# Criando novo usuário e validando formulário

Primeiro é necessário criar um componente de sign-up com a estrutura necessária dentro do form para criação de um novo usuário. É feito no mesma pasta de sign-in.

Feito toda estrutura, criamos uma propriedade do tipo **formGroup,** para criar o formulário é necessário de um **formBuilder** que injetamos no nosso construtor. Implementamos também o **OnInit.** Agora é montado o formulário dizendo que a propriedade criada recebe **formBuilder** utilizando o seu método **group** que é passado como objeto javascript os campos necessários para a criação. Passamos um validators para todas as propriedades com tamanho mínimo e máximo e também que precisa começar com letra minúscula a-z e número só no final 0-9

export class SignUpComponent implements OnInit{

  signForm: FormGroup;

    constructor(private formBuilder: FormBuilder) {}

    ngOnInit(): void {

      this.signForm = this.formBuilder.group({

        email: ['',

        [

          Validators.required,

          Validators.email,

        ]

      ],

        fullName: ['',

        [

          Validators.required,

          Validators.minLength(2),

          Validators.maxLength(40)

        ]

      ],

        userName: ['',

        [

          Validators.required,

          Validators.minLength(2),

          Validators.maxLength(30),

Validators.pattern(/^[a-z0-9\_\-]+$/),

        ]

      ],

        password: ['',

        [

          Validators.required,

          Validators.minLength(2),

          Validators.maxLength(14)

        ]

      ]

      })

    }

}

Agora vamos precisar preparar o formulário para utilizar as validações com o mesmo nome das propriedades nos inputs. O **formGroup** no form e **formControlName** nos campos.

Feito isso, aplicaremos a mesma lógica do componente de mensagem no sign-up acrescentando as outras validações abaixo do input. Após a propriedade de **errors**? acrescentar a validação de acordo com as propriedades no TS.

<appvmessage text="Email is required!" \*ngIf="signupForm.get('email').errors?.requi

red">

</app-vmessage>

<appvmessage text="Email invalid!" \*ngIf="signupForm.get('email').errors?.email">

</app-vmessage>

# Criando o próprio validador

Criando nosso próprio validador onde vai ser usado para todos, vai ficar em shared. É criado sua pasta e dentro o seu TS. Ele será uma function, portanto será um export function -nome da função- e todo validador tem a obrigatoriedade de receber como parâmetro um **control** que vai ser o cara do formulário do tipo **AbstractControl.**

**Control** é sinônimo de input, de coisas do formulário onde interage.

Se não tiver erro de validação é retornado **null.** Se tiver erro de validação é retornado um objeto javascript com uma propriedade **lowerCase**: **true**. Esse nome criado dentro do objeto javascript é o nome utilizado no template depois de erros.

Se: o valor digitado não tiver em branco E se o que tiver lá dentro não seguir a expressão regular vai ser executado o validador.

export function lowerCaseValidator(control: AbstractControl){

  if(control.value.trim() && !/^[a-z\_0-9\_\-]+$/.test(control.value)){

    return { lowerCase: true }

  }

  return null;

}

Agora só precisa ser chamado no OnInit na propriedade que quer validar utilizando o nome da função e no campo input utilizar o nome especificado no return.

userName: ['',

        [

          ...

          lowerCaseValidator,

...

<app-

vmessage text="Apenas letras minusculas e numeros no final do user name." \*ngIf="signupForm.get('userName').errors?.lowerCase"></app-vmessage>

# Validação assíncrona/Validator caso já exista alguém com o mesmo username

Para verificar se o usuário existe ou não no back-end é necessário criar um serviço. O serviço precisa de um **HttpClient** para acessar o método **get.** É criado uma função que recebe como parâmetro o userName o seu retorno é o método **get** com o endereço da api, passando uma string de /**user**/**exists** concatenando com o parâmetro de userName.

export class SignUpService...

constructor(private http: HttpClient){}

checkUserNameTaken(userName: string){

    return this.http.get(API\_URL + '/user/exists/' + userName);

  }

O validador vai precisar injetar o **SignUpService** mas o problema é que um validador não suporta injeção de dependência. Para resolver isso é criado um **outro** **service** na mesma pasta do sign-upcom o nome user-not-taken.validators que vai receber no seu construtor o **signUpService** que contém o método para verificar se já possui o usuário no back.

É criado um método **checkUserNameTaken** nesse serviço que ao ser chamado retorna uma função de validação.

constructor(private signUpService: SignUpService){}

  checkUserNameTaken(){

  }

Quando esse método for chamado vai retornar uma função que recebe um **AbstractControl** em algum momento vai retornar null caso não tenha erro de validação e também retorna um objeto caso tenha erro de validação. Podemos perceber que é uma arrow-function que recebe como parâmetro o AbstractControl.

checkUserNameTaken(){

    (control: AbstractControl) => {

      return null

    }

  }

Agora precisamos injetar esse serviço que retorna uma função no componente TS responsável pela validação que nesse caso é o sign-up componente. Importamos ele no constructor.

Na lista da propriedade que deseja aplicar a validação (**userName**) o segundo parâmetro é um array de validadores síncronos, nossa validação criada é assíncrona então não pode entrar nessa lista. É passado como terceiro parâmetro nosso service criado que retorna a função.

 constructor(…, private userNotTakenValidatorService: UserNotTakenValidadorService)

userName: 1º ['',

       2º [

          Validators.required,

          lowerCaseValidator,

          Validators.minLength(2),

          Validators.maxLength(30),

        ],

       3º this.userNotTakenValidatorService.checkUserNameTaken()

Agora é necessário voltar ao validador e implementar a lógica. A ideia é retornar **null** ou um **objeto** **javascript** mas um validador assíncrono retorna um **Observable** que ao ser acessado consegue pegar o valor no final que vai ser null ou objeto javascript. Não podemos trabalhar com **control**.**value**(o valor digitado no input) mas podemos trabalhar com **control**.**valueChanges** pois é um Observable.

Nesse primeiro momento importamos o **debounceTime** do rxjs/operators para que o validador assíncrono não verifique a cada digito e sim depois de um tempo sem digitar. Passamos o tempo de 300milisegundos ao debounceTime.

import { debounceTime } from 'rxjs/operators';

checkUserNameTaken(){

    return (control: AbstractControl) => {

      return control.valueChanges

      .pipe(debounceTime(300))

    }

  }

}

E agora é necessário fazer a pesquisa na api para verificar se já existe algum userName com o mesmo valor digitado. Vamos precisar do swithMap do rxjs/operators. É passado um pipe e dentro do pipe o swithMap é chamado. Feito isso uma arrow-function é criada recebendo **userName** e retornando o método responsável por ir na api recebendo como parâmetro o userName.

checkUserNameTaken(){

    return (control: AbstractControl) => {

      return control.valueChanges

      .pipe(debounceTime(300))

      .pipe(switchMap(userName => {

        return this.signUpService.checkUserNameTaken(userName);

      }))

    }

  }

Utilizamos o swithMap porque primeiro estamos pegando a emissão do **observable** **valueChanges**, assim que pegar a emissão dele é retornado a emissão do **checkUserNameTaken** para que não tenha as duas emissões simultâneas é necessário parar de executar o fluxo anterior (o observable valueChanges) e troca para o fluxo do **checkUserNameTaken.** A pessoa digita, quando processar o debounceTime não é necessário ouvir mais nada, nenhuma emissão do debounce e agora vai trabalhar com a emissão do checkUserNameTaken.

Agora o resultado disso é true ou false mas nós precisamos que vire null ou objeto javascript e por isso vamos utilizar o **map** do **rxjs** e dizemos para fazer um **map** do resultado chamado **isTaken** e se esse cara é true é jogado um objeto javascript { **userNameTaken**: **true** } e se não der problema nenhum vai retornar **null**.

checkUserNameTaken(){

    return (control: AbstractControl) => {

      return control.valueChanges

      .pipe(debounceTime(300))

      .pipe(switchMap(userName => {

        return this.signUpService.checkUserNameTaken(userName);

      }))

      .pipe(map(isTaken => isTaken ? { userNameTaken: true} : null))

    }

  }

Agora o sistema precisa identificar o momento que foi finalizado, que já passou por todas as etapas porque se ficar desse jeito ele vai ficar escutando o tempo todo. Para forçar que ele complete é importado o operador **first** do **rxjs** e colocamos no pipe dizendo que o primeiro valor que for emitido vai pegar esse primeiro valor e completar esse método. A cada digito depois dos 300 milissegundos é uma emissão.

checkUserNameTaken(){

    return (control: AbstractControl) => {

      return control.valueChanges

      .pipe(debounceTime(300))

      .pipe(switchMap(userName => {

        return this.signUpService.checkUserNameTaken(userName);

      }))

      .pipe(map(isTaken => isTaken ? { userNameTaken: true} : null))

      .pipe(first());

    }

  }

Agora é necessário ir no **template** responsável por registrar um novo usuário e criaremos um novo componente de mensagem de validação e quando houver erro aplicar o **userNameTaken** (nome criado no método checkUserNameTaken). Depois é colocado uma tag **small** pra quando o userName for valido dizer que o userName está disponível.

<app-

vmessage text="Username already taken" \*ngIf="signupForm.get('userName').errors?.userNameTaken"></app-vmessage>

<small \*ngIf="signupForm.get('userName').valid" class="text-success">Username available</small>

Para finalizar, o nosso botão que tem a propriedade de **disable** recebe o operador lógico **OU** || para que sempre que tiver uma validação assíncrona testar também se tem alguma validação assíncrona em processamento.

<button [disabled]="signupForm.invalid || signupForm.pending" class="btn btn-primary btn-

block">Register</button>

# Fazendo submissão do formulário

Para fazer a submissão dos dados do formulário vamos criar um método dentro do TS do componente de **signup** chamado **signup.** E para pegar os campos do formulário vamos utilizar um jeito diferente da forma de login, como são muitos campos no nosso método é criado uma **const** e passamos para essa const a nossa propriedade do tipo **formGroup** (responsável por receber todos os campos do formulário) utilizando o método **getRawValue.** Esse método ele te dá um objeto javascript com todas as propriedades criadas e com os valores digitados no formulário. Feito isso **tipamos** nossa variável dizendo que ela é como (**as -** o tipo dela) a interface criada com as propriedades que vão ser utilizadas.

signup(){

      const newUser = this.signupForm.getRawValue() as NewUser;

    }

export interface NewUser{

  userName: string,

  email: string,

  fullName: string,

  password: string

}

Agora é criado outro método de signup só que no service responsável por consumir api (signup.service). Vamos fazer uma requisição do tipo **post** passando a **const** criada com o endereço da api e com o caminho de /user/signup e como terceiro parâmetro a propriedade **newUser** que é nossa interface declarada no parâmetro do método.

signupService(newUser: NewUser){

    return this.http.post(API\_URL + '/user/signup', newUser)

  }

Agora injetamos nosso service que faz um post na api e usamos ele no nosso método de **signup** passando como parâmetro a const newUser.

signup(){

      const newUser = this.signupForm.getRawValue() as NewUser;

      this.signUpService.signupService(newUser);

    }

Agora fazemos um **subscribe** que se der certo o registro é redirecionado direto pra página de login e para isso é necessário utilizar o **Router** no construtor. Feito a injeção do Router criaremos uma arrow-function chamando o **navigate** e passando uma string em branco(caminho do login).

signup(){

      const newUser = this.signupForm.getRawValue() as NewUser;

      this.signUpService.signupService(newUser)

      .subscribe(() => this.router.navigate(['']),

      error => {

        console.log(error);

      })

    }

EXTRA:

Criando uma mensagem de alerta depois de registrar um usuário.

É criando um novo componente em **shared** pois pode ser usado em outros lugares, e é passado uma interpolação com o texto que vai ser utilizado. Feito isso, no TS do componente de mensagem é criado uma propriedade com o mesmo nome da interpolação e passado um **Input()** para que possa receber valor fora do componente. Passamos ao modulo na declaração e no export pois vai ser utilizado no componente de registro.

Agora no TS responsável por registrar um novo usuário criaremos duas propriedades, uma de **text** onde vai ser feita a property-binding com a propriedade da mensagem. É feita a ligação na declaração do **app**-**componente**. E a outra propriedade recebera um valor booleano **false.** No subscribe do método de registro é chamado as duas propriedades: **this**.**text** recebendo a string e a outra propriedade recebendo um valor true. No **setTimeout** é chamado uma arrow-function que vai ser executado a navegação depois do intervalo de 3000milissegundos. Depois disso, na chamada do ap-mensagem é feito um ngIf que se a propriedade criada recebe um valor booleano true vai executar esse template.

Template e TS

<div class="alert alert-success" role="alert">{{text}}</div>

@Input() text: string = '';

Chamada no template do SignUp

<app-message-signup [text]="text" \*ngIf="msgSuccess"></app-message-signup>

TS SignUp

text: string = ''

msgSuccess: boolean = false;

this.signUpService.signupService(newUser)

      .subscribe(() => {

        this.text = "Registrado com sucesso!"

        this.msgSuccess = true;

        setTimeout(() => {

          this.router.navigate([''])

        }, 1000)

      },

      error => {

        console.log(error);

      })

    }