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-o, 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.

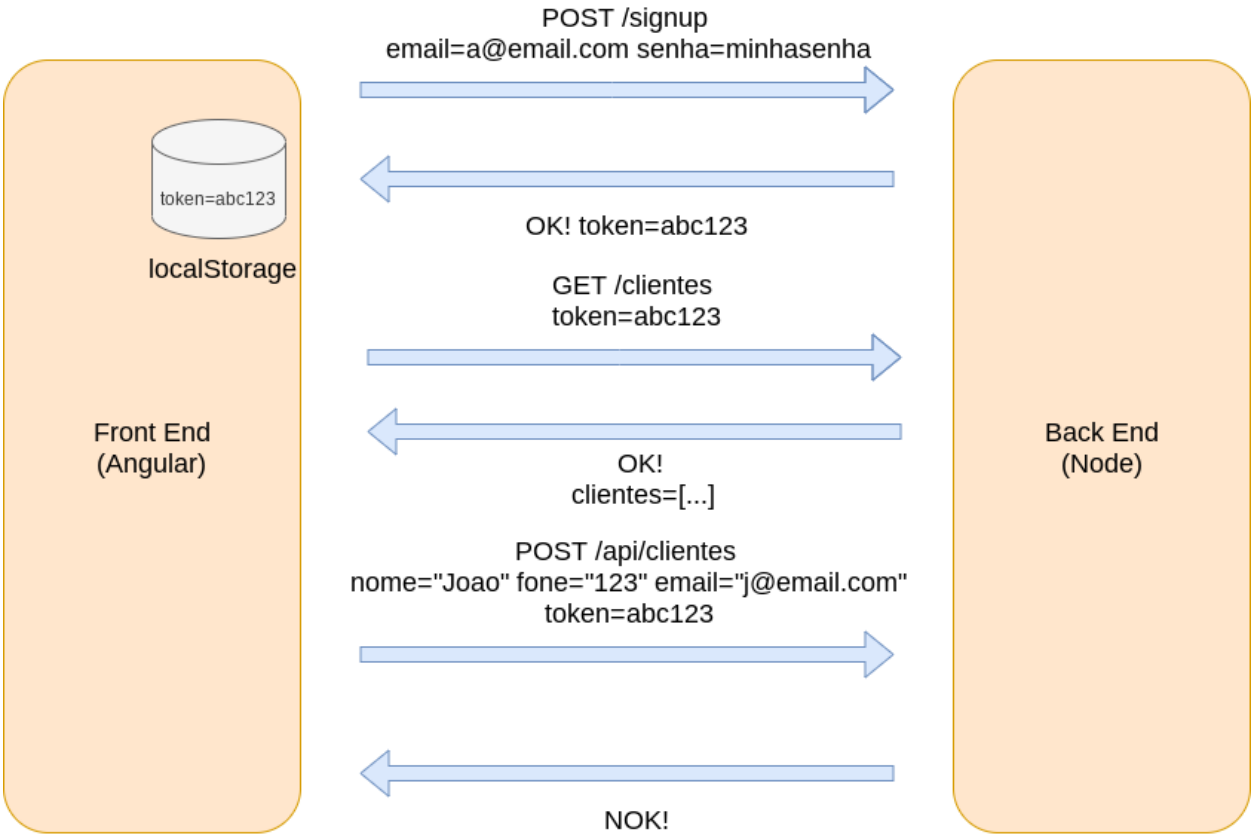


Figura 2.2.1

2.3 (Implementando a autenticação: Back End) Começamos implementando os detalhes necessários para que o usuário possa fazer login na aplicação, ou seja, o processo de autenticação.

- Comece criando uma nova rota no arquivo **backend/rotas/usuarios.js**. Veja a Listagem 2.3.1.

Listagem 2.3.1

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


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

})

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

- No próximo passo, vamos verificar se o e-mail enviado existe na base. Isso pode ser feito com o método **find**. Ele devolve uma promise com o usuário, caso ele exista. Caso não exista, o

atendimento à requisição termina por aí, informamos ao Front End que o e-mail não existe e sequer começamos a manipular a senha. Veja a Listagem 2.3.2.

Listagem 2.3.2

```
router.post('/login', (req, res, next) => {
  Usuario.findOne({ email: req.body.email }).then(u => {
    if (!u) {
      return res.status(401).json({
        mensagem: "email inválido"
      })
    }
  })
})
```

- Caso o e-mail exista na base, devemos verificar se a senha que o usuário enviou está, de fato, associada a este e-mail. A senha que armazenamos na base está criptografada, o que quer dizer que a comparação não pode ser direta. Além disso, não é possível obter a senha a partir do seu código hash (aquele armazenado no banco). Isso faz sentido pois, caso contrário, o mecanismo de criptografia empregado não traria nenhuma vantagem de segurança. Por outro lado, a partir da senha podemos, novamente, obter o código hash. Portanto, é preciso pegar a senha da requisição, calcular seu hash e comparar com o que está armazenado no banco. Faremos isso usando a função **compare** de bcrypt, que já se encarrega de fazer isso. O resultado é uma promise, cujo resultado trataremos a seguir. Veja a Listagem 2.3.3.

Listagem 2.3.3

```
router.post('/login', (req, res, next) => {
  Usuario.findOne({ email: req.body.email }).then(u => {
    if (!u) {
      return res.status(401).json({
        mensagem: "email inválido"
      })
    }
    return bcrypt.compare(req.body.password, u.password);
  })
  .then(result => {

  })
  .catch(err => {

  })
})
```

- O resultado do método compare é um valor booleano. Ou seja, a variável result especificada no bloco then pode valer true ou false. Caso seja false, sabemos que a senha era inválida. Durante esse processo, sabemos que falhas podem acontecer. O bloco catch se encarrega de lidar com elas. Veja a Listagem 2.3.4.

Listagem 2.3.4

```
router.post('/login', (req, res, next) => {
  Usuario.findOne({ email: req.body.email }).then(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"
      })
    }
  })
  .catch(err => {
    return res.status(401).json({
      mensagem: "Login falhou: " + err
    })
  })
})
```

- Caso nenhuma falha aconteça e ambos e-mail e senha estejam corretos, o Back End pode gerar o token para entregar para o Front End. Trata-se de um **JSON Web Token**. Trata-se de um padrão aberto cuja especificação é feita pela **RFC 7519**. Ela pode ser encontrada no Link 2.3.1. Veja também o Link 2.3.2.

Link 2.3.1
https://tools.ietf.org/html/rfc7519

Link 2.3.2
https://jwt.io/

- Utilizaremos o **pacote jsonwebtoken** para isso. Ele pode ser instalado com

npm install jsonwebtoken

- A seguir, podemos criar o token. Isso pode ser feito com o método **sign**. Ele espera alguns argumentos. Utilizaremos:

- um objeto com e-mail e id do usuário
- uma senha de conhecimento do servidor apenas
- um valor que indica em quanto tempo o token expira

Nota: Lembre-se que o token é simplesmente uma sequência de caracteres a partir da qual é possível obter o objeto envolvido em sua criação. Neste exemplo, com a senha em mãos, o Back End é capaz de obter o JSON que contém e-mail e id do usuário.

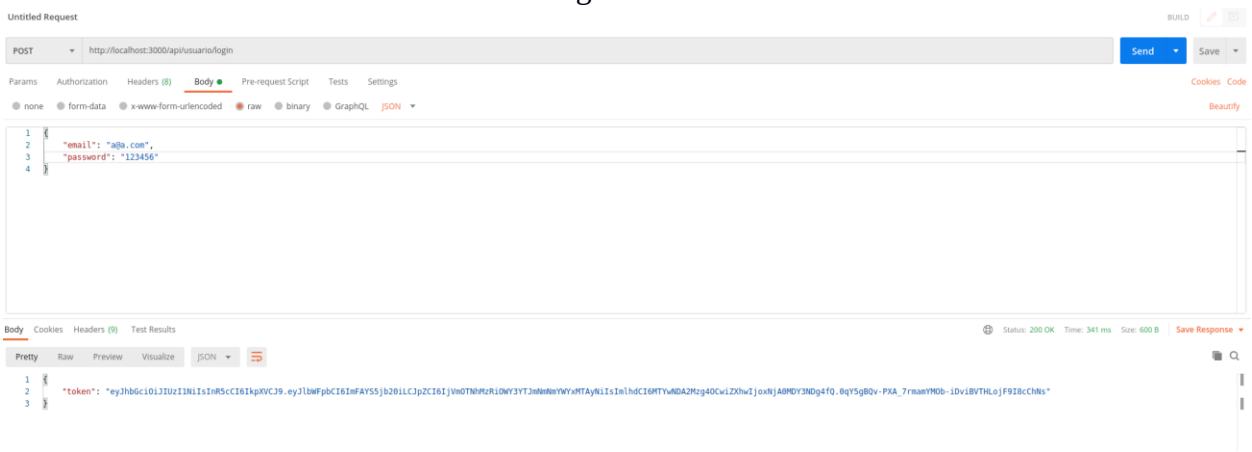
Veja a Listagem 2.3.3.

Listagem 2.3.3

```
...
const jwt = require ('jsonwebtoken');
…
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})
  })
  .catch(err => {
    return res.status(401).json({
      mensagem: "Login falhou: " + err
    })
  })
})
```

- É interessante realizar um teste e ver o resultado. Abra o Postman e faça alguns testes com email/senha válidos e inválidos. Veja um exemplo utilizando e-mail e senha válidos na Figura 2.3.1.

Figura 2.3.1



2.4 (Implementando a autenticação: Front End) Cabe ao serviço de manipulação de usuários realizar o envio de requisições ao endpoint de login disponibilizado pelo Back End. Ele envia e-mail e senha digitados pelo usuário e espera receber o token.

- Comece criando o método destacado na Listagem 2.4.1. Estamos no arquivo **src/app/auth/usuario.service.ts**.

Listagem 2.4.1

```
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)
    });
  }

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

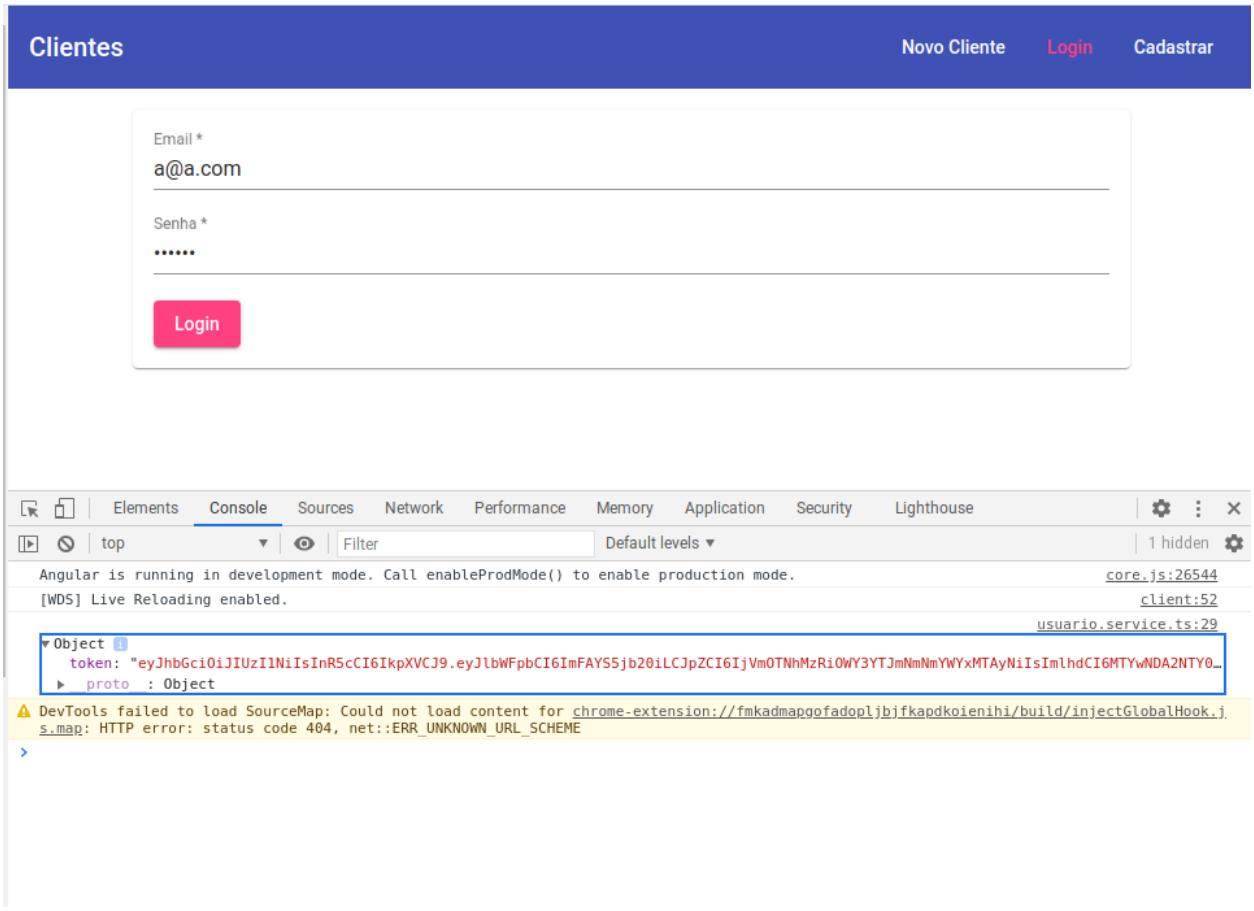
- A seguir, precisamos chamar o método login no componente responsável por essa atividade. Para tal, é preciso injetar uma instância do serviço de manipulação de usuários. O método login é chamado pelo método que entra em execução quando o usuário clica no botão de login. Veja a Listagem 2.4.2. Estamos no arquivo **src/app/login/login.component.ts**.

Listagem 2.4.2

```
onLogin (form: NgForm){
   if (form.invalid) return;
   this.usuarioService.login(form.value.email, form.value.password);
 }
```

- Utilize a aplicação Angular para fazer um novo teste e verifique o resultado no Chrome Dev Tools (CTRL + SHIFT + I, aba console). Veja a Figura 2.4.1.

Figura 2.4.1



2.5 (Autorização: Interceptando requisições) Quando uma requisição é enviada ao Back End, desejamos verificar se quem a envia pode acessar o recurso solicitado ou não. Os recursos que estarão disponíveis publicamente ou que dependem de autorização dependem, evidentemente, da natureza da aplicação. Cabe ao desenvolvedor, com base nas regras de negócio que está implementando, especificar mecanismos de autenticação para os recursos que os requerem. Nesta aplicação, as seguintes operações terão acesso **público**:

- login
- criação de novo usuário
- listagem de clientes

Por outro lado, as seguintes operações terão acesso restrito a usuários autenticados:

- criação de clientes
- atualização de clientes
- remoção de clientes

- A autorização pode ser implementada por meio da especificação de funções que interceptam as requisições enviadas ao Back End. Elas podem ser chamadas de **filtro**, **middleware** ou algo semelhante. O importante é entender que cada requisição, antes de atendida adequadamente pelo Back End, será submetida a uma função que checa se quem a envia está autorizado a acessar o recurso solicitado. Comece criando uma pasta chamada **middleware** (o nome pode ser qualquer) como **subpasta de backend**. Dentro dela, crie um arquivo chamado **check-auth.js** (esse nome também é escolhido pelo desenvolvedor).

- O processo de autenticação envolve verificar

- se a requisição inclui um token
- se o token existente na requisição é válido

No arquivo **check-auth.js**, crie a função da Listagem 2.5.1. Note que ela tenta obter o token da coleção de cabeçalhos da requisição. É comum enviar o token associado a um cabeçalho da requisição, embora não seja obrigatório. Nada impede que o cliente o envie como um parâmetro na URL, por exemplo. Perceba, também o uso do método **split**. Ocorre que estamos utilizando uma convenção comum no mercado, que consiste em enviar o token no seguinte formato:

Bearer tokenaqui

Ou seja, uma string que contém a palavra Bearer seguida de um espaço em branco seguido do token propriamente dito.

Nota: Bearer significa algo como "**detentor**".

Leia mais sobre o uso de tokens "bearer" na RFC 6750, que pode ser encontrada no Link 2.5.1.

Link 2.5.1
https://tools.ietf.org/html/rfc6750

Listagem 2.5.1

```
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];
  }
  //se não existir o header authorization, tratamos o erro
  catch (err){
    res.status(401).json({
      mensagem: "Autenticação falhou"
    })
  }

}
```

- A seguir, usamos o método **verify** de jwt que recebe o token e a senha utilizada para gerá-lo. Se o token for válido, o método somente termina a execução. Caso contrário, um erro é gerado e a execução é direcionada diretamente para o bloco catch. Chamamos, a seguir, a função **next**. Isso implica na chamada à função que o cliente solicitou a princípio. Ou seja, o filtro terminou de operar e o recurso solicitado já pode ser acessado. Veja a Listagem 2.5.2.

Listagem 2.5.2

```
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];
    jwt.verify(token, "minhasenha");
    next()
  }
  //se não existir o header authorization, tratamos o erro
  catch (err){
    res.status(401).json({
      mensagem: "Autenticação falhou"
    })
  }

}
```

- Para utilizar a função que definimos, basta especificá-la em cada rota que desejamos proteger contra acessos não autorizados. Isso é feito de maneira análoga ao que fizemos com o pacote multer. Logo depois da URL do endpoint e antes função alvo, especificamos o nome da função de verificação. As rotas de adição, atualização e remoção serão protegidas. Veja a Listagem 2.5.3. Estamos no arquivo **backend/rotas/clientes.js**.

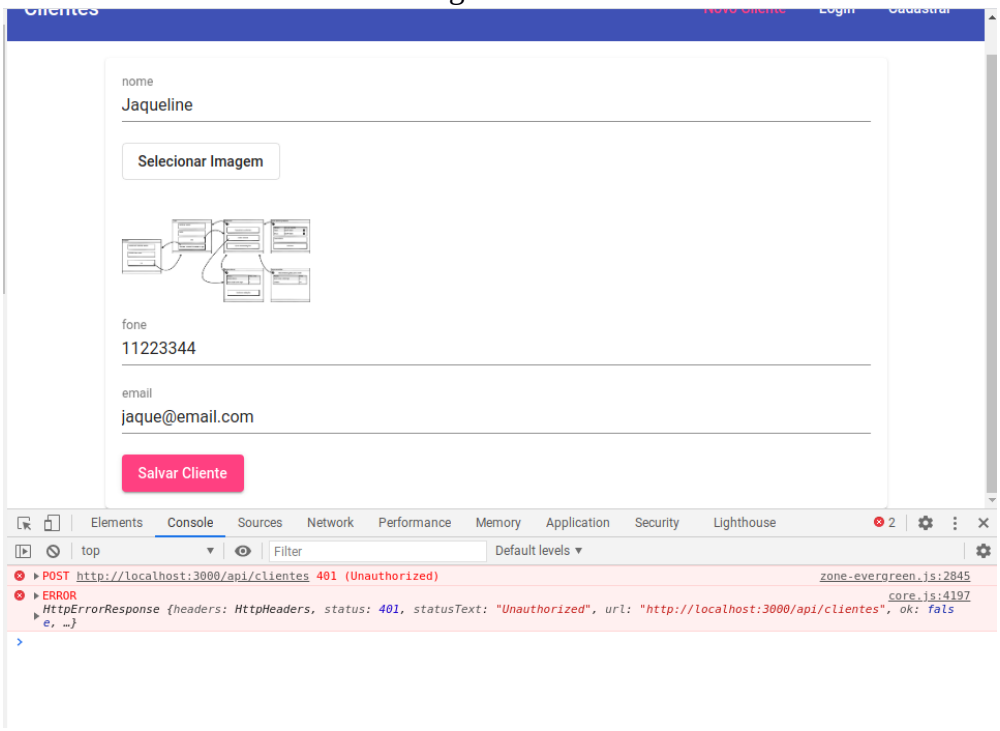
Listagem 2.5.3

```
...
const checkAuth = require ('../middleware/check-auth');
…
router.post('', checkAuth, multer({storage: armazenamento}).single('imagem'), (req, res, next)
=> {
…
router.delete('/:id', checkAuth, (req, res, next) => {
…
router.put(
  "/:id",
  checkAuth,
multer({ storage: armazenamento }).single('imagem'),
(req, res, next) => {
...
```

- Neste momento, acesse a aplicação Angular e tente fazer o cadastro de um novo cliente. Observe, no console do Chrome Dev Tools (CTRL + SHIFT + I, aba console) que o servidor devolveu o código 401, indicando que o cliente não tem autorização para acessar o recurso solicitado. Note,

contudo, que a listagem de clientes aparece corretamente, já que esse recurso não foi protegido (não aplicamos a função **checkAuth** aos métodos **get**). Veja a Figura 2.5.1.

Figura 2.5.1



2.6 (Lidando com o token no Front End) A aplicação Angular recebe o token do Back End quando o usuário informa e-mail e senha válidos. Contudo, ela ainda não o utiliza para acessar as operações às quais o usuário deveria ter acesso, o que ajustaremos agora.

- Abra o arquivo **usuario.service.ts** e crie uma nova propriedade para guardar o token uma vez que ele seja recebido, o que ocorre na função registrada com **subscribe** pela função **login**. Lembre-se de especificar explicitamente qual o tipo de retorno esperado da função post. Veja a Listagem 2.6.1.

Listagem 2.6.1

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

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

  private token: string;

  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)
    });
  }

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

- Outras partes da aplicação precisarão de acesso ao token. Uma delas é o serviço de manipulação de clientes, já que ele é quem envia requisições para cadastro, remoção e atualização de clientes, as quais somente podem ser executadas mediante a existência de um token válido. Por isso, adicione um método "getter" à classe UsuarioService (ainda no arquivo **usuario.service.ts**). Veja a Listagem 2.6.2.

Listagem 2.6.2

```
...
private token: string;
public getToken (): string{
  return this.token;
}
...
```

- A fim de adicionar o token à request, utilizaremos um recurso oferecido pelo Angular chamado **Interceptor**. Seu funcionamento é semelhante ao middleware que escrevemos no Back End. A ideia, como o nome sugere, é interceptar o envio da requisição (antes de ele acontecer, claro) e executar um trecho de código. Neste caso, adicionar o token à requisição. Para utilizar esse recurso, comece criando um arquivo chamado **auth-interceptor.ts** (escolha o nome que preferir) na pasta **src/app/auth**.

- A seguir, escreva uma classe que implementa a interface **HttpInterceptor**. Ao fazer isso, ela **adere a um contrato** que garante a existência do método chamado **intercept**, definido pela interface HttpInterceptor. Isso é fundamental pois é esse método que o Angular irá chamar automaticamente quando uma requisição Http for realizada. O método intercept recebe um objeto que representa a requisição e um objeto que viabiliza a chamada ao "próximo" método, ou seja, ao método que foi chamado a princípio, antes de a requisição ser interceptada. Repare na semelhança com a função **next** que utilizamos váris vezes no Back End. Veja a Listagem 2.6.3.

Listagem 2.6.3

```
import  { HttpHandler, HttpInterceptor, HttpRequest } from '@angular/common/http'
export class AuthInterceptor implements HttpInterceptor{
  intercept(req: HttpRequest<any>, next: HttpHandler){
    return next.handle(req);
  }
}
```

- Nosso objetivo é adicionar o token a um cabeçalho da requisição. Por isso, será necessário injetar uma instância de UsuarioService, já que é ali que definimos o método getToken. Para que a injeção de dependências possa acontecer, teremos de anotar a classe como **Injectable**. Como de costume, a injeção é feita em seu construtor. No método intercept chamamos o método getToken. Veja a Listagem 2.6.4.

Listagem 2.6.4

```
import  { HttpHandler, HttpInterceptor, HttpRequest } from '@angular/common/http'
import { Injectable } from '@angular/core';
import { UsuarioService } from './usuario.service';
@Injectable()
export class AuthInterceptor implements HttpInterceptor{
  constructor(private usuarioService: UsuarioService){

  }
  intercept(req: HttpRequest<any>, next: HttpHandler){
    const token = this.usuarioService.getToken();
    return next.handle(req);
  }
}
```

- A seguir, faremos uma cópia da requisição interceptada, pois alterar a requisição original pode dar origem a efeitos colaterais indesejados, segundo a especificação do framework. Ao fazer a cópia, adicionamos o token associado à chave **Authorization** (no Back End acessamos com letra minúscula, pois o acesso a propriedades é **case insensitive**) usando o método **set**. Talvez pareça que ele substitui o conteúdo anterior, mas não é o caso. O método set mantém o estado do objeto sobre o qual opera e adiciona o cabeçalho que recebe como parâmetro. Ao final, passamos a cópia da request para o método handle. Note a inclusão da palavra **Bearer**, esperada pelo Back End. Veja a Listagem 2.6.5.

Listagem 2.6.5

```
import  { HttpHandler, HttpInterceptor, HttpRequest } from '@angular/common/http'
import { Injectable } from '@angular/core';
import { UsuarioService } from './usuario.service';
@Injectable()
export class AuthInterceptor implements HttpInterceptor{
  constructor(private usuarioService: UsuarioService){

  }
  intercept(req: HttpRequest<any>, next: HttpHandler){
    const token = this.usuarioService.getToken();
    const copia = req.clone({
      headers: req.headers.set('Authorization', `Bearer ${token}`)
    });
    return next.handle(copia);
  }
}
```

- A simples existência da classe não é suficiente. Precisamos informar ao Angular sobre a sua existência explicitamente. Isso pode ser feito no arquivo **app.module.ts**. Ela precisa ser registrada como um **provider**. Há uma constante que o módulo HTTPClientModule utiliza para encontrar possíveis interceptadores, caso existam, é a HTTP_INTERCEPTORS. Associada a essa constante iremos configurador o nosso interceptador.

Nota: Utilizamos a propriedade **multi true** para indicar que HTTP_INTERCEPTORS é um token para um provedor múltiplo, ou seja, um provedor que pode injetar uma coleção de valores, não apenas um só. Visite o Link 2.6.1 para mais detalhes.

Link 2.6.1
https://angular.io/guide/http#provide-the-interceptor

Listagem 2.6.6

```
...
import { HttpClientModule, HTTP_INTERCEPTORS} from '@angular/common/http'
import { AuthInterceptor } from './auth/auth-interceptor'
...
@NgModule({
  declarations: [
    AppComponent,
    ClienteInserirComponent,
    CabecalhoComponent,
    ClienteListaComponent,
    LoginComponent,
    SignupComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    ReactiveFormsModule,
    BrowserAnimationsModule,
    FormsModule,
    MatInputModule,
    MatCardModule,
    MatButtonModule,
    MatToolbarModule,
    MatExpansionModule,
    MatPaginatorModule,
    MatProgressSpinnerModule,
    HttpClientModule,
  ],
  providers: [{provide: HTTP_INTERCEPTORS, useClass: AuthInterceptor}],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

- Faça novos testes na aplicação. Crie um usuário na opção cadastrar. Depois disso, faça login na opção Login. Finalmente, tente cadastrar, editar e remover clientes. Se necessário, reinicie os servidores.

2.7 (Ajustando a interface gráfica e armazenando o token) A aplicação possui o**pções que devem ser exibidas dependendo de existir um usuário logado**. Por exemplo, caso exista um usuário logado, provavelmente não desejamos exibir as opções Login e Cadastrar na barra superior. Quando não houver usuário logado, queremos desabilitar os botões Editar e Remover. Além disso, **o token existe somente na memória.** Isso quer dizer que se a página for atualizada ele é perdido. Precisamos armazená-lo em algum outro meio.

- Começamos adicionando uma opção para "Logout" na barra superior, definida no arquivo **cabecalho.component.html**. Veja a Listagem 2.7.1.

Listagem 2.7.1

```
<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>
    <li><button mat-button>Logout</button></li>
  </ul>
</mat-toolbar>
```

- A exibição das opções existentes na lista é dependente de haver um usuário logado, o que pode ser facilmente detectado pela existência de um token. Contudo, conforme o usuário interage com a aplicação, pode ser o caso de o token deixar de existir, passar a existir novamente etc. Sendo assim, os componentes desejam ser notificados pelo serviço toda vez que o token for alterado. Para isso, vamos adicionar um **Subject** (observável) ao serviço de manipulação de usuários (arquivo **usuario.service.ts**) que será responsável pelas notificações. Precisamos, também, de um método que permita aos componentes obter uma referência ao Subject para que possam se registrar como observadores. Veja a Listagem 2.7.2.

Listagem 2.7.2

```
…
export class UsuarioService {

  private token: string;
  private authStatusSubject = new Subject<boolean>();
  public getToken (): string{
    return this.token;
  }

  public getStatusSubject (){
    return this.authStatusSubject.asObservable();
  }
...
```

- Uma vez que o usuário tenha feito login, é verdade que ele está logado na aplicação. Portanto, o Subject precisa avisar os componentes sobre isso. Isso é feito pelo método next, que entregará **true** para os componentes, indicando que há alguém logado. Veja a Listagem 2.7.3. Ainda estamos no arquivo **usuario.service.ts**.

Listagem 2.7.3

```
login (email: string, senha: string){
   const authData: AuthData = {
     email: email,
     password: senha
   }
   this.httpClient.post<{token: string}>("http://localhost:3000/api/usuario/login",
authData).subscribe(resposta => {
     this.token = resposta.token;
     this.authStatusSubject.next(true);
   });
  }
```

- Podemos agora injetar uma instância do serviço de autenticação no componente que define a barra superior, no arquivo **cabecalho.component.ts**. Ele a utiliza para obter uma referência ao observável (Subject) e para se registrar como um observador (Subscription). Como feito anteriormente, ele guarda uma referência ao objeto Subscription para que, uma vez destruído, possa informar ao Observable que aquela instância já não deseja receber notificações. Ou seja, evitamos vazamento de recursos. Quando notificado, ele guarda o valor booleano recebido em uma variável pública. Ela é pública para que o seu template (arquivo .html) possa acessá-la. Veja a Listagem 2.7.4.

Listagem 2.7.4

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

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

  private authObserver: Subscription;
  public autenticado: boolean = false;
  constructor(
    private usuarioService: UsuarioService
  ) { }
```

```
  ngOnInit(): void {
    this.authObserver =
        this.usuarioService.getStatusSubject().
        subscribe((autenticado) => {
          this.autenticado = autenticado;
        })
  }

  ngOnDestroy(){
    this.authObserver.unsubscribe();
  }
}
```

- Utilizando a variável **autenticado**, o template (arquivo **cabecalho.component.html**) decide se as opções da lista devem aparecer. Veja a Listagem 2.7.5.

Listagem 2.7.5

```
<mat-toolbar color="primary">
  <span><a routerLink="/">Clientes</a></span>
  <span class="separador"></span>
  <ul>
    <li *ngIf="autenticado"><a mat-button routerLink="/criar" routerLinkActive="mat-
accent">Novo Cliente</a></li>
    <li *ngIf="!autenticado"><a mat-button routerLink="/login" routerLinkActive="mat-
accent">Login</a></li>
    <li *ngIf="!autenticado"><a mat-button routerLink="/signup" routerLinkActive="mat-
accent">Cadastrar</a></li>
    <li *ngIf="autenticado"><button mat-button>Logout</button></li>
  </ul>
</mat-toolbar>
```

- Faça novos testes.

- Também desejamos lidar com os botões EDITAR e REMOVER em função de a aplicação ter ou não um usuário logado. Para isso, vamos injetar uma instância do serviço de manipulação de usuários no componente ClienteListaComponent, definido no arquivo **cliente-lista.component.ts**. O processo é análogo. Fazemos a injeção no construtor, declaramos uma variável do tipo Subscription, realizamos o registro para obter notificações no método ngOnInit guardando o resultado na variável para que no método ngOnDestroy, possamos liberar o recurso. Quando o componente é notificado, ele guarda o resultado obtido em uma variável booleana pública, assim seu template (arquivo html) pode acessá-la. Veja a Listagem 2.7.6.

Listagem 2.7.6

```
...
import { Subscription } from 'rxjs';
import { UsuarioService } from 'src/app/auth/usuario.service';
...
@Component({
  selector: 'app-cliente-lista',
  templateUrl: './cliente-lista.component.html',
  styleUrls: ['./cliente-lista.component.css'],
})
export class ClienteListaComponent implements OnInit, OnDestroy {
...
public autenticado: boolean = false;
private authObserver: Subscription;
…
constructor (
   private clienteService: ClienteService,
   private usuarioService: UsuarioService
   ){

 }
…
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
      });
    this.authObserver = this.usuarioService
    .getStatusSubject().
    subscribe((autenticado) => this.autenticado = autenticado)
  }
…
ngOnDestroy(): void {
    this.clientesSubscription.unsubscribe();
   this.authObserver.unsubscribe();
  }
```

- A linha que contém os botões para edição e remoção de clientes terá a sua exibição condicionada de acordo o valor da variável **autenticado**. Veja a Listagem 2.7.7. Estamos no arquivo **cliente-lista.component.html**.

Listagem 2.7.7

```
<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">
      <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>
```

- **Note que há um problema.** Os botões editar e remover não aparecem mesmo quando um usuário está logado. Para entender isso, faça o seguinte teste:

- Atualize a página para começar sem um token.
- Expanda um cliente já existente e perceba que os botões não são exibidos. Isso ocorre pois o componente ClienteListaComponent teve seu método ngOnInit chamado antes de haver alguém logado, portanto o valor de sua varíavel autenticado é false.
- Faça login com um usuário válido.
- Cadastre um novo cliente.
- Clique no canto superior esquerdo, em Clientes, para visualizar a lista. Neste momento, o componente ClienteListaComponent é carregado novamente. Ele faz o registro de seu observador. Porém, como não há nenhuma notificação a ser enviada (ela só é enviada no momento em que um login acontece), a sua variável autenticada permanece com o valor false.

- Para resolver isso, vamos adicionar uma variável booleana no serviço de manipulação de usuários. Assim que ocorrer um login, ele armazena o valor true nesta variável. Além disso, adicionamos um getter que permite aos componentes consultar o valor da nova variável. Veja a Listagem 2.7.8. Estamos no arquivo **usuario.service.ts**. Note que temos um observador dos tipos **pull** e **push** simultaneamente.

Listagem 2.7.8

```
...
@Injectable({
  providedIn: 'root'
})
export class UsuarioService {
  private autenticado: boolean = false;
  private token: string;
...
public isAutenticado() : boolean{
    return this.autenticado;
}
...
login (email: string, senha: string){
    const authData: AuthData = {
      email: email,
      password: senha
    }
    this.httpClient.post<{token: string}>("http://localhost:3000/api/usuario/login", authData).subscribe(resposta =>
{
      this.token = resposta.token;
     if (this.token){
        this.autenticado = true;
        this.authStatusSubject.next(true);
      }
    });
  }
```

- Agora podemos chamar o método isAutenticado no componente ClienteListaComponent, definido no arquivo **cliente-lista.component.ts**, em seu método **ngOnInit**. Veja a Listagem 2.7.9.

Listagem 2.7.9

```
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
     });
   this.autenticado = this.usuarioService.isAutenticado();
   this.authObserver = this.usuarioService
   .getStatusSubject().
   subscribe((autenticado) => this.autenticado = autenticado)
 }
```

- Faça novos testes e verifique se os botões aparecem de maneira condizente com o status do usuário autenticado.

- O botão **Logout** precisa alterar o status das variáveis **autenticado** dos diversos componentes que se interessam por essa informação. Além disso, quando clicado, ele precisa eliminar o token. Para isso, começamos fazendo um *event binding* no botão, definido no arquivo **cabecalho.component.html**. Veja a Listagem 2.7.10

Listagem 2.7.10

```
<mat-toolbar color="primary">
  <span><a routerLink="/">Clientes</a></span>
  <span class="separador"></span>
  <ul>
    <li *ngIf="autenticado"><a mat-button routerLink="/criar" routerLinkActive="mat-
accent">Novo Cliente</a></li>
    <li *ngIf="!autenticado"><a mat-button routerLink="/login" routerLinkActive="mat-
accent">Login</a></li>
    <li *ngIf="!autenticado"><a mat-button routerLink="/signup" routerLinkActive="mat-
accent">Cadastrar</a></li>
    <li *ngIf="autenticado"><button (click)="onLogout()" mat-button>Logout</button></li>
  </ul>
</mat-toolbar>
```

- O método **onLogout** que vinculamos ao botão ainda precisa ser definido, o que pode ser feito no arquivo **cabecalho.component.ts**. Ele chama o método logout do serviço de manipulação de usuários. Esse também não existe ainda. Ele será criado a seguir.

Listagem 2.7.11

```
onLogout(){
   this.usuarioService.logout();
}
```

- No arquivo **usuario.service.ts**, definimos o método logout. Ele deve eliminar o token e avisar aos componentes que não há mais usuário logado, assim eles podem se atualizar visualmente. Veja a Listagem 2.7.12.

Listagem 2.7.12

```
logout(){
  this.token = null;
  this.authStatusSubject.next(false);
}
```

- Faça novos testes.

2.8 (Redirecionando o usuário) Após fazer operações como login e logout, o usuário provavelmente espera ser levado para uma página específica da aplicação, em geral, para a principal. Nessa aplicação, desejamos que a lista de clientes seja renderizada.

- O redirecionamento será feito pelo serviço de manipulação de usuários, pois é ele quem define os métodos de login e logout. Para isso, vamos injetar uma instância do roteador do Angular. A seguir, fazemos o redirecionamento nos métodos adequados. Veja a Listagem 2.8.1. Estamos no arquivo **usuario.service.ts**.

Listagem 2.8.1

```
import { Router } from '@angular/router';

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

  }
login (email: string, senha: string){
   const authData: AuthData = {
     email: email,
     password: senha
   }
   this.httpClient.post<{token: string}>("http://localhost:3000/api/usuario/login",
authData).subscribe(resposta => {
     this.token = resposta.token;
     if (this.token){
       this.autenticado = true;
       this.authStatusSubject.next(true);
       /*renderiza o componente associado à raiz da
       aplicação. Ou seja, a lista de clientes.
       Lembre-se que especificamos um vetor para
       eventualmente montar uma URL em função
       de diversas variáveis*/
       this.router.navigate(['/'])
     }
   });
  }
logout(){
   this.token = null;
   this.authStatusSubject.next(false);
   this.router.navigate(['/'])
  }
```

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.