Angular

Evelise Dib

Frontend na Loft | Mentora na Laboratória | Instrutora na Caelum

Jogadora de Don’t Starve assídua,
eu vim de Santos, cultuo bons
livros e amo os animais 🎵
(tá ligado eu sou o bicho).

Dia 1

Photo: @markusspiske - Unsplash

AngularJs

- Utilizava Javascript
- Foi criado em 2009
- Conceito de S.P.A.
- Mantido pela Google

Angular 2

- Mudança de linguagem
- Semantic Version
- MVC
- Criado em 2016

O que eu preciso para começar a usar?

- Um pouco de café
- Node instalado
- NPM instalado
- Angular CLI instalado

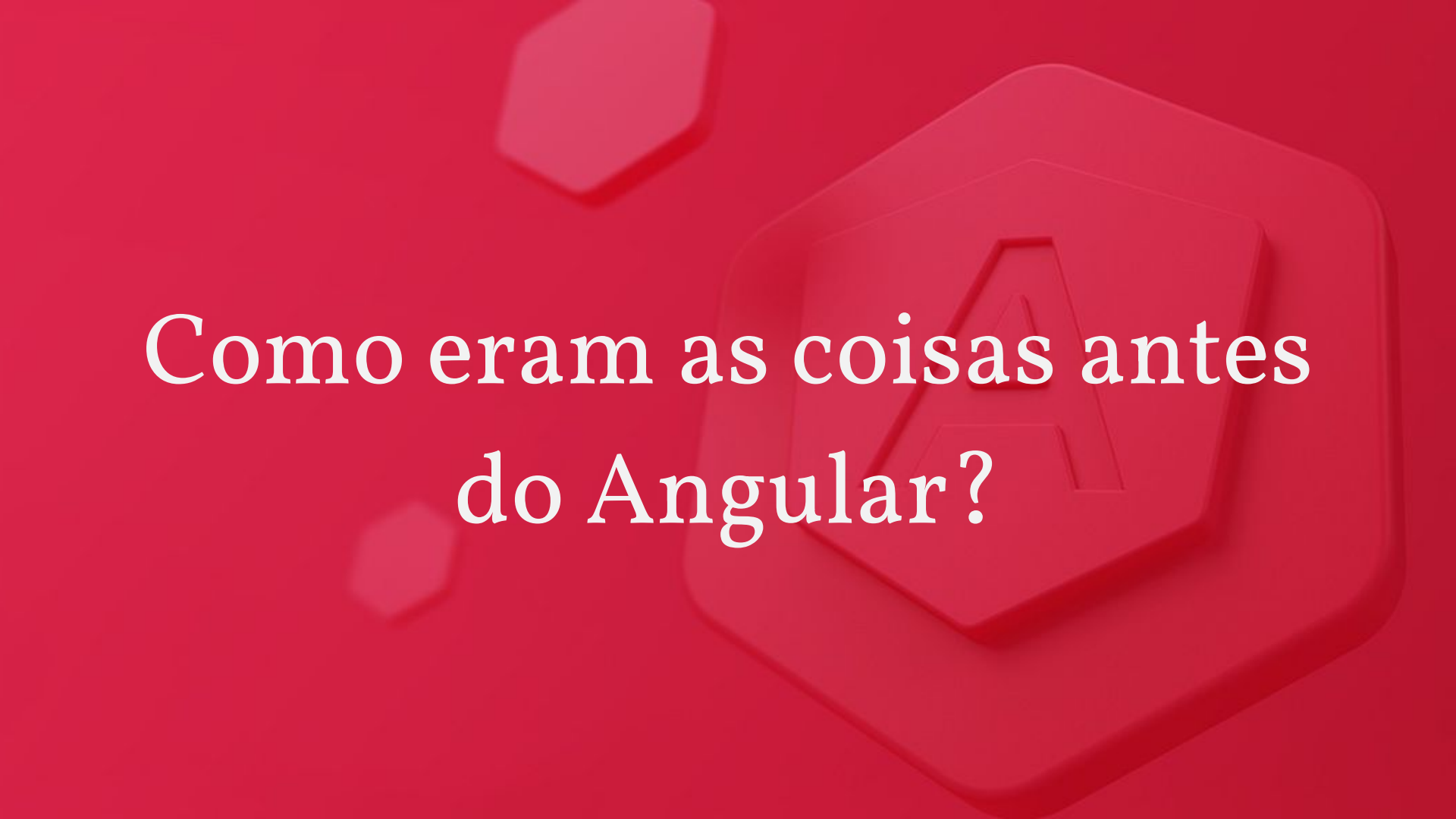
Photo: @nate_dumlao - Unsplash

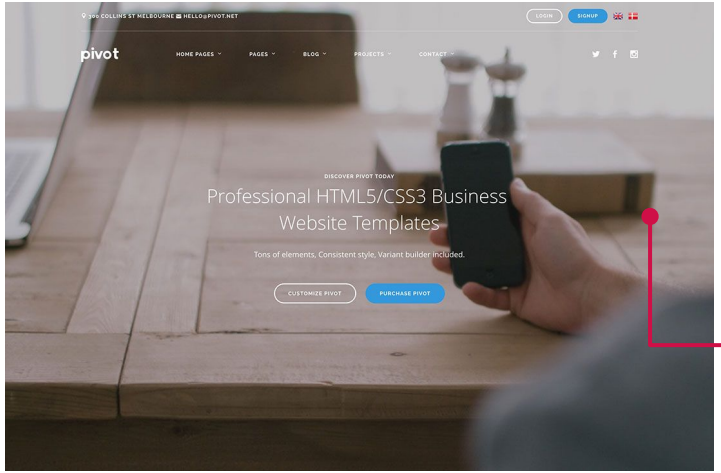
Photo: @markusspiske - Unsplash

CLI

- Command Line Interface
- Mandar comandos escritos para que o computador faça o que você está pedindo

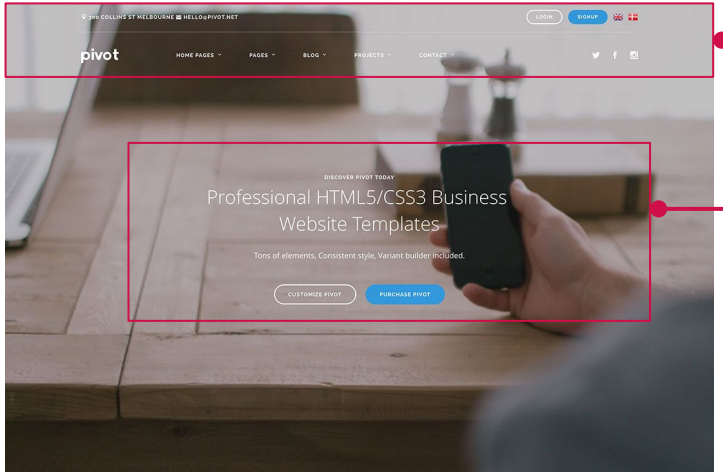
- Abra seu terminal de preferência
- Digite ***cal*** e dê enter


Como eram as coisas antes do Angular?



template (html)	lógica (js)
estilos (css / scss)	dados back-end

E como fica com Angular?



component

template (html)	lógica (ts)
estilos (css / scss)	dados back-end

app.component.ts

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: './app.component.html',
  stylesUrl: ['./app.component.css']
})
export class AppComponent { }
```


 Algum teste aqui
<app-root></app-root>

Dia 2

Photo: @markusspiske - Unsplash

Inicialização

- Ng new é usado para criar uma aplicação do zero
- Ng serve é usado para rodar o servidor local
- A cada modificação de código o servidor é atualizado automaticamente

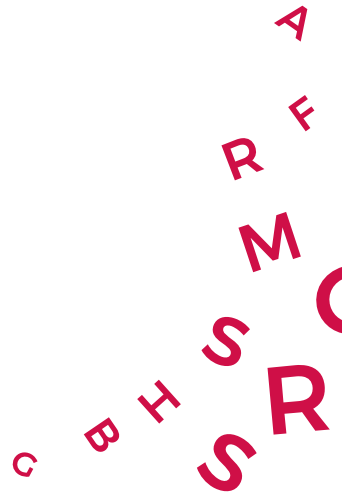
```
> npm install -g @angular/cli
> ng new my-dream-app
> cd my-dream-app
> ng serve
```

Classes

- No final das contas, o Typescript vira Javascript,que é compilado pelo Babel e já vem configurado dentro da aplicação.
- Para que cada component exista, é preciso estar atrelado à uma classe
- A classe tem ser acessível por outras partes internas do código
- Qual a diferença de uma classe comum de um Component?

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: './app.component.html',
  stylesUrl: ['./app.component.css']
})
export class AppComponent { }
```



Decorators

São modificadores de classes, definindo algum tipo ou categoria específica.

É usado com o @

Estilos

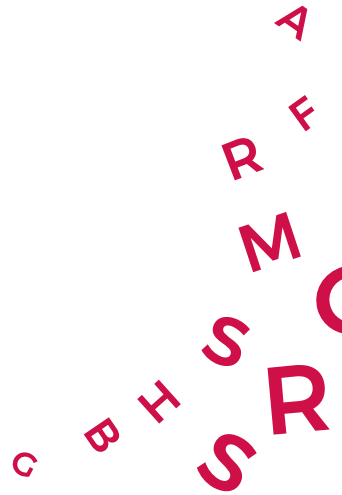
Por component

- É acoplado apenas ao component de referência, não afeta globalmente.

Globalmente

- Afeta o escopo de todo projeto

```
<head>
<meta charset="utf-8">
 <title>Página</title>
 <!-- Material Design -->
 <link rel="stylesheet"
href="https://code.getmdl.io/1.3.0/material.indigo-red.min.css" />
 <!-- ./Material Design -->
</head>
<body>
    <app-root></app-root>
</body>
```



Almoço

Interpolação de template

```
<label>{{textoDaLabel}}</label>
```

```
<div> {{10+13}} </div>
```

- Usado para imprimir no html algum valor fixo ou dinâmico, podendo ser de uma variável, função ou até mesmo um inteiro

Property binding

```
<button disabled>Clique-me</button>
<button [disabled]="false">Clique-me</button>
<button [disabled]="variavel">Clique-me</button>
<!---- ---->
<img [scr]="logoDinamico" />
```

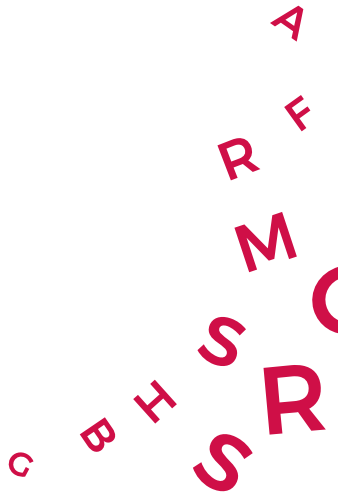
- Através dos [ ] ao redor de alguma atributo é feita a associação (podendo ser um inteiro, booleano, métodos da classe...)

One Way data-binding

```
<button disabled>Clique-me</button>
<button [disabled]="false">Clique-me</button>
<button [disabled]="variavel">Clique-me</button>
<!---- ---->
<img [scr]="logoDinamico" />
```

```
<label>{{textoDaLabel}}</label>
```

- Quando passamos um valor de uma camada para outra.
- Ex.: Da classe até a view



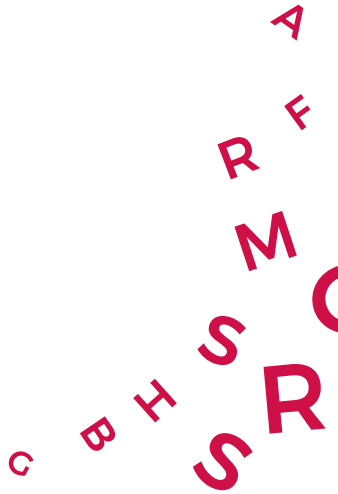
Event Binding

```
<button (click)="onCall()">Clique-me</button>
<input (input)="executar()" />
```

- Associação por eventos.
- Usadas para escutar os eventos padrões do browser e criar alguma lógica através dela.
- A sintaxe é usada com os ( ) em volta do evento

O component root

- Se tudo faz parte de um component, então por onde começar?
- Quando criamos um projeto, o angular seta as configurações de um component raiz para saber de onde partir.
- Dica 1: É possível descobrir a partir do módulo!
- Dica 2: Procuro pela tag no arquivo index.html



Diretivas

- Você pode criar as suas próprias diretivas customizadas
- O ciclo de vida pode ser usado aqui

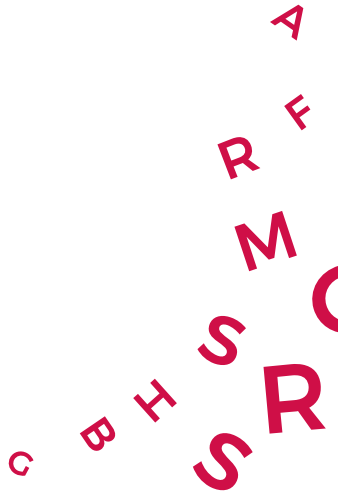
```
import { Directive } from '@angular/core';

@Directive({
  selector: '[customField]'
})

export class CustomFieldDirective {}
```

Arquitetura LIFT

- Locate: Localizar intuitivamente: manter os arquivos relacionados próximos quando
- Identify: Identificar: nomear os arquivos de forma que representa o que ele faz
- Flat: Plano: estrutura de pastas simples -> nunca ultrapassar o limite de 9 camadas
- T-Dry: Não se repetir: Não repita sem necessidade (nomes de view em arquivos com extensão html, por exemplo)



NgClass

- Diretiva específica para manipular classes
- Usada para manipular classes dinâmicas através de algum atributo da classe (quando queremos esconder algum item através de um click por exemplo)

```
<some-element [ngClass]="'first second'">...</some-element>

<some-element [ngClass]="{'first': true, 'second': true, 'third': false}">...</some-element>

<some-element [ngClass]="{'class1 class2 class3' : true}">...</some-element>
```

Sintaxe

Event (Output)

Usado para lidar com os eventos keyDown, keyUp, click

Template

Usados para alteração do DOM (ngIf, ngFor)

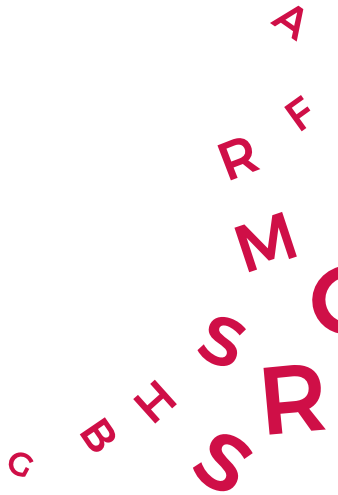
Directive (Input)

Usado para adicionar atributos à tag

Dia 3

Módulos

- Usados para separar melhor os contextos
- Também é usado para o lazy-load
- Imports: Módulos e dependências que você precisa usar
- Exports: Exportação de components ou diretivas
- Declarations: Declaração de components e diretivas
- Bootstrap: Component que será inicializado
- Providers: Serviços (são carregados antes da view)



Ng-content

- Usado para inserir templates dinâmicos
- Funciona como um placeholder
- Insere no lugar marcado o que estiver dentro das custom tags do component

```
<app-child>
    <div header>This should be rendered in header selection of ng-
content</div>
    <div body>This should be rendered in body selection of ng-content</div>
</app-child>
```

```
<div class="header-css-class">
    <ng-content select="[header]"></ng-content>
</div>
<div class="body-css-class">
    <ng-content select="[body]"></ng-content>
</div>
```

Dia 4

$event

- Usado pelo event binding
- Retorno de um ‘handler” de eventos, como se fosse uma resposta do evento

```
<button (click)="cadastrar($event)">
  click-me
</button>
```

NgFor

- Diretiva usada para iterar listas no template
- Funciona como uma estrutura de repetição padrão (for)
- * é usado quando modificamos o DOM (adicionar ou remover algo)

```
<ul>
  <li *ngFor="let usuario of listaUsuarios">
   {{usuario.nome}} - {{usuario.email}}
  </li>
</ul>
```

Refêrencias

- # são usadas para criar referências rápidas de template
- Usada para consultar detalhes dos elementos
- Pode ser usada para atribuir valores

```
<input #nivel value="12" />
{{nivel.value}}
```

NgModel

- NgModel é o que chamamos de two-way-data-binding
- Também é chamado de banana in a box [()]
- Utilizado para formulários (ao mesmo tempo que pega os eventos, salva o valor na variável
- Precisa conter um “name” no campo
- Precisa importar o FormsModule no módulo do component

```
<input #nivel="ngModel" name="nivel" [(ngModel)]="form.nivel" />
```

Dia 5

NgSubmit

- Exclusivo para a tag <form>
- Evita o reload da página (form com action faz reload, por exemplo)
- O botão de ação fica no form e não no button

```
<form #form="ngForm" (ngSubmit)="enviar($event)">
  ...
</form>
```

*ngIf

- Usado para criar regras que devem ser omitidas do template de acordo com alguma variável ou condição

```
<p *ngIf="resultadoPesquisa === null">Nada foi encontrado.</p>
```

Validações de template

- Usadas para formulários menores sem tanta validação
- Precisa importar o módulo do FormsModule na aplicação
- Usado criando referências para acessar as variáveis do template

```
<form #formLogin="ngForm" (ngSubmit)="login(formLogin)">
  <input [(ngModel)]="formulario.nome"  #nome="ngModel" required />
  {{nome.invalid}}
  <input [(ngModel)]="formulario.senha" #senha="ngModel" required />
  {{senha.invalid}}
</form>
```

Dia 6

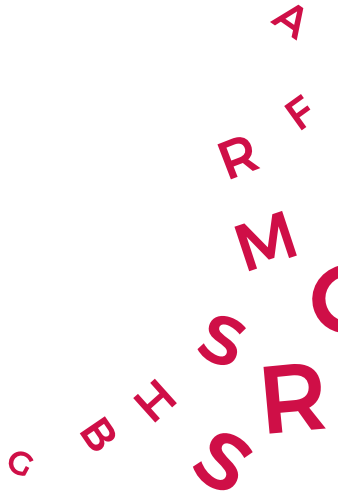
Rotas

- Fica definido o arquivo de módulos da aplicação
- Utiliza o módulo de RouterModule e define um objeto com o caminho e o component
- Precisa usar o component <router-outlet></router-outlet> para definir onde os components serão renderizados (funciona como um placeholder).

```
const appRoutes: Routes = [
  {
    path: '',
    component: HomeComponent
  },
  {
    path: 'cadastro',
    component: HomeComponent
  },
  {
    path: 'ataques',
   component: HomeComponent
  },
  {
    path: '**',
    redirectTo: ''
  }
];
```

Separar módulos

- É importante separar os módulos para garantir o que chamamos de lazy-load
- Cada arquivo e módulo é carregado apenas quando necessário, dessa forma uma S.P.A não é prejudicada na performance.



Dia 7

Input

- Quando existe dependência de components (pai e filho) é possível passar informações para os components existentes.
- Uma forma de enviar os dados é através dos INPUTs

```
<meu-component [dados]="meusDados"></meu-component>
```

Output

- Também é possível enviar dados para fora do component com eventos, dessa forma fazemos o output dos dados
- Para receber os valores no component pai é necessário usar o handler de eventos $event

```
<meu-component [dados]="meusDados" (evento)="algumaFuncao($event)"></meu-component>
```

Broadcast

- Quando precisamos enviar os dados para outros components sem dependência direta, utilizamos o broadcasting dos dados.
- Utilizamos o design pattern de observable para escutar esses valores sendo alterados

```
import { Injectable } from '@angular/core';
import { Subject } from 'rxjs';

@Injectable({
    providedIn: 'root'
})
export class TitlePageService {
 dados = new Subject<string>();

 enviarDados(novoValor: string){
    this.dados.next(novoValor);
 }
}
```

Dia 8

Formulários reativos

- Uma outra forma de validar os dados de um formulário é usando os formulários reativos
- São ótimos quando precisamos validar muitas coisas ou quando a regra de negócio é mais complexa
- Sua lógica fica fixa na maioria das vezes nos arquivos .ts
- É necessário importar o ReactiveFormsModule

```
export class AlgumComponent {
    formCadastroLogin = new FormGroup({
    nome: new FormControl('', Validators.required),
    classe: new FormControl('', Validators.required),
    raca: new FormControl('', Validators.required),
    telefone: new FormControl('', [
        Validators.required,
        Validators.pattern(/[0-9]{4}-?[0-9]{4}[0-9]?/),
    ]),
    nivel: new FormControl('', Validators.required),
  });
}
```

Usando Http

- Para conseguir fazer uma requisição para outra página é necessário usar o Http, dessa forma é possível usar os padrões REST.
- Para utilizar, é preciso importar o módulo HttpClientModule

```
export class AlgumComponent {

  constructor(private httpClient: HttpClient){}

  getLogin() {
    return this.httpClient.get('http://algumaApi');
  }
}
```

Serviços

- São classes com o decorator customizado Injectable.
- São utilizados para separar regras, assim fica fácil de separar as responsabilidades
- É possível utilizar dentro do Injectable a propriedade provideIn, dessa forma, se utilizado com o parâmetro 'root' fica disponível na aplicação toda, não tendo a necessidade de importar no módulo

```
@Injectable()
export class CustomService {


}
```

Dia 9

Pipes

- Pipes são transformadores e assim como tudo no angular, funcionam como uma classe, a diferença é que para utiliza-los da melhor forma, é necessário implementar a interface

```
@Pipe({name: 'algumNome'})
export class AlgumNomePipe implements PipeTransform {
  transform(value: number, exponent?: number): number {
    return Math.pow(value, isNaN(exponent) ? 1 : exponent);
  }
}
```

rxJS

- Biblioteca de programação funcional já importado pelo framework como padrão.
- Utilizado na maioria das vezes para interceptar o retorno de API`s
- Encapsulamos o operador pipe que permite manipular o contexto e adicionar quantos operadores quisermos

```
return this
  .httpClient
  .get(`https://viacep.com.br/ws/${cep}/json/`)
  .pipe(
    map((response: any) => {
      return response.actions;
    }),
    catchError(error => {
      console.log(error);
      throw new Error('Não foi possível completar a requisição`);
    })
  )
```

Dia 10

Desafio

Substitua os components que usamos até agora do Material Light pelo Material Angular!