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 gráficas 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)

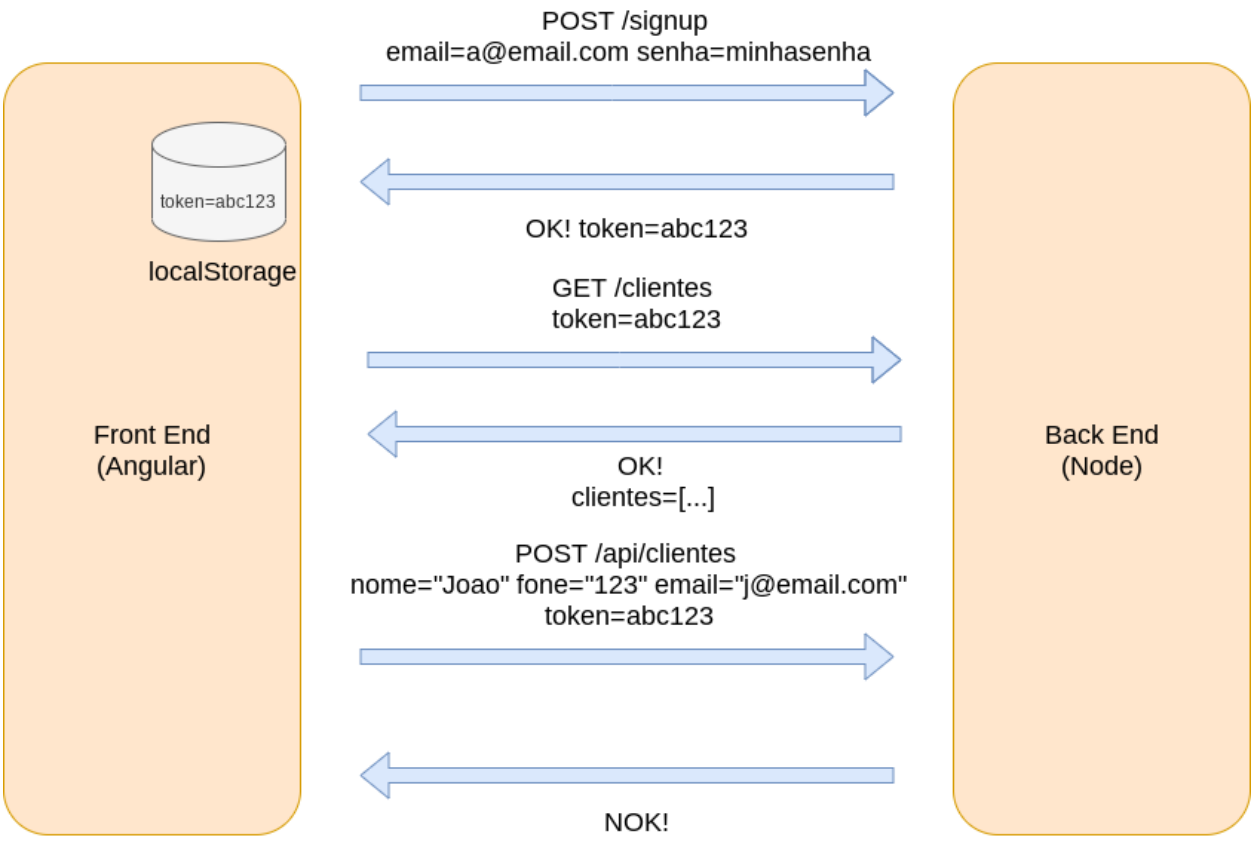
- Execute cada um deles em um terminal separado e mantenha ambos em execução.

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 (Entendendo o processo de autenticação) A fim de acessar determinadas funcionalidades disponibilizadas pelo Back End, o usuário terá de inserir suas "credenciais". Neste caso, seu e-mail e senha. Essa operação utiliza um objeto conhecido como **token**. A aplicação Front End envia os dados de autenticação para o Back End que, a seguir, verificar se os dados são válidos. Em caso positivo, o Back End devolve um **token** (que é uma sequência de caracteres). Cabe à aplicação Front End armazenar esse token localmente (do lado do cliente) e enviar esse token a cada requisição que fizer ao Back End. Para cada requisição, o token é utilizado para o processo de **autorização**. Ou seja, para verificar se a aplicação que solicita determinada operação tem autorização para fazê-lo. Veja a Figura 2.2.1.

Figura 2.2.1



2.3 (Bloqueando acesso às operações de remoção e atualização) Somente usuários que tenham feito o cadastro de um cliente podem fazer a sua remoção ou atualizar os seus dados. Podemos implementar essa funcionalidade utilizando o id de usuário que agora faz parte da definição do modelo de cliente.

- No Back End, o método **put** é o responsável pela atualização. A sua definição aparece no arquivo **backend/rotas/clientes**. Quando fazemos uma atualização no MongoDB, a resposta que obtemos inclui, entre outras coisas, o número de objetos que foram atualizados. Esse valor aparece associado à chave **nModified.** O código da Listagem 2.3.1 mostra como podemos verificar a estrutura do objeto devolvido pelo MongoDB. O resultado deve ser parecido com aquele exibido pela Figura 2.3.1.

Listagem 2.3.1

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

Figura 2.3.1

```
{
  n: 1,
  nModified: 1,
  opTime: {
    ts: Timestamp { _bsontype: 'Timestamp', low_: 7, high_: 1605624542 },
    t: 11
  },
  electionId: 7fffffff000000000000000b,
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp { _bsontype: 'Timestamp', low_: 7, high_: 1605624542 },
    signature: { hash: [Binary], keyId: [Long] }
  },
  operationTime: Timestamp { _bsontype: 'Timestamp', low_: 7, high_: 1605624542 }
}
```

- Perceba como o campo **nModified** pode ser usado para decidir se a operação foi realizada com sucesso ou não. A Listagem 2.3.2 mostra como: se **nModified** for maior do que zero, pelo menos um (neste caso, exatamente um) cliente terá sido atualizado. Devolvemos 200 para o cliente. Caso contrário, não houve atualização e isso irá acontecer caso o cliente sendo atualizado não tenha sido cadastrado pelo usuário tentando a atualização. Neste caso devolvemos 401 para o cliente.

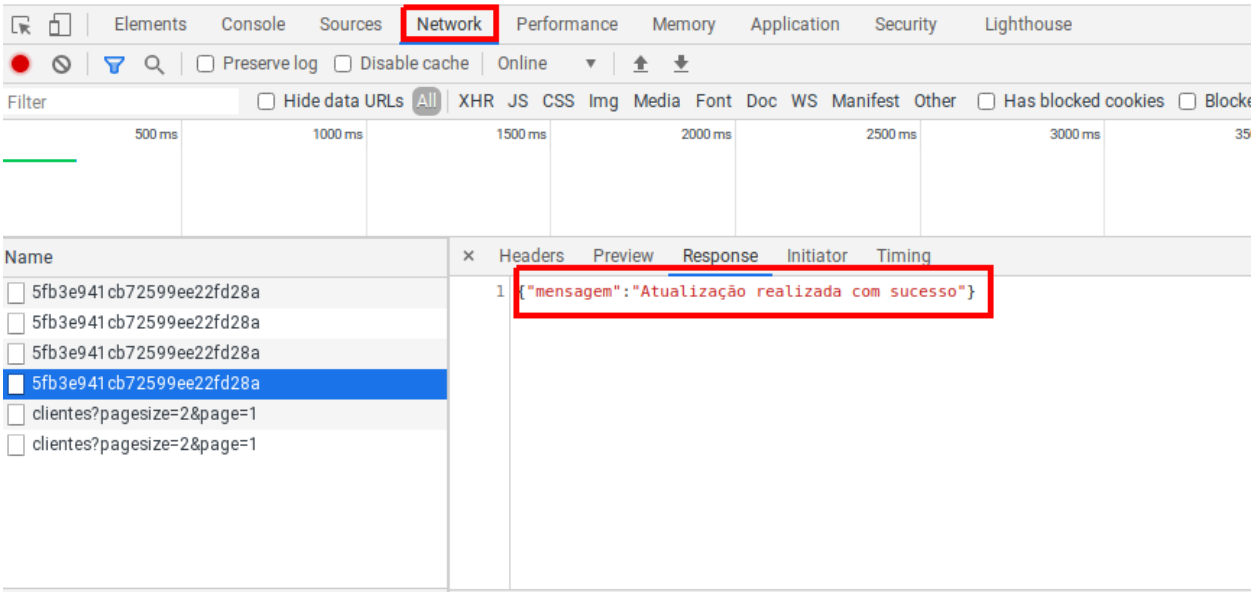
Listagem 2.3.2

```
router.put(
  "/:id",
  checkAuth,
multer({ storage: armazenamento }).single('imagem'),
(req, res, next) => {
  console.log (req.file);
  let imagemURL = req.body.imagemURL;//tentamos pegar a URL já existente
  if (req.file) { //mas se for um arquivo, montamos uma nova
    const url = req.protocol + "://" + req.get("host");
    imagemURL = url + "/imagens/" + req.file.filename;
  }
  const cliente = new Cliente({
    _id: req.params.id,
    nome: req.body.nome,
    fone: req.body.fone,
    email: req.body.email,
    imagemURL: imagemURL
  });
  Cliente.updateOne({ _id: req.params.id, criador: req.dadosUsuario.idUsuario  }, cliente)
    .then((resultado) => {
      //console.log(resultado)
      if (resultado.nModified > 0){
        res.status(200).json({ mensagem: 'Atualização realizada com sucesso' })
      }
      else{
        res.status(401).json({ mensagem: 'Atualização não permitida' })
      }
    });
});
```

- Quando fazemos uma atualização usando o usuário responsável pelo cadastro do cliente sendo atualizado, o resultado é aquele exibido pela Figura 2.3.2. Ela mostra o **Network** do Chrome Dev Tools (CTRL+SHIFT+I). Note que ele pode exibir uma lista de requisições. Clique sobre elas para encontrar a desejada.

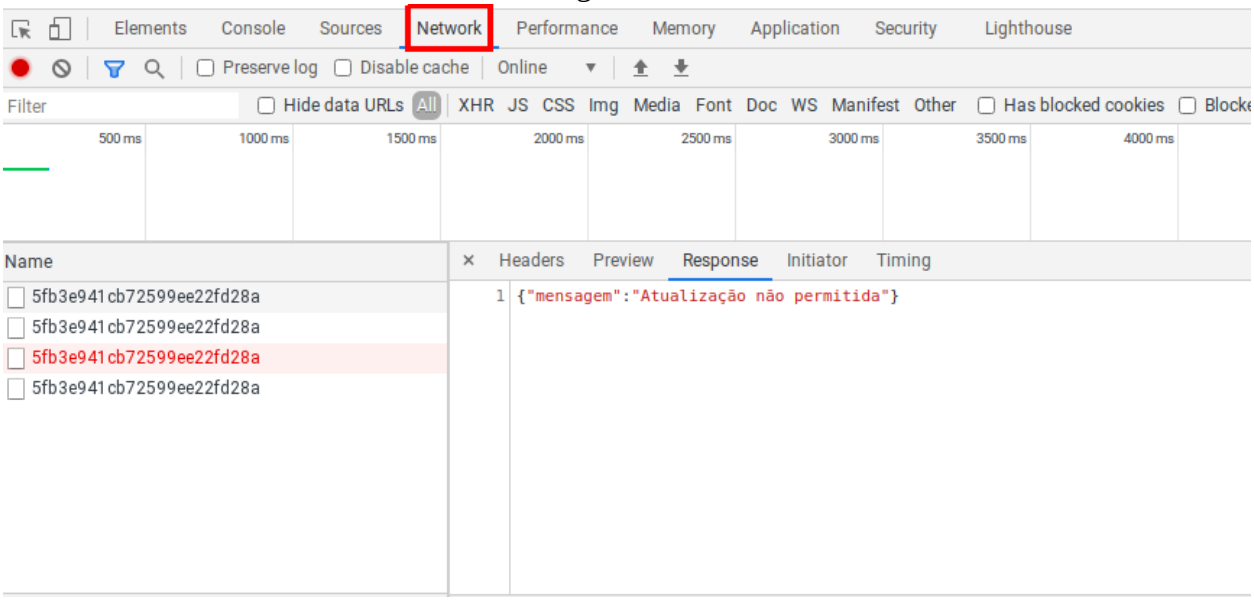
Nota: Pode ser necessário fazer a requisição depois de abrir o Chrome Dev Tools para que ela apareça na aba Network.

Figura 2.3.2



- A Figura 2.3.3 mostra o resultado quando a atualização é feita por um usuário que não fez o cadastro do cliente sendo atualizado. Para simular esse erro, você precisar deslogar da aplicação e logar com um usuário diferente.

Figura 2.3.3



- Note que a aplicação Angular precisa ser ajustada para refletir o retorno que recebeu do Back End. No momento, o spinner deve ficar na tela indefinidamente. Em breve ajustaremos isso.

- A mesma lógica pode ser aplicada para a operação de remoção de clientes. Contudo, o campo **nModified não faz parte** do retorno do MongoDB neste caso. Utilizaremos a variável chamada **n**. Veja a Listagem 2.3.3. Continuamos no arquivo **backend/rotas/clientes.js**.

Listagem 2.3.3

```
router.delete('/:id', checkAuth, (req, res, next) => {
  //console.log("id: ", req.params.id);
  Cliente.deleteOne({ _id: req.params.id, criador:
req.dadosUsuario.idUsuario }).then((resultado) => {
    if (resultado.n > 0){
      res.status(200).json({ mensagem: "Cliente removido" })
    }
    else{
      res.status(401).json({ mensagem: "Remoção não permitida" })
    }
  });
});
```

- Faça novos testes como aqueles feitos para a atualização.

- A validação no Back End evita que as operações sejam realizadas, independentemente de os endpoints serem acessados pela aplicação Angular ou qualquer outro cliente (como o Postman, por exemplo). Contudo, na aplicação Angular, faz sentido deixar de exibir os botões de edição e remoção para os clientes que não possam ser atualizados e removidos. Isso pode ser feito utilizando-se o id do usuário logado: a renderização do componente que contém os dois botões se torna condicional. O primeiro passo é, a partir do Back End, entregar o id do usuário para a aplicação Angular. Isso pode ser feito no arquivo **backend/rotas/usuarios.js**. No método post, onde o login acontece, além de devolvermos o token, devolvemos também o id do usuário. Veja a Listagem 2.3.4.

Listagem 2.3.4

```
router.post('/login', (req, res, next) => {
  let user;
  Usuario.findOne({ email: req.body.email }).then(u => {
    user = u;
    if (!u) {
      return res.status(401).json({
        mensagem: "email inválido"
      })
    }
    return bcrypt.compare(req.body.password, u.password);
  })
  .then(result => {
    if (!result){
      return res.status(401).json({
        mensagem: "senha inválida"
      })
    }
    const token = jwt.sign(
      {email: user.email, id: user._id},
      'minhasenha',
      {expiresIn: '1h'}
    )
    res.status(200).json({
      token: token,
      expiresIn: 3600, //unidade pode ser qualquer, aqui estamos usando segundos
      idUsuario: user._id
    })
  })
  .catch(err => {
    return res.status(401).json({
      mensagem: "Login falhou: " + err
    })})})})
```

- Na aplicação Angular, precisamos considerar a existência do campo **idUsuario** devolvido pelo Back End. Isso é responsabilidade do serviço de manipulação de usuários, definido no arquivo

usuario.service.ts. Começamos definindo uma nova variável de instância, como na Listagem 2.3.5.

Listagem 2.3.5

```
...
export class UsuarioService {
  private autenticado: boolean = false;
  private token: string;
  private tokenTimer: NodeJS.Timer;
  private idUsuario: string;
  private authStatusSubject = new Subject<boolean>();
...
```

- No método **login**, ainda no arquivo **usuario.service.ts**, obtemos o id devolvido pelo Back End e atribuímos à variável recém-criada. Veja a Listagem 2.3.6. Lembre-se de ajustar o tipo genérico, indicando que idUsuario agora faz parte das propriedades do objeto esperado como resposta.

Listagem 2.3.6

```
login (email: string, senha: string){
    const authData: AuthData = {
      email: email,
      password: senha
    }
    this.httpClient.post<{token: string, expiresIn: number, idUsuario:
string}>("http://localhost:3000/api/usuario/login", authData).subscribe(resposta => {
      this.token = resposta.token;
      if (this.token){
        const tempoValidadeToken = resposta.expiresIn;
        this.tokenTimer = setTimeout(() => {
          console.log("rodou logout");
          this.logout()
        },tempoValidadeToken * 1000);
        this.autenticado = true;
        this.idUsuario = resposta.idUsuario;
        this.authStatusSubject.next(true);
        this.salvarDadosDeAutenticacao(this.token, new Date(new Date().getTime() +
tempoValidadeToken * 1000 ));
        this.router.navigate(['/'])
      }
    });  }
```

- Para que os componentes possam conhecer o id do usuário atualmente logado, vamos definir um método que o devolve, ainda no arquivo **usuario.service.ts**. Veja a Listagem 2.3.7.

Listagem 2.3.7

```
public getIdUsuario (){
   return this.idUsuario;
}
```

- Um dos componentes interessados no id do usuário é o ClienteListaComponent, definido no arquivo **cliente-lista.component.ts**. Vamos adicionar a ele uma variável de instância e atribuir a ela o id do usuário logo depois de ele ser construído, no método ngOnInit. Veja a Listagem 2.3.8.

Listagem 2.3.8

```
…
export class ClienteListaComponent implements OnInit, OnDestroy {
  clientes: Cliente[] = [];
  private clientesSubscription: Subscription;
  public estaCarregando = false;
  totalDeClientes: number = 0;
  totalDeClientesPorPagina: number = 2;
  paginaAtual: number = 1; //definir
  opcoesTotalDeClientesPorPagina = [1, 2, 5, 10];
  public autenticado: boolean = false;
  public idUsuario: string;
  private authObserver: Subscription;
…

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

- Quando o usuário fizer logout, precisamos cuidar para que o valor de idUsuario deixe de existir. Isso pode ser feito no próprio método logout, no arquivo **usuario.service.ts**, como mostra a Listagem 2.3.9.

Listagem 2.3.9

```
  logout(){
```

```
  this.token = null;
  this.authStatusSubject.next(false);
  clearTimeout(this.tokenTimer);
  this.idUsuario = null;
  this.removerDadosDeAutenticacao()
  this.router.navigate(['/'])
 }
```

- Assim como o token, o id de usuário também será armazenado em meio persistente do lado do cliente, usando a API **localStorage**. Veja a Listagem 2.3.10. Estamos no arquivo **usuario.service.ts**.

Listagem 2.3.11

```
private salvarDadosDeAutenticacao (token: string, validade: Date, idUsuario: string){
  localStorage.setItem('token', token);
  localStorage.setItem('validade', validade.toISOString());
  localStorage.setItem('idUsuario', idUsuario);
 }
```

- Também precisamos remover o id de usuário no método em que removemos os dados do armazenamento local. Veja a Listagem 2.3.12. Ainda estamos no arquivo **usuario.service.ts**.

Listagem 2.3.12

```
private removerDadosDeAutenticacao (){
  localStorage.removeItem('token');
  localStorage.removeItem('validade');
  localStorage.removeItem('idUsuario')
 }
```

- O método **obterDadosDeAutenticacao**, definido no arquivo **usuario.service.ts**, passa a devolver um objeto que inclui o id de usuário, como mostra a Listagem 2.3.13.

Listagem 2.3.13

```
private obterDadosDeAutenticacao (){
   const token = localStorage.getItem('token');
   const validade = localStorage.getItem('validade');
   const idUsuario = localStorage.getItem("idUsuario");
   return (token && validade) ? { token: token, validade: new Date(validade), idUsuario:
idUsuario} : null;
  }
```

- O método que realiza a autenticação automática quando o usuário fecha o navegador e o reabre sem que o token tenha expirado também se preocupa em trazer dados da memória local, o que precisa incluir o id de usuário. Ele está definido no arquivo **usuario.service.ts**. Veja a Listagem 2.3.14.

Listagem 2.3.14

```
autenticarAutomaticamente (){
   const dadosAutenticacao = this.obterDadosDeAutenticacao();
   if (dadosAutenticacao){
     //pegamos a data atual
     const agora = new Date ();
     //verificamos a diferenca entre a validade e a data atual
     const diferenca = dadosAutenticacao.validade.getTime() - agora.getTime();
     //se a diferença for positiva, o token ainda vale
     console.log (diferenca);
     if (diferenca > 0){
       this.token = dadosAutenticacao.token;
       console.log(dadosAutenticacao);
       this.autenticado = true;
       this.idUsuario = dadosAutenticacao.idUsuario;
       //diferença ja esta em milissegundos, não multiplique!
       this.tokenTimer = setTimeout(() => {
          this.logout()
        }, diferenca);
       this.authStatusSubject.next(true);
     }
   }
  }
```

- Embora espere três argumentos, o método **salvarDadosDeAutenticacao**, definido no arquivo **usuario.service.ts**, o método login o coloca em execução passando somente dois. É preciso ajustar isso também, incluindo o id de usuário na lista de parâmetros. Veja a Listagem 2.3.15.

Listagem 2.3.15

```
login (email: string, senha: string){
   const authData: AuthData = {
     email: email,
     password: senha
   }
   this.httpClient.post<{token: string, expiresIn: number, idUsuario:
string}>("http://localhost:3000/api/usuario/login", authData).subscribe(resposta => {
     this.token = resposta.token;
     if (this.token){
       const tempoValidadeToken = resposta.expiresIn;
       this.tokenTimer = setTimeout(() => {
         console.log("rodou logout");
         this.logout()
       },tempoValidadeToken * 1000);
       this.autenticado = true;
       this.idUsuario = resposta.idUsuario;
       this.authStatusSubject.next(true);
       this.salvarDadosDeAutenticacao(this.token, new Date(new Date().getTime() +
tempoValidadeToken * 1000), this.idUsuario);
       this.router.navigate(['/'])
     }
   });
  }
```

- Cabe ao template do componente **ClienteListaComponent**, definido no arquivo **cliente-lista.component.ts** decidir se o componente que abriga os botões de edição e remoção deve fazer parte da árvore DOM. Uma condição já existe: o usuário precisa estar autenticado. Agora, além disso, também é preciso que seu id seja igual ao id do criador, existente em cada cliente. Veja a Listagem 2.3.16.

Listagem 2.3.16

```
<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 *ngIf="autenticado && idUsuario === cliente.criador">
      <a mat-button color="primary" [routerLink]="['/editar', cliente.id]">EDITAR</a>
      <button mat-button color="warn" (click)="onDelete(cliente.id)">REMOVER</button>
    </mat-action-row>
  </mat-expansion-panel>
</mat-accordion>
<mat-paginator
    *ngIf="clientes.length > 0"
    [length]="totalDeClientes"
    [pageSize]="totalDeClientesPorPagina"
    [pageSizeOptions]="opcoesTotalDeClientesPorPagina"
    (page)="onPaginaAlterada($event)"
></mat-paginator>
<p class="mat-body-1" style="text-align: center;" *ngIf="clientes.length <= 0 && !
estaCarregando">
  Nenhum cliente cadastrado
</p>
```

- O aplicativo apresenta um erro pois o modelo de cliente ainda não inclui o campo criador. Isso pode ser ajustado no arquivo **cliente.model.ts**, como mostra a Listagem 2.3.17.

Listagem 2.3.17

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

- Quando um cliente é atualizado pelo método **atualizarCliente** definido no arquivo **clientes.service.ts**, construímos um objeto do tipo cliente para enviar os dados para o Back End. No momento, esta construção ainda desconsidera o campo criador. Veja a correção na Listagem 2.3.18. O valor, contudo, não será enviado ao Back End pela aplicação cliente: o próprio Back End fará esse tratamento. Isso é uma questão de **segurança**: o usuário final não pode tentar alterar o id de usuário manualmente, por exemplo. Ele é decodificado a partir do token. Dado que somente o Back End tem a senha usada para a sua codificação, somente ele pode decodificá-lo.

Listagem 2.3.18

```
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,
       criador: null
     }
   }
   console.log (typeof(clienteData));
   this.httpClient.put(`http://localhost:3000/api/clientes/${id}`, clienteData)
   .subscribe((res => {
     this.router.navigate(['/'])
   }));
 }
```

- Para tratar o id de usuário no Back End, vamos atualizar o método **put**, definido em **backend/rotas/clientes.js**, como na Listagem 2.3.19.

Listagem 2.3.19

```
router.put(
  "/:id",
  checkAuth,
multer({ storage: armazenamento }).single('imagem'),
(req, res, next) => {
  console.log (req.file);
  let imagemURL = req.body.imagemURL;//tentamos pegar a URL já existente
  if (req.file) { //mas se for um arquivo, montamos uma nova
    const url = req.protocol + "://" + req.get("host");
    imagemURL = url + "/imagens/" + req.file.filename;
  }
  const cliente = new Cliente({
    _id: req.params.id,
    nome: req.body.nome,
    fone: req.body.fone,
    email: req.body.email,
    imagemURL: imagemURL,
    criador: req.dadosUsuario.idUsuario
  });
  Cliente.updateOne({ _id: req.params.id, criador: req.dadosUsuario.idUsuario  }, cliente)
    .then((resultado) => {
      //console.log(resultado)
      if (resultado.nModified > 0){
        res.status(200).json({ mensagem: 'Atualização realizada com sucesso' })
      }
      else{
        res.status(401).json({ mensagem: 'Atualização não permitida' })
      }
    });
});
```

- A aplicação Angular ainda mostra um erro no método **ngOnInit** definido no arquivo **cliente-inserir.component.ts**. Ele registra um Observer que recebe dados do Back End e constrói um cliente sem considerar o campo criador. O ajuste é análogo. Veja a Listagem 2.3.20.

Listagem 2.3.20

```
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: dadosCli.imagemURL,
        criador: dadosCli.criador
       };
      this.form.setValue({
        nome: this.cliente.nome,
        fone: this.cliente.fone,
        email: this.cliente.email,
        imagem: this.cliente.imagemURL
      })
     });
    }
    else{
      this.modo = "criar";
      this.idCliente = null; }});}
```

- Como o retorno do método **getCliente** definido no arquivo **clientes.service.ts** não inclui o campo criador, a aplicação permanece exibindo um erro. O ajuste é exibido na Listagem 2.3.21.

Listagem 2.3.21

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

- Agora faça o seguinte teste: logue na aplicação com um usuário e faça o cadastro de um cliente. Verifique se os botões de remoção e edição podem ser visualizados e utilizados. Deslogue da aplicação e logue com outro usuário. Você deveria ver o cliente cadastrado sem contudo incluir os botões de remoção e edição.

Dica: Se necessário, reinicie o servidor. Como sempre, certifique-se de salvar todos os arquivos.

2.4 (Tratando erros: exibição no Front End) Conforme o usuário interage com a aplicação, ela envia requisições ao HTTP na expectativa de receber os dados desejados pelo usuário. Ocorre que diferentes tipos de erros/exceções podem acontecer. Por exemplo, o servidor pode estar indisponível ou sobrecarregado. O usuário pode estar temporariamente sem Internet. Entre outros. Nesta seção veremos como fazer o tratamento de possíveis erros na aplicação.

- Quando o usuário tenta realizar um novo cadastro (para novo usuário, não cliente) utilizando um e-mail que já foi previamente cadastrado, o Back End devolve o código 500 e uma mensagem de erro. A aplicação Angular, no entanto, não apresenta feedback visual para o usuário final. Vamos começar ajustando o método **criarUsuario** definido no arquivo **usuario.service.ts**. Especificaremos um objeto que contém duas propriedades: **next** e **error**: uma função que entra em execução quando a requisição é atendida com sucesso e outra que executa mediante erro, respectivamente. Ele será entregue ao método subscribe. Veja a Listagem 2.4.1.

Listagem 2.4.1

```
criarUsuario(email: string, senha: string) {
   const authData: AuthData = {
     email: email,
     password: senha
   }
   this.httpClient.post("http://localhost:3000/api/usuario/signup", authData)
   .subscribe({
     //vamos para a página principal pois o usuário foi criado com sucesso
     next: () => this.router.navigate(['/']),
     //notificamos todos os componentes que não há usuário autenticado
     error: () => this.authStatusSubject.next(false)
   });
  }
```

- A seguir, ajustamos o componente SignupComponent, definido no arquivo **signup.component.ts.** Ele passará a receber notificações do serviço de autenticação. Veja a Listagem 2.4.2.

Listagem 2.4.2

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

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

  estaCarregando: boolean = false;
  private authObserver: Subscription;

  constructor(private usuarioService: UsuarioService) { }

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


  ngOnInit(): void {
    this.authObserver = this.usuarioService.getStatusSubject()
    .subscribe(
      authStatus => this.estaCarregando = false
    )
  }

  ngOnDestroy (): void{
    this.authObserver.unsubscribe();
  }

}
```

- Faça novos testes. Crie um novo usuário com um e-mail ainda não existente. Você deveria ser levado à página de listagem de clientes (não logado, você só criou um usuário, não logou). Depois, tente criar outro usuário com o mesmo e-mail. Você deveria permanecer na tela de criação de usuários (ainda não há feedback visual, a menos de o spinner sumir).

- Quando algum tipo de erro acontecer, faremos com que a aplicação exiba um diálogo ou modal explicando o que aconteceu. Para isso, usaremos o componente **Dialog** do Angular Material. Veja a sua documentação no Link 2.4.1.

Link 2.4.1
https://material.angular.io/components/dialog/overview

- O Back End devolve erros representados com códigos do protocolo HTTP. Isso quer dizer que podemos registrar um novo **Interceptor** na aplicação Angular para lidar com eles. A ideia é escrever um único elemento que tem efeito sobre toda a aplicação. Para defini-lo, crie um arquivo chamado **erro-interceptor.ts** na raiz do projeto, lado a lado com o arquivo **app.module.ts**, por exemplo. Veja seu conteúdo inicial na Listagem 2.4.3. Ele é essencialmente o mesmo daquele que utilizamos para lidar com o token. Ele não precisa do serviço de manipulação de usuários, no entanto. Por isso, a injeção de dependência foi removida.

Listagem 2.4.3

```
import  { HttpHandler, HttpInterceptor, HttpRequest } from '@angular/common/http'

export class ErroInterceptor implements HttpInterceptor{

  intercept(req: HttpRequest<any>, next: HttpHandler){
     return next.handle(req);
  }
}
```

- O método **intercept** entra em execução e opera sobre a requisição, antes de ela ser enviada da aplicação Angular para o Back End. Podemos também especificar o que deve ser feito quando o Back End responde. Como desejamos manipular erros, aplicaremos o operador **catchError** de **rxjs/operators**. A função entregue a ele recebe como parâmetro um objeto do tipo HttpErrorResponse que, intuitivamente, contém informações sobre o erro produzido no Back End. Veja a Listagem 2.4.4. Vamos apenas exibir o objeto de erro no console, por enquanto. Para que o método compile, ele precisa devolver um novo Observable, que pode ser produzido pelo operador **throwError** de rxjs.

Listagem 2.4.4

```
import { HttpErrorResponse, HttpHandler, HttpInterceptor, HttpRequest } from
'@angular/common/http'
import { throwError } from 'rxjs';
import { catchError } from 'rxjs/operators'
export class ErroInterceptor implements HttpInterceptor {

  intercept(req: HttpRequest<any>, next: HttpHandler) {
    return next.handle(req).pipe(
      catchError((erro: HttpErrorResponse) => {
        console.log("ErroInterceptor: " + JSON.stringify(erro));
        return throwError(erro);
      }));
  }
}
```

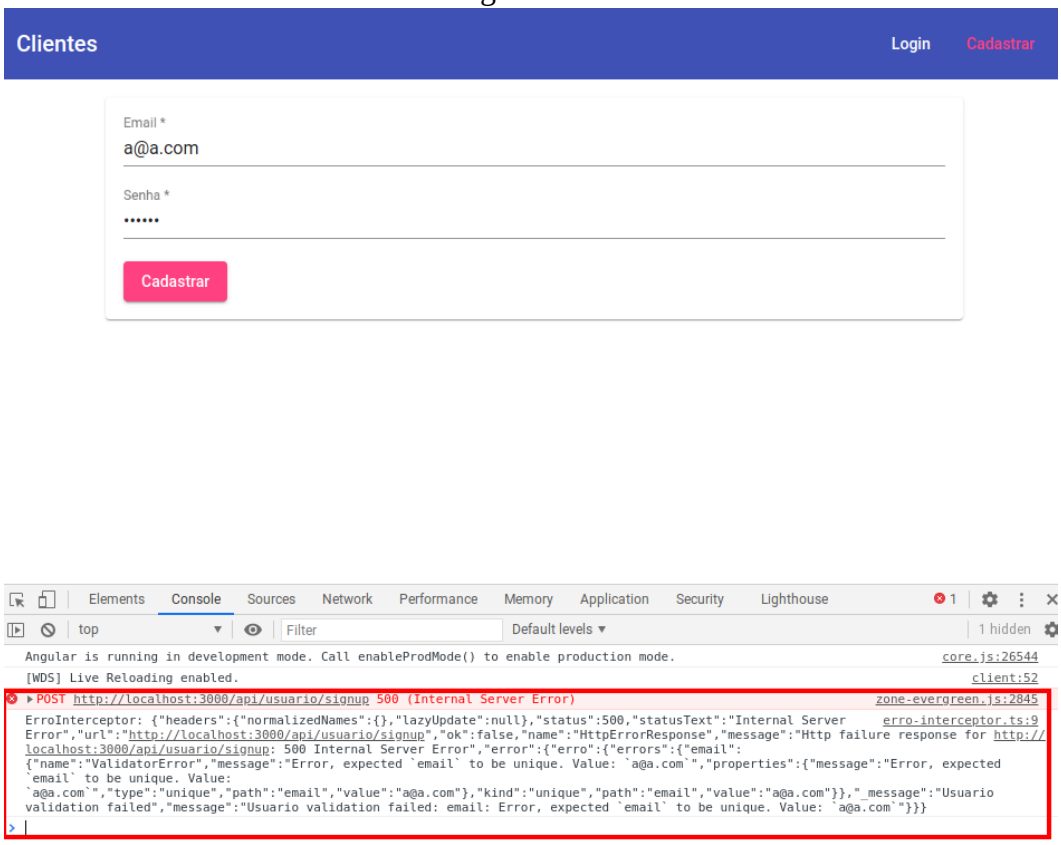
- Para que o Angular tome conhecimento da existência do novo interceptor, ele precisa ser registrado como provider no módulo da aplicação, definido no arquivo **app.module.ts**. Veja a Listagem 2.4.5.

Listagem 2.4.5

```
...
import { ErroInterceptor } from './erro-interceptor'
...
@NgModule({
  declarations: [
    AppComponent,
    ...
  ],
  imports: [
    BrowserModule,
    ...
    HttpClientModule,
  ],
  providers: [
    { provide: HTTP_INTERCEPTORS, useClass: AuthInterceptor, multi: true },
    { provide: HTTP_INTERCEPTORS, useClass: ErroInterceptor, multi: true },

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

- Para testar o novo interceptador, tente fazer um novo cadastro de usuário utilizando um e-mail já existente na base. Veja o resultado no Chrome Dev Tools (CTRL+SHIFT+I). Ele deve ser parecido com o que exibe a Figura 2.4.1.

Figura 2.4.1



- O feedback visual sobre esse possível erro será dado ao usuário por meio de um Dialog do Angular Material. O primeiro passo para utilizá-lo é importar o seu módulo, o que pode ser feito no módulo da aplicação, definido no arquivo **app.module.ts**. Veja a Listagem 2.4.6.

Listagem 2.4.6

```
...
import { MatInputModule } from '@angular/material/input';
import { MatCardModule } from '@angular/material/card';
import { MatButtonModule } from '@angular/material/button';
import { MatDialogModule } from '@angular/material/dialog'
import { MatToolbarModule } from '@angular/material/toolbar';
import { MatExpansionModule } from '@angular/material/expansion';
import { MatProgressSpinnerModule } from '@angular/material/progress-spinner'
import { MatPaginatorModule } from '@angular/material/paginator';
…
@NgModule({
  declarations: [
    AppComponent,
    ...
  ],
  imports: [
   ...
    MatDialogModule,
    ...
  ],
  providers: [
...
```

- A seguir, injetamos uma instância de **MatDialog** no Interceptor, definido no arquivo **erro-interceptor.ts**. Veja a Listagem 2.4.7.

Listagem 2.4.7

```
import { HttpErrorResponse, HttpHandler, HttpInterceptor, HttpRequest } from
'@angular/common/http'
import { Injectable } from '@angular/core';
import { MatDialog } from '@angular/material/dialog';
import { throwError } from 'rxjs';
import { catchError } from 'rxjs/operators'

@Injectable()
export class ErroInterceptor implements HttpInterceptor {

  constructor (private dialog: MatDialog){}

  intercept(req: HttpRequest<any>, next: HttpHandler) {
    return next.handle(req).pipe(
      catchError((erro: HttpErrorResponse) => {
        return throwError(erro);
      }));
  }
}
```

- O component MatDialog funciona da seguinte forma: Construímos um componente Angular comum e especificamos que seu template deve ser renderizado pelo MatDialog. Assim, crie um novo componente com o comando

ng g c erro/erro --skipTests

Ou seja, estamos criando um componente em uma pasta chamada erro (que será criada pelo CLI) e cujo nome será ErroComponent. Ele terá uma propriedade chamada **mensagem** que será exibida em seu template por meio de interpolação. Inclua a propriedade mensagem no arquivo **erro/erro.component.ts** como na Listagem 2.4.8.

Listagem 2.4.8

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

@Component({
  selector: 'app-erro',
  templateUrl: './erro.component.html',
  styleUrls: ['./erro.component.css']
})
export class ErroComponent implements OnInit {
  mensagem: string = "Erro";

  constructor() { }

  ngOnInit(): void {
  }

}
```

- O conteúdo inicial do template do componente de erro é dado na Listagem 2.4.9. Sua definição deve ser feita no arquivo **erro/erro.component.html**.

Listagem 2.4.9

```
<h1>Erro!</h1>
<p>{{mensagem}}</p>
```

- Como sabemos, alguns componentes Angular são instanciados pelo uso de seu seletor: Por exemplo, um componente cujo seletor é cliente-inserir, pode ser instanciado com **<cliente-inserir></cliente-inserir>**. Também sabemos que componentes podem ser instanciados pelo roteador Angular. Com o código **{path: 'login', component: LoginComponent}**, por exemplo, especificamos que o componente chamado LoginComponent deve ser instanciado quando houver uma requisição em host:porta/login. No caso de uso em que estamos, o componente que exibe a mensagem de erro não será instanciado de nenhuma das duas formas. Ele será instanciado dinamicamente, pelo MatDialog. Quando isso ocorre, precisamos informar ao Angular que isso irá acontecer. Podemos fazê-lo adicionando o componente ao vetor associado à chave **entryComponents** do módulo da aplicação, no arquivo **app.module.ts**. Veja a Listagem 2.4.10.

Nota: Isso depende da versão do Angular que você estiver utilizando. Desde a versão 9 essa propriedade já não é necessária. O mecanismo de compilação e ambiente de execução do Angular chamado **Ivy** inclui recursos como compilação AOT (Ahead of Time) que modernizam o funcionamento do framework. Para checar a versão do Angular, use **ng version** no diretório raiz do seu projeto. Procure por Angular (não Angular CLI). Para conhecer mais sobre o Ivy, visite o Link 2.4.1.

Link 2.4.1
https://angular.io/guide/ivy

Listagem 2.4.10

```
...
@NgModule({
  declarations: [
    AppComponent,
    ...
  providers: [
    { provide: HTTP_INTERCEPTORS, useClass: AuthInterceptor, multi: true },
    { provide: HTTP_INTERCEPTORS, useClass: ErroInterceptor, multi: true },

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

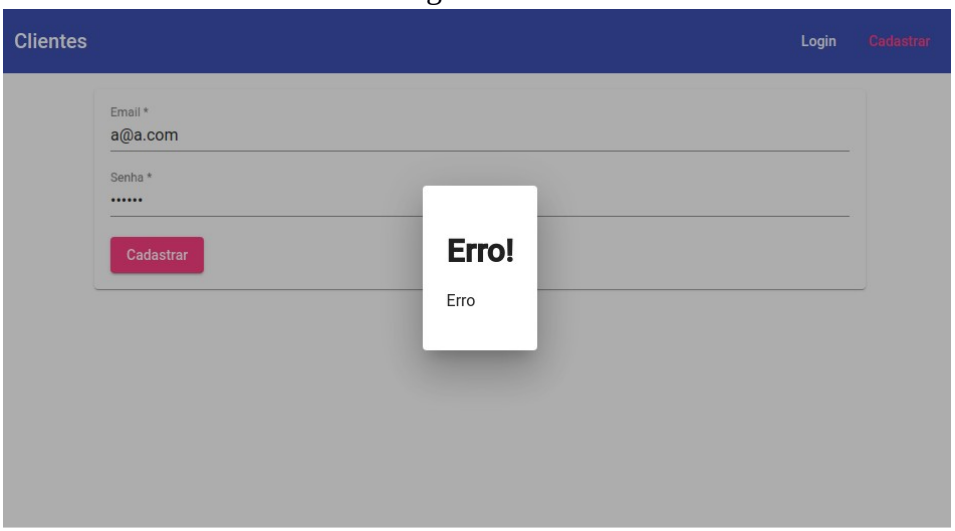
- Utilizamos o método **open** de MatDialog para exibir um diálogo. O método recebe o nome da classe cujo template deve ser exibido no diálogo. Ele será chamado pelo interceptor de erros, definido no arquivo **erro-interceptor.ts**, quando ele detectar algum erro entregue pelo Back End. Veja a Listagem 2.4.11.

Listagem 2.4.11

```
import { HttpErrorResponse, HttpHandler, HttpInterceptor, HttpRequest } from
'@angular/common/http'
import { Injectable } from '@angular/core';
import { MatDialog } from '@angular/material/dialog';
import { throwError } from 'rxjs';
import { catchError } from 'rxjs/operators'
import { ErroComponent } from './erro/erro/erro.component';

@Injectable()
export class ErroInterceptor implements HttpInterceptor {

  constructor (private dialog: MatDialog){}

  intercept(req: HttpRequest<any>, next: HttpHandler) {
    return next.handle(req).pipe(
      catchError((erro: HttpErrorResponse) => {
        this.dialog.open(ErroComponent)
        return throwError(erro);
      }));
  }
}
```

- Faça u m novo teste: tente cadastrar um novo usuário utilizando um e-mail já existente na base. O resultado deve ser parecido com aquele exibido pela Figura 2.4.2.

Figura 2.4.2



- Note que o diálogo está exibindo uma mensagem genérica. Desejamos fazer com que ele exiba uma mensagem que revele exatamente o que aconteceu. O método **open** pode receber um segundo argumento que é um objeto que inclui os dados a serem exibidos. Iremos implementá-lo da seguinte forma. Caso o Back End tenha gerado um erro que possui uma mensagem, iremos exibi-la. Caso contrário, exibimos uma mensagem genérica. Veja a Listagem 2.4.12. Estamos no arquivo **erro-interceptor.ts**.

Listagem 2.4.12

```
import { HttpErrorResponse, HttpHandler, HttpInterceptor, HttpRequest } from
'@angular/common/http'
import { Injectable } from '@angular/core';
import { MatDialog } from '@angular/material/dialog';
import { throwError } from 'rxjs';
import { catchError } from 'rxjs/operators'
import { ErroComponent } from './erro/erro/erro.component';

@Injectable()
export class ErroInterceptor implements HttpInterceptor {

  constructor (private dialog: MatDialog){}

  intercept(req: HttpRequest<any>, next: HttpHandler) {
    return next.handle(req).pipe(
      catchError((erro: HttpErrorResponse) => {
        let msg = "Erro. Tente novamente mais tarde."
        if (erro.error.mensagem) {
          msg = erro.error.mensagem;
        }
        this.dialog.open(ErroComponent, {
          data: {
            message: msg
          }
        })

        return throwError(erro);
      }));
  }
}
```

- O componente ErroComponent precisa acessar a mensagem que lhe está sendo enviada pelo interceptor. Ele pode fazê-lo utilizando a constante chamada **MAT_DIALOG_DATA**, que precisamos injetar por meio de seu construtor, no arquivo **erro/erro.component.ts**. Assim, a mensagem que o componente possuia por padrão pode ser desconsiderada. Veja a Listagem 2.4.13.

Listagem 2.4.13

```
import { Component, Inject, OnInit } from '@angular/core';
import { MAT_DIALOG_DATA } from '@angular/material/dialog';

@Component({
  selector: 'app-erro',
  templateUrl: './erro.component.html',
  styleUrls: ['./erro.component.css']
})
export class ErroComponent implements OnInit {
  mensagem: string;

  constructor(@Inject(MAT_DIALOG_DATA) public data: {message:string}) {
    this.mensagem = data.message;
   }

  ngOnInit(): void {
  }
}
```

- Neste caso, o diálogo deve exibir a mensagem genérica pois neste momento a propriedade **message** não existe. Em breve ajustaremos o Back End para que ela seja produzida e entregue à aplicação Angular adequadamente.

- Podemos estilizar os elementos do template do componente de erro, no arquivo **erro/erro.component.html**, utilizando diretivas próprias do Angular Material, além de adicionar um botão para que o usuário possa fechar o diálogo de maneira mais intuitiva. Veja a Listagem 2.4.14. A diretiva **mat-dialog-close** é responsável por fazer com que o diálogo desapareça quando o botão for clicado.

Listagem 2.4.14

```
<h1 mat-dialog-title>Erro!</h1>
<div mat-dialog-content>
  <p class="mat-body-1">
    {{mensagem}}
  </p>
</div>
<div mat-dialog-actions>
  <button mat-button mat-dialog-close>OK</button>
</div>
```

2.5 (Tratando erros: produzindo mensagens no servidor) Em nossa aplicação, algumasmensagens de erro exibidas no Front End serão produzidas no Back End. Uma delas é aquela que indica que o usuário está utilizando dados inválidos para fazer o cadastro.

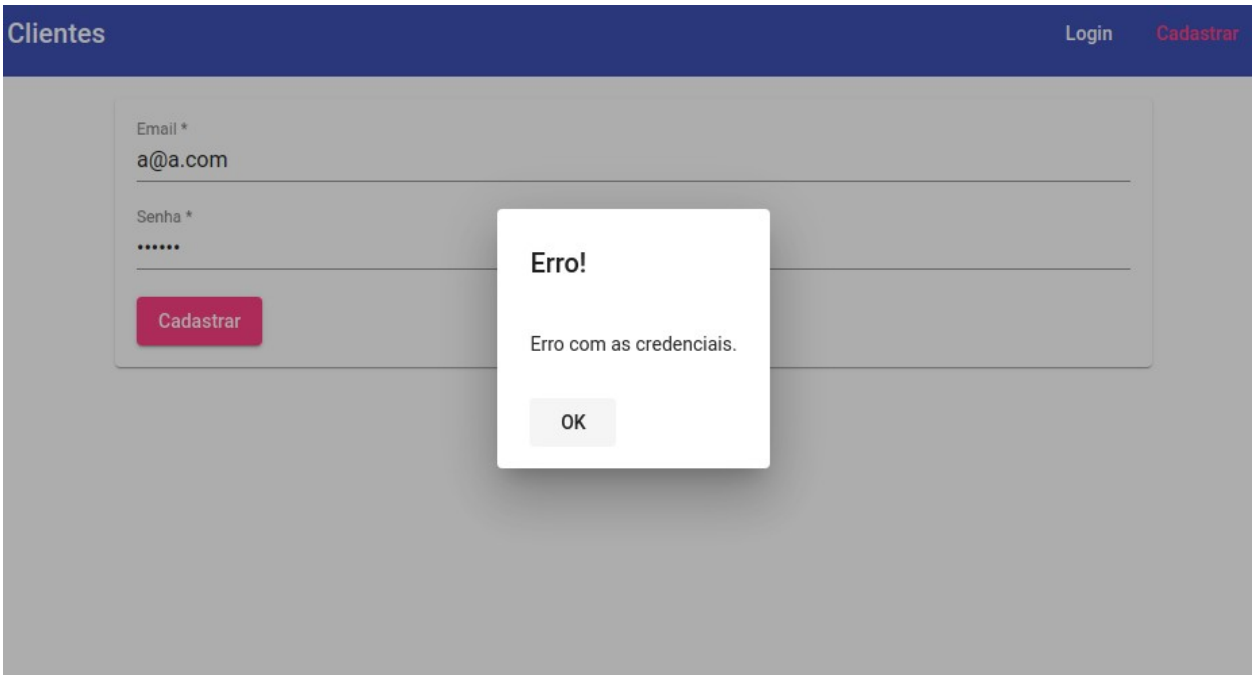
- No arquivo **backend/rotas/usuarios.js**, faça o ajuste da Listagem 2.5.1.

Listagem 2.5.1

```
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({
        mensagem: "Erro com as credenciais."
      })
    })
  })
})
```

- Tente, novamente, cadastrar um usuário com e-mail já existente na base. O resultado deve ser parecido com aquele que a Figura 2.5.1 exibe.

Figura 2.5.1



- No método de login, fazemos um ajuste para evitar que a mensagem exibida ao usuário inclua detalhes sobre o funcionamento da aplicação. Veja a Listagem 2.5.2.

Listagem 2.5.2

```
router.post('/login', (req, res, next) => {
  let user;
  Usuario.findOne({ email: req.body.email }).then(u => {
    user = u;
    if (!u) {
      return res.status(401).json({
        mensagem: "email inválido"
      })
    }
    return bcrypt.compare(req.body.password, u.password);
  })
  .then(result => {
    if (!result){
      return res.status(401).json({
        mensagem: "senha inválida"
      })
    }
    const token = jwt.sign(
      {email: user.email, id: user._id},
      'minhasenha',
      {expiresIn: '1h'}
    )
    console.log ("user: " + JSON.stringify(user));
    res.status(200).json({
      token: token,
      expiresIn: 3600, //unidade pode ser qualquer, aqui estamos usando segundos
      idUsuario: user._id
    })
  })
  .catch(err => {
    return res.status(401).json({
      mensagem: "Erro no login. Tente novamente mais tarde."
    })
  })
})
```

- Na aplicação Angular, faça testes tentando logar com e-mail e/ou senha inválidos. Note que a mensagem “Erro no login. Tente novamente mais tarde.”, neste caso, somente vai aparecer caso uma exceção aconteça, por exemplo, uma tentativa de acesso ao MongoDB que falhou, talvez por ele estar indisponível ou pelo acesso via IP ter sido bloqueado.

- Ajustamos também a mensagem devolvida pelo middleware responsável por verificar o token a cada requisição. Ele está definido no arquivo **backend/middleware/check-auth.js**. Veja a Listagem 2.5.3.

Listagem 2.5.3

```
const jwt = require ('jsonwebtoken');

module.exports = (req, res, next) => {
  //split quebra a string em 2, usando espaço como separador
  //ou seja, gera um vetor em que a primeira posição
  //contém a palavra Bearer e a segunda contém o token desejado
  try{
    const token = req.headers.authorization.split (" ")[1];
    const tokenDecodificado = jwt.verify(token, "minhasenha");
    //as propriedades que acessamos de tokenDecodificado são aquelas que codificamos ao
chamar o método sign, no endpoint de login
    req.dadosUsuario = {
      email: tokenDecodificado.email,
      idUsuario: tokenDecodificado.id
    }
    next()
  }
  //se não existir o header authorization, tratamos o erro
  catch (err){
    res.status(401).json({
      mensagem: "Problemas com o token. Autenticação não realizada."
    })
  }

}
```

- Ainda não há tratamento de erro para a operação de inserção de clientes, que está definida no arquivo **backend/rotas/clientes.js**. Ele pode ser especificado como os demais, usando um bloco catch. Veja a Listagem 2.5.4.

Listagem 2.5.4

```
router.post('', checkAuth, 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}`,
    criador: req.dadosUsuario.idUsuario

  })
  //apagar tudo isso agora
  //temporário
  //console.log(req.dadosUsuario);
  //res.status(200).json({});
  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
        }
      })
    })
    .catch(erro => {
      res.status(500).json({
        mensagem: "Inserção de cliente falhou. Tente novamente mais tarde."
      })
    })
});
```

- De maneira análoga, adicionamos um tratamento de erro à operação de atualização de clientes, também definida no arquivo **backend/rotas/clientes.js**. Veja a Listagem 2.5.5.

Listagem 2.5.5

```
router.put(
  "/:id",
  checkAuth,
multer({ storage: armazenamento }).single('imagem'),
(req, res, next) => {
  console.log (req.file);
  let imagemURL = req.body.imagemURL;//tentamos pegar a URL já existente
  if (req.file) { //mas se for um arquivo, montamos uma nova
    const url = req.protocol + "://" + req.get("host");
    imagemURL = url + "/imagens/" + req.file.filename;
  }
  const cliente = new Cliente({
    _id: req.params.id,
    nome: req.body.nome,
    fone: req.body.fone,
    email: req.body.email,
    imagemURL: imagemURL,
    criador: req.dadosUsuario.idUsuario
  });
  Cliente.updateOne({ _id: req.params.id, criador: req.dadosUsuario.idUsuario  }, cliente)
    .then((resultado) => {
      //console.log(resultado)
      if (resultado.nModified > 0){
        res.status(200).json({ mensagem: 'Atualização realizada com sucesso' })
      }
      else{
        res.status(401).json({ mensagem: 'Atualização não permitida' })
      }
    })
    .catch(erro => {
      res.status(500).json({
        mensagem: "Atualização de cliente falhou. Tente novamente mais tarde."
      })
    })
});
```

- Também adicionamos tratamento de erros à busca de clientes, também definida no arquivo **backend/rotas/clientes.js**, como na Listagem 2.5.6.

Listagem 2.5.6

```
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
      });
    })
    .catch(erro => {
      res.status(500).json({
        mensagem: "Busca de clientes falhou. Tente novamente mais tarde."
      })
    })
});
```

- A Listagem 2.5.7 mostra o tratamento para a operação de busca de um único cliente, ainda no arquivo **backend/rotas/clientes.js**.

Listagem 2.5.7

```
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!" })
  })
    .catch(erro => {
      res.status(500).json({
        mensagem: "Busca de cliente falhou. Tente novamente mais tarde."
      })
    })
});
```

- A Listagem 2.5.8 mostra o tratamento de erro para a operação de remoção.

Listagem 2.5.9

```
router.delete('/:id', checkAuth, (req, res, next) => {
  //console.log("id: ", req.params.id);
  Cliente.deleteOne({ _id: req.params.id, criador:
req.dadosUsuario.idUsuario }).then((resultado) => {
    if (resultado.n > 0) {
      res.status(200).json({ mensagem: "Cliente removido" })
    }
    else {
      res.status(401).json({ mensagem: "Remoção não permitida" })
    }
  })
    .catch(erro => {
      res.status(500).json({
        mensagem: "Remoção de cliente falhou. Tente novamente mais tarde."
      })
    })
});
```

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.