

**WORKSHOP ANGULAR 7**

**WORKSHOP ANGULAR 7**

No módulo 2, realizamos nosso primeiro, iniciamos o entendimento do PrimeNG com um exercício prático e implementamos uma camada de autorização para uma tela de login com o controle de exibição de menu.

Início Módulo 3

Neste módulo trabalharemos com camadas de serviços e autenticação com padrão JWT.

# HTTPCLIENT

Primeiramente, vamos entender o que é e como funciona o Gerenciador de Requisição HttpClient.

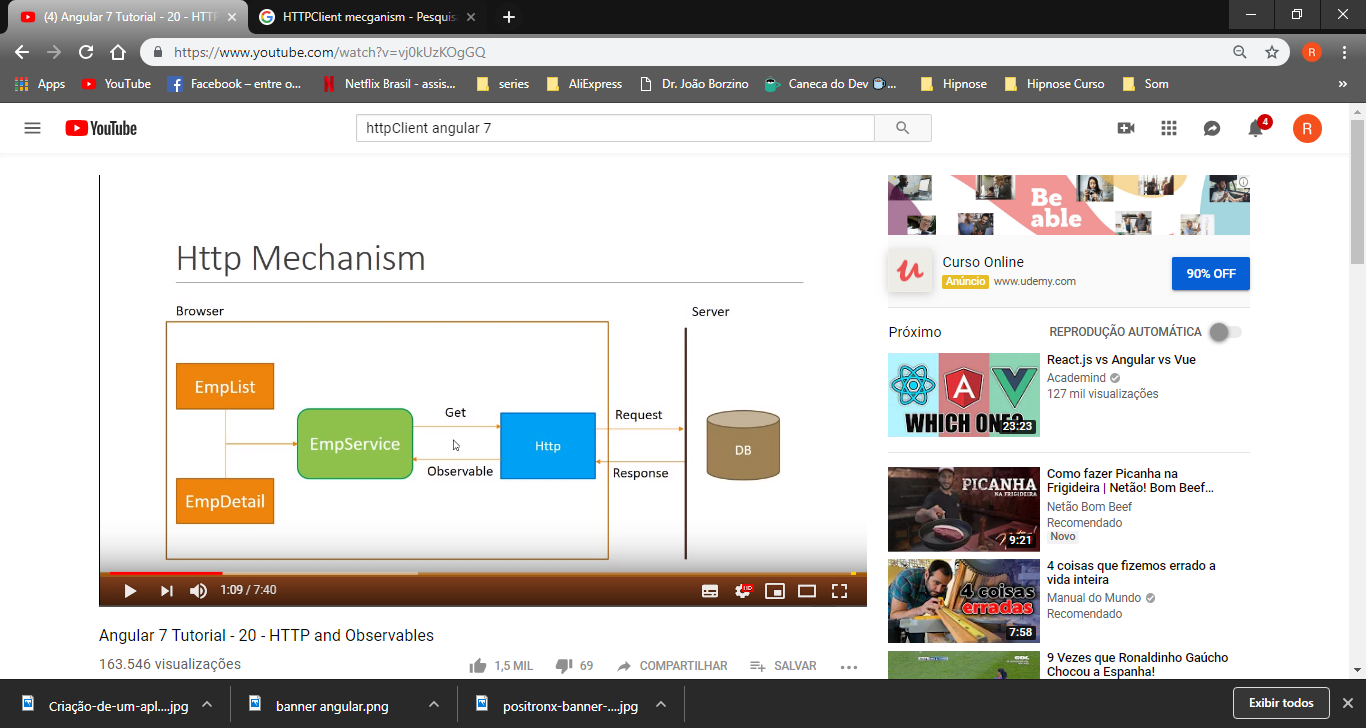
## Introdução ao novo módulo HTTP

As várias versões do módulo HTTP angular têm uma **API baseada em Observables RxJS** . Isso significa que as múltiplas chamadas para o módulo HTTP sempre retornarão um Observable, que precisamos assinar de uma forma ou de outra.

Aqui estão algumas coisas importantes para ter em mente em relação a este tipo específico de Observables retornados pelo módulo HTTP:

* se não assinarmos esses observáveis, nada acontecerá
* se assinarmos várias vezes esses observáveis, várias solicitações HTTP serão acionadas (veja este [post](http://blog.angular-university.io/angular-2-rxjs-common-pitfalls/) para mais detalhes)
* Esse tipo específico de Observables são fluxos de valor único: se a solicitação HTTP for bem-sucedida, esses observáveis ​​emitirão apenas um valor e, em seguida, serão concluídos
* esses observáveis ​​emitirão um erro se a solicitação HTTP falhar, mais sobre isso mais tarde.

## Exercício HttpClient – Criando uma chamada Http



Ng g s shared/seguranca/auth

# TRABALHANDO COM GUARDS

Guards basicamente são guardiões de acesso por rota onde seu papel é verificar se o acesso à rota que está sendo realizado deve ser permitido ou não.

Existem quatro tipos diferentes de Guard:

* **CanActivate**: válida se a rota está ativa
* **CanActivateChild**: válida se a rota filha está ativa
* **CanDeactivate**: verifica se uma rota pode ser desativada
* **CanLoad**: válida se um módulo está utilizando o lazily loading

Ng g g shared/seguranca/auth

## Implementando CanActivate

export class AuthGuard implements CanActivate {

constructor(private authService: AuthService, private router: Router) { }

canActivate(next: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<boolean> {

if (!this.authService.isLoggedIn()) {

this.authService.refreshToken().subscribe(resp => {

if (this.authService.isLoggedIn()) {

return Observable.create().of(true);

} else {

this.router.navigate(['login']);

return Observable.create().of(false);

}

});

this.authService.logout();

this.router.navigate(['login']);

return Observable.create().of(false);

}

}

}

## Incluindo Guards nas rotas privadas

const routes: Routes = [

{ path:'clientes', loadChildren:'./clientes/clientes.module#ClientesModule', canActivate: [AuthGuard]},

{ path: 'home', component: HomeComponent, canActivate: [AuthGuard] },

{ path: 'login', component: LoginComponent},

{ path: '', redirectTo: '/home', pathMatch: 'full' },

{ path: '\*\*', component: PaginaNaoEncontradaComponent },

];

# JWT

**JWT** ou **JSON Web Token** nada mais é que um objeto **JSON** definido na [RFC 7519](https://tools.ietf.org/html/rfc7519) para realizar transferência informação de permissões de acesso entre duas pontas. Ele é codificado e assinado e possuí o seguinte formato:

header.payload.signature

No **header** (cabeçalho) ficam os dados do **token**, que informam seu tipo e o algoritmo utilizado em sua assinatura:

{

"alg": "HS256",

"typ": "JWT"

}

No **payload** (carga) ficam os dados do usuário e alguns metadados como a expiração do **token**:

{

"sub": "1234567890",

"name": "John Do",

"iat": 1516239022

}

Finalmente na **signature** (assinatura) os dados de **header** e **payload** codificados em **base 64** e unidos por . (ponto) para serem assinados usando o algoritmo definido no **header**:

HMACSHA256(

base64UrlEncode(header) + "." +

base64UrlEncode(payload),

secret

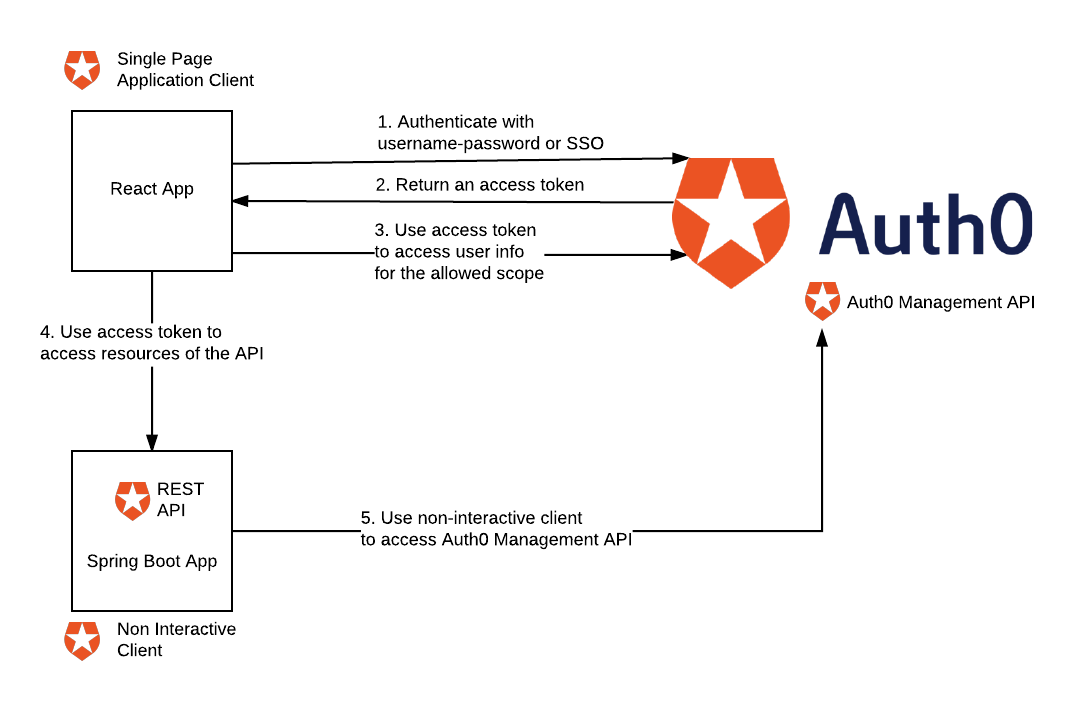
)

Com a assinatura é possível verificar se o **token** não foi alterado no caminho, garantindo sua integridade. Com ela também é possível confirmar a autenticidade de sua fonte.

Estes três blocos unidos por . (ponto) cada um codificado em **base 64** compõem o **JWT Token**:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.PcmVIPbcZl9j7qFzXRAeSyhtuBnHQNMuLHsaG5l804A

## Fluxo de Autenticação com JWT



Para auxiliar na implementação do **frontend** utilizaremos algumas bibliotecas:

npm install -s moment

npm install -s jwt-decode

npm install -s @types/jwt-decode

A biblioteca [moment](https://momentjs.com/" \t "_blank) facilitará o trabalho com tempo, já que precisamos controlar a expiração do **token**e sua renovação enquanto a biblioteca [jwt-decode](https://github.com/auth0/jwt-decode" \t "_blank) cuidará do **token** em si.

## Resumindo

A classe AuthService é a responsável por no autenticar de fato no sistema. Ela implemanta as funções de login, logout e refreshToken para fazer a manutenção da sessão no sistema, auxiliada das funções setSessionque salva a sessão em **local storage**, getExpiration que realiza o cálculo de expiração para a função refreshToken decidir se é hora de atualizar o token ou não, os pares isLoggedIn e isLoggedOut que são utilizados para verificar se o usuário está logado e o **getter** token que retorna o **JWT** para ser utilizado nas requisições autenticadas.

A classe AuthInterceptor implementa os interceptadores do Angular, que neste caso intercepta todas as requisições **http** realizadas e, caso o usuário esteja logado, injeta o cabeçalho Authorization JWT <token> na requisição para realizar chamadas autenticadas na **API**.

A classe AuthGuard serve como um escudo que impede o acesso de usuário não logado nas rotas em que ela for vinculada, se um usuário não logado tenta acessar determinada rota protegida por ela, o mesmo será redirecionado para a tela de login.

# SEGREGANDO NÍVEL DE ACESSO POR ROLES

Como podemos observar na autenticação com o uso do JWT, existem algumas propriedades no response da autenticação e um destes atributos, é o authorities.

Nele é possível obtermos todas as roles que o usuário logado possui e está roles, são as chaves que darão acesso as rotas, aos componentes, aos eventos, ao o que for descrito como nível de acesso.

Com estas informações e a aplicação bem estruturada, a segregação de acesso, menu e chaveamento ficará simples e muito fácil de fazer.

## Passo 1:

Implementar métodos para verificar se o usuário logado possui roles de acesso:

hasPermission(role: string) {

return this.payload && this.payload.authorities.includes(role);

}

hasAnyPermission(roles: string[]) {

for (const role of roles) {

if (this.hasPermission(role)) {

return true;

}

}

}

## Passo 2:

Implementar verificação de acesso no AuthGuard baseando-se nas roles também

import { AuthService } from "./auth.service";

import { Injectable } from '@angular/core';

import { ActivatedRouteSnapshot, RouterStateSnapshot, CanActivate, Router } from '@angular/router';

import { Observable } from 'rxjs';

import { of } from 'rxjs';

@Injectable({

providedIn: 'root'

})

export class AuthGuard implements CanActivate {

constructor(private authService: AuthService, private router: Router) { }

canActivate(next: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<boolean> {

let loggedIn: boolean;

this.authService.isLoggedin.subscribe(x => { loggedIn = x });

if (!loggedIn) {

this.authService.refreshToken().subscribe(resp => {

if (this.authService.isLoggedIn()) {

return of(next.data.roles && this.authService.hasAnyPermission(next.data.roles));

} else {

this.router.navigate(['login']);

return of(false);

}

});

this.authService.logout();

this.router.navigate(['login']);

return of(false);

} // if !loggedIn

return of(next.data.roles && this.authService.hasAnyPermission(next.data.roles));

}

}

Nos routings modules, os menus protegidos pelo AuthGuard, basta passar no atributo data as roles do módulo, contexto, funções conforme o exemplo abaixo:

const routes: Routes = [

{ path:'clientes', loadChildren:'./clientes/clientes.module#ClientesModule', canActivate: [AuthGuard],

data:{ roles: ['ROLE\_CLIENTE\_CADASTRAR', 'ROLE\_CLIENTE\_EDITAR', 'ROLE\_CLIENTE\_CONSULTAR', 'ROLE\_CLIENTE\_EXCLUIR'] }},

{ path: 'home', component: HomeComponent, canActivate: [AuthGuard]},

{ path: 'login', component: LoginComponent},

{ path: '', redirectTo: '/home', pathMatch: 'full' },

{ path: '\*\*', component: PaginaNaoEncontradaComponent },

];

Para manipular o acesso de menus com a roles basta utilizar o métodos do auth service de verificação de permissão passando as roles desejadas como parâmetros para cada item da lista de menus:

this.items = [

{ label: 'Home', icon: 'fa fa-home', routerLink:['/'] },

{ label: 'Clientes', icon: 'fa fa-users', routerLink:['/clientes'],

visible: this.authService.hasAnyPermission(['ROLE\_CADASTRAR', 'ROLE\_CONSULTAR']),

}

];

## Exemplo de um Token de Acesso e seu Payload

### ACCESS\_TOKEN

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c3IiOiJyYW1vcmltIiwibmFtZSI6IlJPTkVZIEFVR1VTVE8gQU1PUklNI

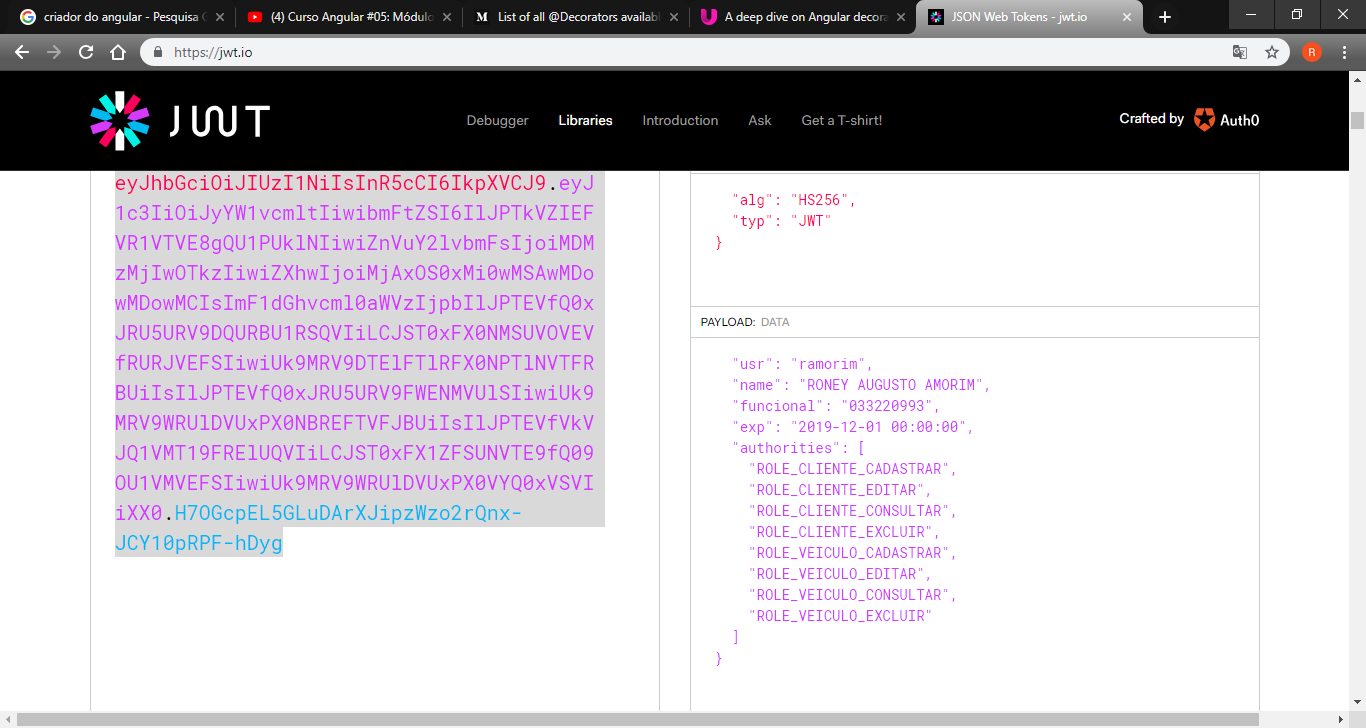
iwiZnVuY2lvbmFsIjoiMDMzMjIwOTkzIiwiZXhwIjoiMjAxOS0xMi0wMSAwMDowMDowMCIsImF1dGhvcml0aWVzIjpbIlJPTEV

fQ0xJRU5URV9DQURBU1RSQVIiLCJST0xFX0NMSUVOVEVfRURJVEFSIiwiUk9MRV9DTElFTlRFX0NPTlNVTFRBUiIsIlJPTEVfQ

0xJRU5URV9FWENMVUlSIiwiUk9MRV9WRUlDVUxPX0NBREFTVFJBUiIsIlJPTEVfVkVJQ1VMT19FRElUQVIiLCJST0xFX1ZFSUN

VTE9fQ09OU1VMVEFSIiwiUk9MRV9WRUlDVUxPX0VYQ0xVSVIiXX0.H7OGcpEL5GLuDArXJipzWzo2rQnx-JCY10pRPF-hDyg

### PAYLOAD



# PRATIACANDO - EXERCÍCIO FINAL