

# Projeto de Interfaces WEB

## Refatoração do Projeto CRUD Aula 14

# Introdução

- O objetivo dessa aula é refatorar o projeto Angular CRUD no que diz respeito aos seguintes aspectos:
  - Criação de Módulos
  - Lazy-loading de componentes (novo roteamento)
  - Validação de formulário via HTML
  - Uso de sessionStorage para login
  - Uso de Guardas para prevenir acesso a páginas não logadas
  - Segurança no Login (criptografia e tokens)



# Módulos e Lazy Loading

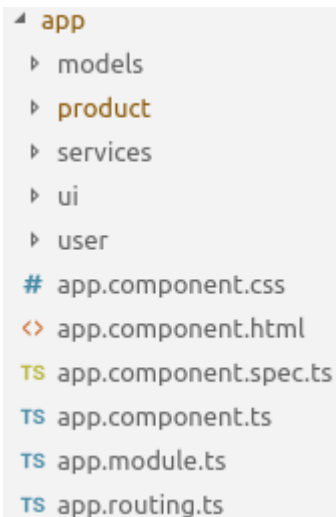
(<https://malcoded.com/posts/angular-fundamentals-modules/>)

# O que são módulos

- De forma simples, módulos são **classes**, assim como **componentes(components)** e **serviços(services)**.
- Você pode pensar em módulos como pacotes que organizam arquivos-fonte para um uso específico (por exemplo, um módulo para tratar apenas de componentes da interface gráfica).
- O módulo mais importante (módulo raiz) é o **App-Module**, o qual chama (diretamente ou indiretamente), todos os outros módulos da aplicação.
- O operador **@NgModule** define todas as propriedades do módulo (bootstrap, exports, declarations, imports, providers).

# Criação de módulos

- Módulos servem não **apenas** para deixar nosso projeto mais organizado, mas também irão auxiliar no lazy-loading de componentes.



```
└─ app
  ├── models
  ├── product
  ├── services
  ├── ui
  ├── user
  ├── app.component.css
  ├── app.component.html
  ├── app.component.spec.ts
  ├── app.component.ts
  ├── app.module.ts
  └── app.routing.ts
```

- Para o nosso projeto, iremos criar os **módulos** “product”, “ui”, “user”.
  - **product** – componentes (CRUD) relacionados com a entidade “product”.
  - **user** – componentes (CRUD) relacionados com a entidade “user”.
  - **ui** – componentes relacionados à interface gráfica com o usuário
- Para criar um novo módulo:
  - **ng g m <nome do módulo>**

# Criação de módulos

- O próximo passo é criar os componentes dentro dos módulos.

```
└─ product
  ├── edit-product
  ├── list-product
  ├── register-product
  ├── product.module.ts
  ├── product.routing.ts
  └── services
└─ ui
  ├── footer
  ├── menu
  ├── ui.module.ts
└─ user
  ├── edit-user
  ├── list-user
  ├── login-user
  ├── register-user
  ├── user.module.ts
  └── user.routing.ts
```

- Para o módulo **product**:

- **edit-product**;
- **list-product**;
- **register-product**;

- Para o módulo **ui**:

- **footer**
- **menu**

- Para o módulo **user**:

- **edit-user**;
- **list-user**;
- **login-user**;
- **register-user**

Para criar um componente dentro de um módulo:

**-ng g c <nome do módulo>/<nome do componente>**

# Lazy-loading de componentes

- Ao usar módulos, podemos configurar com que nossa aplicação carregue apenas os componentes de um determinado módulo por vez.
- Essa é uma característica útil caso estejamos trabalhando com dispositivos de clientes com pouca memória.
- Na verdade, sistema reais podem crescer bastante em quantidade de módulos, sendo inviável carregá-los todos ao mesmo tempo.
- O lazy-loading (carregamento preguiçoso) é possível graças ao uso de módulo em conjunto com arquivos de roteamento (routing).
- Em Angular **não-existe** uma hierarquia de módulos. Todos os módulos são “compilados” em um. No entanto, ao usar **lazy-loading**, módulos são injetáveis de forma hierárquica.

# Lazy-loading de componentes

- Os módulos **user** e **product** devem ter os arquivos:
  - `user.routing.ts`
  - `product.routing.ts`
- São nesses arquivos que iremos definir as rotas para os componentes (antes fazíamos isso, ingenuamente, no `app.module.ts`).
- Iremos, depois, referenciar esses arquivos no `.module` de cada módulo:
  - `user.module.ts`
  - `product.module.ts`



# Lazy-loading de componentes

- user.routing.ts (rotas de user)

```
import { GuardService } from '../services/guard.service';
import { EditUserComponent } from '../edit-user/edit-user.component';
import { RegisterUserComponent } from '../register-user/register-user.component';
import { ListUserComponent } from '../list-user/list-user.component';
import { LoginUserComponent } from '../login-user/login-user.component';
import { Routes, RouterModule } from '@angular/router';
import { ModuleWithProviders } from '@angular/core';
```

```
const routes: Routes = [
  {path: '', component: LoginUserComponent},
  {path: 'login', component: LoginUserComponent},
  {path: 'list', component: ListUserComponent, canActivate: [GuardService]},
  {path: 'register', component: RegisterUserComponent},
  {path: 'edit/:id', component: EditUserComponent, canActivate: [GuardService]},
];
```

```
export const routing: ModuleWithProviders = RouterModule.forChild(routes);
```

Rotas que antes fazíamos em app.module.ts.

Note que também criamos o **GuardService**, classe que previne o acesso aos componentes ListUser e EditUser. Vamos explicar o **GuardService** mais adiante.

Não cria o serviço raiz de rotas mas cria todas as rotas-filho deste módulo.

# Lazy-loading de componentes

- product.routing.ts (rotas de product)

```
import { GuardService } from '../services/guard.service';
import { EditProductComponent } from './edit-product/edit-product.component';
import { RegisterProductComponent } from './register-product/register-product.component';
import { ListProductComponent } from './list-product/list-product.component';
import { Routes, RouterModule } from "@angular/router";
import { ModuleWithProviders } from '@angular/core';

const routes:Routes = [
  {path:'list',component:ListProductComponent, canActivate: [GuardService]},
  {path:'register',component:RegisterProductComponent, canActivate: [GuardService]},
  {path:'edit/:id',component>EditProductComponent, canActivate: [GuardService]},
];

export const routing: ModuleWithProviders = RouterModule.forChild(routes);
```

Rotas que antes fazíamos em app.module.ts.

Mesma ideia aqui. No entanto, o **GuardService** protege todos os componentes de **product**.

# Lazy-loading de componentes

- user.module.ts

```
import { NgModule } from '@angular/core';  
import { FormsModule } from '@angular/forms';  
import { CommonModule } from '@angular/common';
```

```
import { RegisterUserComponent } from '../user/register-user/register-user.component';  
import { ListUserComponent } from '../user/list-user/list-user.component';  
import { EditUserComponent } from '../user/edit-user/edit-user.component';  
import { LoginUserComponent } from '../user/login-user/login-user.component';  
import { routing } from './user.routing';
```

Import do arquivo de rotas, dos slides anteriores (user.routing.ts).

```
@NgModule({  
  declarations: [RegisterUserComponent, ListUserComponent, EditUserComponent, LoginUserComponent],  
  imports: [  
    CommonModule,  
    FormsModule,  
    routing  
  ]  
})  
export class UserModule { }
```

Importando o **routing** no Module.

# Lazy-loading de componentes

- product.module.ts

```
import { NgModule } from '@angular/core';  
import { FormsModule } from '@angular/forms';  
import { CommonModule } from '@angular/common';
```

```
import { RegisterProductComponent } from './register-product/register-product.component';  
import { EditProductComponent } from './edit-product/edit-product.component';  
import { ListProductComponent } from './list-product/list-product.component';  
import { routing } from './product.routing';
```

Import do arquivo de rotas, dos slides anteriores (product.routing.ts).

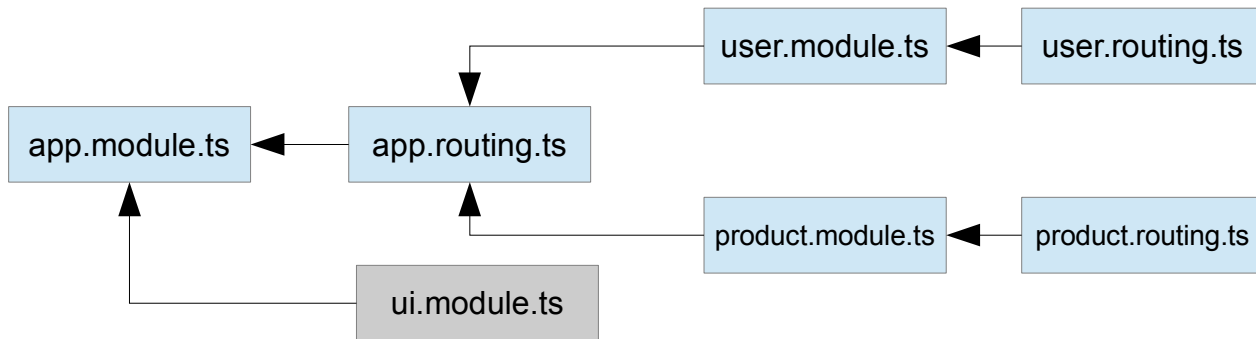
```
@NgModule({  
  declarations: [RegisterProductComponent, EditProductComponent, ListProductComponent],  
  imports: [  
    CommonModule,  
    FormsModule,  
    routing  
  ]  
})  
export class ProductModule { }
```

Importando o **routing** no Module.

# Lazy-loading de componentes

```
└─ app
  ├── models
  ├── product
  ├── services
  ├── ui
  └── user
# app.component.css
<> app.component.html
TS app.component.spec.ts
TS app.component.ts
TS app.module.ts
TS app.routing.ts
```

- Devemos ainda criar, no diretório app, o arquivo de roteamento raiz, o **app.routing.ts**
- **app.routing.ts** irá importar os módulos-filho:
  - o arquivo **user.module.ts**, o qual importa o arquivo **user.routing.ts**
  - o arquivo **product.module.ts**, o qual importa o arquivo **product.routing.ts**
- Por fim, **app.routing.ts**, será importado por **app.module.ts**, o módulo raiz.



# Lazy-loading de componentes

- app.routing.ts

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

```
const routes:Routes = [  
  //caso não digite nada, leve para user  
  { path: '', pathMatch: 'full', redirectTo: 'user' },  
  //importando as rotas-filho  
  { path:'user',loadChildren:'./user/user.module#UserModule'},  
  { path:'product',loadChildren:'./product/product.module#ProductModule'},  
  //path errado, leva para user. DEIXAR ESSE PATH SEMPRE POR ÚLTIMO.  
  { path: '**', redirectTo: 'user' }  
];
```

```
export const routing:ModuleWithProviders = RouterModule.forRoot(routes);
```

Importando os módulos-filhos, em lazy-loading.

Cria o serviço de rota raiz e carrega todas as rotas filho.

Exportando as rotas.

# Lazy-loading de componentes

- app.module.ts

```
import { NgModule } from '@angular/core';  
import { AppComponent } from './app.component';  
import { HttpClientModule } from '@angular/common/http';  
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';  
import { ToastrModule } from 'ngx-toastr';
```

```
import { routing } from './app.routing';  
import { UiModule } from './ui/ui.module';
```



Variável routing mestre.

# Lazy-loading de componentes

- app.module.ts (cont.)

```
@NgModule({  
  declarations: [  
    AppComponent  
  ],  
  imports: [  
    BrowserModule,  
    ToastrModule.forRoot(),  
    HttpClientModule,
```

→ Não há necessidade de importar mais o BrowserModule pois já estamos importando o BrowserModule.

```
    //created by this application  
    UiModule,  
    routing  
  ],  
  providers: [],  
  bootstrap: [AppComponent]  
})  
export class AppModule { }
```

→ importante a variável routing, criada acima.





# **Validação do Formulário**

# Validação Formulário

- Validar o HTML previne que sejam enviados dados inconsistentes ao Back-End.
- Cada input de entrada tem as suas regras, e ao submeter ao Angular, o mesmo irá checar se todo o formulário está correto (segundo as regras de cada input) antes de enviar.
- A mensagens de erro são mostradas por input fazendo uso de CSS.

# Validação Formulário

```
<form (ngSubmit)="onSubmit(registerForm)" #registerForm="ngForm">
```

```
  <div class="form-group">
```

```
    <label for="login">Login</label>
```

```
    <input type="text" class="form-control" name="login" id="login"  
      [(ngModel)]="user.login"
```

```
    #login="ngModel" required
```

```
    [ngClass]="{ 'is-invalid': registerForm.submitted && login.invalid }">
```

```
  <div *ngIf="login.invalid && registerForm.submitted" class="text-danger mt-1">
```

```
    <div *ngIf="login.errors['required']">Login is required.</div>
```

```
  </div>
```

```
</div>
```

...

Note que o formulário é representado por uma variável (`#registerForm="ngForm"`) e o método passa o formulário `onSubmit(registerForm)` como parâmetro para o componente.

Exemplo básico. Vamos quebrar nos slides seguintes para entender melhor!

# Validação Formulário

```
<form (ngSubmit)="onSubmit(registerForm)" #registerForm="ngForm">
```

```
  <div class="form-group">
```

```
    <label for="login">Login</label>
```

```
    <input type="text" class="form-control" name="login" id="login"  
      [(ngModel)]="user.login"
```

➡ Objeto `user` do componente.

```
    #login="ngModel" required
```

```
    [ngClass]="{ 'is-invalid': registerForm.submitted && login.invalid }">
```

```
  <div *ngIf="login.invalid && registerForm.submitted" class="text-danger mt-1">
```

```
    <div *ngIf="login.errors['required']">Login is required.</div>
```

```
  </div>
```

```
</div>
```

...

Até o momento, o input está sem nenhuma validação. Ele apenas faz um 2-way data binding (`[(ngModel)]="user.login"`) para que seus dados reflitam no componente e vice-versa

# Validação Formulário

```
<form (ngSubmit)="onSubmit(registerForm)" #registerForm="ngForm">
```

```
...<!-- SUPRIMIDO!-->
```

```
  <div *ngIf="login.invalid && registerForm.submitted" class="text-danger mt-1">
```

```
    <div *ngIf="login.errors['required']">Login is required.</div>
```

```
  </div>
```

```
</div>
```

```
...
```

A primeira div irá testar a se o login é inválido e se o formulário já foi submetido ao menos 1 vez (escolha do desenvolvedor) Sendo `*ngIf` um irá aparecer apenas se for `login.invalid && registerForm.submitted` verdade.

A div mais interna depende da div externa anterior. Ela irá especificar qual erro realmente aconteceu. Como temos apenas um tipo de erro para o login, então só iremos testar ele (`*ngIf="login.errors['required']"`). Qual seria um outro tipo de regra interessante para o login, a nível de interface?

# Validação Formulário

- O componente register-user.component.ts
  - Ao clicar em enviar, o método onSubmit deste componente será chamado, recebendo como parâmetro o formulário inteiro:

```
onSubmit(registerForm: NgForm){  
  if(registerForm.invalid){  
    this.toast.error("All fields are required.");  
    return;  
  }  
  
  this.userService.register(this.user).subscribe(  
  ...
```

Caso algum das regras do formulário esteja com problema (registerForm.invalid), ele não será submetido (return).

# Validação Formulário

- A mesma ideia serve para outros inputs:

```
<div class="form-group">
  <label for="email">E-mail</label>
  <input type="text" class="form-control" name="email" id="email"
    [(ngModel)]="user.email"
    #email="ngModel" required email
    [ngClass]="{ 'is-invalid': registerForm.submitted && email.invalid }">
```

Esse é o parâmetro HTML que dita as regras de um e-mail bem formado. Não confunda com o nome da variável `#email`.

```
<div *ngIf="email.invalid && registerForm.submitted" class="text-danger mt-1" >
  <div *ngIf="email.errors['required']">E-mail is required.</div>
  <div *ngIf="email.errors['email']">Must be a valid e-mail address.</div>
</div>
</div>
```

Nesse caso, temos dois tipos de erro: o e-mail é obrigatório (`required`) e tem um formato (`xxx@yyy.com`). O formato é especificado pelo parâmetro do próprio HTML (`email`), não sendo necessário criar nenhum pattern.

# Validação Formulário

```
<form (ngSubmit)="onSubmit(registerForm)" #registerForm="ngForm">
```

```
<div class="form-group">
```

```
<label for="login">Login</label>
```

```
<input type="text" class="form-control" name="login" id="login"  
[(ngModel)]="user.login"
```

```
#login="ngModel" required
```

```
[ngClass]="{ 'is-invalid': registerForm.submitted && login.invalid }">
```

```
<div *ngIf="login.invalid && registerForm.submitted" class="text-danger mt-1">
```

```
<div *ngIf="login.errors['required']">Login is required.</div>
```

```
</div>
```

```
</div>
```

...

O próximo passo é criar uma variável de referência (`#login="ngModel"`) a qual será usada para testar se o input está inválido. No caso, a única regra é que ele é obrigatório (`required`).

Caso o formulário tenha sido submetido **E** o login seja inválido (`registerForm.submitted && login.invalid`), o `ngClass` irá disparar o CSS `is-invalid` o qual modificará a aparência do input!



# Validação Formulário

- um último caso:

```
<div class="form-group">
  <label for="password">Password</label>
  <input type="password" class="form-control" name="password" id="password"
    [(ngModel)]="user.password"
    #password="ngModel" required minlength="6"
    pattern="^(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])[a-zA-Z0-9]+$"
    [ngClass]="{ 'is-invalid': registerForm.submitted && password.invalid }">

  <div *ngIf="password.invalid && registerForm.submitted" class="text-danger mt-1" >
    <div *ngIf="password.errors['required']">Password is required.</div>
    <div *ngIf="password.errors['minlength']">Password must be at least 6 characters.</div>
    <div *ngIf="password.errors['pattern']">Must be a valid password.</div>
  </div>

</div>
```

Quais são as regras desse caso? O que tornaria um password válido?

# Login, Validação no Servidor, Criptografia e Token (LVSCT)



# LVSCT - Angular

- O login-user.ts envia o login e senha digitados no input para um serviço específico:

```
onSubmit(registerForm: NgForm){  
  if(registerForm.invalid){  
    this.toasty.error("All fields are required.");  
    return;  
  }  
  
  this.authUser.login(this.user.login, this.user.password);  
}
```

# LVSCT - Angular

- AuthUserService: por enquanto, vamos nos concentrar apenas no método login:

```
login(login:string, password:string){  
  this.userService.login(login,password)  
    .subscribe(  
    (res:User)=>{  
      if(res!=null){  
        sessionStorage.setItem("user_login",JSON.stringify(res));  
        localStorage.setItem("access_token",res.token);  
        this.userBehaviorSubject.next(res);  
        this.router.navigate(["user/list"]);  
      }else{  
        this.toasty.error("Invalid user or/and password!")  
      }  
    })  
  );  
}
```

chama o endpoint do back-end, passado um objeto json via post.

Caso o login tenha obtido sucesso, um res!=null é enviado ao cliente.

Salva o usuário logado em sessão e o token vindo do servidor em um cookie.

Veremos adiante!

Redireciona pra página de listagem.

# LVSCT - Angular

- userService (comunicação com o back-end)

```
login(login:string,password:string){  
  let inputLogin = {"login":login,"password":password}  
  return this.httpClient.post<User>(`${this.loginUrl}`,inputLogin); //mongo  
}  
  
register(user:User):Observable<User>{  
  //return this.httpClient.post<User>(this.url,user); // json-server  
  return this.httpClient.post<User>(`${this.loginUrl}/register`,user); //express  
}
```

Dois serviços que não necessitam de credenciais com o servidor.

# LVSCT - Express

```
var loginService = require('../services/login.service.mongo');  
var express = require('express');  
var router = express.Router();
```

```
router.post('/', function (req, res, next) {  
  loginService.login(req, res);  
});
```

recebe a requisição com  
login e senha.

```
router.post('/register', function (req, res, next) {  
  loginService.register(req, res);  
});
```

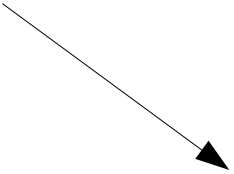
recebe a requisição um  
usuário inteiro para salvar  
no banco.

```
module.exports = router;
```

# LVSCT - Express

- login.service.mongo.js

```
static register(req,res){  
  let rcvUser = req.body;  
  rcvUser.password = bcrypt.hashSync(rcvUser.password, 10)  
  UserModel.create(rcvUser).then(  
    (user)=>{  
      res.status(201).json(user);  
    }  
  ).catch(  
    (error)=>{  
      res.status(500).json(error);  
    }  
  );  
}
```



O método hashSync criptografa a senha do usuário ANTES de guardá-la no banco.

# LVSCT - Express

```
static login(req,res){  
  let loginForm = req.body;  
  UserModel.findOne({'login':loginForm.login})  
    .then(  
      (user)=>{  
        if(bcrypt.compareSync(loginForm.password,user.password)){  
          //LOGIN ENCONTRADO E PASS BATEM (CRIAR TOKEN)  
          let token = jwt.sign({user: user}, 'secret');  
          res.status(201).json({  
            'firstName':user.firstName,  
            'lastName': user.lastName,  
            'login':user.login,  
            'token':token  
          });  
        }else{  
          //LOGIN ENCONTRADO MAS PASS NÃO BATE  
          res.status(201).json(null);  
        }  
      }  
    )  
  )  
}
```

Procura o usuário com login enviado pelo formulário.

Compara o password passado pelo form com o password criptografado do banco.

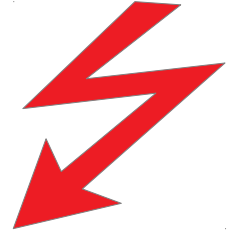
Cria um token a ser enviado ao cliente. Apenas requisições com o token poderão acessar certos endpoints.



# LVSCT - Express

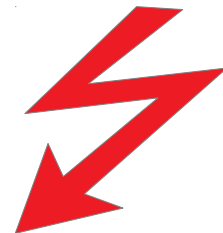
```
.then(undefined,  
  (err)=>{  
    //ERRO EM CASO DE LOGIN NÃO ENCONTRADO  
    res.status(201).json(null);  
  })  
  .catch(  
    (error)=>{  
      res.status(500).json(error);  
    }  
  );  
}
```

# LVSCT – Express - Apêndice



- Uma pequena pausa na aplicação para explicar criptografia e tokens no Node.
- Os textos a seguir foram cedidos pelo professor **Vitor Farias**.

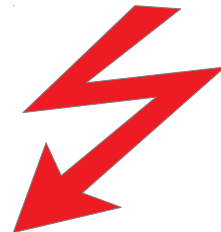
# LVSCT – Express - Apêndice



## **Autenticação via Tokens**

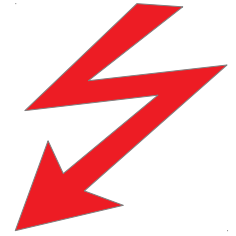
- Token serve para identificar uma aplicação
- Ao fazer o login, o servidor retorna um token para o cliente
  - ○ Esse token contém um identificador da sessão, data de
- validade do token, id do usuário ...
- Sempre que formos acessar algum recurso no servidor, temos que passar também o token para mostrar que estamos logados
  - A partir do token, o servidor consegue saber qual é o usuário logado

# LVSCT – Express - Apêndice

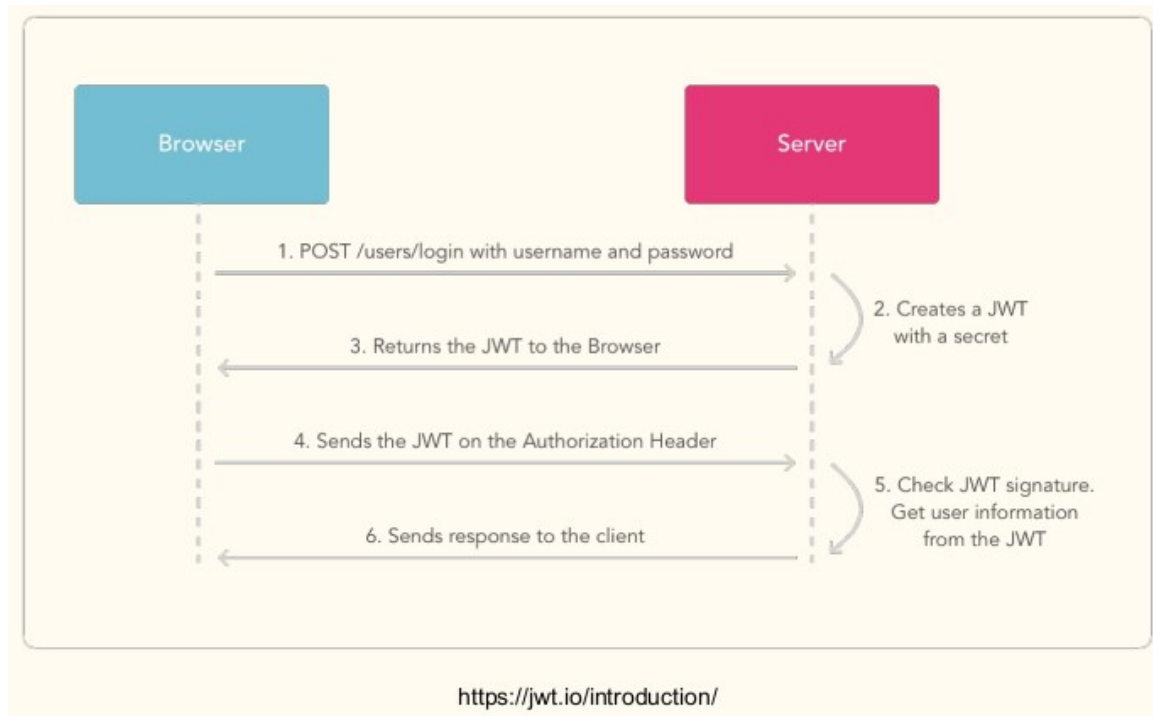


- Json Web Token
- Json Web Token (JWT)
  - Padrão (RFC 7165)
  - Criação e transmissão segura de objetos JSON via token
- Um JWT é dividido em 3 partes
  - Header - informações como algoritmo de criptografia
  - Payload
  - Signature - informações para validar token
- No payload, é possível armazenar qualquer objeto, inclusive dados do usuário
- **npm install --save jsonwebtoken**

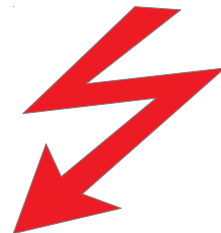
# LVSCT – Express - Apêndice



- JWT - Fluxo



# LVSCT – Express - Apêndice



- **Criar token**

- Função `jwt.sign(payload, secretOrPrivateKey)`
- **payload** é os dados que vão ser embutidos no token (no nosso caso, o user)
- **secretOrPrivateKey** é a chave/senha privada que só o servidor pode conhecer (uma string sua)
- Retorna token

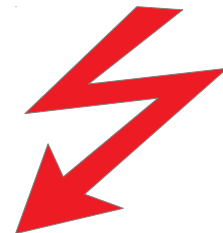
- **Validar token**

- Função `jwt.verify(token, secretOrPublicKey)`
- **token** a ser validado
- **secretOrPublicKey** é a chave que foi usada para criar o token (uma string sua)
- Retorna true se token é válido ou false, caso contrário (na verdade, retorna o token decodificado e validado)

- **Decodificar token**

- Função `jwt.decode(token)`
- Recebe token a ser decodificado
- Retorna objeto representando payload
- obs: não valida token!

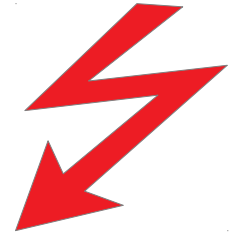
# LVSCT – Express - Apêndice



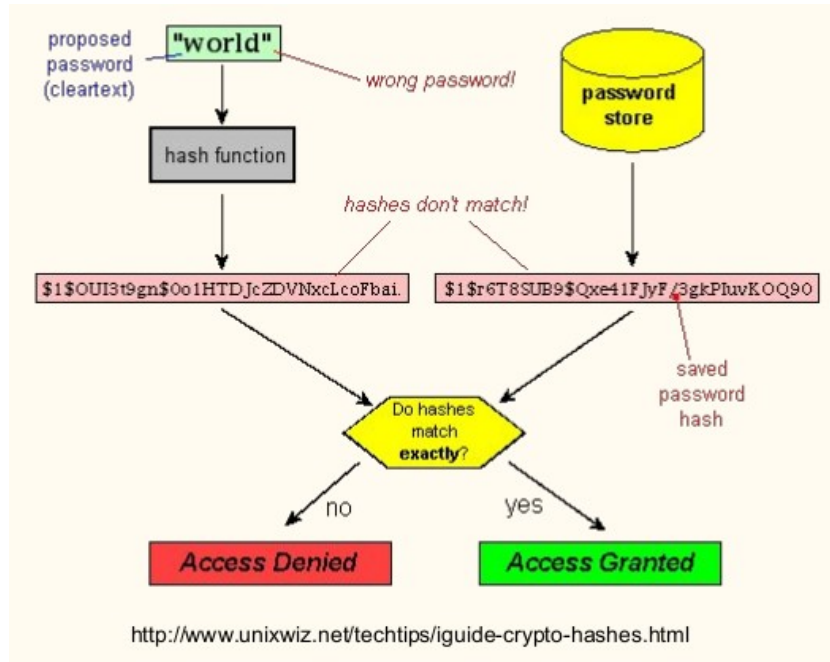
## Funções Hash

- Hash é um função que recebe dados de tamanho variável e retorna um dado de tamanho fixo
  - Esse retorno é uma cadeia de caracteres que chamamos de assinatura hash
- Propriedade importante das funções hash
  - A partir da assinatura, não é possível obter o dado original
  - Ao produzir a assinatura, se perde informação
- Assim, não guardaremos a senha em banco
  - Guardamos apenas a assinatura hash da senha
- Desse modo, mesmo que um invasor tenha posse das assinaturas hash, não é possível obter a senha original.

# LVSCT – Express - Apêndice

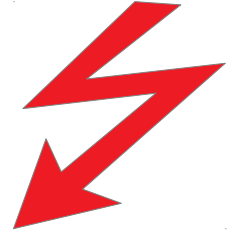


- Fluxo



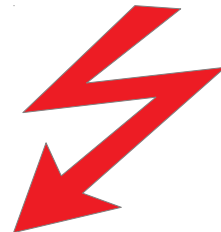


# LVSCT – Express - Apêndice



- Criptografia
- **BCrypt** no Node
  - Usaremos o BCrypt para hashear nossas senhas
  - Instalação:
    - `npm install --save bcrypt`

# LVSCT – Express - Apêndice



- **Como usar**
  - Para criar hash
  - Função `bcrypt.hashSync(data, salt)`
  - data é o dado a ser hashado
  - salt representa um inteiro usado para criar uma string que será concatenada com o dado (valor 10, por exemplo)
  - Retorna Hash
- **Para comparar dois hashes**
  - Função `bcrypt.compareSync(hash1, hash2)`
  - Retorna true caso sejam iguais ou false, caso contrário

# LVSCT – Conclusão...

- Até o momento, temos o seguinte fluxo na ação de login:
  - Do lado do cliente:
    - Usuário preenche “login” e “senha”;
    - O serviço de autenticação se comunica com o endpoint, passando o login se senha.
  - Do lado do servidor:
    - Servidor recebe login e senha.
    - Pesquisa em banco (usando o mongoose) por um usuário com o login passado.
    - Caso exista, compara sua senha criptograda com a senha passada pelo cliente.
    - Caso a senha esteja ok, retorna sucesso para o cliente com um objeto Json contendo alguns dados do cliente e o token gerado.
  - Voltando ao cliente:
    - Se tudo deu certo, o cliente recebe um objeto Json contendo o **token** gerado pelo servidor.
    - Cliente armazena usuário logado na sessão e o token em um cookie (não é obrigatório seguir esse procedimento!).

# LVSCT - Conclusão...

- **Perguntas:**

- Como o cliente envia, em suas requisições o token novamente para o servidor para que o mesmo teste sua validade?
- Como o servidor testa a validade?

# LVSCT – Token no cliente

- Do lado do cliente, toda requisição tem um **cabeçalho (header)**. Uma abordagem interessante é “**embutir**” o token no cabeçalho da requisição quando for pedir algo ao servidor.
- Mas...como colocar o token no cabeçalho de cada requisição?
  - Uma forma é fazer “no braço”, colocando de uma a uma
  - Outra forma é “interceptar” toda requisição e automaticamente colocar o token no cabeçalho.
    - Para essa abordagem iremos criar um novo serviço: **auth-interceptor.service.ts**

# LVSCT – Token no cliente

```
import { Injectable } from '@angular/core';  
import { HttpEvent, HttpHandler, HttpInterceptor, HttpRequest } from '@angular/common/http';  
import { Observable } from 'rxjs';
```

```
@Injectable({  
  providedIn: 'root'  
})
```

**auth-interceptor.service.ts**

```
export class AuthInterceptor implements HttpInterceptor {
```

```
  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {  
    const token = localStorage.getItem("access_token");
```

```
    if (!token) {  
      return next.handle(req);  
    }
```

```
    const req1 = req.clone({  
      headers: req.headers.set('Authorization', token),  
    });
```

—► Colocando o token dentro da requisição (header)

```
    return next.handle(req1);  
  }  
}
```

# LVSCT – Token no cliente

- Em app.module, carregue o serviço:

```
@NgModule({  
  declarations: [  
    AppComponent  
  ],  
  imports: [  
    BrowserModule,  
    ToastrModule.forRoot(),  
    HttpClientModule,  
  
    //created by this application  
    UiModule,  
    routing  
  ],  
  providers: [{ provide: HTTP_INTERCEPTORS, useClass: AuthInterceptor, multi: true }],  
  bootstrap: [AppComponent]  
})
```

classe criada em auth-interceptor.service.ts



# LVSCT – Token no servidor

- Cada requisição pode ser interceptada ou, pode-se colocar o teste em cada endpoint que necessite ser testado quanto ao token (infelizmente a primeira opção não terminou como esperado, vamos ficar com a segunda :)).
- Inicialmente, vamos criar uma função de autenticação do lado do servidor:
  - `auth-service.js`



# LVSCT – Token no servidor

## auth-service.js

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

```
module.exports.check = function (token, res) {  
  if(token==null || token==undefined){  
    res.status(401).json({  
      title: 'Not Authenticated'  
    })  
    return false;  
  }  
  if (!jwt.verify(token, 'secret')) {  
    res.status(401).json({  
      title: 'Not Authenticated'  
    });  
    return false;  
  }  
  return true;  
}
```

Caso não venha nenhum token, o cliente não está autorizado.

Caso venha algum token, ele é verificado junto a a chave que o criou ('secret'). Se não bater, o cliente não é autorizado.

# LVSCT – Token no servidor

- Mas como chamar a função?
  - para cada endpoint, teste o token vindo no header do request! Só não faça isso para o login e para o register. Exemplo:

```
static list(req,res){  
  if(!auth.check(req.headers.authorization,res)) return;  
  UserModel.find().then(  
    (users)=>{  
      res.status(201).json(users);  
    }  
  ).catch(  
    (error)=>{  
      res.status(500).json(error);  
    }  
  );  
}
```

Pegando a propriedade “authorization” do cabeçalho, colocada lá no cliente (interceptor). Caso a função check retorne **false**, a função não dá prosseguimento (**return**).

Faça esse procedimento para **TODO** serviço que você queria autenticar.



# **Gerenciando o usuário logado no Angular**

# Angular - Login

- Como prevenir que usuário não logados acessem páginas que eles não tem permissão?
- A ideia é interceptar algumas requisições no Angular para impedir, por exemplo, que um usuário não logada possa acessar a página de listagem de produtos.
- Caso isso aconteça, o Angular deve redirecionar o usuário para a tela de login.
- No entanto, antes devemos voltar novamente para o código do serviço **AuthUserService**, do arquivo **auth-user.service.ts**

# Angular - Login

- **auth-user.service.ts**

```
import { User } from '../models/user.model';
import { UserService } from './user.service';
import { Router } from '@angular/router';
import { Injectable } from '@angular/core';
import { BehaviorSubject, Observable } from 'rxjs';
import { ToastrService } from 'ngx-toastr';

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

  private userBehaviorSubject: BehaviorSubject<User>;
  public userObservable: Observable<User>;

  constructor(private router: Router,
               private userService: UserService,
               private toasty: ToastrService) {

    this.userBehaviorSubject = new BehaviorSubject<User>(JSON.parse(
      sessionStorage.getItem("user_login")
    ));
    this.userObservable = this.userBehaviorSubject.asObservable();
  }
}
```

Esses dois objetos permitem criar um observável para o usuário, ou seja, toda vez que objeto User for modificado (login e logout), quem estiver inscrito, será notificado. Isso é interessante para uma interface gráfica como por exemplo, o Navbar só ser mostrado caso exista um usuário logado.

UserBehaviour é iniciado vazio aqui e transformado em um observável logo depois.

# Angular - Login

```
login(login:string, password:string){
  this.userService.login(login,password)
    .subscribe(
      (res:User)=>{
        if(res!=null){
          sessionStorage.setItem("user_login",JSON.stringify(res));
          localStorage.setItem("access_token",res.token);
          this.userBehaviorSubject.next(res);
          this.router.navigate(["user/list"]);
        }else{
          this.toasty.error("Invalid user or/and password!")
        }
      }
    );
}

logout(){
  sessionStorage.removeItem("user_login");
  localStorage.removeItem("access_token");
  this.userBehaviorSubject.next(null);
  this.router.navigate([""]);
}

getLoggedUser():User{
  let lu = JSON.parse(sessionStorage.getItem("user_login"));
  return lu;
}
```

O valor de userbehaviour é modificado.  
Quem estiver inscrito em seu observável  
ser notificado.

# Angular - Login

- Existem dois tipos de “códigos”, interessados em um usuário logado:
  - **Passivo**: se inscreve no observável e só quer se notificado quando o usuário for modificado (login e logout). Deve se inscrever no observável de usuário.
  - **Ativo**: deseja saber quem é o usuário logado para tomar uma ação. Esse código “corre atrás” e não “fica esperando”. Deve chamar o método `getLoggedUser`.

# Angular – Login - Passivo

- Problema: queremos mostrar o navbar APENAS quando um usuário estiver logado.
- Obviamente não iremos escrever um código que fica checando em um laço se tem alguém logado (busy-wait).
- A solução é se inscrever no observável e, quando o usuário for modificado, tomar alguma ação.
- Vejamos, no módulo **ui**, o arquivo **menu.component.ts** (próximo slide)



# Angular – Login - Passivo

```
export class MenuComponent implements OnInit{

  user:User = null;

  constructor(private authUserService:AuthUserService){

    ngOnInit(): void {
      this.authUserService.userObservable.subscribe(
        (res:User)=>{
          this.user = res;
        }
      );
    }

    logout(){
      this.user = null;
      this.authUserService.logout();
    }
  }
}
```

O componente se inscreve no observável, tornando-se observador do estado de User. Quando um User é modificado, a variável local recebe esse valor `this.user = res;`, alertando então o seu HTML.

# Angular – Login - Passivo

- O HTML do navbar, menu.component.html
  - Note que o navbar só será renderizado caso existe um user:
    - `<nav class="navbar navbar-expand navbar-dark bg-dark" *ngIf="user">`

# Angular – Login - Ativo

- Problema: queremos que apenas certas páginas sejam acessíveis para usuários logados. Sendo assim, toda vez que um cliente clicar em um link, o nosso código deve “ir atrás” para saber se tem alguém logado, ou seja, ele é ativo.
- Para fazer isso, ele deve acessar o método `getLoggedUser`, do serviço de autenticação.
- No entanto, seria interessante interceptar o clique numa rota (link) com uma espécie de **Guarda**. Essa guarda seria uma classe especializada em verificar se existe um usuário logado ou não.
- Caso exista, a Guarda deixa o cliente continuar com a requisição. Caso contrário, volta pra tela de login.

# Angular – Login - Ativo

- guard.service.ts

```
export class GuardService implements CanActivate {
```

```
  constructor(private authService: AuthUserService,  
               private router: Router) { }
```

```
  canActivate(route: ActivatedRouteSnapshot, state:  
RouterStateSnapshot) {
```

```
    if (this.authService.isLoggedIn()) {  
      return true;
```

```
    }
```

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

```
    return false;
```

```
  }
```

```
}
```



Verifica se existe alguém logado.

# Angular – Login - Ativo

- Interceptar as rotas com a guarda (exemplo nas rotas de user):

```
import { GuardService } from '../services/guard.service';
```

```
const routes: Routes = [  
  {path:"",component:LoginUserComponent},  
  {path:'login',component:LoginUserComponent},  
  {path:'register',component:RegisterUserComponent},  
  {path:'list',component:ListUserComponent, canActivate: [GuardService]},  
  {path:'edit/:id',component>EditUserComponent, canActivate: [GuardService]},  
];
```

Esses não precisam de Guarda!

