1 e 2.4.2 ilustram o procedimento.

Link 2.4.1
mongodb.com

Figura 2.4.1

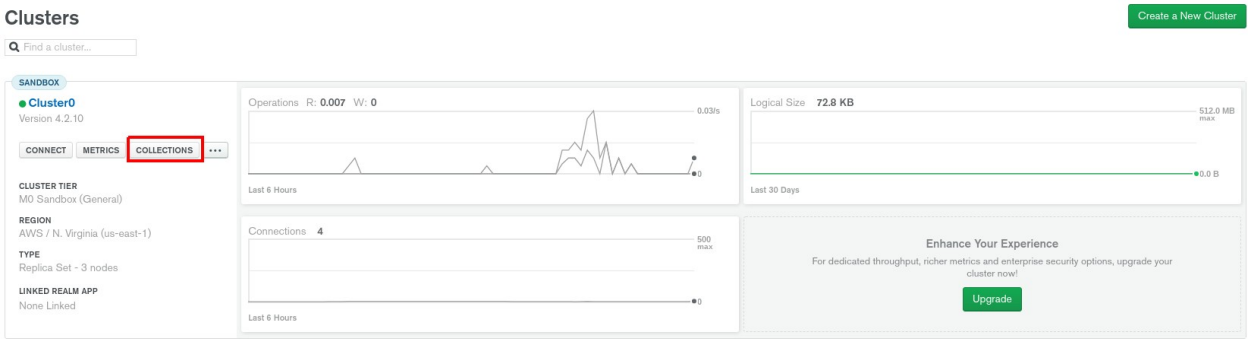
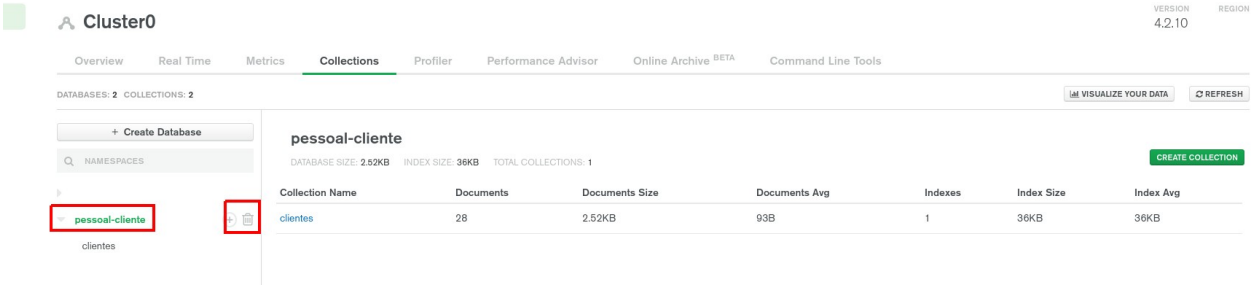


Figura 2.4.2



- O próximo passo é exibir a imagem de cada cliente. Para isso, vamos adicionar uma div com um img como filho no template do componente **ClienteListaComponent**, ou seja, no arquivo **cliente-lista.component.html**. Veja a Listagem 2.4.11.

Listagem 2.4.11

```
<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>
<p class="mat-body-1" style="text-align: center;" *ngIf="clientes.length <= 0 && !
estaCarregando">
  Nenhum cliente cadastrado
</p>
```

- Note que ainda não é possível visualizar as fotos pois, no servidor, elas se encontram na pasta backend/imagens. Por outro lado, a propriedade imagemURL aponta para uma pasta chamada imagens na raiz que não existe. Essa é uma boa prática pois podemos restringir quais rotas podem acessar nossos arquivos. Porém, precisamos fazer o mapeamento adequado. Para tal, faça os ajustes ilustrados na Listagem 2.4.12. Estamos no arquivo **backend/app.js**.

Listagem 2.4.12

```
const path = require ('path');
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('/imagens', express.static(path.join("backend/imagens")));
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;
```

- As regras envolvendo a classe cliente-imagem precisam ser criadas no arquivo **cliente-lista.component.css**, como mostra a Listagem 2.4.13.

Listagem 2.4.13

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

mat-spinner {
  margin: auto;
}

.cliente-imagem {
 width: 100%;
 display: flex;
 justify-content: center;
}

.cliente-imagem > img {
  max-width: 50%;
}
```

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.