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 (Exibindo a quantidade correta de clientes no elemento de paginação) O elemento de paginação está mostrando um total de clientes fixo, igual a dez. Desejamos, evidentemente, que ele exiba a quantidade real de clientes na base. Essa informação pode ser obtida no Back End.

- No Back End, precisamos fazer com que a busca inclua a quantidade de itens, que pode ser encontrada utilizando-se o método **count**. No momento, já executamos um método (**find**) para buscar todos os clientes. Para usar o método count, iremos encadear o uso de promises. Executamos uma consulta que devolve uma Promise e, com o resultado em mãos, executamos outra consulta que devolve outra Promise. E assim por diante. Veja a Listagem 2.2.1. Estamos no arquivo **clientes.ts**.

Listagem 2.2.1

```
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
  let clientesEncontrados;
  if (pageSize && page){
    consulta
      .skip(pageSize * (page - 1))
      .limit(pageSize);
  }
  consulta.then(documents => {
    clientesEncontrados = documents;
    //devolve uma Promise, tratada com o próximo then
    return Cliente.count();
  })
  .then((count) => {
    res.status(200).json({
      mensagem: "Tudo OK",
      clientes: clientesEncontrados,
      //devolvendo o count para o Front
      maxClientes: count
    });
  })

});
```

- No Front End, passamos a utilizar o valor associado a maxClientes que o Back End devolve. O responsável pela interação com o Back End é o serviço, portanto, abra o arquivo **clientes.service.ts**. O método getClientes, no momento, especifica o recebimento de um objeto JSON com duas propriedades: mensagem e clientes. É preciso adicionar uma terceira, indicando que agora temos o número de clientes também. Veja a Listagem 2.2.2.

Listagem 2.2.2

```
getClientes(pagesize: number, page: number): void {
   const parametros = `?pagesize=${pagesize}&page=${page}`;
   this.httpClient.get <{mensagem: string, clientes: any, maxClientes:
number}>('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]);
     }
   )
  }
```

- Ainda no método getClientes, observe que a variável chamada **dados** representa aquilo que o Back End entrega para o Front End. Ou seja, um objeto JSON contendo uma mensagem, uma lista de clientes e, agora, o número de clientes. Lembre-se que utilizamos a função **map** para explicar que, dado um objeto desse tipo, desejamos extrair somente a coleção de clientes, que deve ser o resultado do método. Contudo, agora também desejamos que o número de clientes faça parte do resultado do método. Para isso, basta ajustar o mapeamento: o objeto resultante deixa de ser uma única lista de clientes e passa a ser um objeto JSON que contém a lista de clientes e, ainda, o número de clientes. Veja a Listagem 2.2.3.

Listagem 2.2.3

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

- O método getClientes ainda requer um ajuste. O método subscribe recebe um objeto que, até então, era a lista de clientes, o que viabiliza a atribuição feita por ele. Temos agora um objeto que contém a lista desejada e, assim, precisamos acessá-lo para só então acessar a lista de clientes. Veja a Listagem 2.2.4.

Listagem 2.2.4

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

- Quando uma busca pela lista de clientes termina, o serviço se encarrega de notificar os seus observadores - os componentes Angular interessados na lista. No momento, a notificação envolve somente a lista. Contudo, a quantidade de clientes existentes na base também é de interesse. Assim, precisamos alterar o tipo com o qual o Subject lida, que passará a ser um objeto JSON que contém a lista de clientes e o número de clientes como suas propriedades. Veja a Listagem 2.2.5. Ainda estamos no arquivo **clientes.service.ts**.

Listagem 2.2.5

```
…
@Injectable({ providedIn: 'root' })
export class ClienteService {

  private clientes: Cliente[] = [];
  private listaClientesAtualizada = new Subject<{clientes: Cliente[], maxClientes: number}>();
...
```

- De volta ao método **getClientes**, ajustamos a chamada que ele faz ao método next para que a notificação inclua também o número de clientes, em um objeto JSON apropriado. Veja a Listagem 2.2.6.

Listagem 2.2.6

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

2.3 (Removendo redundâncias nos métodos atualizarCliente e adicionarCliente) Quando ambos os métodos atualizarCliente e adicionarCliente, eles fazem uma requisição ao Back End e, uma vez que recebam uma resposta, se encarregam de notificar os seus observadores. Além de

fazer isso, eles utilizam o **roteador** do Angular para navegar para o componente mapeado à raiz da aplicação, ou seja, o componente que exibe a lista. Devido a essa navegação, o método **ngOnInit** do componente que exibe a lista executa. Observe que ele próprio chama o método getClientes, o que já faz com que a lista seja atualizada. Isso quer dizer que os envios de notificação feitos pelos métodos atualizarCliente e adicionarCliente são desnecessários. Vamos, portanto, removê-los, como mostram as listagens 2.3.1 e 2.3.2.

Listagem 2.3.1

```
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 => {
     //apagar tudo isso
     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(['/'])
   }));
  }
```

Listagem 2.3.2

```
adicionarCliente(nome: string, fone: string, email: string, imagem: File) {
   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) => {
       //apagar tudo isso
       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(['/']);
      }
    )
  }
```

2.4 (Ajustando o método delete) Quando a operação de remoção de clientes acontece, também desejamos atualizar a lista. A atualiação da lista depende dos valores das variáveis **totalDeClientes** e **totalDeClientesPorPagina**, os quais são conhecidos somente pelo componente ClienteListaComponent. Assim, faremos com que o serviço devolva o Observable para que o próprio ClienteListaComponent possa chamar o método getClientes. As listagens 2.4.1 e 2.4.2 mostram as atualizações a serem feitas nos arquivos **clientes.service.ts** e **cliente-lista.component.ts**, respectivamente. Também queremos exibir o componente de carregamento da página enquanto a operação não termina, o que também é feito no método onDelete.

Listagem 2.4.1

```
//lembre-se de ajustar o tipo de retorno do método
removerCliente (id: string){
   return this.httpClient.delete(`http://localhost:3000/api/clientes/${id}`);
}
```

Listagem 2.4.2

```
onDelete (id: string): void{
   this.estaCarregando = true;
   this.clienteService.removerCliente(id).subscribe(() => {
     this.clienteService.getClientes(this.totalDeClientesPorPagina, this.paginaAtual);
   });
 }
```

- Ainda no arquivo **cliente-lista.component.ts**, note que o método ngOnInit deixou de funcionar também. Isso ocorre pois ele ainda espera receber uma lista de clientes nas notificações enviadas pelo serviço. Como sabemos, o dado incluído nas notificações agora é um objeto JSON que contém a lista e o número de clientes na base. Assim, basta ajustar a forma como ele acessa a lista de clientes. Aliás, podemos aproveitar para atualizar a variável **totalDeClientes** também, pois agora temos acesso ao valor a ser atribuído a ela. Ajuste, também, o seu valor inicial, para zero. Veja a Listagem 2.4.3.

Listagem 2.4.3

```
totalDeClientes: number = 0;

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

Nota: O número total de clientes existentes na base no Back End é potencialmente diferente do número de clientes existentes na lista exibida no Front End. Isso ocorre pois os elementos existentes na lista do Front End são somente aqueles que estão de acordo com os parâmetros de paginação enviados para o Back End.

2.5 (Autenticação: Front End) A aplicação oferece funcionalidades que podem ser acessadas por qualquer pessoa. Nesta seção, passaremos a implementar o Front End para um mecanismo de autenticação capaz de restringir o acesso a determinadas funcionalidades. Por exemplo, poderíamos permitir que todo tipo de usuário veja todos os clientes, alguns usuários possam fazer cadastros e clientes somente podem ser removidos ou atualizados por usuários que os tenham cadastrados.

- Comece criando um componente responsável pelo form para que o usuário possa fazer login na aplicação. Para tal, use o comando a seguir. Note que ele ficará em uma subpasta chamada **auth**, que não é obrigatória. Trata-se somente de uma questão de organização. Chamaremos o componente de **login**.

ng g c auth/login --skipTests

- Crie também um componente responsável pelo cadastro de novos usuários chamado **signup**. Use o comando a seguir.

ng g c auth/signup --skipTests

- Vamos começar elaborando o form para login. Ele possui campos para e-mail e senha, além de um botão para submissão. Por ser um form muito simples, utilizaremos um **template driven form**. A Listagem 2.5.1 mostra a sua definição, que deve ficar no arquivo **login.component.html**.

Listagem 2.5.1

```
<mat-card>
  <mat-spinner></mat-spinner>
  <form>
    <mat-form-field>
      <input matInput type="email" placeholder="Email">
      <mat-error>Digite um email válido</mat-error>
    </mat-form-field>
    <mat-form-field>
      <input matInput type="password" placeholder="Senha">
      <mat-error>Senha inválida</mat-error>
    </mat-form-field>
    <button mat-raised-button color="accent" type="submit">Login</button>
  </form>
</mat-card>
```

- A aplicação deve estar exibindo um erro no momento envolvendo a mensagem **No Provider for ControlContainer**. Isso ocorre pois o form que acabamos de criar é um **template driven form**. O form que a aplicação já possuía é um **reactive form** e, por isso, o módulo que importamos é o **ReactiveFormsModule**. Para utilizar template-driven forms, precisamos importar o módulo **FormsModule**, o que pode ser feito no arquivo **app.module.ts**, ilustrado na Listagem 2.5.2.

Listagem 2.5.2

```
import { FormsModule, ReactiveFormsModule } from '@angular/forms'
...
@NgModule({
  declarations: [
    AppComponent,
    ClienteInserirComponent,
    CabecalhoComponent,
    ClienteListaComponent,
    LoginComponent,
    SignupComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    ReactiveFormsModule,
    BrowserAnimationsModule,
    FormsModule,
    MatInputModule,
    MatCardModule,
    MatButtonModule,
    MatToolbarModule,
    MatExpansionModule,
    MatPaginatorModule,
    MatProgressSpinnerModule,
    HttpClientModule,
  ],
...
```

- A exibição do spinner e do form deve ser mutuamente exclusiva: somente um deles deve ser exibido a qualquer momento. Para controlar isso, adicione uma variável à classe do componente, no arquivo **login.component.ts** e, a seguir, adicione diretivas estruturas a ambos os elementos. As listagens 2.5.3 e 2.5.4 ilustram os ajustes mencionados.

Listagem 2.5.3

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css']
})
export class LoginComponent implements OnInit {
  estaCarregando: boolean = false;
  constructor() { }
  ngOnInit(): void {
  }
}
```

Listagem 2.5.4

```
<mat-card>
  <mat-spinner *ngIf="estaCarregando"></mat-spinner>
  <form *ngIf="!estaCarregando">
    <mat-form-field>
      <input matInput type="email" placeholder="Email">
      <mat-error>Digite um email válido</mat-error>
    </mat-form-field>
    <mat-form-field>
      <input matInput type="password" placeholder="Senha">
      <mat-error>Senha inválida</mat-error>
    </mat-form-field>
    <button mat-raised-button color="accent" type="submit">Login</button>
  </form>
</mat-card>
```

- O padrão de acesso ao componente de login será **host:porta/login.** Para isso, basta adicionar uma nova rota, o que pode ser feito no arquivo **app-routing.module.ts**. Veja a Listagem 2.5.5.

Listagem 2.5.5

```
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';
import { LoginComponent } from './auth/login/login.component';
const routes: Routes = [
  { path: '', component: ClienteListaComponent },
  {path: 'criar', component: ClienteInserirComponent},
  {path: 'editar/:idCliente', component: ClienteInserirComponent},
  {path: 'login', component: LoginComponent}
];
@NgModule({
  imports: [
    RouterModule.forRoot(routes)
  ],
  exports: [
    RouterModule
  ]

})
export class AppRoutingModule{

}
```

- A página inicial da aplicação passará a exibir um link para login. Ele será exibido em seu cabeçalho, o que quer dizer que precisamos ajustar o arquivo **cabecalho.component.html**. Veja a Listagem 2.5.6.

Listagem 2.5.6

```
<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>
    <li><a mat-button routerLink="/login" routerLinkActive="mat-accent">Login</a></li>
  </ul>
</mat-toolbar>
```

- Abra a página inicial e veja que os links no menu merecem ajustes de estilos. Queremos que eles fiquem lado a lado. Para isso, basta alterar o display da lista não ordenada para flex. Essa é uma regra CSS que deve ser escrita no arquivo **cabecalho.component.css.** Veja a Listagem 2.5.7.

Listagem 2.5.7

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

.separador{
  flex: 1;
}

ul {
  display: flex;
}
```

- Teste novamente a página para ver o resultado.

- Clicando no link de login, vemos que o form de login também merece alguns ajustes visuais. Neste momento, seus elementos devem estar um do lado do outro. Para dar-lhe um novo visual, defina as regras CSS da Listagem 2.5.8 no arquivo **login.component.css**.

Listagem 2.5.8

```
mat-form-field {
  width: 100%;
}
mat-spinner{
  margin: auto;
}
```

- Faremos com que o Angular **intercepte** a **submissão** do form e nos entregue um objeto JSON com os dados dele, os quais serão utilizados no método a ser registrado no evento **submit**. Além disso, adicionamos **validações** típicas aos campos. A Listagem 2.5.9 ilustra esses detalhes. Estamos no arquivo **login.component.html**.

Listagem 2.5.9

```
<mat-card>
  <mat-spinner *ngIf="estaCarregando"></mat-spinner>
  <form (submit)="onLogin(loginForm)" #loginForm="ngForm" *ngIf="!estaCarregando">
    <mat-form-field>
      <input matInput name="email" ngModel #emailInput="ngModel" type="email"
placeholder="Email" required email>
      <mat-error *ngIf="emailInput.invalid">Digite um email válido</mat-error>
    </mat-form-field>
    <mat-form-field>
      <input matInput name="password" #passwordInput="ngModel" ngModel type="password"
placeholder="Senha" required>
      <mat-error *ngIf="passwordInput.invalid">Senha inválida</mat-error>
    </mat-form-field>
    <button mat-raised-button color="accent" type="submit">Login</button>
  </form>
</mat-card>
```

- Defina o método onLogin no arquivo **login.component.ts** e exiba o **value** do form recebido no console, como na Listagem 2.5.10.

Listagem 2.5.10

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

@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css']
})
export class LoginComponent implements OnInit {
  estaCarregando: boolean = false;
  onLogin (form: NgForm){
    console.log (form.value);
  }
  constructor() { }
  ngOnInit(): void {  }}
```

- Preencha os campos e-mail e senha com valores quaisquer (use um e-mail com formato válido) e verifique o console do Chrome Dev Tools (CTRL+SHIFT+I, aba console).

- A seguir, vamos implementar o form para **cadastro de novos usuários**. O cadastro utilizada e-mail e senha e, portanto, os forms são praticamente idênticos. Assim, copie e cole o conteúdo do arquivo **login.component.html** para o arquivo **signup.component.html** e, a seguir, faça os ajustes destacados na Listagem 2.5.11.

Listagem 2.5.11

```
<mat-card>
  <mat-spinner *ngIf="estaCarregando"></mat-spinner>
  <form (submit)="onSignup(signupForm)" #signupForm="ngForm" *ngIf="!estaCarregando">
    <mat-form-field>
      <input matInput name="email" ngModel #emailInput="ngModel" type="email"
placeholder="Email" required email>
      <mat-error *ngIf="emailInput.invalid">Digite um email válido</mat-error>
    </mat-form-field>
    <mat-form-field>
      <input matInput name="password" #passwordInput="ngModel" ngModel type="password"
placeholder="Senha" required>
      <mat-error *ngIf="passwordInput.invalid">Senha inválida</mat-error>
    </mat-form-field>
    <button mat-raised-button color="accent" type="submit">Cadastrar</button>
  </form>
</mat-card>
```

- Crie o método **onSignup** e a varíavel **estaCarregando** no arquivo **signup.component.ts**, como mostra a Listagem 2.5.12.

Listagem 2.5.12

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

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

  onSignup(form: NgForm){
    console.log (form.value);
  }

  estaCarregando: boolean = false;

  constructor() { }

  ngOnInit(): void {
  }

}
```

- Para permitir que a visualização do form de cadastro, iremos adicionar uma nova rota ao roteador do Angular, no arquivo **app-routing.module.ts**. O componente será acessível por meio do padrão host:porta/signup. Veja a Listagem 2.5.13.

Listagem 2.5.13

```
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';
import { LoginComponent } from './auth/login/login.component';
import { SignupComponent } from './auth/signup/signup.component';
const routes: Routes = [
  { path: '', component: ClienteListaComponent },
  {path: 'criar', component: ClienteInserirComponent},
  {path: 'editar/:idCliente', component: ClienteInserirComponent},
  {path: 'login', component: LoginComponent},
  {path: 'signup', component: SignupComponent}
];
@NgModule({
  imports: [
    RouterModule.forRoot(routes)
  ],
  exports: [
    RouterModule
  ]

})
export class AppRoutingModule{

}
```

- O cabeçalho, exibido na página inicial, dará acesso também a esse novo form. Para isso, basta adicionar um novo item de lista no arquivo **cabecalho.component.html**. Veja a Listagem 2.5.14.

Listagem 2.5.14

```
<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>
    <li><a mat-button routerLink="/login" routerLinkActive="mat-accent">Login</a></li>
    <li><a mat-button routerLink="/signup" routerLinkActive="mat-accent">Cadastrar</a></li>
  </ul>
</mat-toolbar>
```

- Assim como feito com o componente de login, defina as regras CSS da Listagem 2.5.15 no arquivo **signup.component.css.**

Listagem 2.5.15

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

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

2.6 (Autenticação: Back End) O Back End precisa lidar com requisições para lidar com o cadastro de novos usuários e com a verificação para login de usuários existentes. Para isso, definiremos novas rotas analogamente ao que foi feito para o tratamento de clientes.

- Comece criando um novo arquivo chamado **usuarios.js** na pasta **backend/rotas.** A primeira rota que criaremos lidará com o cadastro. A Listagem 2.6.1 mostra o conteúdo inicial do arquivo **usuarios.js**.

Listagem 2.6.1

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

router.post('/signup', (req, res, next) => {

});



module.exports = router;
```

- No arquivo **app.js**, precisamos registrar também o roteador utilizado para manipulação de usuários. Veja a Listagem 2.6.2.

Listagem 2.6.2

```
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');
const usuarioRoutes = require ('./rotas/usuarios');


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);
app.use ('/api/usuario', usuarioRoutes);


module.exports = app;
```

- Para cadastrar um novo usuário, precisamos começar definindo um novo modelo do Mongoose, como mostra a Listagem 2.6.3. Crie um arquivo chamado **usuario.js** na pasta **model** do Back End para fazer essa definição.

Listagem 2.6.3

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

const usuarioSchema = mongoose.Schema({
  email: {type: String, required: true, unique: true},
  password: {type: String, required: true}
})

module.exports = mongoose.model("Usuario", usuarioSchema);
```

Nota: **required** é um validador. Caso o campo envolvido não tenha valor associado, um erro acontece. No entando, **unique** não é um validador. Ele somente indica ao Mongoose e ao MongoDB que o campo envolvido não possui repetições, o que os permite realizar certas otimizações. Cadastrar dois documentos com valor igual ao campo com restrição unique não causa um erro.

- Para fazer validações a campos que utilizam **unique**, utilizaremos um pacote próprio para isso, que pode ser instalado com

npm install --save mongoose-unique-validator

- Para colocá-lo em funcionamento, basta aplicá-lo ao modelo como um plugin. Veja a Listagem 2.6.4.

Listagem 2.6.4

```
const mongoose = require ('mongoose');
const uniqueValidator = require ('mongoose-unique-validator');

const usuarioSchema = mongoose.Schema({
  email: {type: String, required: true, unique: true},
  password: {type: String, required: true}
});

usuarioSchema.plugin(uniqueValidator);

module.exports = mongoose.model("Usuario", usuarioSchema);
```

- Com o modelo pronto, podemos continuar com a implementação do endpoint que faz inserções na base. Para isso, basta construir um novo objeto do tipo Usuario configurando as suas propriedades com os valores recebidos na requisição. A seguir, chamamos o método save. Veja a Listagem 2.6.5. Estamos no arquivo **usuarios.js** do Back End.

Listagem 2.6.5

```
const express = require ('express');
const router = express.Router();
const Usuario = require ('../models/usuario');

router.post('/signup', (req, res, next) => {
  const usuario = new Usuario ({
    email: req.body.email,
    password: req.body.password
  })
  usuario.save();
});



module.exports = router;
```

- A aplicação está salvando a senha do usuário sem criptografia, o que pode ser uma péssima ideia. Não queremos que o sistema seja invadido e que os invasores consigam visualizar as senhas. Não queremos nem mesmo que nós mesmos tenhamos acesso às senhas. Por isso, vamos instalar um pacote que oferece funcionalides de criptografia. Ele pode ser instalado com

npm install --save bcrypt

- O método **hash** do pacote bcrypt é capaz de criptografar um texto recebido como parâmetro. Passamos também um número inteiro (o conhecido **salt round**) que indica quantas “unidades de tempo” estamos dispostos a esperar para que o hash esteja completo. Quanto maior o número, mais tempo leva para o hash ficar pronto e mais difícil será para descobrir a senha por força bruta.Neste exemplo, usaremos o valor 10. Pode ser interessante usar um valor aleatório de salt round para cada geração nova, o que tende a dificultar ainda mais a descoberta de senhas por força bruta e tende a fazer com que o hash gerado seja diferente mesmo para senhas iguais. Veja seu uso na Listagem 2.6.6.

Listagem 2.6.6

```
const express = require ('express');
const router = express.Router();
const Usuario = require ('../models/usuario');
const bcrypt = require ('bcrypt');

router.post('/signup', (req, res, next) => {
  bcrypt.hash (req.body.password, 10)
  .then(hash => {
    const usuario = new Usuario ({
      email: req.body.email,
      password: hash
    })
    usuario.save();
  })
});



module.exports = router;
```

- A seguir, tratamos a resposta vinda do MongoDB. A requisição pode ser atendida com sucesso e também pode falhar, talvez por conta da validação imposta pelo **unique**. Vamos tratar esse possível erro com um bloco **catch**. Veja a Listagem 2.6.7.

Listagem 2.6.7

```
router.post('/signup', (req, res, next) => {
  bcrypt.hash (req.body.password, 10)
  .then(hash => {
    const usuario = new Usuario ({
      email: req.body.email,
      password: hash
    })
    usuario.save()
    .then(result => {
      res.status(201).json({
        mensagem: "Usuario criado",
        resultado: result
      });
    })
    .catch(err => {
      res.status(500).json({
        erro: err
      })
    })
  })
});
```

2.7 (Autenticação: Requisições do Front ao Back) Para enviar requisições para o Back End solicitando o cadastro de novos usuários a partir da aplicação Angular, iremos criar um novo serviço cuja finalidade será disponibilizar funcionalidades que envolvem dados de usuários, de maneira análoga ao serviço de manipulação de clientes que temos no momento.

- Comece criando o serviço com

ng g s auth/usuario --skipTests

- A seguir, injete uma instância do cliente HTTP e crie o método que será utilizado para a criação de usuários, como ilustra a Listagem 2.7.1.

Listagem 2.7.1

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http'

@Injectable({
  providedIn: 'root'
})
export class UsuarioService {

  constructor(private httpClient: HttpClient) {

   }
   criarUsuario (email: string, senha: string){

   }
}
```

- Crie um novo modelo para representar os dados que são enviados para o servidor em operações de autenticação. Ele terá ambos os campos e-mail e senha obrigatórios. Para isso, crie um novo arquivo chamado **auth-data.model.ts** na pasta **src/app/auth/** da aplicação Angular. Seu conteúdo aparece na Listagem 2.7.2.

Listagem 2.7.2

```
export interface AuthData{
  email: string;
  password: string;
}
```

- De volta ao arquivo **usuario.service.ts**, construa um objeto do tipo AuthData usando os dados recebidos como parâmetro pelo método criarUsuario. A seguir, faça uma requisição POST. Veja a Listagem 2.7.3.

Listagem 2.7.3

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http'
import { AuthData } from './auth-data.model'

@Injectable({
  providedIn: 'root'
})
export class UsuarioService {

  constructor(private httpClient: HttpClient) {

  }
  criarUsuario(email: string, senha: string) {
    const authData: AuthData = {
      email: email,
      password: senha
    }
    this.httpClient.post("http://localhost:3000/api/usuario/signup", authData).subscribe(resposta
=> {
      console.log(resposta)
    });
  }
}
```

- A seguir, chamamos o método criarUsuario quando o usuário clica no botão de cadastro. Para isso, precisamos pedir ao Angular que injete uma instância de UsuarioService no componente **SignupComponent**, definido no arquivo **signup.component.ts**. Antes da submissão, verificamos se o form passou em todos os testes de validação. Veja a Listagem 2.7.4.

Listagem 2.7.4

```
import { Component, OnInit } from '@angular/core';
import { NgForm } from '@angular/forms';
import { UsuarioService } from '../usuario.service'

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

  onSignup(form: NgForm){
    if (form.invalid)return;
    this.usuarioService.criarUsuario(form.value.email, form.value.password);
  }

  estaCarregando: boolean = false;

  constructor(private usuarioService: UsuarioService) { }

  ngOnInit(): void {
  }

}
```

- Tente cadastrar um usuário. A seguir, tente cadastrar um novo usuário usando o mesmo e-mail. Verifique os resultados no Chrome Dev Tools (CTRL+SHIFT+I, aba console).

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.