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)

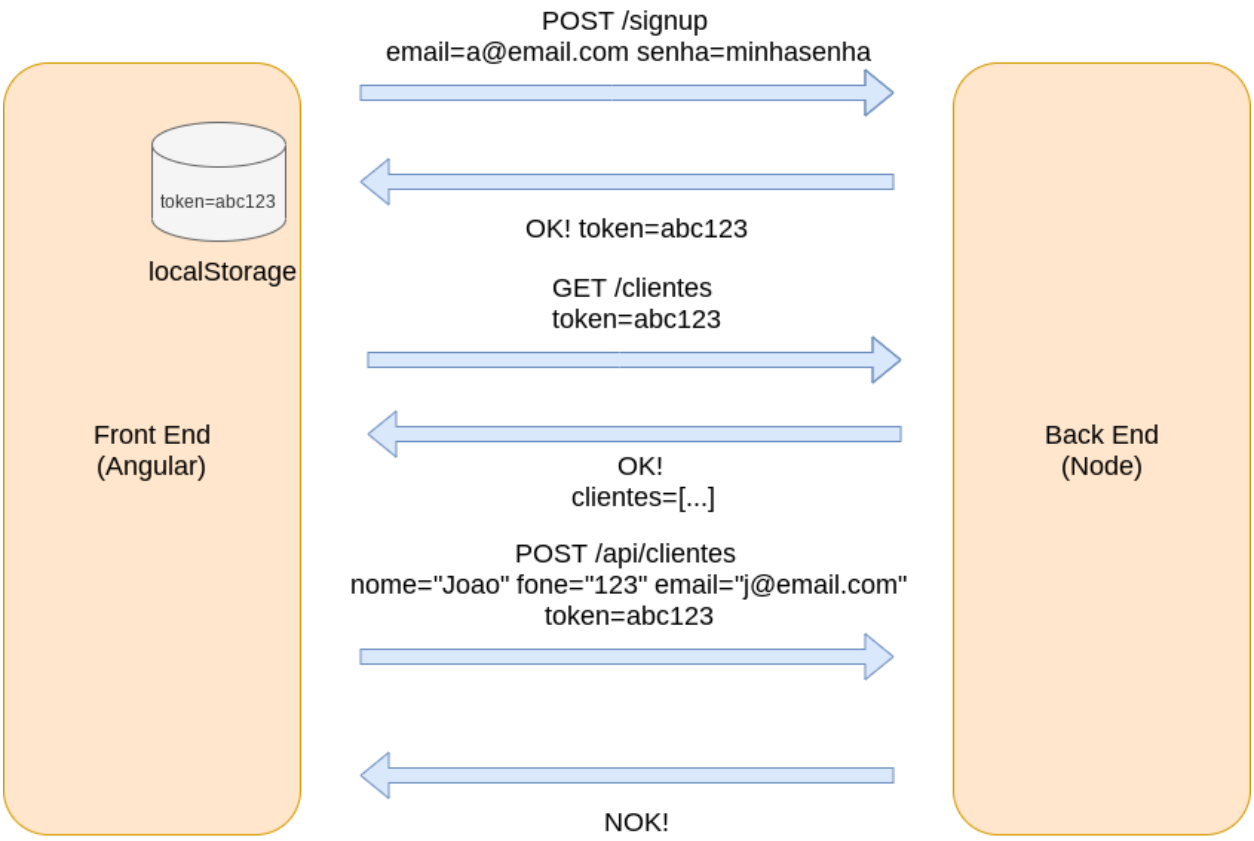
- Execute cada um deles em um terminal separado e mantenha ambos em execução.

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 (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 (Completando o logout) 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 2.3.1.

Listagem 2.3.1

```
<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 deve ser feito no arquivo **cabecalho.component.ts**, como mostra a Listagem 2.3.2. Ele chama o método logout do serviço de manipulação de usuários. Esse também não existe ainda. Ele será criado logo depois.

Listagem 2.3.2

```
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 2.3.3.

Listagem 2.3.3

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

- Faça novos testes.

2.4 (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 2.4.1. Estamos no arquivo **usuario.service.ts**.

Listagem 2.4.1

```
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(['/'])
  }
```

2.5 (Protegendo acesso direto a rotas que requerem login) Embora o usuário possa fazer login para ter acessso à funcionalidade de cadastro de novos clientes, ele não necessariamente passa

por essa fase para acessá-la. Nada impede que ele digite host:porta/criar diretamente em seu navegador, por exemplo. Se fizer isso, terá acesso à funcionalidade sem estar logado. Precisamos proteger as rotas que requerem login contra esse tipo de acesso. Isso pode ser feito utilizando-se um mecanismo do próprio roteador do Angular: **Route Guards**.

- Comece criando um novo arquivo chamado **auth.guard.ts** na pasta **auth** do seu projeto Angular.

- O Angular oferece **interfaces** que definem **métodos** que são **chamados automaticante** mediante determinados **eventos**. Para o evento "**uma rota foi acessada**", por exemplo, há um método que pode ser chamado automaticamente pelo Angular cuja finalidade é verificar se o acesso a essa rota está autorizado ou não. Para utilizar tais interfaces, iremos escrever uma classe chamada **AuthGuard** no arquivo **auth.guard.ts**, como na Listagem 2.5.1.

Listagem 2.5.1

```
export class AuthGuard{

}
```

- A interface **CanActivate** define um método chamado **canActivate** que tem como finalidade decidir se uma rota pode ser acessada ou não. A sua definição é dada na Listagem 2.5.2.

Listagem 2.5.2

```
import { ActivatedRouteSnapshot, CanActivate, RouterStateSnapshot, UrlTree } from "@angular/router";

import { Observable } from "rxjs";


export class AuthGuard implements CanActivate{

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean | UrlTree | Observable<boolean | UrlTree> | Promise<boolean | UrlTree> {


  }

}
```

- Observe que o **tipo** de **retorno** do método pode variar: ele pode devolver um **valor booleano** ou um objeto do tipo **UrlTree**, caso opere de maneira **síncrona**. Caso opere de maneira **assíncrona**, ele devolve um **Observable** ou uma **Promise**, os quais podem dar como resultado um valor booleano ou um objeto do tipo UrlTree.

- A ideia é simples: O método devolve **true** (ou um Observable/Promise que resulta em true) caso a rota possa ser acessada e **false** caso contrário.

- As rotas poderão ser acessadas caso o usuário esteja logado, portanto, precisamos obter essa informação de algum lugar. Ela está disponível no serviço de manipulação de usuários. Trata-se de uma dependência a ser injetada. Além disso, caso o usuário não esteja logado, desejamos fazer com que ele seja levado para a página de login, o que pode ser feito pelo roteador do Angular. Outra dependência a ser injetada. Veja a Listagem 2.5.3.

Listagem 2.5.3

```
import { Injectable } from "@angular/core";
import { ActivatedRouteSnapshot, CanActivate, Router, RouterStateSnapshot, UrlTree } from
"@angular/router";
import { Observable } from "rxjs";
import { UsuarioService } from './usuario.service';

@Injectable()
export class AuthGuard implements CanActivate{

  constructor(
    private usuarioService: UsuarioService,
    private router: Router
    ){

    }
  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean | UrlTree |
Observable<boolean | UrlTree> | Promise<boolean | UrlTree> {

    const isAutenticado = this.usuarioService.isAutenticado();
    if (!isAutenticado){
      this.router.navigate(['/login']);
    }
    return isAutenticado;
  }

}
```

- A existência da classe AuthGuard ainda não é de conhecimento do Angular. Precisamos registrá-la como um **provider** em um dos módulos da aplicação. Dada a sua natureza, pode ser de interesse fazer isso no módulo de roteamento (arquivo **app-routing.module.ts**). Veja a Listagem 2.5.4.

Listagem 2.5.4

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

import { ClienteInserirComponent } from './clientes/cliente-inserir/cliente-inserir.component';
import { ClienteListaComponent } from './clientes/cliente-lista/cliente-lista.component';
import { LoginComponent } from './auth/login/login.component';
import { SignupComponent } from './auth/signup/signup.component';

import { AuthGuard } from './auth/auth.guard'
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
  ],
  providers: [AuthGuard]

})
export class AppRoutingModule{

}
```

- Após registrar o componente, podemos utilizá-lo. Os objetos de mapeamento definidos no arquivo **app-routing.module.ts** possuem uma propriedade chamada **canActivate**. A ela associamos uma expressão cujo resultado será utilizado para determinar se a rota pode ser acessada. Veja a Listagem 2.5.5. Note que as rotas que desejamos “proteger” são aquelas que dão acesso às funcionalidade de adição e edição de clientes.

Listagem 2.5.5

```
const routes: Routes = [
  { path: '', component: ClienteListaComponent },
  {path: 'criar', component: ClienteInserirComponent, canActivate: [AuthGuard]},
  { path: 'editar/:idCliente', component: ClienteInserirComponent, canActivate: [AuthGuard]},
  {path: 'login', component: LoginComponent},
  {path: 'signup', component: SignupComponent}
];
```

- Faça o seguinte teste: no navegador, digite localhost:4200/criar sem estar logado na aplicação. Veja o resultado.

2.6 (Atualizações após a expiração do token) Lembre-se que, quando criamos o JWT, uma de suas características é o **tempo de validade**. Uma vez que esse tempo se esgote, o token já não é válido e a aplicação Angular já não deve ser capaz de acessar recursos que requerem login. Contudo, a expiração do token acontece **somente no back end** e a aplicação Angular não é notificada sobre isso.

- Para resolver esse problema, o primeiro passo será devolver o tempo de validade do token para a aplicação Angular logo após ele ter sido criado no Back end. Isso pode ser feito no arquivo **backend/rotas/usuarios.js**. Veja a Listagem 2.6.1.

Listagem 2.6.1

```
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
    })
  })
  .catch(err => {
    return res.status(401).json({
      mensagem: "Login falhou: " + err
    })
  })
})
```

- Na aplicação Angular, podemos obter o valor entregue pelo Back End no serviço de manipulação de usuários (**usuario.service.ts**). Iremos extrair o valor e configurar um “timer” que fará com que o usuário seja automaticamente “deslogado” quando o tempo vencer. Veja a Listagem 2.6.2. A função **setTimeout** é própria do Javascript. Ela coloca em execução a função que recebe como primeiro argumento após o tempo especificado como segundo argumento passar. Ela espera um valor em milissegundos e o Back End devolve um valor em segundos, o que quer dizer que precisamos fazer uma conversão simples.

Listagem 2.6.2

```
login (email: string, senha: string){
   const authData: AuthData = {
     email: email,
     password: senha
   }
   this.httpClient.post<{token: string, expiresIn:
number}>("http://localhost:3000/api/usuario/login", authData).subscribe(resposta => {
     this.token = resposta.token;
     if (this.token){
       const tempoValidadeToken = resposta.expiresIn;
       setTimeout(() => {
         this.logout()
       },tempoValidadeToken * 1000);
       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(['/'])
     }
   });
  }
```

- Também precisamos tomar o cuidado de parar a função de timeout caso o usuário faça logout manualmente. Para isso, vamos guardar uma referência ao timer em uma variável de instância, cuja declaração aparece na Listagem 2.6.3.

Listagem 2.6.3

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

- Note que **NodeJS.Timer** é um tipo provido pelo compilador Typescript. É possível que seja necessário adicionar o tipo **node** ao arquivo **tsconfig.app.json** como na Listagem 2.6.4. Ele fica na raiz da aplicação. Talvez você queira substituir o tipo **NodeJS.Timer** por **any** também.

Listagem 2.6.4

```
/* To learn more about this file see: https://angular.io/config/tsconfig. */
{
  "extends": "./tsconfig.base.json",
  "compilerOptions": {
    "outDir": "./out-tsc/app",
    "types": ["node"]
  },
  "files": ["src/main.ts", "src/polyfills.ts"],
  "include": ["src/**/*.d.ts"]
}
```

- Feita a declaração, podemos armazenar o valor devolvido pela função setTimeout e utilizá-lo para reconfigurar o timer caso o usuário faça logout. Veja a Listagem 2.6.5. Estamos no arquivo **usuario.service.ts**.

Listagem 2.6.5

```
login (email: string, senha: string){
    const authData: AuthData = {
      email: email,
      password: senha
    }
    this.httpClient.post<{token: string, expiresIn:
number}>("http://localhost:3000/api/usuario/login", authData).subscribe(resposta => {
      this.token = resposta.token;
      if (this.token){
        const tempoValidadeToken = resposta.expiresIn;
        this.tokenTimer = setTimeout(() => {
          this.logout()
        },tempoValidadeToken * 1000);
        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(['/'])
      }
    });
  }
```

- No método logout definido no arquivo **usuario.service.ts**, reconfiguramos o timer, como mostra a Listagem 2.6.6.

Listagem 2.6.6

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

 }
```

2.7 (Armazenamento em meio persistente do lado do cliente) O token e o tempo de expiração são valores que o Back End entrega para a aplicação cliente e ela tem a responsabilidade de armazená-los. No momento, ela os armazena em memória volátil. Isso quer dizer que se o usuário atualizar a tela, por exemplo, os valores serão perdidos. Precisamos de um mecanismo que nos permita armazenar dados em meio persistente do lado do cliente. Para isso, utilizaremos uma API disponível em Javascript que se chama **localStorage**.

- Para começar, vamos criar um método no no serviço de manipulação de usuários (arquivo **usuario.service.ts**) que tem como finalidade armazenar o token e o seu tempo de validade. Veja a Listagem 2.7.1. A API localStorage nos permite armazenar pares chave/valor em meio persistente (lembre-se: do lado do cliente, na máquina dele, em memória persistente a que o seu navegador tem acesso) usando o método **setItem**. O método **toISOString** converte o objeto Date para uma representação textual padronizada que permite que a string gerada possa ser convertida novamente para o tipo Date.

Listagem 2.7.1

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

- Quando um logout acontecer, desejamos remover essas informações da memória persistente. O método da Listagem 2.7.2, também definido no arquivo **usuario.service.ts**, é responsável por essa tarefa. Repare na definição do método e também na sua chamada, no método logout.

Listagem 2.7.2

```
private removerDadosDeAutenticacao (){
   localStorage.removeItem('token');
   localStorage.removeItem('validade');
}
logout(){
   this.token = null;
   this.authStatusSubject.next(false);
   clearTimeout(this.tokenTimer);
   this.removerDadosDeAutenticacao()
   this.router.navigate(['/'])
}
```

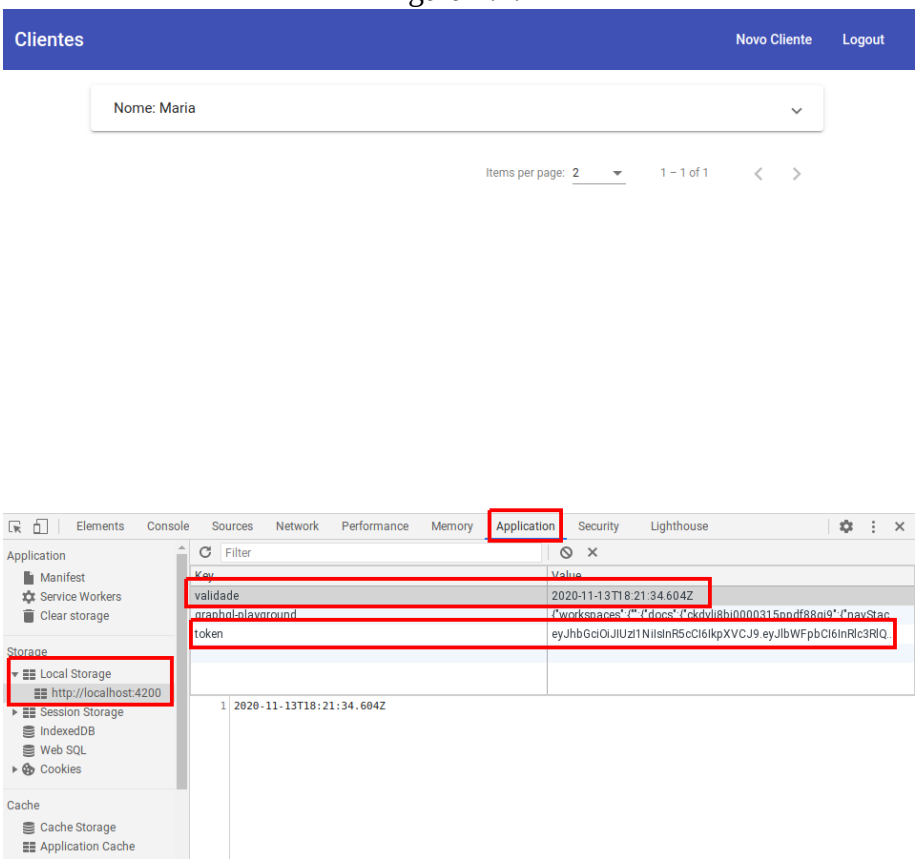
- O método **salvarDadosDeAutenticacao** também precisa ser chamada, o que é responsabilidade do método login, já que é ele quem tem acesso ao token e a seu tempo de validade assim que o usuário loga no sistema. Veja a Listagem 2.7.3.

Listagem 2.7.3

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

- Faça um login na aplicação e abra o Chrome Dev Tools (CTRL+SHIFT+I). Como mostra a Figura 2.7.1, verifique que o token e seu tempo de validade estão armazenados em localStorage.

Figura 2.7.1



Nota: O mecanismo **localStorage** é próprio para uso quando se deseja que os dados sejam acessíveis em diferentes abas e janelas do navegador. Eles são mantidos mesmo quando o navegador é fechado ou o sistema operacional reinicializado. Use **sessionStorage** caso deseje dados válidos somente para a aba atual. Eles são mantidos mesmo mediante uma atualização de página. Não sobrevivem, porém, caso a aba seja fechada.

Nota: O mecanismo que utilizamos é **mais seguro** do que o uso de **cookies**, pois é armazenado localmente somente e o servidor não o conhece. Além disso, cada “origin” (caracterizada por protocolo, host e porta) tem seu próprio objeto de **storage** e não tem como acessar os demais.

- Quando o usuário acessa a aplicação, ele pode fazer login, realizar algumas atividades e fechar o navegador. Pode ser que ele volte a utilizar a aplicação sem que o seu token tenha expirado. Neste caso, desejamos que a aplicação o considere automaticamente logado, sem que ele tenha que clicar novamente no botão login e inserir seus dados. Para isso, vamos escrever dois métodos, ambos no arquivo **usuario.service.ts**. Um deles obtém os dados armazenados em **localStorage**, caso existam. O outro faz o login automático. Veja a Listagem 2.7.4.

Listagem 2.7.4

```
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;
       //diferença ja esta em milissegundos, não multiplique!
       this.tokenTimer = setTimeout(() => {
          this.logout()
        }, diferenca);
       this.authStatusSubject.next(true);
     }
   }
 }
```

Nota: Perceba que o valor **diferença** foi entregue diretamente ao método setTimeout. Não fizemos uma multiplicação por 1000. Isso ocorre pois o valor já está em milissegundos. Fazendo a multiplicação, potencialmente obteríamos um valor grande que não pode ser representado por um inteiro de 32 bits, o que causaria uma exceção interna. Por padrão, mediante esta situação, a função setTimeout executa a função que lhe foi entregue após 1 milissegundo. Ou seja, o usuário seria praticamente imediadamente deslogado após a aplicação ter feito seu login automático.

- Sabemos que o componente principal, definido no arquivo **app.component.ts** executa antes de todos os demais. Ele pode ser um bom candidato a ter como responsabilidade a realização da autenticação automática. Para isso, precisamos injetar uma instância de UsuarioService. Implementamos o método **ngOnInit** da interface **OnInit** e chamamos o método **autenticarAutomaticamente** de UsuarioService ali. Veja a Listagem 2.7.4.

Listagem 2.7.4

```
import { Component, OnInit } from '@angular/core';
import { UsuarioService } from './auth/usuario.service';
import { Cliente } from './clientes/cliente.model';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent implements OnInit{
  constructor(private usuarioService: UsuarioService){

  }
  ngOnInit(){
    this.usuarioService.autenticarAutomaticamente();
  }
}
```

Nota: Se seu arquivo **app.component.ts** ainda tem definições de coleções ou métodos você pode removê-las: elas foram criadas antes do processo de refatoração feito em aulas passadas.

- A função **autenticarAutomaticamente** pode executar mais rapidamente do que a renderização do componente CabecalhoComponent. Se isso acontecer, o cabeçalho exibirá as opções de um usuário que não está logado mesmo que, de fato, exista um usuário logado. Isso acontece pois o CabecalhoComponent se registra como observador de UsuarioService e isso pode acontecer depois de a notificação ter sido enviada. Já resolvemos esse problema anteriormente. Além de fazer o registro como observador, o CabecalhoComponent precisa, também, consultar o estado de autenticação diretamente. Veja a Listagem 2.7.5. Estamos no arquivo **cabecalho.component.ts**.

Listagem 2.7.5

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

2.8 (Autorização: usuários só podem editar e remover clientes que criaram) Neste próximo passo, desejamos realizar o que chamamos de autorização. Uma vez autenticado, um usuário pode acessar somente os recursos para os quais tenha autorização. Nesta aplicação, somente pode editar e/ou remover um cliente o usuário que o tenha cadastrado a princípio.

- Para implementar a autorização descrita, começaremos ajustando o modelo no Back End. O modelo que descreve clientes irá especificar, também, o id do usuário criador. **ObjectId** é um tipo manipulado internamente pelo Mongoose. A propriedade **ref** indica a qual modelo se refere o id que estamos inserindo. Note a semelhança com o conceito de chave estrangeira do modelo relacional. Veja a Listagem 2.8.1. Estamos no arquivo **backend/models/cliente.js**.

Listagem 2.8.1

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

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

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

- Neste momento, pode ser uma boa ideia remover todos os clientes da sua base. Assim podemos garantir que todos terão um id de usuário associado.

- O endpoint de cadastro de clientes precisa ter acesso ao id do usuário que está fazendo a inserção para enviá-lo ao MongoDB. Embora a aplicação Angular não o esteja enviando explicitamente, podemos obtê-lo por meio do Token, já que ele foi codificado quando o Token foi gerado. Lembre-se que temos uma função que intercepta as requisições no Back End a fim de verificar se elas possuem justamente o Token. Ela está definida no arquivo **middleware/check-auth.js**. Ali, podemos obter o Token decodificado (o que a função **verify** já nos entrega) e "anexar" partes de interesse à requisição, antes de ela ser repassada para o endpoint destino. Veja a Listagem 2.8.2.

Listagem 2.8.2

```
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];
    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: "Autenticação falhou"
    })
  }

}
```

- Faça o ajuste temporário da Listagem 2.8.3 no método de criação de clientes (arquivo **backend/rotas/clientes.js**) para verificar se o objeto com os dados do usuário pode, de fato, ser acessado a partir da requisição.

Listagem 2.8.3

```
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}`

  })
  //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
        }
      })
    })
});
```

- A partir da aplicação Angular, faça a inserção de um novo cliente e observe o console que está executando o Node. Ele deveria exibir o e-mail e o id do usuário que está logado no momento. Você deveria ver algo parecido com o que a Figura 2.8.1 mostra. O erro de validação do campo “criador” é esperado, afinal, nosso modelo o especifica como obrigatório e ele ainda não foi utilizado para criar o objeto cliente. Isso será resolvido em breve.

Figura 2.8.1

```
(node:28158) DeprecationWarning: collection.ensureIndex is deprecated. Use createIndexe
s instead.
Conexão OK
entrou
file: [object Object]
{ email: 'teste@email.com', idUsuario: '5fa5538485414d260739ed59' }
(node:28158) UnhandledPromiseRejectionWarning: ValidationError: Cliente validation fail
ed: criador: Path `criador` is required.
    at model.Document.invalidate (/home/rodrigo/workspaces/2020_2/angular/pessoal-crud-
cliente/node_modules/mongoose/lib/document.js:2598:32)
    at /home/rodrigo/workspaces/2020_2/angular/pessoal-crud-cliente/node_modules/mongoo
se/lib/document.js:2418:17
    at /home/rodrigo/workspaces/2020_2/angular/pessoal-crud-cliente/node_modules/mongoo
```

- Agora apague o código que foi inserido temporiamente, somente como teste, exibido na Listagem 2.8.4.

Listagem 2.8.4

```
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}`
  })
  //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
        }
      })
    })
});
```

- O objeto cliente a ser inserido pode agora incluir o id do usuário responsável pela sua inserção. Basta adicionarmos a ele o campo criador. Veja a Listagem 2.8.5.

Listagem 2.8.5

```
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
        }
      })
    })
});
```

- Na aplicação Angular, quando obtemos a coleção de clientes, cada um deles contém o id de seu criador. Contudo após o mapeamento realizado (para traduzir _id para id, lembra?), o campo criador é ignorado. Podemos passar a considerá-lo como mostra a Listagem 2.8.6. Faça um console.log para verificar o resultado. Estamos no arquivo **cliente.service.ts**. O logo aparecerá no Chrome Dev Tools (CTRL+SHIFT+I).

Listagem 2.8.6

```
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,
           criador: cliente.criador
         }
       }),
       maxClientes: dados.maxClientes
     }
   }))
   .subscribe(
     (dados) => {
       console.log(dados.clientes);
       this.clientes = dados.clientes;
       this.listaClientesAtualizada.next({
         clientes: [...this.clientes],
         maxClientes: dados.maxClientes
       });
     }
   )
 }
```

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.