

Treinamento Front-End – ReactJS –  
TypeScript - Angular

# Sumário

ReactJs.....	4
Recursos ReactJS .....	4
Configuração e Instalação .....	4
Estrutura de Diretórios do Aplicativo React.....	6
O que é um componente?.....	7
Comunicação de Componentes .....	15
O que é props? .....	15
Flux - Redux .....	24
Usando Flux.....	25
TypeScript.....	34
O que é o TypeScript? .....	34
Recursos do TypeScript .....	34
TypeScript e ECMAScript.....	35
Componentes do TypeScript .....	35
Sintaxe básica .....	36
Compilar e executar um programa TypeScript .....	36
Tipos .....	38
Tipo any .....	39
Variáveis .....	39
Declaração variável no TypeScript .....	40
Inferência de tipo .....	41
Declaração <i>let</i> .....	42
Declaração <i>Const</i> .....	44
Loops .....	46
Loop definido - <i>for</i> .....	47
O loop <i>for ... in</i> .....	47
Loop <i>for ... of</i> .....	48
Loop indefinido - <i>while</i> .....	49
Loop <i>While</i> .....	49
do... while loop .....	51
Classes .....	53
TypeScript e orientação a objeto .....	53
O framework Angular .....	54

A arquitetura do framework Angular.....	55
Aplicações Angular .....	57
Visão geral.....	57
Angular - Configuração do ambiente .....	58
Configuração do Projeto .....	61
Data Binding .....	70
O que é Data Binding?.....	70
Necessidade do Data Binding.....	70
Interpolação (Interpolation).....	71
Routing .....	73
Ligação/Vinculação de dados baseada em propriedades (Property Binding) .....	77
Vinculação/ligação por evento (Event Binding) .....	79
Vinculação/ligação de dados bidirecional (Two-Way Data Binding) .....	82
Input property .....	86
Output Property .....	89
Diretivas .....	94
Diretivas de estrutura.....	94
*ngIf.....	94
ngIf else .....	95
ngIf then else.....	96
ngFor .....	101
Diretivas de atributo .....	107
ngStyle .....	110
Diretivas de componente .....	113
Pipes .....	116
Pipe personalizado .....	120
Forms.....	123
Template Driven Form .....	123
Model Driven Form .....	127
Validação de formulário .....	130
Services.....	134
O que é um service.....	135
Para que services são utilizados.....	135

Vantagens ao se usar services.....	135
Como criar um service no Angular .....	135
Angular Dependency Injection.....	141
O que é dependência .....	141
Angular Dependency Injection -definição.....	141
Estrutura da Angular Dependency Injection.....	144
Como funciona a injeção de dependência no Angular .....	145
Como usar a dependency Injection (injeção de dependência).....	146
Injetando Service em Service .....	148
Referencias:.....	154

## ReactJs

### O que é React?

- React é uma biblioteca JavaScript — uma das mais populares.
- é um projeto de código aberto criado pelo Facebook.
- não é um framework (ao contrário do Angular).
- é usado para construir interfaces de usuário (UI) no front-end.
- é usado para criar aplicativos de página única (SPAs).
- React é a camada de visualização de um aplicativo MVC (Model View Controller)

ReactJS é uma biblioteca JavaScript usada para construir componentes reutilizáveis da interface do usuário. Isso significa que é uma biblioteca de visualizações que usa componentes para alterar o conteúdo da página sem atualizar, que é o princípio central por trás de aplicativos de página única.

### Recursos ReactJS

- JSX - JSX significa JavaScript XML. JSX é uma extensão de sintaxe JavaScript. JSX nos permite escrever HTML no React. Não é necessário usar JSX no desenvolvimento do React, mas é recomendado. Com JSX, você pode escrever expressões dentro de chaves { }, incluindo variáveis, funções e propriedades.
- Fluxo de dados unidirecional - O React implementa um fluxo de dados unidirecional. No React, os valores de dados são passados para cada componente como propriedades em suas tags HTML. O componente não pode modificar diretamente nenhuma propriedade, mas pode passar uma função de retorno de chamada e, com a ajuda disso, pode modificar os dados.

### Configuração e Instalação

- 1) Instale o [Node.js](#) e o [npm](#) globalmente em sua máquina. (use a versão LTS node.js)

- 2) pelo prompt de comando faça a instalação da lib `create-react-app` inserindo o comando abaixo:

```
npm install -g create-react-app
```

- 3) O Facebook criou o ambiente [Create React App](#) que criará um servidor de desenvolvimento e compilará automaticamente o React

e todas as outras coisas. Para criar qualquer aplicativo React, execute o seguinte código no seu terminal:

```
npx create-react-app novo-app
```

**create-react-app** configurará tudo o que você precisa para executar um aplicativo React. Recomendação: certifique-se de não usar letras maiúsculas para o nome do seu aplicativo.

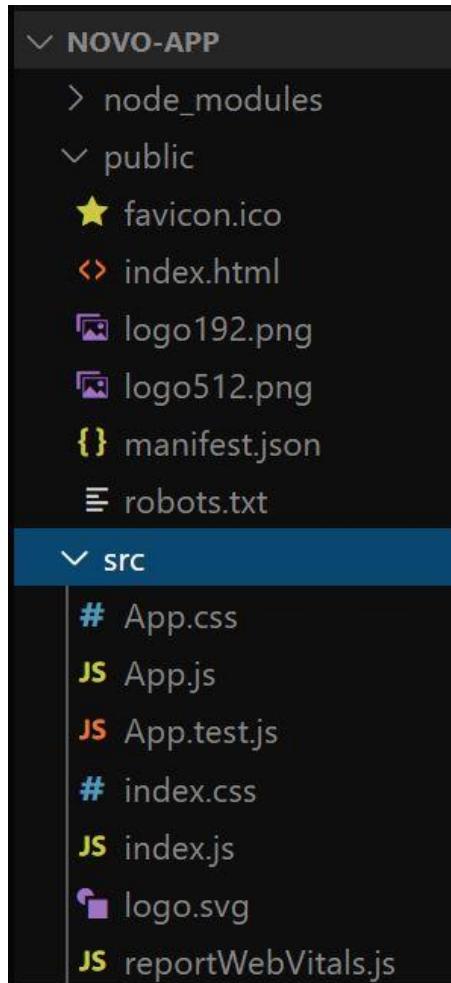
Então vamos conferir nosso navegador – executando o comando **npm start** - e podemos ver que nosso aplicativo está instalado e funcionando em <http://localhost:3000>



## Estrutura de diretórios

Observe a estrutura de diretórios que o React CLI criou para o projeto.

Abra seu projeto - recém-criado - no VS Code.



## Estrutura de Diretórios do Aplicativo React

O primeiro diretório é `node_modules` e isso está tendo toda a dependência da biblioteca de `package.json`.

### Diretório public.

Neste diretório, nosso arquivo importante é `index.html`. Como você sabe, o React é um aplicativo de página única(SPA), por isso existe um único html ou - podemos dizer - um único arquivo que é “*renderizado*” quando executamos o aplicativo. Além disso, concentre-se no elemento descrito com a `<div>` com `id="root"`.

```
<div id="root"></div>
```

Esta `<div>` será substituída pelo componente raiz.

Dentro deste diretório, você também encontrará `manifest.json`. Este arquivo é para PWAs — Progressive Web Apps.

O próximo é o diretório **src**. Ele conterá todo o nosso código React. Possui o arquivo **App.js** - o primeiro componente - que será *renderizado* quando executarmos nossa aplicação. Mas quem está *renderizando* este componente do aplicativo? A resposta é: o arquivo **index.js**. Você pode ver que este arquivo está composto, também, pelo método **ReactDOM.render()**.

```
ReactDOM.render(<App />,
document.getElementById('root'));
```

Este método de renderização vai renderizar a *div root* do arquivo index.html.

Ele substituirá essa *div* pelo componente app e seu conteúdo. É assim que seu primeiro componente funcionará.

- No diretório **src**, também observamos o arquivo **index.css**.
- Segundo em frente, observamos o arquivo **package.json**. Ele contém todas as dependências do nosso aplicativo React.

### O que é um componente?

Um Component é um “pequeno pedaço reutilizável” de código. Os componentes são os principais blocos de construção dos aplicativos React.

Um aplicativo React pode ser visto como uma árvore de componentes - tendo um componente raiz (“App”) e muitos componentes filhos aninhados. São como funções javascript. Eles aceitam entradas arbitrárias (chamadas de “props”) e precisam retornar / renderizar algum código JSX (elementos React) que descreve o que deve aparecer na tela.

Também podemos usar vários JSX em um componente. Variáveis e condições devem estar dentro do método de renderização.

Ao criar um componente React, o nome do componente deve começar com uma letra maiúscula.

### Tipos de Componentes

Basicamente, os componentes têm dois tipos:

- Componentes de classe
- Componentes de função

### **Componentes de classe**

Vamos criar um componente baseado em classe (a partir do nosso projeto criado anteriormente). Portanto, agora, para organizar adequadamente a estrutura de nosso aplicativo, crie um diretório nomeado como **componentes** dentro do diretório **src**. Todos os componentes deste projeto serão criados dentro deste diretório.

Agora clique com o botão direito do mouse em seu diretório **componentes** e crie um novo arquivo – nomeie-o como **Demo.js**.

Dentro deste arquivo - *Demo.js* -, será criado um componente de classe. Para isso, é necessário seguir as etapas abaixo:

- i) Primeiro, importe o pacote React da biblioteca React. Observe a instrução abaixo:

```
import React, {Component} from 'react';
```

- ii) Agora, crie uma classe e dê a ela qualquer nome. Esta classe deve estender uma interface de componente. Agora, crie um método *render()* nesta classe.

O método *render()* será chamado automaticamente pelo ReactJs quando necessário.

#### ***Demo.js***

```
import React, {Component} from 'react';

class Demo extends Component{
    render() {
```

```

        return (
            <div>
                <h2>Ola Mundo ReactJS!</h2>
            </div>
        )
    }

export default Demo;

```

O que quer que você queira exibir em seu componente, é necessário escrevê-los dentro deste método render () .

*iii)* A última etapa é **exportar** o componente.

```
export default Demo;
```

Portanto, um componente de classe é criado usando a sintaxe descrita acima.

Agora, é necessário exibir este componente em nosso navegador. Para isso, temos que importar / chamar o *Componente Demo* dentro do arquivo *App.js* , pois *App.js* é o único componente que será *renderizado* quando a aplicação for executado.

Abra o arquivo ***App.js*** e implemente declaração abaixo:

### ***App.js***

```

import './App.css';

// importando componentes
import Demo from './componentes/Demo';

function App() {
    return (
        <div className="App">
            <header className="App-header">
                <Demo />
            </header>

```

```

        </div>
    );
}

export default App;

```

Chame o componente `<Demo />` dentro do método de retorno.

Agora, é possível ver “Ola Mundo ReactJS” no seu navegador.

## Ola Mundo ReactJS

Também existe outra forma de exportar este componente Demo. Podemos escrever a exportação como uma instrução embutida.

No arquivo `Demo.js`, remova toda a linha `'export default app,'` e escreva `'export default'` antes da palavra-chave `'class'`.

```

export default class Demo extends Component {
    .....

```

E isso vai funcionar da mesma forma que antes.

### ***Demo.js***

```

// importado os recursos necessários para a criação do componente de classe
import React, {Component} from 'react';

// criando a classe que se tornará nosso componente
export default class Demo extends Component{
    render() {
        return(
            <div>
                <h2>Ola Mundo ReactJS!</h2>
            </div>
        )
    }
}

//expor o componente Demo.js para os outros elementos do projeto
//export default Demo;

```

A palavra - chave `default` é responsável por possibilitar essa abordagem. Se usarmos `default` com a nossa declaração de classe, em seguida, podemos importar este componente com qualquer nome em `App.js`.

Por exemplo, se o nome da classe – dentro do arquivo `App.js` – for alterado como indicado abaixo, ainda assim é possível fazer uso do recurso indicado em `Demo.js`. Observe o código abaixo:

### **App.js**

```
import './App.css';
// importando componentes
import Demo15 from './componentes/Demo';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <Demo15 />
      </header>
    </div>
  );
}

export default App;
```

Agora salve e execute o aplicativo; o mesmo resultado será exibido.

Agora, se a palavra reservada '`default`' for excluída, ao executar novamente a aplicação, o resultado será semelhante ao indicado na imagem abaixo:

---

### Failed to compile

```
./src/App.js
Attempted import error: './components/Demo' does not contain a default export (imported as 'Demo').
```

This error occurred during the build time and cannot be dismissed.

Portanto, para importar um componente que não possui essa palavra-chave **default**, é preciso usar chaves {} na instrução de importação. Dentro dessas chaves, temos que escrever o que estamos exportando do arquivo **App.js**.

```
import './App.css';
// importando componentes
import { Demo } from './componentes/Demo';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <Demo />
      </header>
    </div>
  );
}

export default App;
```

## 2) Componentes Funcionais

Agora vamos criar outro componente que seria nosso componente funcional (construído a partir de uma função).

Crie um novo arquivo e nomeie-o como **DemoFunc.js** dentro do diretório **componentes**.

Agora, implemente a função **DemoFunc** usando a palavra-chave de **function** como abaixo e retorne um texto HTML para exibir no navegador.

### **DemoFunc.js**

```
// criar um componente funcional - baseado em função
function DemoFunc() {
  return(
    <div>
      <h2>Um salve do componente funcional
ReactJS!</h2>
    </div>
  )
}
```

```
export default DemoFunc;
```

(Observação: o funcionamento da palavra-chave '*default*' é o mesmo para ambos os tipos de componentes)

Agora no arquivo ***App.js***, é preciso importar este componente ***DemoFunc*** recém-criado.

Portanto, abra o arquivo *App.js* e escreva uma declaração abaixo - e também chame `<DemoFunc />` no método `return()`. Observe o código abaixo:

### ***App.js***

```
import './App.css';
// importando componentes
import Demo from './componentes/Demo';
import DemoFunc from './componentes/DemoFunc';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <h1>App Component</h1>
        <Demo />
        <DemoFunc />
      </header>
    </div>
  );
}

export default App;
```

Agora execute o aplicativo.

Um componente funcional também pode ser criado de outra forma. Usando a sintaxe ES6 (ECMAScript 6) para criar um componente funcional é possível fazer uso da *arrow function*. Crie um novo arquivo e nomeie-o como ***FuncSeta.js*** dentro do diretório ***componentes***.

Observe o código abaixo:

### ***FuncSeta.js***

```
const FuncSeta = () => {
  return (
    <div>
      <h2>Cum salve do componente funcional - arrow
function - Reactjs</h2>
    </div>
  )
}

// exportar o componente
export default FuncSeta;
```

Agora no arquivo ***App.js***, é preciso importar este componente ***FuncSeta*** recém-criado.

Portanto, abra o arquivo ***App.js*** e escreva uma declaração abaixo - e também chame `<DemoFunc />` no método `return()`. Observe o código abaixo:

### ***App.js***

```
import './App.css';
// importando componentes
import Demo from './componentes/Demo';
import DemoFunc from './componentes/DemoFunc';
import FuncSeta from './componentes/FuncSeta';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <h1>App Component</h1>
        <Demo />
        <DemoFunc />
        <FuncSeta />
      </header>
    </div>
  );
}

export default App;
```

Você pode verificar seu navegador agora e ele está mostrando o mesmo texto de antes.

## Ola Mundo ReactJS

**Um salve do componente funcional - arrow function - ReactJS!**

### Comunicação de Componentes

- O que é props?
- Usando props com componentes de classe
- Usando props com Componentes Funcionais
- O que é estado (state)?
- Estado(state) x props

### O que é props?

- Props é o nome abreviado de propriedades e são imutáveis.
- Qualquer informação que é passada de um componente para outro é conhecida como 'props'.
- As props são passados para os componentes por meio de atributos HTML.
- O componente recebe o argumento como um objeto de props.
- Para acessar essas informações, temos que usar a expressão JSX **this.props**.

### Usando props com componentes de classe

Neste passo, serão criados dois componentes (componentes de classe): um como assumirá o papel de “pai” – componente *Parent.js* - e outro assumirá o papel de filho – *Child.js*.

#### ***Parent.js***

```
import React, {Component} from 'react';

// criar a classe
class Parent extends Component{
    // chamando o método render
    render() {
        return(
            // desenhado a view
            <div>
                <h2>Eu sou o componente-pai</h2>
            </div>
        )
    }
}
// exportando o componente
```

```
export default Parent;
```

### **Child.js**

```
import React, {Component} from 'react';

// implementando o componente-filho
class Child extends Component{
    render(){
        return(
            // desenhando a view
            <div>
                <h3>Olá! sou o componente-filho!</h3>
            </div>
        )
    }
}
// exportando o arquivo
export default Child;
```

Agora, as seguintes tarefas serão realizadas:

1. o componente filho será chamado dentro do arquivo componente pai
  2. Será passado algum valor do componente pai para o componente filho
- 1) Chamando o componente **Child** dentro do componente **Parent**.  
 Para isso, será necessário importar este componente filho em **Parent.js**

```
import React, {Component} from 'react';
// importando o componente Child.js
import Child from './Child';
```

E agora apenas chame `<Child />` dentro do método **return()**.

### **Parent.js**

```
class Parent extends Component{
    // método render
    .
    render(){
        return(
            <div>
                <h1>Eu sou o componente pai</h1>
                <Child />
            </div>
        );
    }
}
```

```
}
```

```
export default Parent;
```

Navegue até o arquivo **App.js**, importe o componente **Parent.js** e declare o componente dentro de **return()**. Observer a imagem abaixo:

### **App.js**

```
import './App.css';
// importando componentes
import Demo from './componentes/Demo';
import DemoFunc from './componentes/DemoFunc';
import Parent from './componentes/Parent';

function App () {
  return (
    <div className="App">
      <header className="App-header">
        <Demo />
        <DemoFunc />
        <Parent />
      </header>
    </div>
  );
}

export default App;
```

Agora execute este aplicativo e podemos ver o conteúdo dos componentes pai e filho.

Ola Mundo ReactJS

Um salve do componente funcional - arrow function - ReactJS!

Eu sou o componente-pai

Olá! sou o componente-filho!

### **2) Passando dados do componente pai para o componente filho:**

Em **Parent.js**, será declararado uma propriedade nomeada como '**Dados**' a partir da referencia da tag '**Child**' e uma string será atribuída à ela. Implemente o código abaixo como se segue:

### **Parent.js**

```
import React, {Component} from 'react';
// importando o componente Child.js
import Child from './Child';

// criar a classe
class Parent extends Component{
    // chamando o método render
    render(){
        return(
            // desenhando a view
            <div>
                <h2>Eu sou o componente-pai</h2>
                <Child Dados = "Sou o texto enviado do
componente-pai para o componente-filho"/>
            </div>
        )
    }
}
// exportando o componente
export default Parent;
```

E agora em **Child.js**, é necessário receber o contexto do envio dos dados definido no componente-Pai. Este atributo – *Dados* – é acessado usando a declaração **this.props.Dados**. Para fazer isso, será necessário usar a sintaxe JSX e usar {} para imprimir o valor. Imprimente o código abaixo como se segue:

### **Child.js**

```
//importar o módulo de dependencia
import React, { Component } from 'react';

//criar o componente filho

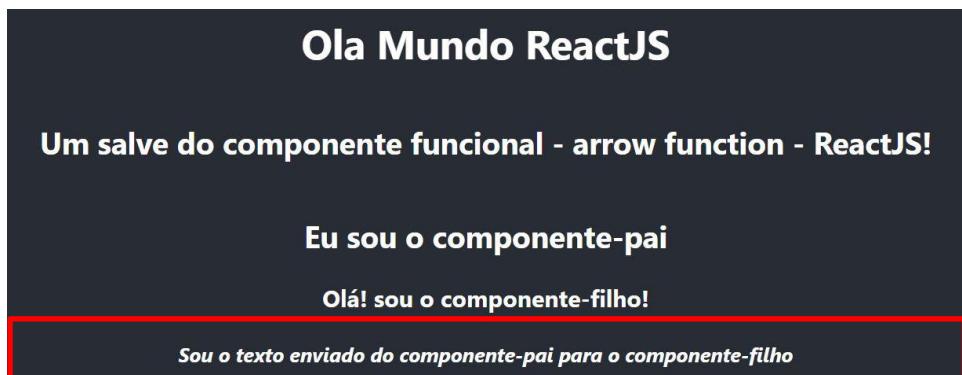
class Child extends Component{
    render(){
        return(
            <div>
                <h3>Eu sou o componente filho!</h3>
                <h3>{this.props.Dados}</h3>
            </div>
        );
    }
}
```

```

        }
    }
export default Child;

```

Verifique o navegador e observe o resultado:



## Usando props com Componentes Funcionais

Em **Parent.js**, chame o componente funcional **DemoFunc.js**.

Importe e renderize o componente **FuncSeta** como indicado abaixo:

**Parent.js**,

```

//fazer o import do modulo de dependencia Component
//do módulo react
import React, { Component } from 'react';

//importar o componente filho para dentro do componente
//pai
import Child from './Child';

//chamando o componente funcional DemoFunc
import FuncSeta from './FuncSeta';

//criar o componente de classe
//parent

class Parent extends Component{
    //criar o método render
    render(){
        return(
            <div>
                <h1>Eu sou o componente pai</h1>
                <Child Dados = "Sou o texto enviado do
componente-pai para o componente-filho"

```

```

/>           <FuncSeta DadosNovos = "Texto para o comp
onente funcional passado por uma props" />
    </div>
)
}
export default Parent;

```

Lembre-se de que no componente funcional não existe a palavra-chave **default** (associada a estrutura inicial do componente) para buscar o valor de **prop**. Portanto, aqui será passado o **props** como um argumento para o componente funcional. Dessa forma é possível usar **props.DadosNovos** para obter o valor. Observe o código abaixo e implemente como se segue:

### **FuncSeta.js**

```

const FuncSeta = (props) => {
  return (
    <div>
      <h2>Componente funcional () seta - dando um
alô!</h2>
      <h4><i>{props.novosDados}</i></h4>
    </div>
  )
}

// exportar o componente
export default FuncSeta;

```

Execute e verifique a saída:

**Um salve do componente funcional - arrow function - ReactJS!**

*Texto para DemoFunc de seu querido pai*

### **O que é estado?**

- O comportamento do aplicativo / componente em um determinado momento é definido pelo estado.

- Os dados dos componentes serão armazenados no estado do componente.
- Este estado pode ser modificado com base na ação do usuário ou outra ação.
- Quando o estado de um componente é alterado, o React irá renderizar novamente o componente para o navegador.

Observe os seguintes passos:

- criar um componente de classe e nomeá-lo como **State.js**.
- nesse caso, em primeiro lugar, simplesmente será uma variável e depois exibido seu valor. Então, o projeto terá um Botão e, clicando neste botão, o valor original da variável será alterado.

### **State.js**

```
import React, {Component} from 'react';
// criando a class component
class State extends Component{
    // criando o state
    state = {
        texto: 'Este é o valor representando o estado
original da aplicação'
    }
    render() {
        return(
            <div>
                <h3>{this.state.texto}</h3>
            </div>
        )
    }
}

export default State;
```

Como é possível observar, declaramos uma variável dentro do estado e para recuperá-la, a instrução **this.state.texto** usa o mesmo método já observado e implementado anteriormente.

Agora, importe o componente **State.js** para dentro do arquivo **App.js**. Declare-o em **return()**. Observe o código abaixo e implemente-o com o que segue:

```
import './App.css';
// importar os componentes
```

```

import Demo from './componentes/Demo';
import DemoFunc from './componentes/DemoFunc';
import Parent from './componentes/Parent';
import State from './componentes/State';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <Demo />
        <DemoFunc />
        <Parent />
        <Sample />
      </header>
    </div>
  );
}

export default App;

```

Agora execute-o e verifique no navegador, você verá que a frase “*Estado original*” será exibida no browser

**Este é o valor representando o estado original da aplicação**

Agora, será criado um manipulador de estado – para que o state original receba um novo valor – e, na sequencia, crie um botão e o associe ao evento **onClick**. Clicando no botão, o valor orginal da variavel será alterado a partir da chamado do manipulador de estado. Implemente o código abaixo com se segue:

### **State.js**

```

import React, {Component} from 'react';
// criando a class compoent
class State extends Component{
  // criando o state
  state = {
    texto: 'Este é o valor representando o estado
original da aplicação'
  }

  // criando um manipulador de estado
  manipuladorState = () => {

```

```

        // chamada da função log para observar o
comportamento do evento disparado
        console.log('Botão clicado')
        // configurar o novo state
        this.setState(
            {
                texto: 'Você acaba de alterar o estado
original da aplicação - a partir da ação do usuário'
            }
        )
    }

    render() {
        return (
            <div>
                <h3>{this.state.texto}</h3>
                <button onClick =
{this.manipuladorState}>Clique aqui</button>
            </div>
        )
    }
}

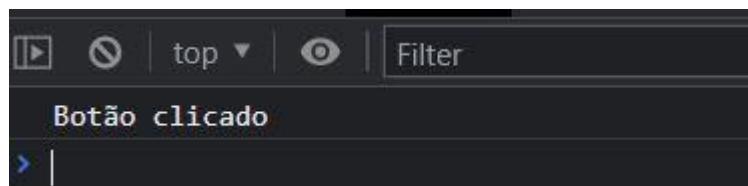
export default State;

```

Foi criado o método **manipuladorState** e, atribuído á ele uma arrow function `() => {}` e dentro desse evento, está sendo alterado o valor da variável texto. Observe o resultado:

**Você acaba de alterar o estado original da aplicação - a partir da ação do usuário**

Clique aqui



Então, quando o botão é clicado, o texto alterado.

## Estado x props

- Os props são imutáveis, ou seja, uma vez definido, não podem ser alterados, enquanto o estado é um objeto observável que é usado para

armazenar dados que podem mudar com o tempo e também para controlar o comportamento após cada mudança.

- Enquanto Props são definidos pelo componente pai, State geralmente é atualizado por manipuladores de eventos.
- Então, agora, cobrimos todos os fundamentos do ReactJs. Vimos como começar com a criação de componentes e, em seguida, como passar dados entre eles também.

## Flux - Redux

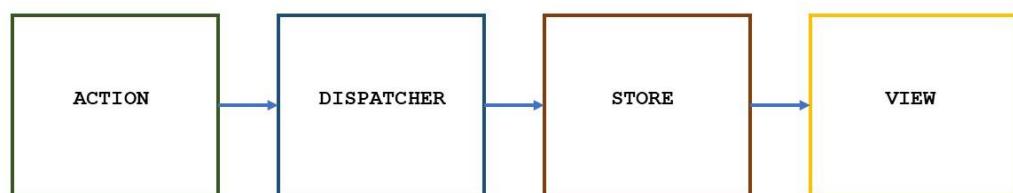
**Flux** é um conceito de programação, onde os dados são **unidirecionais**. Esses dados entram no aplicativo e fluem por ele em uma direção até que sejam renderizados na tela.

### Elementos de fluxo

Observe a definição do conceito de **flux** - a partir de suas características:

- **Action** - as ações são enviadas ao despachante para acionar o fluxo de dados.
- **Dispatcher** - Este é um hub central do aplicativo. Todos os dados são despachados e enviados para as stores.
- **Store** - Store é o local onde o estado e a lógica do aplicativo são mantidos. Cada store está mantendo um determinado estado e será atualizado quando necessário.
- **View** - A **visualização** receberá dados da store e renderizará novamente o aplicativo.

O fluxo de dados é ilustrado na imagem a seguir.



## Flux Pros

- O fluxo de dados direcional único é fácil de entender.
- O aplicativo é mais fácil de manter.
- As partes do aplicativo são desacopladas.

## Usando Flux

O objetivo é observar cada passo necessário para conectar **Redux e React**.

### Etapa 0 – Criar o novo projeto

Para criar o novo projeto, abra o prompt de comando, acesse o local – pelo prompt de comando – onde seja possível armazenar o projeto e execute a seguinte instrução:

```
C:\Users\username\Desktop\pasta-projeto>npx create-react-app flux-app
```

### Etapa 1 - Instalar Redux

Instalar o Redux através da janela do **prompt de comando**:

```
C:\Users\username\Desktop\flux-app>npm install -g react-redux
```

E, na sequencia instale o redux com o comando abaixo:

```
C:\Users\username\Desktop\flux-app>npm install -g redux
```

### Etapa 2 - Criar arquivos e pastas dentro do novo projeto

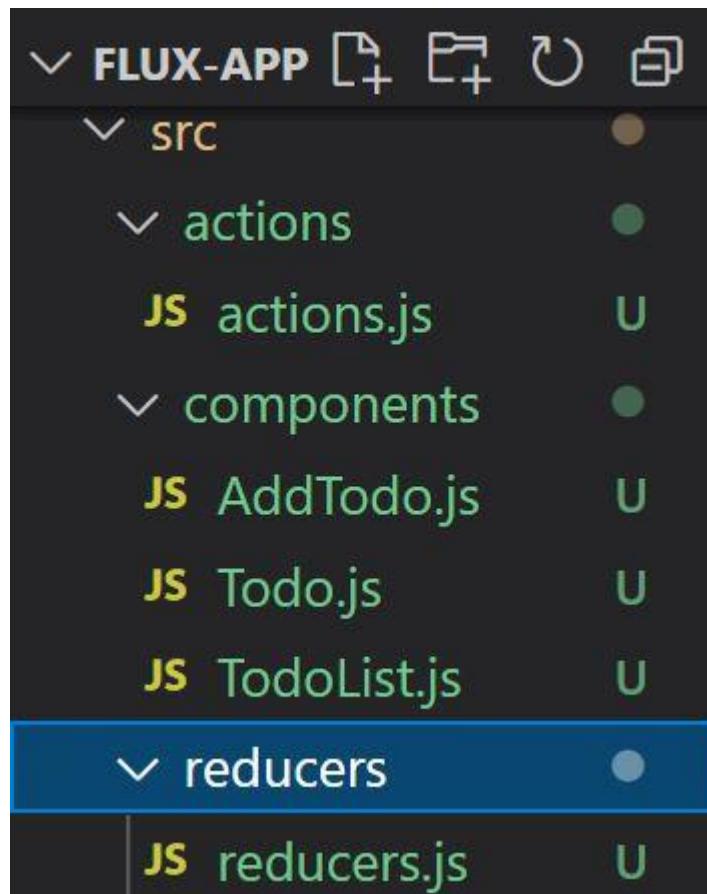
Nesta etapa, serão criados as pastas e arquivos para as **ações (actions)**, **redutores (reducers)** e **componentes (components)**. Ao final, a estrutura de pastas dispostas na aplicação será como se segue abaixo:

```
C:\Users\username\Desktop\flux-app\src>mkdir actions
```

```
C:\Users\username\Desktop\flux-app\src>mkdir components
```

```
C:\Users\username\Desktop\flux-app\src>mkdir reducers
```

```
C:\Users\username\Desktop\flux-app\src>actions/actions.js
C:\Users\username\Desktop\flux-app\src>reducers/reducers.js
C:\Users\username\Desktop\flux-app\src>components/AddTodo.js
C:\Users\username\Desktop\flux-app\src>componentes/Todo.js
C:\Users\username\Desktop\flux-app\src>components/TodoList.js
```



### Etapa 3 – Actions (Ações)

Actions, como visto anteriormente, são objetos JavaScript que usam a propriedade **type** para informar sobre os dados que devem ser enviados para a store. Estamos definindo a ação **ADD\_TODO** que será usada para adicionar um novo item à nossa lista. A função **addTodo** é um criador de ação que retorna nossa ação e define um **id** para cada item criado.

#### actions / actions.js

```
//declarar a constante que detem o nome da action
export const ADD_TODO = 'ADD_TODO'
```

```
// criar let para o incremento da lista de Todos
let nextTodoId = 0;
//criar função para exportar a action
export function addTodo(text) {
    return{
        type: ADD_TODO,
        id: nextTodoId++,
        text
    }
}
```

## Etapa 4 – Reducers(Redutores)

Embora as ações apenas açãoem mudanças no aplicativo, os **redutores** especificam essas mudanças. Estamos usando a instrução **switch** para pesquisar uma ação **ADD\_TODO**. O redutor é uma função que usa dois parâmetros ( **estado** e **ação** ) para calcular e retornar um estado atualizado.

A primeira função será usada para criar um novo item, enquanto a segunda irá “empurrar” esse item para a lista. No final, estamos usando a função auxiliar **combineReducers** , onde é possível adicionar quaisquer novos redutores que possamos usar no futuro.

### *reducers / reducers.js*

```
//fazendo os imports necessários
import {combineReducers} from 'redux'
//importar a action
import {ADD_TODO} from '../actions/actions';

//criar a função para a criação de cada um dos itens
//da lista

function todo(state, action){
    //verificar o que a action esta fazendo
    switch(action.type){
        case ADD_TODO:
            return{
                id: action.id,
                text: action.text,
            }
        default:
    }
}
```

```

        return state
    }

}

//implementar a função para criar uma lista de itens
function todos(state = [], action) {
    switch (action.type) {
        case ADD_TODO:
            return [
                ...state,
                todo(undefined, action)
            ]
    }

    default:
        return state
    }
}

//criar uma constante para receber a "junção" a partir da
//função combineReducers

const todoApp = combineReducers({
    todos
} )

//exportar o reducer
export default todoApp

```

## Etapa 5 - Armazenar

A store é um local que mantém o estado do aplicativo. É muito fácil criar uma store depois de ter reducers. Está sendo passado a propriedade da store para o elemento **provider**, que envolve o componente de root.

### **index.js**

```

import React from 'react';
//import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
//import reportWebVitals from './reportWebVitals';

```

```
// novos imports necessários para criar a store que
// armazenará o state da aplicação
import {render} from 'react-dom';
import { legacy_createStore as createStore } from 'redux'
import {Provider} from 'react-redux';

import todoApp from './reducers/reducers';

// criar a store
let store = createStore(todoApp)

let rootElement = document.getElementById('root')

render(
  <Provider store = {store}>
    <App />
  </Provider>,
  rootElement
)
```

## Etapa 6 - Componente Principal

O componente ***App.js*** é o componente principal do app. Apenas o componente principal deve “estar ciente” de um elemento redux. A parte importante a notar é a função de ***connect*** - que é usada para conectar o ***aplicativo*** – a partir do componente principal à ***store***.

Esta função – ***connect*** - leva a função de ***select*** como um argumento. A função ***select*** “pega” o estado da store e retorna os props ( ***visibleTodos*** ) que pode ser usado nos componentes. Implemente o código abaixo como se segue:

### ***App.js***

```
import './App.css';
//fazer os imports necessários
import React, {Component} from 'react';
import {connect} from 'react-redux';
import {addTodo} from './actions/actions';
import AddTodo from './components/AddTodo';
import TodoList from './components/TodoList';

class App extends Component {
  //criando o método render
  render(){
    const { dispatch, visibleTodos } = this.props
```

```

        return (
            <div className="App">
                <header className="App-header">
                    <div>
                        <AddTodo onClick = {text =>
dispatch(addTodo(text))} />
                        <TodoList todos = {visibleTodos}>/</TodoList>
                    </div>
                </header>
            </div>
        );
    }
}

//criar a função para selecionar o estado atual da store
//criada anteriormente
function select(state) {
    return{
        visibleTodos: state.todos
    }
}

//criar a conexão entre aquilo que está vindo da store
//com aquilo que será exibido a partir das props do arquivo
//principal
export default connect (select) (App);

```

## Etapa 7 - Os componentes

Esses componentes não devem precisar saber da existencia do redux.

Abra o arquivo **AddTodo.js** dentro da pasta **components e implemente o código com se segue:**

### **components / AddTodo.js**

```

import React, {Component} from 'react';

//criar o componente de classe e a estrutura para o
elemento de referencia
class AddTodo extends Component{
    constructor(props)
    {
        super(props)
        this.input = React.createRef();
    }
}

```

```

    }

//criar o manipulador de evento
inserirItem(e) {
  e.preventDefault();
  var node = this.input.current.value
  var text = node.trim()
  this.props.onAddClick(text)
}

render() {
  return(
    <div>
      <input type = 'text' ref = {this.input}/>
      <button className = "btn-insert" onClick =
{ (e) => this.inserirItem(e)}>
        Incluir Item
      </button>
    </div>
  )
}

export default AddTodo;

```

## **components / Todo.js**

```

import React, {Component} from 'react'

//criar o componente de classe
class Todo extends Component{
  render() {
    return(
      <li>
        {this.props.text}
      </li>
    )
  }
}
export default Todo;

```

Criamos o arquivo *TodoList.js* dentro da pasta components

## **components / TodoList.js**

```
import React, {Component} from 'react';
```

```
//importar o componente Todo
import Todo from './Todo';

//criar o componente de classe

class TodoList extends Component{
    render(){
        return(
            <ul className = 'list-clean'>
                {this.props.todos.map(todo =>
                    <Todo
                        key = {todo.id}
                        {...todo}
                    />
                ) }
            </ul>
        );
    }
}
export default TodoList;
```

Altere os seguintes arquivos css como indicado abaixo:

### ***App.css***

```
.App {
    text-align: center;
    list-style: none;
}

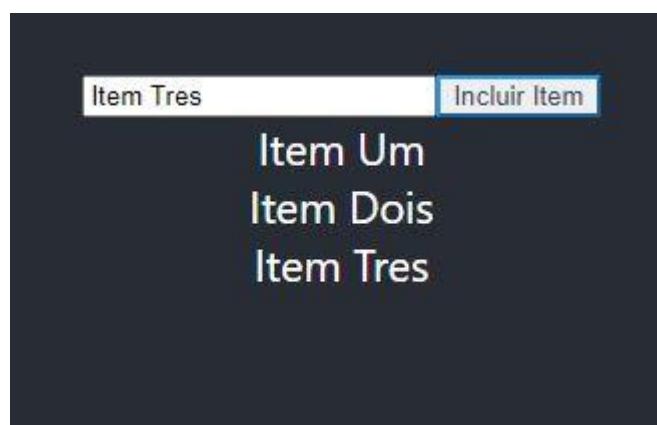
.App-header {
    background-color: #282c34;
    min-height: 100vh;
    display: flex;
    flex-direction: column;
    align-items: center;
    justify-content: center;
    font-size: calc(10px + 2vmin);
    color: white;
}
```

### ***index.css***

```
body {
    margin: 0;
```

```
    font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', 'Roboto', 'Oxygen',  
    'Ubuntu', 'Cantarell', 'Fira Sans', 'Droid Sans',  
    'Helvetica Neue',  
    sans-serif;  
    -webkit-font-smoothing: antialiased;  
    -moz-osx-font-smoothing: grayscale;  
    list-style: none;  
}  
  
code {  
    font-family: source-code-pro, Menlo, Monaco, Consolas,  
    'Courier New',  
    monospace;  
}  
.list-clean{  
    list-style: none;  
    margin: 0;  
    padding: 0;  
}  
  
.btn-insert {  
    border-color: #2196F3;  
    color: rgb(53, 72, 92);  
}  
  
.btn-insert:hover {  
    background: #2196F3;  
    color: rgb(41, 39, 39);  
}
```

Quando executar o aplicativo, será possível adicionar itens à lista.  
Observe o resultado:



## TypeScript

### Visão geral

O JavaScript foi adotado como uma linguagem de programação para o chamado cliente-side (lado do cliente) – ou, ainda – front-end. O desenvolvimento do Node.js também marcou o JavaScript como uma tecnologia emergente do lado do servidor (server-side) – ou, ainda – back-end middleware. No entanto, à medida que o código JavaScript cresceu, ficou relativamente mais “confuso”. Essa percepção dificulta a manutenção e a reutilização do código. Além disso, sua “falha” em adotar os recursos de Orientação a objetos (OOP), forte verificação de tipagem de dado e verificação de erro em tempo de compilação impede que o JavaScript seja bem-sucedido, naquilo que se conheço como nível corporativo, como uma tecnologia completa do servidor (servr-side). O **TypeScript** foi apresentado para preencher essa lacuna.

### O que é o TypeScript?

Por definição, "TypeScript é **JavaScript** para desenvolvimento de aplicações / ou aplicativos em escala".

É uma linguagem compilada, fortemente tipada e orientada a objetos. Foi projetada por **Anders Hejlsberg** (designer do C #) na Microsoft. O TypeScript é uma linguagem e um conjunto de ferramentas que podemos utilizar para trabalhar com projetos web. É, também, considerada um superconjunto de JavaScript compilado em JavaScript. Em outras palavras, o TypeScript é JavaScript, acoplada de recursos adicionais.

### Recursos do TypeScript

**TypeScript é apenas JavaScript**. Começa com JavaScript e termina com JavaScript. Adota os blocos de construção básicos do seu programa a partir do JavaScript. Portanto, você só precisa saber JavaScript para usar o TypeScript. Todo o código TypeScript é convertido em seu equivalente em JavaScript para fins de execução.

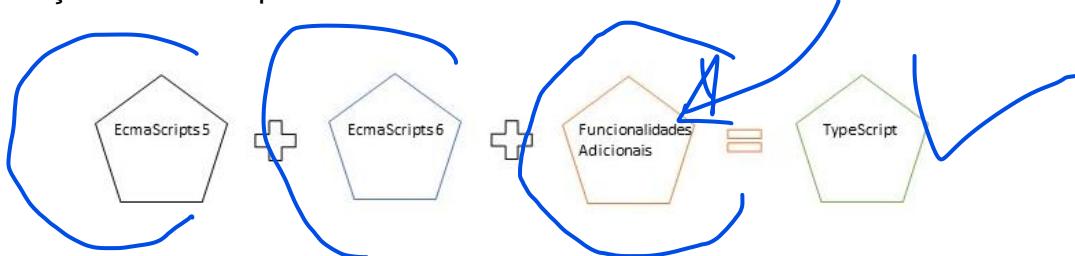
O TypeScript suporta outras bibliotecas JS. É compilado, pode ser consumido a partir de qualquer código JavaScript. O JavaScript gerado pelo TypeScript pode reutilizar todas as estruturas, ferramentas e bibliotecas JS existentes.

**JavaScript** é **TypeScript**. Isso significa que qualquer arquivo **.js** válido pode ser renomeado para **.ts** e compilado com outros arquivos TypeScript.

**TypeScript é portável**. O TypeScript é portável em navegadores, dispositivos e sistemas operacionais. Pode ser executado em qualquer ambiente em que o JavaScript seja executado. Diferentemente de suas contrapartes, o TypeScript não precisa de uma VM dedicada ou de um ambiente de tempo de execução específico para executar.

### TypeScript e ECMAScript

A especificação ECMAScript é uma especificação padronizada de uma linguagem de script. Existem seis edições do ECMA-262 publicadas. A versão 6 da norma possui o codinome "Harmony". O TypeScript está alinhado com a especificação ECMAScript6.



TypeScript adota seus recursos básicos de linguagem da especificação ECMAScript5, ou seja, a especificação oficial para JavaScript. Os recursos da linguagem TypeScript, como módulos e orientação baseada em classe, estão alinhados com a especificação EcmaScript 6. Além disso, o TypeScript também inclui recursos como genéricos e anotações de tipo que não fazem parte da especificação EcmaScript6.

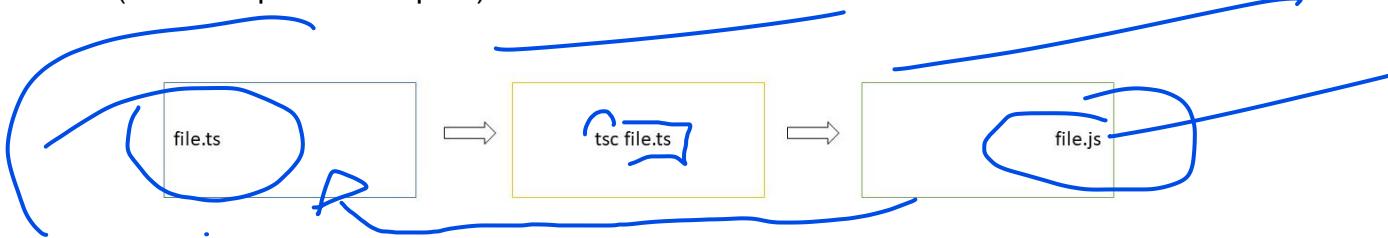
### Componentes do TypeScript

No coração, o TypeScript possui os três componentes a seguir -

- **Idioma** - É composto pela sintaxe, pelas palavras-chave e pelas anotações de tipo.
- **O compilador TypeScript** - O compilador TypeScript (tsc) converte as instruções escritas em TypeScript em seu equivalente em JavaScript.
- **O TypeScript Language Service - serviço de linguagem TypeScript** - O "Serviço de linguagem" expõe uma camada adicional ao redor do pipeline do compilador principal que são aplicativos semelhantes a editor. O serviço de idioma suporta o conjunto comum de operações típicas de um editor, como conclusão de instruções, ajuda de assinatura, formatação e estrutura de códigos, colorização, etc.

### O compilador TypeScript

O compilador TypeScript é ele próprio um arquivo **.ts** compilado no arquivo JavaScript (.js). O TSC (TypeScript Compiler) é um compilador fonte a fonte (transcompiler / transpiler).



O TSC gera uma versão JavaScript do arquivo **.ts**, que é passado para ele. Em outras palavras, o TSC produz um código-fonte JavaScript equivalente - sempre a partir do arquivo Typescript fornecido, como uma entrada, para ele `-tsc`. Este processo é denominado como **transpilação**.

No entanto, o compilador rejeita qualquer arquivo JavaScript bruto que seja passado. Para o compilador podemos passar apenas arquivos **ts** ou **.d.ts**.

Para instalar o *typescript* basta inserir o comando:

```
npm install typescript
```

Também é possível gerar um arquivo `tsconfig.json` – caso seja necessário fazer algum tipo de configuração adicional. Para gerar este arquivo, basta inserir a instrução abaixo em seu prompt de comando (no caminho da pasta onde se encontram ou onde salvar os projetos *typescript*):

```
tsc -init
```

Se tudo correr bem, o resultado do arquivo `tsconfig.json` é este indicado abaixo:



## Sintaxe básica

Sintaxe define um conjunto de regras para escrever programas. Toda especificação de idioma define sua própria sintaxe. Um programa TypeScript é composto por -

- Módulos
- Funções
- Variáveis
- Declarações e expressões
- Comentários

## Compilar e executar um programa TypeScript

Vamos ver como compilar e executar um programa TypeScript usando o Visual Studio Code. Siga as etapas abaixo:

**Etapa 1** - Salve o arquivo com extensão .ts. Vamos salvar o arquivo como teste.ts. O editor de código marca erros, caso eles existam, enquanto você o salva.

**Etapa 2** - Clique com o botão direito do mouse no arquivo TypeScript na opção Arquivos de Trabalho no Painel de Exploração do VS Code. Selecione Abrir na opção Prompt de Comando (Command Prompt) ou Abrir no Terminal (Open Terminal).

**Etapa 3** - Para compilar o arquivo, use o seguinte comando na janela do terminal.

tsc teste.ts

**Etapa 4** - O arquivo é compilado para teste.js. Para executar o programa – agora, convertido no formato .js –, digite a seguinte instrução no terminal que você abriu:

node teste.js

Agora, implemente o código abaixo como se segue:

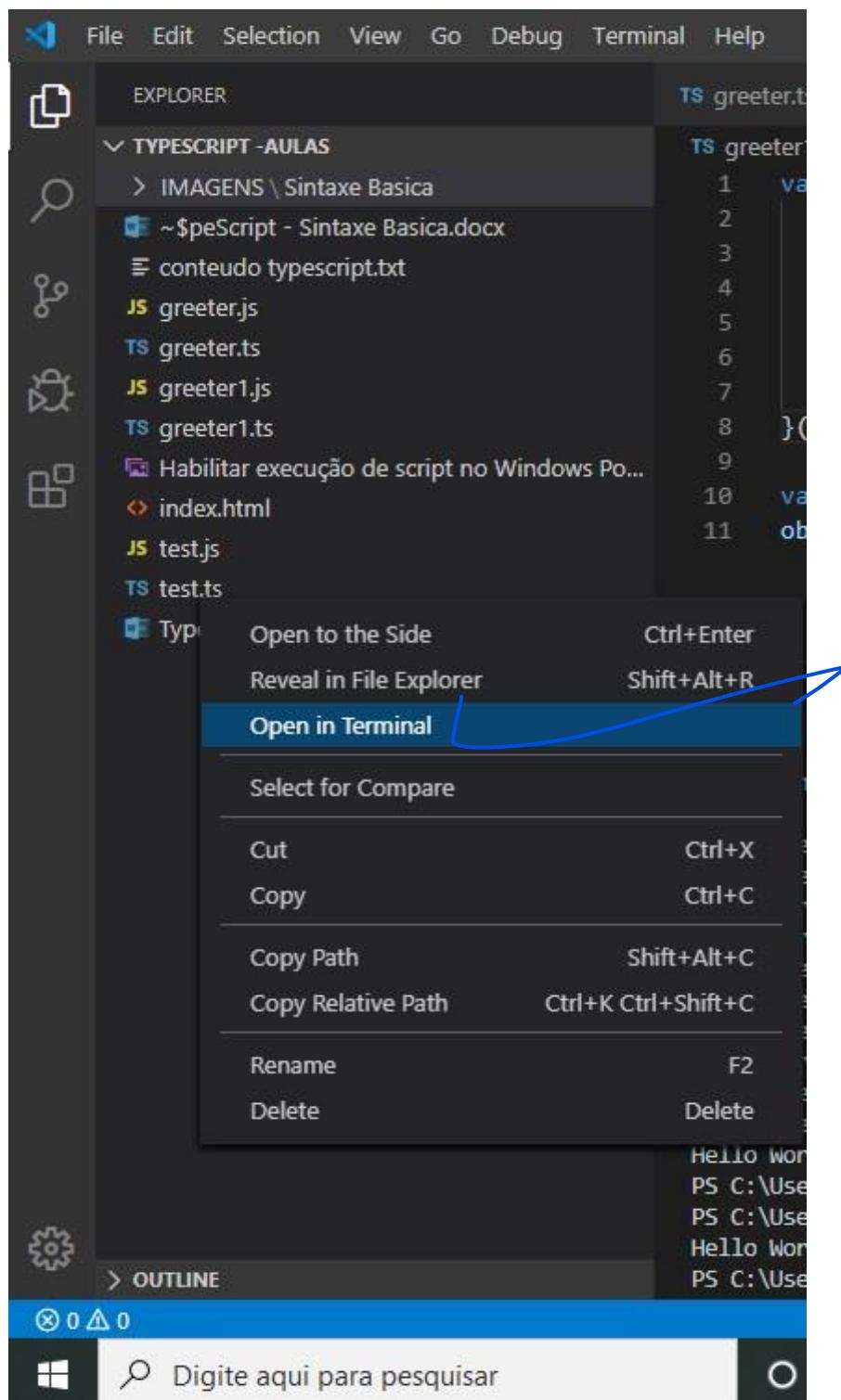
### Seu primeiro código TypeScript

```
var mensagem: string = "Hello Mundo"
console.log(mensagem)
```

Ao compilar, ele gera o seguinte código JavaScript.

```
var mensagem = "Hello Mundo";
console.log(mensagem);
```

- A linha 1 declara uma variável pela mensagem de nome. Variáveis são um mecanismo para armazenar valores em um programa.
- A linha 2 imprime o valor da variável no prompt. Aqui, console refere-se à janela do terminal. O log da função () é usado para exibir o texto na tela.

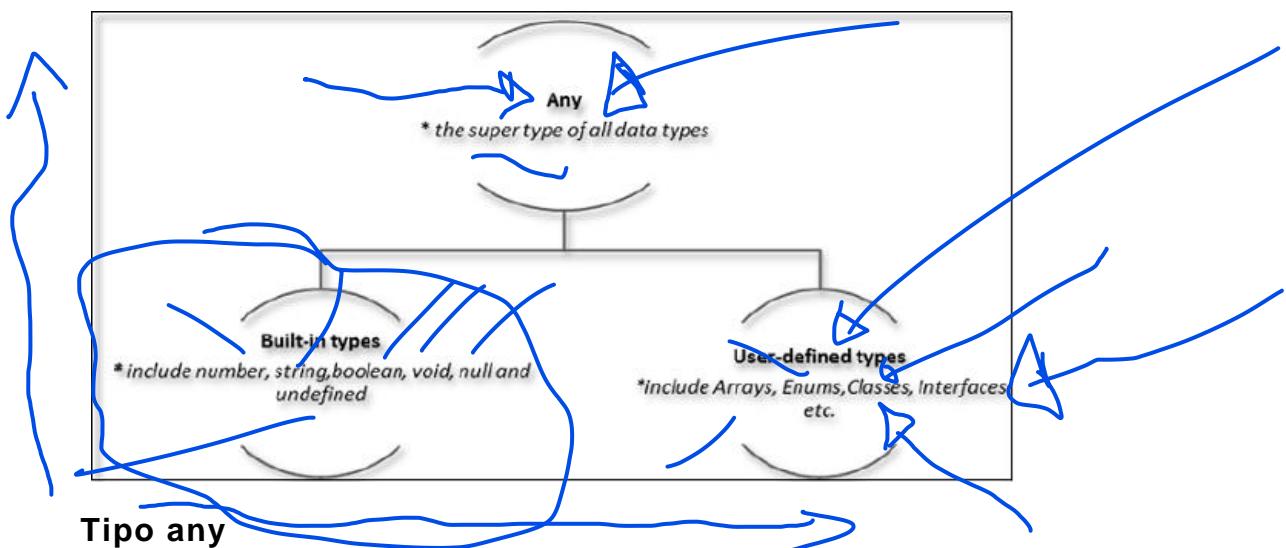


## Tipos

O sistema de tipos representa os diferentes tipos de valores suportados pela linguagem *TypeScript*. O sistema de tipos verifica a validade dos valores fornecidos, antes de serem armazenados ou manipulados pelo programa. Isso garante que o código se comporte conforme o esperado. O *Type System*

também permite dicas de código mais avançadas e documentação automatizada.

O *TypeScript* fornece tipos de dados como parte de seu sistema de tipos opcional. A classificação do tipo de dados é a seguinte:



O *any data type* é o supertipo de todos os tipos no *TypeScript*. Denota um tipo dinâmico. Usar o tipo *any* é equivalente a desativar a verificação de tipo para uma variável.

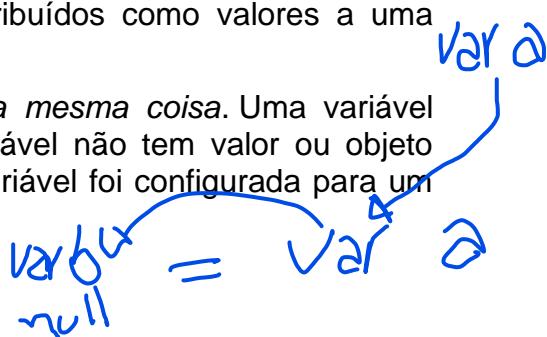
*var c: undefined*

### Null (Null) e Undefined (undefined) – São iguais?

Os tipos de dados **null** e **undefined** são frequentemente uma fonte de confusão. O **null** e **undefined** não podem ser usados para referenciar o tipo de dado de uma variável. Eles só podem ser atribuídos como valores a uma variável.

No entanto, **null** e **undefined** não são a mesma coisa. Uma variável inicializada com **undefined** significa que a variável não tem valor ou objeto atribuído a ela, enquanto **null** significa que a variável foi configurada para um objeto cujo valor é **undefined**.

### Tipos definidos pelo usuário



Tipos definidos pelo usuário incluem enumerations (enums), classes, interfaces, matrizes e tuplas.

## Variáveis

Uma variável, por definição, é "um espaço nomeado na memória" que armazena valores. Em outras palavras, ele atua como um contêiner para valores em um programa. As variáveis *TypeScript* devem seguir as regras de nomenclatura *JavaScript*:

- Os nomes de variáveis podem conter alfabetos e dígitos numéricos.
- Eles não podem conter espaços e caracteres especiais, exceto o sublinhado (\_) e o sinal de dólar (\$).



- Os nomes de variáveis não podem começar com um dígito.

Uma variável deve ser declarada antes de ser usada. Use a palavra-chave **var** para declarar variáveis.

### Declaração variável no TypeScript

A sintaxe do tipo para declarar uma variável no TypeScript é incluir dois pontos (:) após o nome da variável, seguido por seu tipo. Assim como no JavaScript, usamos a palavra-chave **var** para declarar uma variável.

Ao declarar uma variável, você tem quatro opções:

- Declare seu tipo e valor em uma instrução.

**var**

[**identifier**]

**:**

[**type-annotation**]

**=**

**value**

**;**

- Declara o tipo, mas sem valor. Nesse caso, a variável será definida como `undefined`.

**var**

[**identifier**]

**:**

[**type-annotation**]

**;**

- Declara seu valor, mas nenhum tipo. O tipo de variável será definido para o tipo de dado do valor atribuído.

**var**

[**identifier**]

**=**

**value**

**;**

- Não declara nem o valor nem o tipo. Nesse caso, o tipo de dado da variável será `any` e será inicializado como indefinido.

**var**

[**identifier**]

**;**

A tabela a seguir ilustra a sintaxe válida para a declaração de variável, conforme discutido acima:

### Criando variáveis com diferentes definições de tipo

Observe o código abaixo:

```
var nome:string = "Chespirito"
var pontuacao:number = 110
var pontuacao2:number = 11.35
var soma = pontuacao + pontuacao2
console.log("nome: "+nome)
console.log("primeira pontuação: "+pontuacao)
console.log("segunda pontuação: "+pontuacao2)
console.log("soma das pontuações: "+soma)
```

Ao compilar, ele gera o seguinte código JavaScript.

```
var nome = "Chespirito";
var pontuacao = 110;
var pontuacao2 = 11.35;
var soma = pontuacao + pontuacao2;
console.log("nome: " + nome);
console.log("primeira pontuação: " + pontuacao);
console.log("segunda pontuação: " + pontuacao2);
console.log("soma das pontuações: " + soma);
```

A saída do programa acima é dada abaixo -

```
nome: Chespirito
primeira pontuação: 110
segunda pontuação: 11.35
soma das pontuações: 121.35
```

## Inferência de tipo

Dado o fato de que o TypeScript é fortemente tipado, esse recurso é opcional. O TypeScript também incentiva a digitação dinâmica de variáveis. Isso significa que, o TypeScript incentiva a declaração de uma variável sem um tipo. Nesses casos, o compilador determinará o tipo da variável com base no valor atribuído a ela. O TypeScript encontrará o primeiro uso da variável no código, determinará o tipo para o qual ela foi definida inicialmente e assumirá o mesmo tipo para essa variável no restante do seu bloco de código.

O mesmo é explicado no seguinte *snippet* de código:

### Exemplo

```
var umNumero = 2; //infeirndo o tipo do dado da variavel
console.log("valor da variavel umNumero é :" + umNumero);
```

No *snippet* de código acima:

- O código declara uma variável e define seu valor como 2. Observe que a declaração da variável não especifica o tipo de dado. Portanto, o programa usa *typing inferido* para determinar o tipo de dados da variável, ou seja, atribui o tipo do primeiro valor ao qual a variável está configurada. Nesse caso, **umNumero** é definido como *type number* (*tipo number*).
- Quando o código tenta definir o valor da variável como string, o compilador gera um erro, já que o tipo da variável já está definido como número.

O compilador TypeScript gerará erros, se tentarmos atribuir um valor a uma variável que não é do mesmo tipo. Portanto, o TypeScript segue a Strong Typing. A sintaxe de Strong Typing garante que os tipos especificados em ambos os lados do operador de atribuição (=) sejam os mesmos. É por isso que o código a seguir resultará em um erro de compilação -

```
var num:number = "typescript"
```

*a instrução acima retornará um erro de compilação*

## Declaração *let*

Para resolver problemas com declarações ***var***, o ES6 introduziu dois novos tipos de declarações de variáveis no JavaScript, usando as palavras-chave ***let*** e ***const***. O *TypeScript*, sendo um *superconjunto* de *JavaScript*, também suporta esses novos tipos de declarações de variáveis. Observe o código abaixo:

```
var nomeFuncionario = "Miguelito";  
  
ou  
  
let nomeFuncionario = "Miguelito";
```

As declarações *let* seguem a mesma sintaxe que as declarações *var*. Diferente das variáveis declaradas com *var*, as variáveis declaradas com *let* possuem um escopo de bloco. Isso significa que o escopo das variáveis *let* é limitado ao seu bloco de contenção. Considere o seguinte exemplo.

### Exemplo

```
let numeroUm:number = 1;  
  
function declarandoLet() {  
    let numeroDois:number = 2;  
  
    if (numeroUm > numeroDois) {  
        let numeroTres: number = 3;  
        numeroTres++;  
        console.log(numeroTres);  
    }  
  
    while(numeroUm < numeroDois) {  
        let numeroQuatro: number = 4;  
        numeroUm++;  
        console.log(numeroUm);  
    }  
  
    console.log(numeroUm); //OK  
    console.log(numeroDois); //OK  
    //console.log(numeroTres); //Compiler Error: Cannot find name 'numeroTres'  
    //console.log(numeroQuatro); //Compiler Error: Cannot find name 'numeroQuatro'  
}
```

```
declarandoLet();
```

No exemplo acima, todas as variáveis são declaradas usando *let*. A variável *numeroTres* é declarada no bloco *if*, portanto, seu escopo é limitado ao bloco *if* e não pode ser acessado fora do bloco. Da mesma maneira, *numeroQuatro* é declarado no bloco *while* para que não possa ser acessado fora do bloco. Assim, ao acessar *numeroTres* e *numeroQuatro* fora dos respectivos blocos um erro será exibido no compilador.

O mesmo exemplo com a declaração *var* é compilado sem erro.

### **Exemplo: escopo de variáveis usando var**

```
// criando e declarando variaveis com a palavra let
var numeroUm: number = 1

// criando a função para trabalhar com let
function declarandoLet() {
    var numeroDois: number = 2
    var numeroTres: number = 3
    // bloco if
    if(numeroUm < numeroDois) {
        numeroTres++

    }
    var numeroQuatro:number = 4
    // bloco while
    while(numeroUm < numeroDois) {
        numeroQuatro++
        numeroUm++
    }

    // chamando algumas propriedades
    console.log(numeroUm)
    console.log(numeroDois)
    console.log(numeroTres)
    console.log(numeroQuatro)
}
declarandoLet()
```

### **Vantagens de usar let em relação a var**

As variáveis com escopo no bloco não podem ser lidas ou gravadas antes de serem declaradas.

No exemplo acima, o compilador *TypeScript* emitirá um erro se usarmos variáveis antes de declará-las usando *let*, enquanto que não ocorrerá um erro ao usar variáveis antes de declará-las usando *var*.

### **Permitir que variáveis não possam ser declaradas novamente**

O compilador *TypeScript* emitirá um erro quando variáveis com o mesmo nome (com distinção entre maiúsculas e minúsculas) são declaradas várias vezes no mesmo bloco usando *let*.

### **Exemplo: Várias variáveis com o mesmo nome**

```
let numero:number = 1; // OK
let Numero:number = 2; // OK

let numero:number = 5;// Compiler Error: Cannot redeclare
block-scoped variable 'numero'
let Numero:number = 6;// Compiler Error: Cannot redeclare
block-scoped variable 'Numero'
```

No exemplo acima, o compilador *TypeScript* trata os nomes das variáveis como distinção entre maiúsculas e minúsculas; No entanto, ocorrerá um erro para as variáveis com o mesmo nome e caso.

### **Declaração Const**

As variáveis podem ser declaradas usando *const* semelhante às declarações *var* ou *let*. A *const* torna uma variável uma constante em que seu valor não pode ser alterado. As variáveis *const* possuem as mesmas regras de escopo que as variáveis *let*.

### **Exemplo: Variável Const**

```
const numeroConst:number = 100;
numeroConst = 200; //Compiler Error: Cannot assign to 'numeroConst' because it is a constant or read-only property
```

As variáveis `const` devem ser declaradas e inicializadas em uma única instrução. Declaração e inicialização separadas não são suportadas; esse tipo de variável permite alterar as subpropriedades de um objeto, mas não a estrutura do objeto. Observe o código abaixo:

### **Exemplo: `const Object`**

```
// implementando a primeira constante
const idParticipantes = {
    participanteA: 1,
    participanteB: 2,
    participanteC: 3,
    participanteD: 4
}
console.log(idParticipantes)
// acessar uma propriedade do objeto const
idParticipantes.participanteC = 167

// exibindo o valor alterado da propriedade do objeto
console.log(idParticipantes)

console.log(idParticipantes.participanteC)
```

Mesmo se a tentativa for de alteração da estrutura do objeto, o compilador apontará esse erro. Observe o código abaixo:

```
const idParticipantes = {
    participanteA : 1,
    participanteB : 2,
    participanteC : 3,
    participanteD : 4
};
idParticipantes.participanteC = 105; // OK

idParticipantes = {           //Compiler Error: Cannot assign to
playerCodes because it is a constant or read-only
    participanteA : 50,
    participanteB : 10,
    participanteC : 13,
    participanteD : 20
};
```

Isso não deve ser confundido com a ideia de que os valores a que se referem são imutáveis. Observe o próximo bloco de código:

```

let vidaGatos = 9

// criando o objeto literal
const dadosGato = {
    nome: 'Mila Burns',
    qtdeVidas: vidaGatos
}
// exibir o objeto
console.log(dadosGato)

// aplicando modificações nas propriedades do objeto
dadosGato.nome = 'Frajola Jenkins'
dadosGato.qtdeVidas = 67

// exibindo as alterações nas propriedades do objeto
console.log('Quantidade de vidas de ', dadosGato.nome, 'é igual a ', dadosGato.qtdeVidas)

// exibindo o objeto
console.log(dadosGato)

```

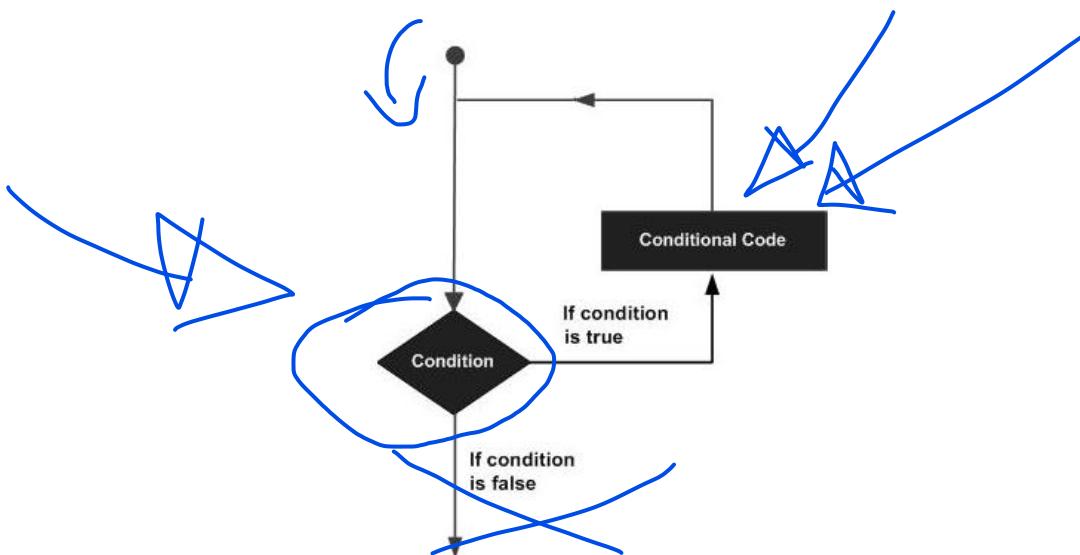
A menos que se tome medidas específicas para evitá-lo, o estado interno de uma *const* ainda pode ser modificado. Felizmente, o *TypeScript* permite especificar os membros de um objeto *readonly*.

## Loops

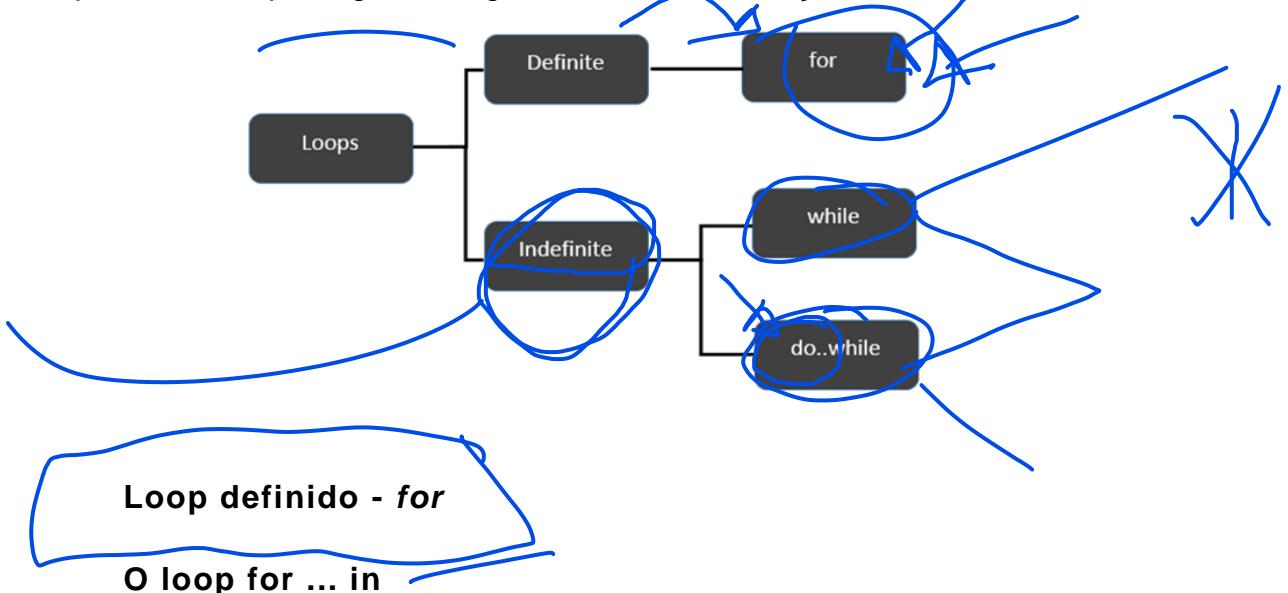
Quando o cenário necessita da aplicação de loops (laços) significa que, em geral, as instruções são executadas sequencialmente: A primeira instrução em uma função é executada primeiro, seguida pela segunda e assim por diante.

Todas as linguagens de programação fornecem várias estruturas de controle que permitem caminhos de execução mais complexos ou menos complexos.

Uma declaração de loop nos permite executar uma declaração ou grupo de declarações várias vezes. Dada a seguir é a forma geral de uma instrução de loop na maioria das linguagens de programação.



O *TypeScript* fornece diferentes tipos de loops para lidar com os requisitos de loop. A figura a seguir ilustra a classificação dos loops:



Outra variação do de ciclo é o loop *for ... in*. O loop *for... in* pode ser usado para iterar sobre um conjunto de valores, como no caso de um array (matriz) ou de uma tupla. A sintaxe para o mesmo é fornecida abaixo -

O loop *for ... in* é usado para iterar através de uma lista ou coleção de valores. O tipo de dados *val* aqui deve ser string ou qualquer. A sintaxe do loop ***for..in*** é a seguinte:

### Sintaxe

```
for (var x in colecao) {
    // declaracoes
}
```

Observe o exemplo a seguir:

### Exemplo

```
// vamos criar nosso primeiro loop for
var y: any // essa é nossa variável iteradora

// criando a coleção de dados
var z: any = 'a b c'

// criando o loop
for(y in z){
    console.log(z[y])
}
```

Ao compilar, ele gera o seguinte código JavaScript:

```
// vamos criar nosso primeiro loop for
var y; // essa é variável iteradora
// criando a coleção de dados
var z = 'a b c';
// criando o loop
for (y in z) {
    console.log(z[y]);
}
```

A execução produzirá a seguinte saída:

a  
b  
c

### **Loop for ... of**

O loop *for ... of* retorna elementos de uma coleção, por exemplo, array, lista ou tupla e, portanto, não é necessário usar o loop for tradicional.

Veja o exemplo abaixo:

```
// este é o loop for of
var y: any
//criando a coleção de dados
let umArray: Array<number> = [380, 1067, 2087, 4780, 6750]
//criando o loop
for(y of umArray){
    console.log(y)
}
```

O código acima exibirá a seguinte saída:

**380**

**1067**

**2087**

**4780**

**6750**

### **Loop indefinido - while**

Um loop indefinido é usado quando o número de iterações em um loop é indeterminado ou desconhecido.

Loops indefinidos podem ser implementados usando:

Loops e descrição
<p><b>Loop while</b></p> <p>O loop <i>while</i> executa as instruções sempre que a condição especificada é avaliada como verdadeira.</p>
<p><b>do....while</b></p> <p>O loop <i>do ... while</i> é semelhante ao loop <i>while</i>, exceto que o loop <i>do ... while</i> não avalia a condição pela primeira vez que o loop é executado.</p>

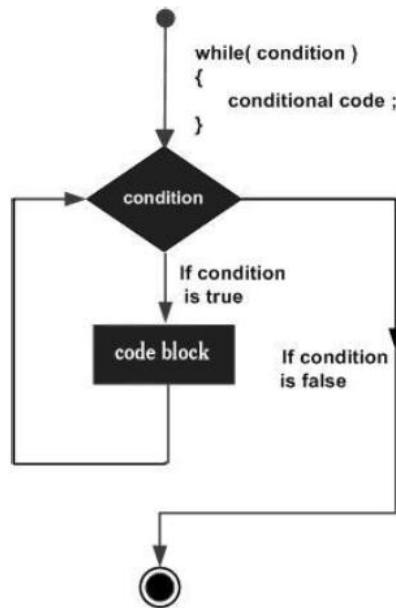
### **Loop While**

O Loop *while* executa as instruções cada vez que a condição especificada for avaliada como *True*. Em outras palavras, o loop avalia a condição antes da execução do bloco de código.

### **Sintaxe**

```
while(expressão_teste) {
    // declaracoes
}
```

### **Diagrama de fluxo**



### Exemplo: loop While

Observe o código abaixo. A premissa é a mesma do loop for – calculo do factorial – agora, executado com o *loop while*.

```

console.log("O factorial é uma operação muito importante para o estudo e desenvolvimento da análise combinatória")
console.log("Conhecemos como factorial de um número natural a multiplicação desse número por seus antecessores com exceção do zero")
// primeiro loop while
// vamos estabelecer as variaveis
var numero: number = 10
var fatorial: number = 1

// vamos estabelecer o loop while
while(numero >= 1){
    fatorial = fatorial * numero
    numero--
}

console.log('O valor do factorial de 10 até 1 é : ', fatorial)
  
```

Ao compilar, a execução gera o seguinte código JavaScript:

```

console.log("O factorial é uma operação muito importante para o estudo e desenvolvimento da análise combinatória");
  
```

```

console.log("Conhecemos como factorial de um número natural
a multiplicação desse número por seus antecessores com exce-
ção do zero");
// primeiro loop while
// vamos estabelecer as variaveis
var numero: number = 10
var factorial: number = 1

// vamos estabelecer o loop while
while(numero >= 1) {
    factorial = factorial * numero
    numero--
}

}
console.log('O valor do factorial de 10 até 1 é : ', factorial)

```

Produz a seguinte saída:

O valor do factorial é da variável umNúmero é: 3628800



### do... while loop

O *loop do ... while* é semelhante ao *loop while*, exceto que o *loop do ... while* não avalia a condição pela primeira vez que o loop é executado. No entanto, a condição é avaliada para as iterações subsequentes. Em outras palavras, o bloco de código será executado pelo menos uma vez em um *loop do ... while*.

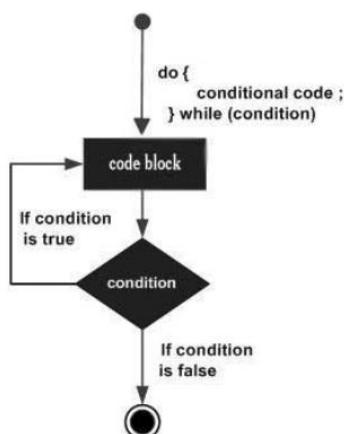
### Sintaxe

```

do {
    //declarações
} while(expressão_teste)

```

### Fluxograma



### Exemplo: do... while

```
// implementando o loop do...while
var novoNum: number = 13
console.log('Valores encontrados durante a iteração do loop
do..while')

// estabelecendo o loop
do{
    console.log(novoNum)
    novoNum--
}while(novoNum >= 0)
```

Ao compilar, ele gera o seguinte código JavaScript:

```
// implementando o loop do...while
var novoNum = 13;
console.log('Valores encontrados durante a iteração do loop
do..while');
// estabelecendo o loop
do {
    console.log(novoNum);
    novoNum--;
} while (novoNum >= 0);
```

O exemplo imprime números de 0 a 13 na ordem inversa.

Valores encontrados durante a iteração do loop  
do..while

13  
12  
11  
10  
9  
8  
7  
6  
5  
4  
3  
2  
1  
0

## Classes

### TypeScript e orientação a objeto

TypeScript é JavaScript orientado a objetos. A Orientação a Objetos é um paradigma de desenvolvimento de software que segue a modelagem do mundo real. A Orientação a Objetos considera um programa como uma coleção de objetos que se comunicam por meio de um mecanismo chamado métodos. O TypeScript também suporta esses componentes orientados a objetos.

- Objeto - Um objeto é uma representação em tempo real de qualquer entidade. De acordo com Grady Brooch, todo objeto deve ter três recursos -
  - Estado - descrito pelos atributos de um objeto
  - Comportamento - descreve como o objeto agirá
  - Identidade - um valor único que distingue um objeto de um conjunto de objetos semelhantes.
- Classe - Uma classe em termos de POO é um plano para criar objetos. Uma classe encapsula dados para o objeto.
- Método - Métodos facilitam a comunicação entre objetos.

### Criando uma classe

Para criar uma classe em Typescript é necessário iniciar com o uso da palavra-reservada `class` – como na maioria das linguagens de programação. Observe a sintaxe abaixo:

#### Sintaxe

```
class class_name {
    //class scope
}
```

#### Exemplo: TypeScript e orientação a objeto

```
//vamos criar nosso primeira class typesccript

class Saudacao{
    saudacao():void{
        console.log("Olá mundo a partir do POO TS!")
    }
}

//vamos criar nosso objeto a partir da classe Saudacao
//assim, podemos fazer uso do método saudação()
```

```
var obj = new Saudacao();
obj.saudacao();
```

O exemplo acima define uma classe `Saudacao`. A classe possui um método `saudacao()`. O método imprime a sequência "Hello World" no terminal. A palavra-chave `new` cria um objeto da classe (obj). O objeto chama o método `saudacao()`.

Ao compilar, ele gera o seguinte código JavaScript.

```
//vamos criar nosso primeira class typescript
var Saudacao = /** @class */ (function () {
    function Saudacao() {
    }
    Saudacao.prototype.saudacao = function () {
        console.log("Ola mundo a partir do POO TS!");
    };
    return Saudacao;
}());
//vamos criar nosso objeto a partir da classe Saudacao
//assim, podemos fazer uso do método saudacao()
var obj = new Saudacao();
obj.saudacao();
```

A saída do programa acima é dada abaixo:

**Ola mundo a partir do POO TS!**

*TypeScript* suporta recursos de programação orientada a objetos, como *classes*, *interfaces*, etc. Uma classe em termos de OOP é um modelo para a criação de objetos. Uma classe encapsula dados para o objeto. O *TypeScript* oferece suporte embutido para esse conceito chamado classe. O *JavaScript* ES5 ou anterior não suporta classes. O *TypeScript* obtém esse recurso do ES6.

Uma definição de classe pode incluir o seguinte:

- **Campos** (fields)- Um campo é qualquer variável declarada em uma classe. Campos representam dados pertencentes a objetos
- **Construtores** (constructor) - Responsável por alocar memória para os objetos da classe
- **Funções** (function) - Funções representam ações que um objeto pode executar. Eles também são chamados de métodos

Esses componentes juntos são denominados como membros de dados da classe.

## O framework Angular

**Angular** é um *framework* para desenvolver aplicações em diversas plataformas, mantido e desenvolvido pela **Google**.

Ele é uma reescrita completa do antigo *angularjs* e foi escrito em **TypeScript**.

Ele vem com um **conjunto de bibliotecas** poderosas que podemos importar, possibilitando construir aplicações com uma qualidade e produtividade surpreendente.

### A arquitetura do framework Angular

A arquitetura do **Angular** permite organizar a aplicação por módulos através dos *NgModules*, que fornecem um contexto para os componentes serem compilados.

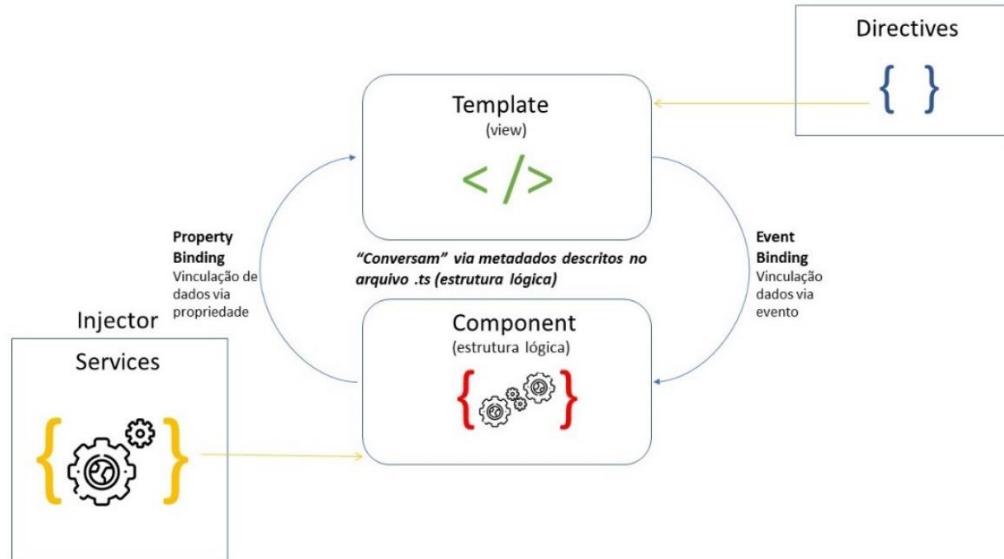
Uma aplicação sempre tem ao menos um **módulo raiz (app module)** que habilita a inicialização e, normalmente, possui outros módulos de bibliotecas.

Os componentes deliberam as visualizações — que são conglomerados de elementos e funcionalidades de tela — que o Angular modifica de acordo com a lógica e os dados da aplicação.

Esses componentes usam serviços que fornecem funcionalidades específicas e que são indiretamente relacionadas a essas visualizações.

Os **service providers** podem ser injetados nos componentes como dependências, tornando seu código modular e reutilizável.

Services e *components* são simples classes com *decorators*, que definem o tipo da estrutura lógica implementada e fornecem metadados para informar o Angular como usá-los. Observe o diagrama abaixo:



### A instrução @NgModules

Tem como objetivo declarar e agrupar tudo que criamos no Angular. Existem duas estruturas principais, que são: *declarations* e o *providers*.

**Declarations** é onde declaramos os itens que iremos utilizar nesse módulo, como por exemplo componentes e diretivas, já nos **Providers** informamos os serviços.

Assim como módulos **JavaScript**, o *NgModule* também pode importar funcionalidades de outros *NgModules* e permitir que suas próprias funcionalidades também sejam exportadas.

Um exemplo claro disso é que para usar o serviço de roteador no seu app basta importar o *RouterModule*.

```
@NgModule({
  declarations: [ AppComponent ],
  imports: [ AppRoutingModule ],
  providers: [ UmService ]
})
```

## Components

A maior parte do desenvolvimento quando se utiliza o *framework Angular* é feito nos componentes.

**Cada componente define uma classe**, que contém dados e lógicas do aplicativo e está sempre associada a um *template HTML*, onde são definidas as visualizações deste componente.

O decorador `@Component()` identifica a classe imediatamente como um componente e oferece o modelo e os metadados específicos dele.

Os metadados configuram, por exemplo, como o componente pode ser referenciado no **HTML** e também quais os serviços devem ser utilizados.

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
```

## Directives

As diretivas são como marcadores no elemento DOM que comunicam ao **Angular** para incluir um comportamento específico.

Existem três tipo de diretivas no Angular, que são: Diretivas de atributos, Diretivas estruturais e Componentes.

**Diretivas de atributos:** Alteram a aparência ou o comportamento de um elemento, componente ou outra diretiva, como por exemplo, *NgClass* e *NgStyle*.

**Diretivas estruturais:** Modificam o *layout* adicionando ou removendo elementos do DOM, como por exemplo, *NgIf* e *NgFor*.

**Diretivas de Componentes:** São diretivas com um modelo.

## Aplicações Angular

Como mencionado anteriormente, o Angular é dos frameworks mais usados na Web. Vou listar alguns deles aqui:

- **Comunidade apoiada pelo Google** - O Google apoia ativamente o Angular e seu desenvolvimento. Angular é usado em vários Google Apps.
- **Interface do usuário declarativa** - Angular usa HTML como linguagem de exibição e amplia sua funcionalidade. Ajuda no tratamento da diferenciação da interface do usuário versus o código, e a interface do usuário é pouco acoplada ao código.
- **TypeScript** - O TypeScript é um super conjunto de javascript e é fácil de depurar. É altamente seguro e é orientado a objetos.
- **Estrutura modular** - O desenvolvimento angular é altamente modular, é baseado em componentes e é de fácil manutenção.
- **Supporte multiplataforma** - O código angular funciona bem em todas as plataformas sem muita alteração no código

### Visão geral

**Angular** é propriedade do Google e a versão estável foi feito em maio de 2021. A ultima versão lançada é o Angular 12.

Abaixo está a lista de versões angulares lançadas até agora -

Versão	Data de lançamento
JS angular	Outubro de 2010
Angular 2.0	Set 2016
Angular 4.0	Março de 2017
Angular 5.0	Novembro 2017
Angular 6.0	Maio 2018
Angular7.0	Outubro 2018

Angular 8.0	Março / abril de 2019
Angular 9.0	Fevereiro 2020
Angular 10.0	Junho de 2020
Angular 11.0	Novembro 2020
Angular 12.*	Maio de 2021 – versão beta
Angular 13.*	A partir de Novembro de 2021
Angular 14*	A partir de Fevereiro de 2022

O Google planeja lançar a principal versão angular a cada 6 meses. A versão lançada até o momento é compatível com versões anteriores e pode ser atualizada para a mais recente com muita facilidade.

## Angular - Configuração do ambiente

Neste passo, discutiremos a configuração do ambiente necessária para o Angular. Para instalar o Angular, é necessário ter instalados em seu computador:

- Nodejs
- Npm
- CLI angular
- IDE para escrever seu código

### Nodejs

Para verificar se o nodejs está instalado no seu sistema, digite **node -v** no terminal. Isso ajudará você a ver a versão do nodejs atualmente instalada no seu sistema.

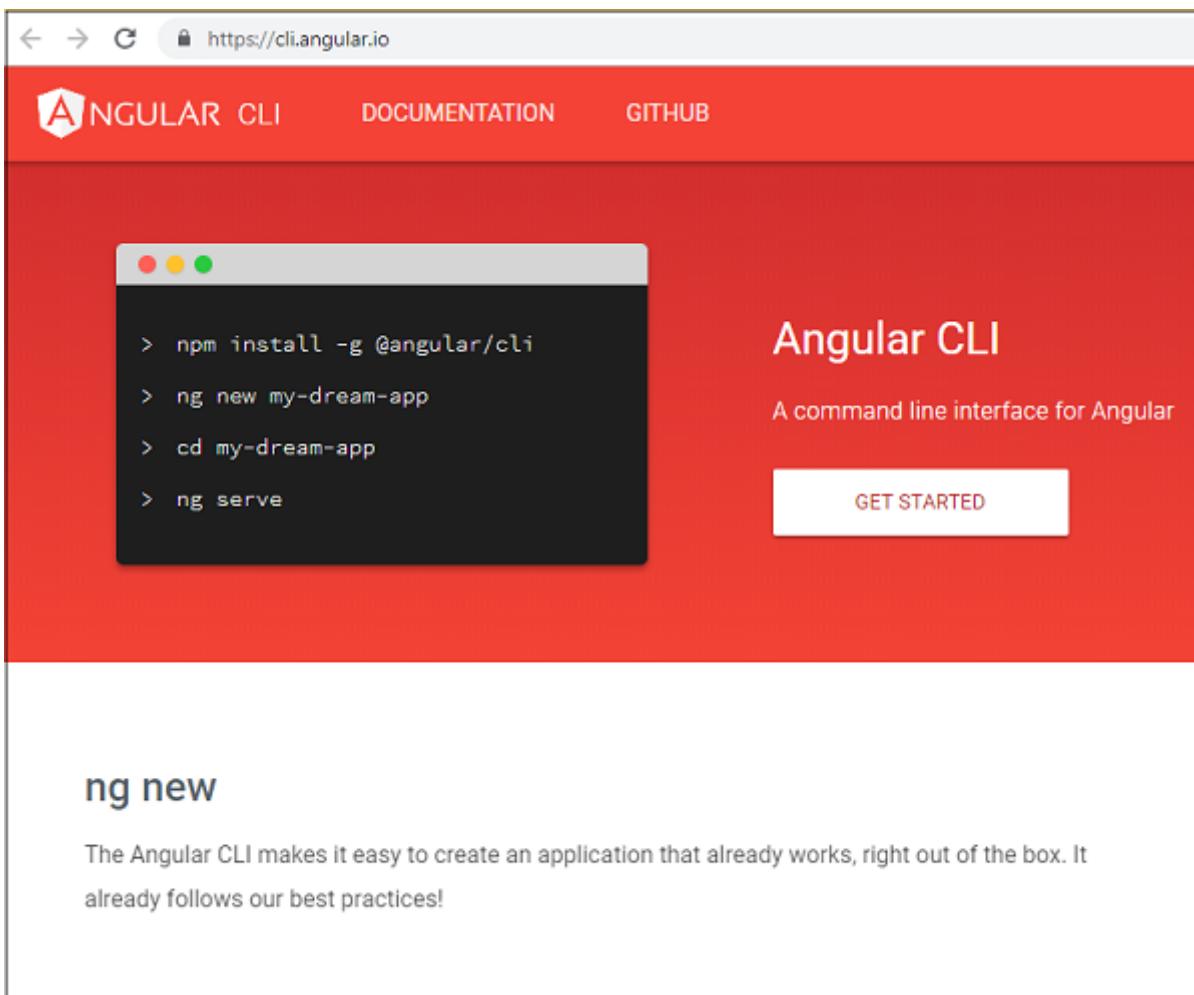
Nodejs deve ser maior que 8.x ou 10.x, e npm deve ser maior que 5.6 ou 6.4.

```
C:\>node -v  
v10.15.1
```

Com base no seu sistema operacional, instale o pacote necessário. Depois que o nodejs estiver instalado, o npm também será instalado junto com ele. Para verificar se o npm está instalado ou não, digite npm -v no terminal, conforme indicado abaixo. Ele exibirá a versão do npm.

```
C:\>npm -v  
6.4.1
```

As instalações do Angular são muito simples com a ajuda da CLI angular. Visite a página inicial <https://cli.angular.io/> do angular para obter a referência do comando.



Digite **npm install -g @angular/cli** no prompt de comando para instalar o angular cli no seu sistema. Vai demorar um pouco para instalar e, uma vez feito, você pode verificar a versão usando o comando abaixo:

```
npm install -g @angular/cli // comando instalar o
Angular
  ng new nome_do_projeto // nome do projeto
  cd nome-da-pasta
  ng serve
```

Os comandos acima ajudam a obter a configuração do projeto no Angular.

Vamos criar uma pasta chamada **projetos-angular** e instalar **angular/cli** como mostrado abaixo

Depois que a instalação estiver concluída, verifique os detalhes dos pacotes instalados usando o comando **ng version**, como mostrado abaixo:

```
Angular CLI: 13.1.2
Node: 16.13.1
Package Manager: npm 8.5.0
OS: win32 x64

Angular: undefined
...
Package          Version
-----
@angular-devkit/architect    0.1301.2
@angular-devkit/core         13.1.2
@angular-devkit/schematics   13.1.2
@angular/cli                 13.1.2
@schematics/angular          13.1.2
```

Ele fornece a versão para o Angular CLI e as versões dos pacotes disponíveis para o Angular.

Terminamos a instalação do Angular, agora começaremos com a configuração do projeto.

## Atualização angular

O Angular é uma versão importante em que, na estrutura do núcleo angular, Angular CLI, Angular Materials são atualizados. Caso você esteja usando o Angular 5 ou 6 e queira atualizar para o Angular na sua versão atual, abaixo está o comando que atualizará seu aplicativo para a versão recente do Angular:

```
ng update @angular/cli
```

## Configuração do Projeto

### Projeto angular-alpha

Vamos criar uma pasta na área de trabalho. Essa pasta abrigará nossos projetos angular. Para criar uma pasta na área de trabalho – via prompt de comando – basta inserir no prompt as seguintes intruções:

```
C:\Users\SeuUsuario\Desktop\mkdir projetos-angular
C:\Users\SeuUsuario\Desktop\cd projetos-angular
C:\Users\SeuUsuario\Desktop\projetos-angular\ng new angular-alpha
```

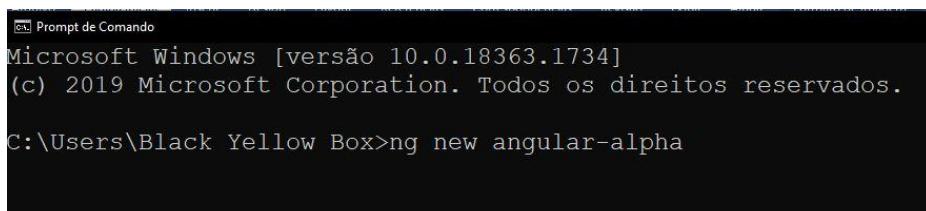
O conjunto de instruções acima mostra que: devemos encontrar e inserir no prompt o path (caminho) completo do diretório “Área de trabalho (Desktop)”. Na sequencia, inserimos o comando mkdir e o nome da nossa pasta (projetos-angular). Feito isso, basta clicar Enter em seu teclado. Sua pasta para os projetos Angular está criada.

Na sequencia, basta inserir ainda no Prompt de comando a instrução: `cd projetos-angular`. Para criar um projeto Angular, foi usado o seguinte comando: a partir do prompt de comando:

```
ng new angular-alpha
```

Use o *nome do projeto* de sua escolha. Agora, será executado o comando acima através do prompt.

Aqui, usamos o *nome do projeto* como *angular-alpha*. Depois de executar o comando, ele perguntará sobre o roteamento, como mostrado abaixo:



```
Prompt de Comando
Microsoft Windows [versão 10.0.18363.1734]
(c) 2019 Microsoft Corporation. Todos os direitos reservados.

C:\Users\Black Yellow Box>ng new angular-alpha
```

Digite *y* para adicionar roteamento à configuração do seu projeto.

A próxima pergunta é sobre a folha de estilo:

```
npm
Microsoft Windows [versão 10.0.18363.1734]
(c) 2019 Microsoft Corporation. Todos os direitos reservados.

C:\Users\Black Yellow Box>ng new angular-alpha
? Would you like to add Angular routing? (y/N) y
```

As opções disponíveis são CSS, Sass, Less e Stylus. Na captura de tela acima, a seta está no CSS. Para alterar, você pode usar as teclas de seta para selecionar a necessária para a configuração do seu projeto. No momento, discutiremos o CSS para a configuração do projeto.

```
npm
Microsoft Windows [versão 10.0.18363.1734]
(c) 2019 Microsoft Corporation. Todos os direitos reservados.

C:\Users\Black Yellow Box>ng new angular-alpha
? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use? (Use arrow keys)
> CSS
SCSS  [ https://sass-lang.com/documentation/syntax#scss           ]
Sass   [ https://sass-lang.com/documentation/syntax#the-indented-syntax ]
Less    [ http://lesscss.org                                         ]
```

O projeto ***angular-alpha*** foi criado com sucesso. Ele instala todos os pacotes necessários para que nosso projeto seja executado no Angular. Vamos agora entrarna pasta do projeto criado, que está no diretório ***angular-alpha***.

Altere o diretório na linha de comando usando a linha de código fornecida:

```
cd angular-alpha
```

## Visual Studio Code

Este é um IDE de código aberto do Visual Studio. Está disponível para plataformas Mac OS X, Linux e Windows. O VScode está disponível em - <https://code.visualstudio.com/>

Instalação no Windows - se ainda não estive instalado em seu ambiente, basta seguir as mesmas estapas da instalação anterior (para typescript).

Não iniciamos nenhum projeto nele. O que precisamos fazer, agora, é trazer o projeto que criamos - usando o angular-cli e seu respectivo comando para criação – para dentro da nossa IDE Visual Studio Code. O procedimento para executarmos essa tarefa é o seguinte:

Clique em File > ao clicar em File no menu de tarefas no topo da IDE se abrirá uma janela de contexto: ao abri a janela selecione Open Folder. Na sequencia, encontre o local onde seu projeto Angular foi gerado; clique na pasta que contem seu projeto e, dessa forma ele será importado para dentro da IDE.

Vamos abrir o ***angular-alpha*** e ver como é a estrutura da pasta.

Agora que temos a estrutura de arquivos do nosso projeto, vamos compilar nosso projeto com o seguinte comando:

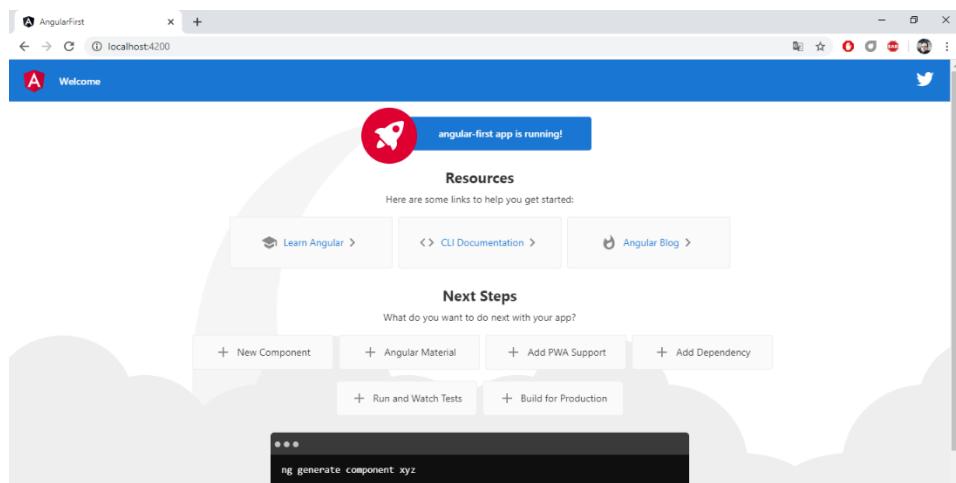
### **ng serve**

O comando `ng serve` cria o aplicativo e inicia o servidor web:

Você verá o abaixo quando o comando começar a executar:

O servidor da web inicia na porta 4200. Digite o URL "**`http://localhost:4200/`**" no navegador e veja a saída. Depois que o projeto for compilado, você receberá a seguinte saída:

Depois de executar o URL, **`http://localhost:4200/`** no navegador, você será direcionado para a seguinte tela:



Vamos concluir a configuração do projeto. Se você vir que usamos a porta 4200, que é a porta padrão usada pelo `angular - cli` durante a compilação. Você pode alterar a porta, se desejar, usando o seguinte comando:

```
ng serve --host 0.0.0.0 -port 4205 por exemplo
```

A pasta `angular-alpha` / tem a seguinte **estrutura de pastas**:

- **node\_modules** / - O pacote npm instalado é `node_modules`. Você pode abrir a pasta e ver os pacotes disponíveis.
- **src** / - Esta pasta é onde iremos trabalhar no projeto usando o Angular. Especialmente usando os arquivos localizados dentro da pasta `src / app` que foi criada durante a instalação do projeto e contém todos os arquivos necessários para trabalharmos no `angular-alpha`.

A pasta `angular-alpha` contém alguns arquivos em sua estrutura:

- **editorconfig** - Este é o arquivo de configuração do editor.
- **.gitignore** - Um arquivo `.gitignore` deve ser confirmado no repositório, para compartilhar as regras de ignorar com outros usuários que cloram o repositório.
- **angular.json** - basicamente contém o nome do projeto, a versão do cli etc.

- **package.json** - O arquivo package.json informa quais bibliotecas serão instaladas no node\_modules quando você executar o npm install. Caso você precise adicionar mais bibliotecas, adicione-as no arquivo package.json e execute o comando npm install.
- **tsconfig.json** - basicamente contém as opções do compilador necessárias durante a compilação.
- **tslint.json** - Este é o arquivo de configuração com regras a serem consideradas durante a compilação.

A pasta **src /** é a pasta principal, que possui internamente uma estrutura de arquivos diferente.

### Pasta App

Ela contém os arquivos descritos abaixo. Esses arquivos são instalados pelo angular-cli por padrão:

#### **app-routing.module.ts**

Este arquivo tratará do roteamento necessário para o seu projeto. Ele está conectado ao módulo principal, ou seja, *app.module.ts*.

Observe a estrutura do arquivo, abaixo:

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

const routes: Routes = [];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

#### **app.component.html**

O código html que será exibido em tela, geralmente, estará disponível neste arquivo.

A estrutura principal do código é a seguinte:

```
<div class="toolbar" role="banner">
  <img
    width="40"
    alt="Angular Logo"/>
  <span>Bem-vindo</span>
  <div class="spacer"></div>
</div>
```

```

<div class="content" role="main">

    <!-- Highlight Card -->
    <div class="card highlight-card card-small">

        <svg id="rocket" xmlns="http://www.w3.org/2000/svg" width="101.678" height="101.678" viewBox="0 0 101.678 101.678">
            <title>Rocket Ship</title>
            <g id="Group_83" data-name="Group 83" transform="translate(-141 -696)">
                <circle id="Ellipse_8" data-name="Ellipse 8" cx="50.839" cy="50.839" r="50.839" transform="translate(141 696)" fill="#dd0031"/>
                <g id="Group_47" data-name="Group 47" transform="translate(165.185 720.185)">
                    <path id="Path_33" data-name="Path 33" transform="translate(0.371 3.363)" fill="#fff"/>
                    <path id="Path_34" data-name="Path 34" fill="#fff"/>
                </g>
            </g>
        </svg>

        <span>{{ title }} app is running!</span>

        <svg id="rocket-smoke" xmlns="http://www.w3.org/2000/svg" width="516.119" height="1083.632" viewBox="0 0 516.119 1083.632">
            <title>Rocket Ship Smoke</title>
            <path id="Path_40" fill="#f5f5f5"/>
        </svg>

    </div>

    <svg id="clouds" xmlns="http://www.w3.org/2000/svg" width="2611.084" height="485.677" viewBox="0 0 2611.084 485.677">
        <title>Gray Clouds Background</title>
        <path id="Path_39"/>
    </svg>

</div>

```

Este é o código html padrão (aqui, com a exclusão de identificação de imagens svg) atualmente disponível com a criação do projeto.

### app.component.spec.ts

Esses são arquivos gerados automaticamente que contêm testes de unidade para o componente de origem.

### app.component.ts

A classe (estrutura lógica) para as regras de funcionamento do componente é definida neste arquivo. Você pode fazer o processamento da estrutura html no arquivo `.ts`. O processamento incluirá atividades como conectar-se ao banco de dados, interagir com outros componentes, roteamento, services, etc.

Observe a estrutura de código abaixo:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'angular-alpha';
}
```

### app.module.ts

Se você abrir o arquivo, verá que o código faz referência a diferentes bibliotecas importadas. O Angular-cli usou essas bibliotecas padrão para a importação: angular / core.

Os próprios nomes explicam o uso das bibliotecas. Eles são importados e salvos em variáveis como declarações, importações, providers e autoinicialização.

Podemos ver também **app-routing.module**. Isso ocorre porque selecionamos o roteamento no início da instalação. O módulo é adicionado por @ angular / cli.

Observe a estrutura de código abaixo:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

```

] ,
imports: [
  BrowserModule,
  AppRoutingModule
],
providers: [],
bootstrap: [AppComponent]
} )
export class AppModule { }

```

O `@NgModule` é importado de `@ angular/core` e possui um objeto com as seguintes propriedades -

**Declarations** - nas declarações, a referência aos componentes é armazenada. O componente App é o componente padrão criado sempre que um novo projeto é iniciado. Aprenderemos sobre a criação de novos componentes em uma seção diferente.

**Imports** - Os módulos serão importados conforme mostrado acima. No momento, o `BrowserModule` faz parte das importações importadas do `@ angular / platform-browser`. Há também o módulo de roteamento adicionado `AppRoutingModule`.

**Providers** - Isso terá referência aos services criados. O service será discutido em um capítulo subsequente.

**Bootstrap** - Isso tem referência ao componente padrão criado, ou seja, `AppComponent`.

**app.component.css** - Você pode escrever o css do componente neste arquivo. No momento, adicionamos a cor de fundo à div, como mostrado abaixo.

## Assets

Você pode salvar suas imagens, arquivos js nesta pasta.

## Environment

Esta pasta possui detalhes para a produção ou o ambiente de desenvolvimento. A pasta contém dois arquivos.

- `environment.prod.ts`
- `environment.ts`

Ambos os arquivos têm detalhes sobre se o arquivo final deve ser compilado no ambiente de produção ou no ambiente de desenvolvimento.

A estrutura adicional do arquivo `angular-alpha / folder` inclui o seguinte:

## favicon.ico

Esse é um arquivo geralmente encontrado no diretório raiz de um site.

## index.html

Este é o arquivo que é exibido no navegador.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>AngularAlpha</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

O corpo tem **<app-root></app-root>**. Esse é o seletor usado no arquivo **app.component.ts** e exibirá os detalhes do arquivo **app.component.html**.

## main.ts

*main.ts* é o arquivo de onde começamos o desenvolvimento do projeto. Começa com a importação do módulo básico que precisamos. No momento, se você ver angular / core, angular / plataforma-dinâmica do navegador, app.module e ambiente, é importado por padrão durante a instalação do angular-cli e a configuração do projeto.

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));
```

O `platformBrowserDynamic().BootstrapModule(AppModule)` possui a referência do módulo pai `AppModule`. Portanto, quando é executado no navegador, o arquivo é chamado `index.html`. `Index.html` refere-se internamente a `main.ts` que chama o módulo pai, ou seja, `AppModule` quando o código a seguir é executado:

```
platformBrowserDynamic().bootstrapModule(AppModule).catch((err => console.error(err)) );
```

Quando o `AppModule` é chamado, ele chama `app.module.ts`, que chama mais o `AppComponent` com base na autoinicialização da seguinte maneira:

```
bootstrap: [AppComponent]
```

### **polyfill.ts**

Isso é usado principalmente para compatibilidade com versões anteriores.

### **styles.css**

Este é o arquivo de estilo necessário para o projeto.

### **test.ts**

Aqui, os casos de teste de unidade para testar o projeto serão tratados.

### **tsconfig.app.json**

Isso é usado durante a compilação, possui os detalhes de configuração que precisam ser usados para executar o aplicativo.

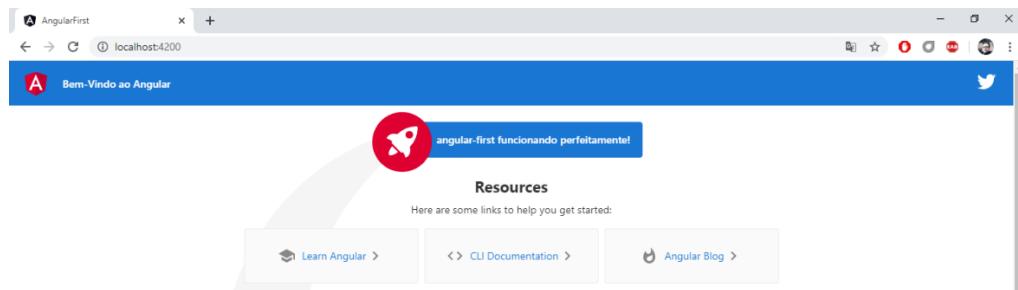
### **tsconfig.spec.json**

Isso ajuda a manter os detalhes para o teste.

A estrutura final do arquivo será a seguinte:

No `app.component.ts`, existe um seletor: `app-root`, usado no arquivo `index.html`. Isso exibirá o conteúdo presente em `app.component.html`.

A imagem abaixo, se correu tudo certo com a criação do projeto, ao executar a instrução `ng serve`, representa a renderização do projeto exibido no browser através do endereço: `http://localhost:4200`



## Data Binding

### O que é Data Binding?

A ligação de dados (*data binding*) é um dos mais úteis recursos da estrutura Angular. Devido a esse recurso, é possível escrever menos código em comparação com qualquer outra biblioteca ou estrutura do *client-side*. Em função do data binding - em um aplicativo Angular - a sincronização automática de dados entre o modelo (model) e os componentes da visualização (view) é automático. No Angular, sempre podemos tratar o modelo (model) como uma única *single-source-of-truth* de dados em nosso aplicativo Web usando *data binding*. Dessa forma, a interface do usuário ou a visualização (view) sempre representa o modelo (model) de dados o tempo todo. Com a ajuda da data binding, podemos estabelecer uma relação entre a interface do usuário do aplicativo e a lógica de negócios. Se estabelecermos a ligação de dados (data binding) de maneira correta e os dados fornecerem as notificações apropriadas para a estrutura, quando o usuário fizer alterações nos dados na visualização, o aplicativo será atualizado e exibirá as informações que desejamos.

### Necessidade do Data Binding

A estrutura Angular fornece esse recurso especial e poderoso chamado Data Binding, que traz flexibilidade em qualquer aplicativo Web. Com a ajuda da ligação de dados (data binding), podemos obter melhor controle sobre o processo e as etapas relacionadas ao processo de comunicação entre componentes. Esse processo facilita a vida do desenvolvedor e reduz o tempo de desenvolvimento em relação a outras estruturas. Alguns dos motivos

relacionados aos motivos pelos quais a ligação de dados é necessária para qualquer aplicativo baseado na Web estão listados abaixo

1. Com a ajuda da ligação de dados (data binding), as páginas Web baseadas em dados podem ser desenvolvidas de maneira rápida e eficiente.
2. Sempre obtemos o resultado desejado com poucas linhas de código.
3. Devido a esse processo, o tempo de execução aumenta. Como resultado, melhora a qualidade do aplicativo.

Com a ajuda do emissor do evento (event emitter), podemos obter um melhor controle sobre o processo de ligação de dados.

### **Diferentes Tipos de Ligação de Dados (Data Binding)**

No Angular, existem quatro tipos diferentes de processos de ligação de dados disponíveis. São eles:

- Interpolação (Interpolation)
- Ligação de propriedade (Property Binding)
- Ligação através de evento (Event Binding)
- Ligação através de evento (Event Binding)
- Ligação bidirecional (Two-Way Binding)

### **Interpolação (Interpolation)**

A ligação de dados de interpolação (interpolation) é a maneira mais popular e mais fácil de ligação de dados no Angular. Esse recurso também está disponível nas versões anteriores da estrutura Angular. Na verdade, o **contexto entre chaves** é a expressão do modelo (model) que o Angular avalia primeiro e depois converte em sequências de caracteres (string). A interpolação (interpolation) usa a expressão de chaves, ou seja, {{}} para renderizar o valor associado ao componente (component). Pode ser uma sequência estática, valor numérico ou um objeto do seu modelo (model) de dados. No

Angular, usamos assim: {{firstName}}. O exemplo abaixo mostra como podemos usar a interpolação no componente para exibir dados no front end.

```
<span>{{ title }} aplicativo angular funcionando!</span>
```

## Implementação Interpolation (Interpolação)

Neste passo, vamos implementar cada um dos data binding estudados até este momento. No exemplo abaixo, faremos uso da Interpolação (Interpolation) no Angular para diferentes tipos de dados.

Crie um novo componente e nomeie-o como ***interpolation*** dentro da pasta app do seu projeto – para criar este novo componente - via angular cli – é necessário usar o comando abaixo:

```
ng generate component interpolation
```

Abra a pasta do componente e edite os arquivos conforme indicado abaixo:

### ***interpolation.component.ts***

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

@Component({
  selector: 'app-interpolation',
  templateUrl: './interpolation.component.html',
  styleUrls: ['./interpolation.component.css']
})
export class InterpolationComponent{
  // criando as propriedades para data binding interpolation
  public x: number = 10
  public umArray: Array<number> = [10, 22, 14]
  public dataHoje: Date = new Date()

  public statusBooleano: boolean = true

  public exibirTexto(): string{
    return 'Texto retornado a partir de uma função!'
  }
}
```

### ***interpolation.component.html***

```
<!-- vinculando as propriedades descritas no ts -->
<div>
  <h1>Interpolation Data Binding</h1>

  <span>O valor da variavel x é: {{ x }}</span>
  <br /><br />
```

```

<span>O array de números é composto pelos seguintes valores: {{ umArray }}</span>
<br /><br />

<span>A data de hoje é: {{ dataHoje }}</span>
<br /><br />

<span>O status booleano indicado na variável é: {{ statusBooleano }}</span>
<br /><br />

<span>{{ statusBooleano ? "Este é o texto para o status True" : "Este é o texto para o status False!" }}</span>
<br /><br />
<span>{{ exibirTexto() }}</span>
<br /><br />
</div>

```

Antes de observar o resultado da primeira implementação, no browser, navegue até o arquivo **app-routing.module.ts** e implemente o código abaixo com o que segue. Neste arquivo serão criadas todas as rotas para todos os componentes deste projeto.

***Obs.: este passo deve ser repetido para todos os novos componentes criados dentro deste projeto***

## Routing

Routing (Rotear) significa basicamente navegar entre as páginas. Aqui as páginas às quais existem referências estarão na forma de componentes. Durante a configuração do projeto, já incluímos o módulo de roteamento; está disponível dentro do arquivo **app.module.ts**, como mostrado abaixo:

### **app.module.ts**

```

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';

import { AppComponent } from './app.component';
import { AlterarTextoDirective } from './alterar-
texto.directive';
import { ComponenteFilhoComponent } from './componente-
filho/componente-filho.component';

```

```

import { RaizQuadrada } from './pipe-raiz-quadrada';

@NgModule({
  declarations: [
    AppComponent,
    AlterarTextoDirective,
    ComponenteFilhoComponent,
    RaizQuadrada
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

*AppRoutingModule* é adicionado - como mostrado acima - e incluído na array *imports*.

Os detalhes do arquivo *app-routing.module.ts* estão indicados abaixo:

### ***app-routing.module.ts***

```

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

const routes: Routes = [];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }

```

Aqui, é preciso observar que esse arquivo é gerado por padrão quando o roteamento é adicionado durante a configuração do projeto. Se não forem adicionados, deverá ser criado manualmente.

Portanto, no arquivo acima, foram importadas as dependencias *Routes* e *RouterModule* de *@angular/router*.

Há uma const *route* definida, que é do tipo *Routes*. É um array onde serão implementadas todas as rotas dos componentes deste projeto.

As const *route* é fornecida ao *RouterModule* como mostrado em *@NgModule*. Para exibir os detalhes de roteamento para o usuário, é preciso adicionar a diretiva **<router-outlet></router-outlet>** onde queremos que a exibição seja mostrada – por padrão, as tags são referenciadas no arquivo *app.component.html* - a view principal do projeto - como mostrado abaixo:

```
<router-outlet></router-outlet>
```

Agora, a partir da criação do componente **Interpolation**, crie um novo componente chamado **Home** – seguindo os mesmos passos realizados para criar o componente **Interpolation** – este novo componente servirá de base para o projeto. Na sequência, crie as rotas necessárias para que eles sejam acessados e o resultado da implementação seja exibido no browser. Implemente o código abaixo como segue:

### **app-routing.module.ts**

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
// importar os componentes aqui
import { HomeComponent } from './componentes/home/home.component';
import { InterpolationComponent } from './componentes/interpolation/interpolation.component';

const routes: Routes = [
  {path: '', redirectTo:'home', pathMatch:'full'},
  {path: 'home', component:HomeComponent},
  {path: 'interpolation', component:InterpolationComponent},
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Mais um passo precisa ser dado. Navegue até o arquivo `app.component.html` e crie os `routerLinks` necessários para, através de um clique no link, seja possível exibir o componente no browser. Implemente o código abaixo como se segue:

### **app.component.html**

```
<!-- Toolbar -->
<div class="toolbar" role="banner">
  .
  .
  .
<!-- AQUI, VAMOS COLOCAR NOSSOS CÓDIGOS -->

  <nav>
    <a routerLink = "/home">Home</a>
    <a routerLink = "/interpolation">Interpolação</a>
  </nav>
```

```
<!-- AQUI, VAMOS COLOCAR NOSSOS CÓDIGOS -->
.
.
.
</div>
<router-outlet></router-outlet>
```

Navegue até o arquivo *app.compoennte.css* e implemente – ao final das instruções – o código abaixo:

### ***app.compoennte.css***

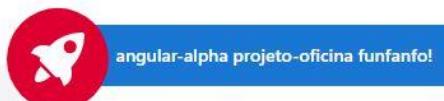
```
/*atributos dos links para o roteamento*/
nav{
    margin-top: 30px;
}
a:link, a:visited {
    background-color: #848686;
    color: white;
    padding: 10px 25px;
    text-align: center;
    text-decoration: none;
    display: inline-block;
}
a:hover, a:active {
    background-color: #BD9696;
}
```

Agora, navegue até o arquivo *styles.css* e implemente o bloco de código a seguir:

### ***styles.css***

```
/* You can add global styles to this file, and also import
other style files */
.content {
    display: flex;
    margin: 82px auto 32px;
    padding: 0 16px;
    max-width: 960px;
    flex-direction: column;
    align-items: center;
}
```

Ao final destes passos, o resultado será semelhante ao indicado abaixo:



Home      Interpolação

## Olá! seja bem-vindo ao meu projeto-oficina Angular!

Finalmente, é possível clicar no link Interpolação e exibir o resultado da implementação feita no componente. Observe o resultado. Salve o projeto e observe a saída.

### Implementação Interpolation data binding

O valor da variavel x é: 10

O array de numeros é composto pelos seguintes valores: 10,22,45

A data de hoje - composta - é: Tue Jul 12 2022 17:18:32 GMT-0300 (Horário Padrão de Brasília)

O status booleano indicado na variavel é: false

Este é o texto para quando o status for False

Texto retornado a partir de uma função

## Ligaçāo/Vinculaçāo de dados baseada em propriedades (Property Binding)

No Angular, existe outro mecanismo de ligação, chamado de Vinculação de Propriedade (property binding). Em sua essência, vinculação/ligação de dados baseada em propriedade é exatamente o mesmo que interpolação. É também chamado de ligação unidirecional. Para ligação de propriedade (property binding) é usado [ ] para enviar os dados do componente para o modelo HTML. A maneira mais comum de usar o property binding é atribuir

qualquer propriedade da tag do elemento HTML ao [ ] com o valor da propriedade do componente; observe o snippet abaixo:

```
Exibir valor com property-
binding : <input [value]="umvalor" />
```

## Implementação property Binding

Nesta sequencia, vamos implementar o data binding - conforme exemplo abaixo – baseado em propriedade (property binding). Crie um novo componente e nomeie-o como ***property-binding*** dentro da pasta app do seu projeto – para criar este novo componente - via angular cli – é necessário usar o comando abaixo:

```
ng generate component property-binding
```

Abra a pasta do componente e edite os arquivos conforme indicado abaixo:

### ***property-binding.component.ts***

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

@Component({
  selector: 'app-property-binding',
  templateUrl: './property-binding.component.html',
  styleUrls: ['./property-binding.component.css']
})
export class PropertyBindingComponent implements OnInit {

  constructor() { }

  ngOnInit(): void {
    // criar uma propriedade para ser vinculada via property
    // binding na view
    public umNumero: number = 10
  }
}
```

### ***property-binding.component.html***

```
<!-- vincular a propriedade via property binding -->
<div class = "content">
    <h2>Implementação Property Binding</h2>

    <span>Vinculando uma propriedade via property-binding
<input [value] = "umNumero" /></span>
</div>
```

### ***app.component.html***

```
<!-- AQUI, VAMOS COLOCAR NOSSOS CÓDIGOS -->
<nav>
    <a routerLink = "/home">Home</a>
    <a routerLink = "/interpolation">Interpolação</a>
    <a routerLink = "/propbinding">Property Binding</a>
</nav>
<!-- AQUI, VAMOS COLOCAR NOSSOS CÓDIGOS -->
```



## **Property-binding**

O valor da variável x é:

## **Vinculação/ligação por evento (Event Binding)**

A ligação de eventos (event binding) é outra das técnicas de ligação de dados (data binding) disponíveis no Angular. Essa técnica não funciona com o valor dos elementos da interface do usuário - funciona com as **atividades de eventos dos elementos da interface do usuário**, como **evento de clique**, **evento de desfoco** (blur-event) etc. Nas versões anteriores do Angular, são usadas diferentes tipos de diretivas (directives). como ng-click, ng-blur para vincular qualquer ação de evento específica de um controle (control) HTML. Porém, na versão Angular atual, precisamos usar a mesma propriedade do

elemento HTML (como clicar, alterar etc.) e usá-la entre parênteses. No Angular, para propriedades, usamos colchetes e, em eventos, parênteses.

```
<div>
    <h2>Exemplo Event Binding Angular</h2>
    <input type="button" value="Click" (click)="exibirAlerta()" />
    <br /><br />
    <input type="button" value="Mouse Enter" (mouseenter)="exibirAlerta()" />
</div>
```

### **Implementação Event Binding (Ligaçāo/Vinculaçāo de dados baseada em evento)**

Agora, vamos implementar o data binding - conforme exemplo abaixo – baseado em evento (event binding). Crie um novo componente e nomeie-o como **event-binding** dentro da pasta app do seu projeto – para criar este novo componente - via angular cli – é necessário usar o comando abaixo:

***ng generate component event-binding***

Abra a pasta do componente e edite os arquivos conforme indicado abaixo:

#### ***event-binding.component.ts***

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

@Component({
  selector: 'app-event-binding',
  templateUrl: './event-binding.component.html',
  styleUrls: ['./event-binding.component.css']
})
export class EventBindingComponent {

  // vamos implementar uma função para ser vinculada na view
  public exibirAlerta():void{
    console.log('Evento disparado...')
    alert('Evento disparado...')
  }
}
```

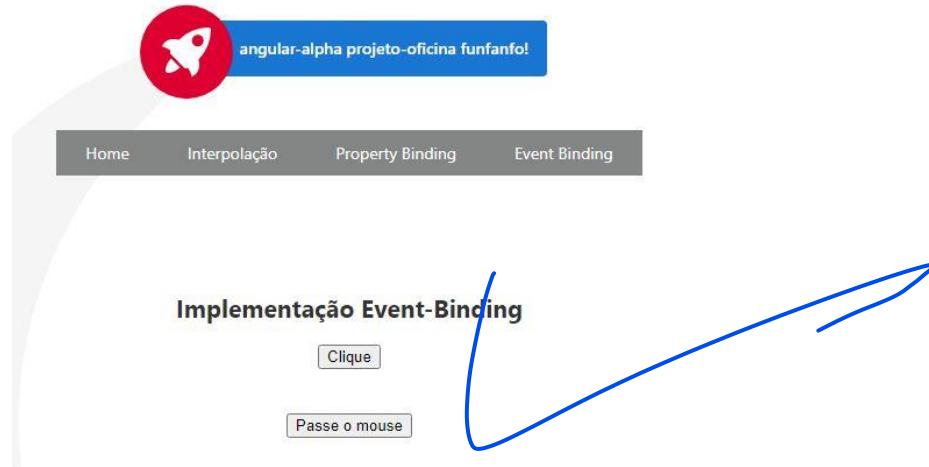
### ***event-binding.component.html***

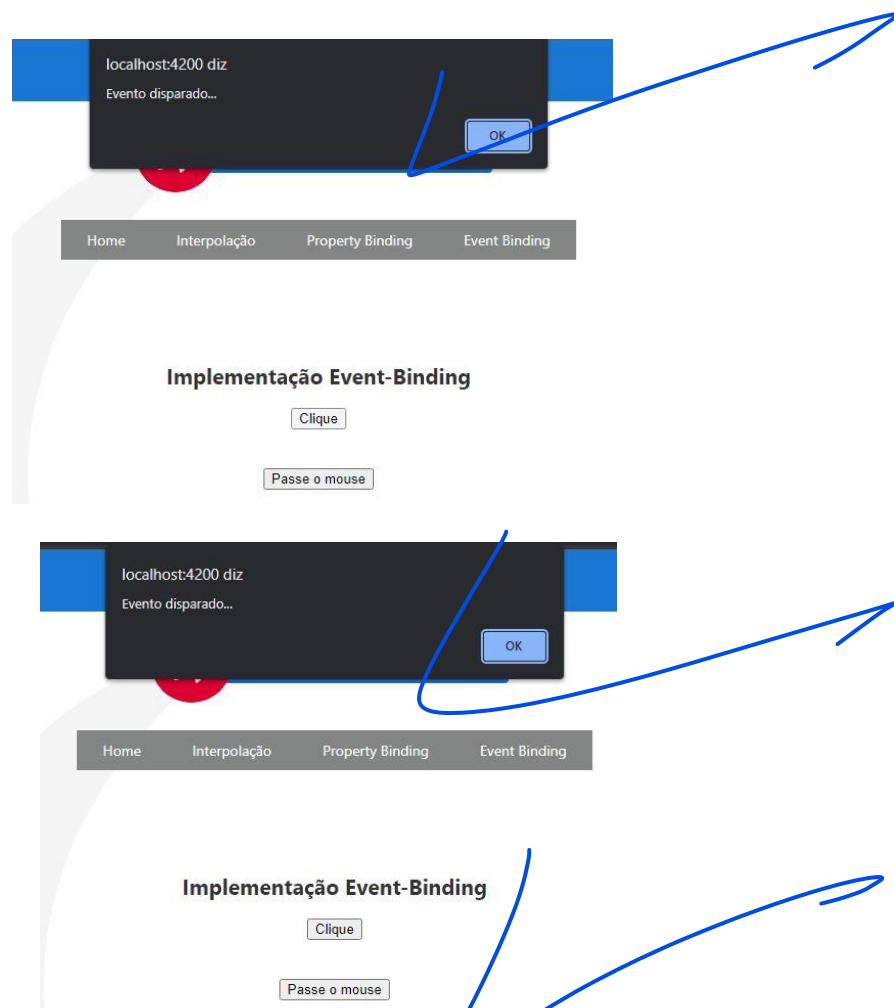
```
<div>
  <h2>Implementação Event Binding</h2>
  <input type = "button" value = "Clique" (click) = "exibirAlerta()" />
  <br />
  <input type = "button" value = "Passe o mouse" (mouseenter) = "exibirAlerta()" />
</div>
```

### ***app.component.html***

```
<!-- AQUI, VAMOS COLOCAR NOSSOS CÓDIGOS -->
<nav>
  <a routerLink = "/home">Home</a>
  <a routerLink = "/interpolation">Interpolação</a>
  <a routerLink = "/propbinding">Property Binding</a>
  <a routerLink = "/evento">Event Binding</a>
</nav>
<!-- AQUI, VAMOS COLOCAR NOSSOS CÓDIGOS -->
```

A saída será a seguinte:





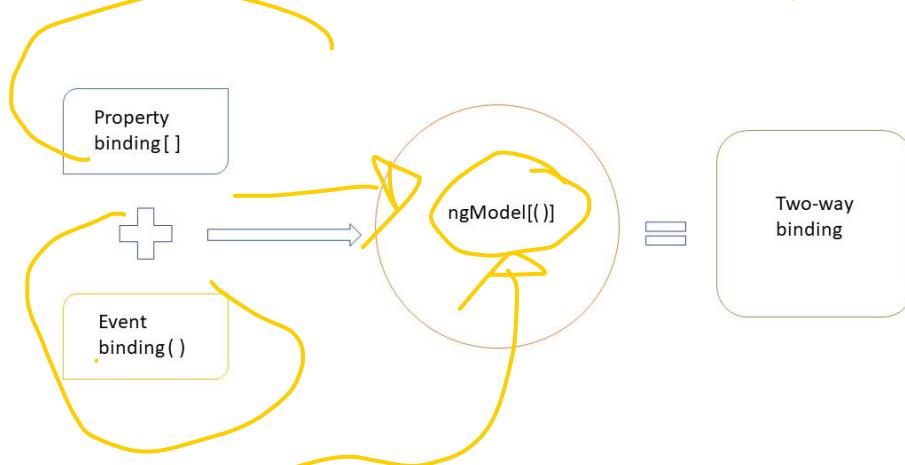
## Vinculação/ligação de dados bidirecional (Two-Way Data Binding)

No Angular Framework, as técnicas de ligação de dados (data binding) mais usadas e importantes são conhecidas como Vinculação de Dados Bidirecional (two-way data binding). A ligação bidirecional é usada principalmente no campo do tipo de entrada ou em qualquer elemento de formulário em que o usuário possa fornecer valores de entrada do navegador ou fornecer qualquer valor ou, ainda, alterar qualquer valor de controle por meio do navegador. Por outro lado, o mesmo é atualizado automaticamente nas variáveis do componente e vice-versa. Da mesma forma, no Angular, temos uma diretiva chamada `ngModel`, e ela precisa ser usada como abaixo:

```
<div>
  <div>
    <span>Informe seu nome </span>
    <input [(ngModel)]="x" type="text"/>
  </div>
```

```
<div>
  <span>Seu nome é: </span>
  <span>{ { x } }</span>
</div>
</div>
```

Usamos `[ ]`, pois na verdade é uma ligação de propriedade (property binding), e parênteses são usados para o conceito de ligação de evento (event binding), ou seja, a notação de ligação de dados bidirecional é `[()]`.



`ngModel` executa a ligação de propriedades (property binding) e a ligação de eventos (event binding). Na verdade, a ligação de propriedade (property binding) do `ngModel` (ou seja, `[ngModel]`) realiza a atividade para atualizar o elemento de entrada com um valor. Enquanto que (`ngModel`) (evento `(ngModelChange)`) instrui o “mundo externo” quando qualquer alteração ocorreu no elemento DOM.

### Implementação Two Way Data Binding (Vinculação/Ligação de dados bideracional)

Agora, nesse exemplo, mostraremos como usar a ligação de dados bidirecional em angular. Para esse fim, Crie um novo componente e nomeie-o como **two-way** dentro da pasta app do seu projeto – para criar este novo componente - via angular cli – é necessário usar o comando abaixo:

**`ng generate component two-way`**

Abra a pasta do componente e edite os arquivos conforme indicado abaixo:

Quando a diretiva `ngModel` ou ligação de dados bidirecional em nossos componentes são implementados, é necessário incluir o módulo de dependencia `FormsModule` do Angular no arquivo `app.module.ts`, como abaixo - o `ngModel` não funcionará sem a inclusão do `FormsModule`:

### **`app.module.ts`**

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
// importando o modulo Forms
import { FormsModule } from '@angular/forms';

@NgModule({
  declarations: [
    AppComponent,
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

### **`two-way.component.ts`**

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

@Component({
  selector: 'app-two-way',
  templateUrl: './two-way.component.html',
  styleUrls: ['./two-way.component.css']
})
export class TwoWayComponent {
  // propriedade que será vinculada via two-way binding
  public nome: string = ''
```



}

***two-way.component.html***

```
<div>
  <div>
    <span>Insira seu nome</span>
    <input type = "text" [(ngModel)] = "nome" />
  </div>
  <br /><br />
  <div>
    <span>Seu nome é:</span>
    <span>{{ nome }}</span>
  </div>
</div>
```



```
<!-- AQUI, VAMOS COLOCAR NOSSOS CÓDIGOS -->
<nav>
  <a routerLink = "/home">Home</a>
  <a routerLink = "/interpolation">Interpolação</a>
  <a routerLink = "/propbinding">Property Binding</a>
  <a routerLink = "/evento">Event Binding</a>
  <a routerLink = "/two-way">Two-way Binding</a>
</nav>
<!-- AQUI, VAMOS COLOCAR NOSSOS CÓDIGOS -->
```

A saída é a seguinte:



## Input property

No Angular Framework, todos os componentes são usados como componente sem estado (stateless component) ou com estado (statefull component). Normalmente, os componentes são usados e definidos como: sem estado (stateless component). Mas, às vezes, precisamos usar alguns componentes com estado (statefull component). O motivo por trás do uso de um componente com estado (statefull component) em um aplicativo Web é devido à passagem ou recebimento de dados do componente atual para o componente pai ou um componente filho. Portanto, dessa forma, precisamos indicar ao Angular que tipo de dados ou quais dados podem chegar ao nosso componente atual. Para implementar esse conceito, precisamos usar o decorador `@Input()` em qualquer variável. Os principais recursos do decorador `@Input()` são os seguintes:

- `@Input` é um decorador (decorator) para marcar uma propriedade de entrada. Com a ajuda dessa propriedade, podemos definir uma propriedade de parâmetro de entrada, assim como os atributos normais da tag HTML, para transmitir e vincular esse valor ao componente como uma ligação de propriedade (property binding).
- O decorador (decorator) do `@Input` sempre fornece uma **comunicação de dados unidirecional do componente pai para o componente filho**. Usando esse recurso, podemos fornecer parte do valor em relação a qualquer propriedade de um componente filho dos componentes pai.
- A propriedade `component` deve ser anotada com o decorador `@Input` para atuar como uma propriedade de entrada. Um componente pode receber valor de outro componente usando a ligação de propriedade do componente.

Ele pode ser anotado como qualquer tipo de propriedade, como número, string, array ou classe definida pelo usuário. Para usar um *alias* para o nome da propriedade de ligação (property binding), precisamos atribuir um nome alternativo como `@Input(alias)`. Uso de `@Input` com o tipo de dados string.

```
@Input() public mensagem :string = '';
@Input('alerta') public outraMensagem :string= ''
```

### Implementação Input Property

Nesse exemplo, será possível observar a maneira com a qual é possível implementar a propriedade de entrada de um componente. Para esse propósito, é necessário desenvolver o primeiro componente no qual a propriedade de entrada será definida.

Para esse fim, crie dois novos componentes e nomeie-o como **c-input-filho** e **c-pai**, respectivamente, dentro da pasta app do seu projeto – para criar este novo componente - via angular cli – é necessário usar o comando abaixo:

```
ng generate component c-input-filho
```

```
ng generate component c-pai
```

Abra a pasta do componente e edite os arquivos conforme indicado abaixo:

#### **c-input-filho.component.ts**

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

@Component({
  selector: 'app-c-input-filho',
  templateUrl: './input.component.html',
  styleUrls: ['./input.component.css']
})
export class CInputFilhoComponent{
  // elementos para fazer uso do Input Property para data binding
  @Input() public mensagem: string = '';
  @Input('alerta') public outraMensagem: string = '';

  public exibirMensagemAlerta(): void{
    alert(this.outraMensagem)
  }
}
```

#### **c-input-filho.component.html**

```
<div>
  Mensagem: <h3>{{ mensagem }}</h3>
  <input type = "button" value = "Exibir mensagem" (click)
) = "exibirMensagemAlerta()"/>
</div>
```

Agora, é necessário consumir esse componente das entradas de dados em outro componente e passar o valor de entrada usando as propriedades de entrada (input properties). Para este fim, será usado o componente **c-pai**. Implemente o código abaixo como se segue:

#### **c-pai.component.ts**

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class CPaiComponent {
  title = 'angular-alpha';

  // criando as propriedades cujo valores serão enviados para o
  // componente secundário
  public texto: string = 'Este é o texto que será exibido na janela de alerta!'
  public outroTexto: string = 'Texto enviado do componente-principal para o componente-secundário!'
}
```

#### **c-pai.component.html**

```
<app-c-input-
filho [alerta] = "texto" [mensagem] = "outroTexto"></app-c-
input-filho>
```

#### **app.componetne.html**

```
<!-- AQUI, VAMOS COLOCAR NOSSOS CÓDIGOS -->
<nav>
  <a routerLink = "/home">Home</a>
  <a routerLink = "/interpolation">Interpolação</a>
  <a routerLink = "/propbinding">Property Binding</a>
  <a routerLink = "/evento">Event Binding</a>
  <a routerLink = "/two-way">Two-way Binding</a>
  <a routerLink = "/primario">Input Prop</a>
</nav>
<!-- AQUI, VAMOS COLOCAR NOSSOS CÓDIGOS -->
```

A saída é a seguinte:

Mensagem:

Este é o texto que será exibido no input.component.html

Exibir Alerta

**Output Property**

No Angular, o `@Output` é um decorador (decorator) usado normalmente como uma propriedade de saída. `@Output` é usado para definir propriedades de saída para obter ligação de evento (event binding) personalizada. `@Output` será usado com a instância do Event Emitter. Os principais recursos do decorador (decorator) `@Output()` são os seguintes:

- O decorador (decorator) `@Output` vincula uma propriedade de um componente para enviar dados de um componente (componente filho) para um componente de chamada (componente pai).
- Essa é uma comunicação unidirecional de um filho para um componente pai.
- `@Output` vincula uma propriedade da classe `EventEmitter`. Se assumirmos o componente como um controle (control), a propriedade de saída atuará como um evento desse controle.
- O decorador (decorator) `@Output` também pode fornecer opções para personalizar o nome da propriedade usando o nome alternativo como `@Output` (alias). Nesse caso, esse nome alternativo personalizado atuará como um nome de propriedade de ligação de evento do componente.

Ele pode ser inserido em qualquer tipo de propriedade, como número, string, array ou classe definida pelo usuário. Para usar um *alias* para o nome da propriedade de ligação (property binding), precisamos atribuir um nome alternativo como @Output (alias). Uso de @Output com o tipo de dados string.

Assim como o decorador de entrada, o decorador de saída também precisa primeiro declarar como abaixo

```
@Output() anunciador = new EventEmitter<any>();
```

Agora, é necessário que o emissor (emitter) faça a “emissão” – o anúncio do evento - de um ponto no componente para que o evento possa ser gerado e rastreado pelo componente pai.

```
public enviarDados(): void{
    this.anunciador.emit(this.objDados)
}
```

Se queremos passar qualquer valor através desse emissor de evento (event emitter), precisamos passar esse valor como parâmetro através do emissor () .

No componente pai, essa propriedade de saída será definida como abaixo

```
<div>
    <app-
output (anunciador)="recebendoDados ($event)"></app-
output>
</div>
```

### Implementação Output Property

Agora, nesse exemplo, abordaremos como usar a propriedade de saída de qualquer componente. Para esse fim, crie um novo componente e nomeie-o como **output** dentro da pasta app do seu projeto – para criar este novo componente - via angular cli – é necessário usar o comando abaixo:

```
ng generate component output
```

Abra a pasta do componente e edite os arquivos conforme indicado abaixo:

### ***output.component.ts***

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

@Component({
  selector: 'app-output',
  templateUrl: './output.component.html',
  styleUrls: ['./output.component.css']
})
export class OutputComponent{
  // criar um "anunciador" do envetno que
  // enviará um dataset para o componente-pai
  @Output() anunciador = new EventEmitter<any>()

  // criar um objeto literal para receber o conjunto de
dados
  public objDados: any = {}

  // criar uma função para - usando o anunciador - enviar
dados
  public enviarDados(): void{
    this.anunciador.emit(this.objDados)
  }
}
```

### ***output.component.html***

```
<h2>Output Property</h2>
<div class = "content">
  <h2>Implementação Output Property</h2>
  <br />
  Informe seu nome: <input type = "text" [(ngModel)] =
"objDados.nome" />
  <br />
  Informe seu email: <input type = "email" [(ngModel)] =
"objDados.email" />
  <div class = "box">
    <input type = "button" value = "Enviar" (click) =
"enviarDados()" />
  </div>
</div>
```

Para melhorar a visualização dos componentes em tela, implemente este trecho de propriedades dentro do arquivo css do componente:

### ***output.component.css***

```
.box {
  display: flex;
  flex-direction: column;
  align-items: center;
}

input[type=text], input[type=email] {
  width: 100%;
  padding: 12px 20px;
  margin: 8px 0;
  display: inline-block;
  border: 1px solid #040670;
  box-sizing: border-box;
}

input[type="button"] {
  width: 100%;
  margin-top: 8px;
  padding: 12px 20px;
  background-color: #1976D2;
  color: #FFFFFF;
  font-weight: 700;
  border: none;
  border-radius: 4px;
  transition: 0.25s;
}

input[type="button"]:hover {
  background-color: #DD0031;
}
```

Agora, precisamos consumir esse evento no componente-pai, como mostrado abaixo:

### ***c-pai.component.ts***

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

@Component({
  selector: 'app-c-pai',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class CPaiComponent {
```

```
// criar função para receber os dados e vincular na view
do comp-pai

public recebendoDados(dadosRecebidos:any) {
  let receptora: string = 'Obrigado por se cadastrar ' +
dadosRecebidos.nome + '.' + '\n';

  receptora = receptora + 'O email ' +
dadosRecebidos.email + ' foi cadastrado com sucesso!'

  alert(receptora)
}

}
```

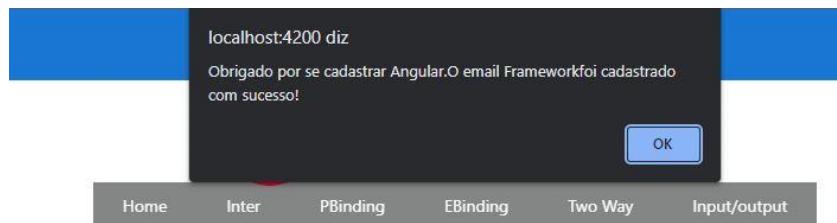
### **c-pai.component.html**

```
<div class = "content">
  <app-output (anunciador) =
"recebendoDados($event)"></app-output>
</div>
```

### **app.component.html**

```
<!-- estes são os links de navegação entre
componentes/telas-->
<nav>
  <a routerLink = "/home">Home</a>
  <a routerLink = "/interpolation">Inter</a>
  <a routerLink = "/p-binding">PBinding</a>
  <a routerLink = "/e-binding">EBinding</a>
  <a routerLink = "/two-way">Two Way</a>
  <a routerLink = "/input-output">Input/output</a>
</nav>
```

A saída é a seguinte:



### Implementação OutPut Property

Informe seu nome:

Informe seu email:

**Enviar**

## Diretivas

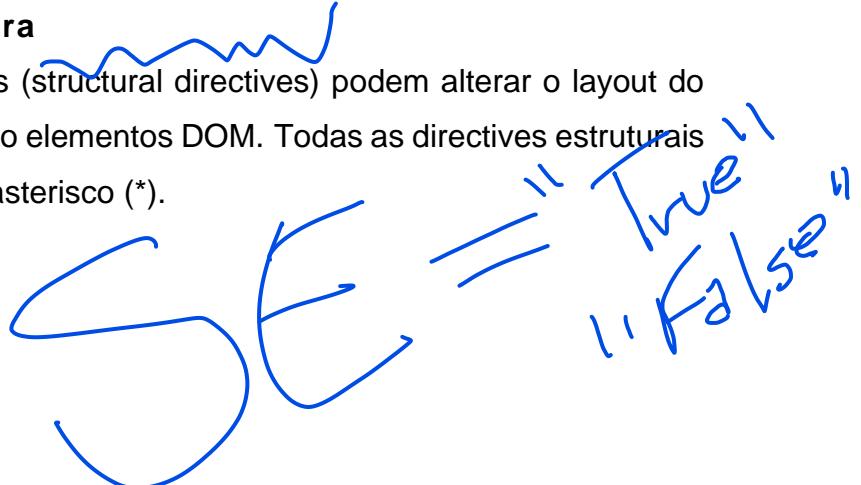
As *Directives Angular* proporcionam a manipulação do DOM. Fazendo das directives é possível alterar aparência, comportamento ou a estrutura do layout de um elemento DOM.

Existem três tipos de directives no Angular:

1. Diretivas de estrutura (Structural Directives)
2. Diretivas de atributo (Attribute Directives)
3. Diretivas de componentes ou customizadas (Directive Component)

### Diretivas de estrutura

As directives estruturais (structural directives) podem alterar o layout do DOM adicionando e removendo elementos DOM. Todas as directives estruturais são precedidas pelo símbolo asterisco (\*).



A diretiva `ngIf` é uma das diretivas Angular estrutural mais usadas; fazendo uso dessa diretiva é possível incluir/excluir elementos DOM considerando alguma condição para tomada de decisão. Seu uso parte da premissa de acoplamento da diretiva a um elemento DOM – pode ser adicionada a qualquer elemento DOM como `div`, `p`, `h1`, *seletor de componente*, entre outros. Como qualquer outra diretiva estrutural, é prefixado com \*(asterisco)

A descrição como se segue - `*ngIf` - é vinculada a uma expressão condicional. Dessa forma, a expressão é, então, avaliada pela diretiva. O retorno da verificação deve ser “`true`” ou “`false`”. Se a expressão for avaliada como `false`, o Angular remove o elemento inteiro do DOM. Se `true`, o elemento será inserido no DOM.

Usando o ***NOT lógico*** – caracter exclamação `()` - , é possível criar uma imitação da condição `else`. Observe o sippet abaixo:

```
<p *ngIf="!condicao">
    conteúdo exibido quando a condição for false
</p>
```

**condição** - aqui, a condição pode ser qualquer coisa; uma propriedade da classe do componente – por exemplo; também um método na classe do componente. Mas deve ser avaliado como `true/false`. A diretiva `ngIf` tentará forçar o valor para verificação booleana.

Considerando este cenário, a implementação adequada é usar o bloco `else` – como opção.

### **ngIf else**

`ngIf` possibilita a definição do bloco `else` - como opção - usando o ***ng-template***; a documentação oficial define `ng-template` como: “*a diretiva ng-template representa um template Angular: isso significa que o conteúdo desta tag irá conter parte de um template (estrutura de view), que pode então ser composto junto com outros templates para formar o template do componente final.*”

A expressão algorítmica começa com uma condição seguida por um ponto e vírgula. Na sequencia, observa-se a implementação da instrução `else`

acoplada a um template denominado *blocoElse*. O *template* pode ser definido em qualquer lugar do bloco de código acima, usando a diretiva *ng-template*. Uma boa prática é indica-lo logo após a implementação da diretiva *ngIf* facilitando, assim, a leitura do código implementado.

Quando a condição é avaliada como *false*, a diretiva *ng-template* com o nome *#blocoElse* é, então, processada pela diretiva *ngIf*.

### **ngIf then else**

É possível, também, definir o bloco *then else* usando o *ng-template*. Quando a condição é *true*, o template *blocoThen* é exibido. Caso a condição seja avaliada como *false*, o template *blocoElse* é exibido.

### **Atributo hidden**

Observe o snippet abaixo:

```
<p [hidden]="condição">
    Algum conteúdo a ser exibido
</p>
```

Considerando o uso da diretiva *ngIf* – em relação a propriedade *hidden* observa-se a seguinte definição: *nlf* não oculta o elemento DOM. Ele remove o elemento inteiro junto com sua sub-árvore; também remove o estado correspondente, liberando os recursos anexados ao elemento; o atributo *hidden* não remove o elemento do DOM, apenas esconde. Essa é a diferença entre *[hidden]='false'* e *\*ngIf='false'*; o primeiro método simplesmente oculta o elemento. O segundo método *ngIf* remove o elemento completamente do DOM.

### **Implementação**

Agora, crie um novo projeto chamado *directive* ou continue usando o projeto angular-alpha para implementar a diretiva *ngIf*.

Para esse fim, crie um novo componente e nomeie-o como ***ng-if*** dentro da pasta app do seu projeto – para criar este novo componente - via angular cli – é necessário usar o comando abaixo:

```
ng generate component ng-if
```

Abra a pasta do componente e edite os arquivos conforme indicado abaixo:

### ***ng-if.component.ts***

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

@Component({
  selector: 'app-ng-if',
  templateUrl: './ng-if.component.html',
  styleUrls: ['./ng-if.component.css']
})
export class NgIfComponent{
  // criando a propriedade para ser avaliada na view pela diretiva *ngIf
  exiba!: boolean
}
```

### ***ng-if.component.html***

```
<div class = "content">
  <h2>Implementação diretiva *ngIf</h2>
  <div>
    Propriedade "exiba": <input type = "checkbox"
[ (ngModel) ] = "exiba" />
  </div>
  <!-- uso direto da diretiva *ngIf-->
  <h2>Uso direto de *ngIf</h2>
  <p *ngIf = "exiba">
    Texto exibido quando o checkbox estiver "ativado" -
quando for TRUE
  </p>
  <p *ngIf = "!exiba">
    Texto exibido quando o checkbox estiver
"desativado" - quando for FALSE
  </p>

  <!-- usando *ngIf/else -->
  <h2>Usando *ngIf/else</h2>
  <p *ngIf = "exiba; else primeiroElse">
    Texto exibido quando o checkbox estiver "ativado" -
quando for TRUE
  </p>
  <!--contruindo a "miniview"-->
  <ng-template #primeiroElse>
    Texto exibido quando o checkbox estiver
"desativado" - quando for FALSE
  </ng-template>
```

```

</ng-template>

<!--*ngIf - then - else -->
<h2>Usando *ngIf - then - else</h2>
<p *ngIf = "exiba; then miniViewThen; else novoElse">
    -----
    -----
    </p>

    <!-- criando a mini-view then-->
    <ng-template #miniViewThen>
        Texto exibido quando o checkbox estiver "ativado" -
TRUE
    </ng-template>

    <!-- criando a mini-view else-->
    <ng-template #novoElse>
        Texto exibido quando o checkbox estiver
        "desativado" - FALSE
    </ng-template>

    <!--usando o atributo hidden-->
    <h2>Usando o atributo hidden</h2>
    <p [hidden] = "exiba">
        Texto exibido quando o checkbox estiver
        "desativado" - FALSE
    </p>
    <p [hidden] = "!exiba">
        Texto exibido quando o checkbox estiver "ativado" -
TRUE
    </p>

</div>

```



No arquivo `app.module.ts` é necessário importar o módulo de dependência `FormsModule`. Observe o código abaixo:

### **`app.module.ts`**

```

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { FormsModule } from '@angular/forms';

```



```
@NgModule ({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

### ***app.component.html***

```
<!-- estes são os links de navegação entre
componentes/telas-->
<nav>
  <a routerLink = "/home">Home</a>
  <a routerLink = "/interpolation">Inter</a>
  <a routerLink = "/p-binding">PBinding</a>
  <a routerLink = "/e-binding">EBinding</a>
  <a routerLink = "/two-way">Two Way</a>
  <a routerLink = "/input-output">Input/output</a>
  <a routerLink = "/ng-if">ng-if</a>
  <!--<a routerLink = "/ng-for">ng-for</a>
</nav>
```

O resultado no browser deve ser semelhante ao indicado abaixo. Com o checkbox desativado:

**Implementação diretiva \*ngIf**

Propriedade "exiba":

**Uso direto de \*ngIf**

Texto exibido quando o checkbox estiver "desativado" - quando for FALSE

**Usando \*ngIf/else**

Texto exibido quando o checkbox estiver "desativado" - quando for FALSE

**Usando \*ngIf - then - else**

Texto exibido quando o checkbox estiver "desativado" - FALSE

**Usando o atributo hidden**

Texto exibido quando o checkbox estiver "desativado" - FALSE

Agora, com o checkbox ativado:

**Implementação diretiva \*ngIf**

Propriedade "exiba":

**Uso direto de \*ngIf**

Texto exibido quando o checkbox estiver "ativado" - quando for TRUE

**Usando \*ngIf/else**

Texto exibido quando o checkbox estiver "ativado" - quando for TRUE

**Usando \*ngIf - then - else**

Texto exibido quando o checkbox estiver "ativado" - TRUE

**Usando o atributo hidden**

Texto exibido quando o checkbox estiver "ativado" - TRUE

Para observar o que ocorreu com a implementação do atributo `hidden` abra o console do browser e veja que o projeto Angular renderiza os dois elementos. Observe as imagens abaixo:

```
<h1 _ngcontent-bqi-c50>Usando o hidden</h1>
  <p _ngcontent-bqi-c50>
    " Fazendo uso da hidden property binding - este conteúdo será exibido quando a condição for "true" "
  </p>
  <p _ngcontent-bqi-c50 hidden>
    " Fazendo uso da hidden property binding - este conteúdo será exibido quando a condição for "false" "
  </p>
```

```
<h1 _ngcontent-bqi-c50>Usando o hidden</h1>
  <p _ngcontent-bqi-c50 hidden>
    " Fazendo uso da hidden property binding - este conteúdo será exibido quando a condição for "true" "
  </p>
  <p _ngcontent-bqi-c50>
    " Fazendo uso da hidden property binding - este conteúdo será exibido quando a condição for "false" "
  </p>
```

## ngFor

O `ngFor` é uma *structural directive Angular*, que repete uma parte do template HTML uma vez por cada item de uma lista iterável (coleção); observe o snippet abaixo:

```
<tr *ngFor="let item of items;">
  <td>{{ item.items }}</td>
</tr>
```

`<tr></tr>` - é o elemento sobre o qual a diretiva `ngFor` é aplicada. Este elemento `<tr></tr>` será repetido para cada item encontrado dentro da coleção.

`*ngFor` - é a sintaxe que indica o uso da diretiva. O elemento `*` (asterisco) representa sua sintaxe como diretiva estrutural.

`let <item>of<items>` - `item` é a variável auxiliar (de contagem/iteração) do template. Representa o elemento atualmente iterado da coleção `<items>`, que deve ser exibida ao usuário. Normalmente a coleção é array de elementos.

O escopo da variável `item` está dentro do `<td></td>`. Você pode acessá-lo em qualquer lugar dentro dele (`<td></td>`), mas não fora dele.

## Implementação

Agora, crie um novo projeto chamado directive ou continue usando o projeto angular-alpha para implementar a diretiva ngFor.

Para esse fim, crie um novo componente e nomeie-o como **ng-for** dentro da pasta app do seu projeto – para criar este novo componente - via angular cli – é necessário usar o comando abaixo:

***ng generate component ng-for***

Abra a pasta do componente e edite os arquivos conforme indicado abaixo:

***ng-for.component.ts***

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

@Component({
  selector: 'app-ng-for',
  templateUrl: './ng-for.component.html',
  styleUrls: ['./ng-for.component.css']
})
export class NgForComponent {
  // criar a propriedade com a expressão de uso da diretiva
  *ngFor
  subtitulo: string = '5 filmes sensacionais!'
  // criando o conjunto de dados
  filmes: Filme[] = [
    {titulo:'O poderoso Chefão', direcao:'Francis Ford
Coppola', elenco:'Marlon Brando, Al Pacino, James
Caan', anoLancamento:'10 de setembro, 1972'},
    { titulo: 'Um Estranho no Ninho', direcao: 'Milos
Forman', elenco: 'Jack Nicholson LouiseFletcher, Michael
Berryman', anoLancamento: '26 de maio, 1976' },
    { titulo: 'A lista de Schindler', direcao: 'Steven
Spielberg', elenco: 'Liam Neeson, Ralph Fiennes, Ben
Kingsley', anoLancamento: '11 de março, 1993' },
    { titulo: 'Forrest Gump - O contador de histórias',
direcao: 'Robert Zemeckis', elenco: 'Tom Hanks, Robin
Wright, Gary Sinise', anoLancamento: '7 de dezembro, 1994'
},
    { titulo: 'Laranja Mecânica', direcao: 'Stanley
Kubrik', elenco: 'Malcolm McDowell, Patrick Magee, Michael
Bates ', anoLancamento: '04 de setembro, 1971' }
  ]
}
```

```
// criando o model domain - para dar "formato" a dataset
class Filme {
  titulo!: string
  direcao!: string
  elenco!: string
  anoLancamento!: string
}
```

Para implementar a diretiva ngFor será necessário:

1. Criar um bloco de elementos HTML, que pode exibir um único filme.
2. Usar o ngFor para repetir o bloco para cada filme.

Navegue até o arquivo *app.component.html* e implemente o código abaixo – como se segue:

#### ***ng-for.component.html***

```
<div class = "content">
<h2>{{ subtítulo }}</h2>

<ul>
  <li *ngFor="let i of filmes">
    {{ i.titulo }} - {{ i.direcao }}
  </li>
</ul>
</div>
```

O elemento *<ul></ul>* foi usado para exibir os filmes. O elemento *</li></li>* exibe um único filme. É necessário, então, repetir o elemento *</li></li>* para cada filme. Portanto, aqui, se aplica a diretiva *\*ngFor* - no elemento *</li></li>*.

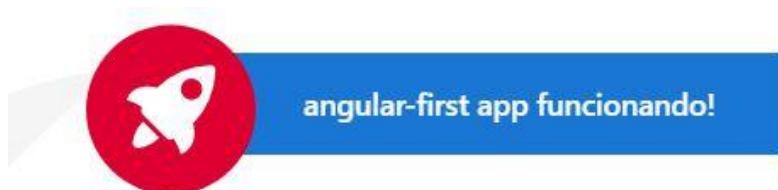
A instrução *let i of filmes* irá iterar sobre a coleção *filmes*, que é uma propriedade da classe do componente. A variável *i* é a variável auxiliar do *template*, que representa o filme iterado da coleção. Foi usada a vinculação de interpolação angular para exibir o título do filme e o nome do diretor.

#### ***app.componet.html***

```
<!-- estes são os links de navegação entre
componentes/telas-->
<nav>
  <a routerLink = "/home">Home</a>
```

```
<a routerLink = "/interpolation">Inter</a>
<a routerLink = "/p-binding">PBinding</a>
<a routerLink = "/e-binding">EBinding</a>
<a routerLink = "/two-way">Two Way</a>
<a routerLink = "/input-output">Input/output</a>
<a routerLink = "/ng-if">ng-if</a>
<a routerLink = "/ng-for">ng-for</a>
</nav>
```

Observe o resultado no browser:



- O poderoso Chefão - Francis Ford Coppola
- Um Estranho no Ninho - Milos Forman
- A lista de Schindler - Steven Spielberg
- Forrest Gump - O contador de histórias - Robert Zemeckis
- Laranja Mecânica - Stanley Kubrik

No console do browser, é possível observar que o projeto Angular gerou um elemento maracado para cada filme encontrado:

```

▼<ul _ngcontent-ksi-c16>
  ▼<li _ngcontent-ksi-c16>
    ::marker
    " O poderoso Chefão - Francis Ford Coppola "
  </li>
  ▼<li _ngcontent-ksi-c16> == $0
    ::marker
    " Um Estranho no Ninho - Milos Forman "
  </li>
  ▼<li _ngcontent-ksi-c16>
    ::marker
    " A lista de Schindler - Steven Spielberg "
  </li>
  ▼<li _ngcontent-ksi-c16>
    ::marker
    " Forrest Gump - O contador de histórias - Robert Zemeckis "
  </li>
  ▼<li _ngcontent-ksi-c16>
    ::marker
    " Laranja Mecânica - Stanley Kubrik "
  </li>
  <!--bindings={}
    "ng-reflect-ng-for-of": "[object Object],[object Object"
  }-->
</ul>

```

Da mesma forma, é possível usar o elemento *table* para exibir os filmes. Aqui, será necessário repetir o elemento *tr* para cada filme. Portanto, basta aplicar a diretiva *\*ngFor* sobre o elemento *tr*. Observe o código abaixo e implemente-o como se segue – o arquivo *app.component.ts* não sofre alterações neste passo:

### ***ng-for.component.html***

```

<div class = "content">
  <h2>{{ subtítulo }}</h2>

  <table class = "bordaTabela">
    <thead>
      <tr>
        <th>Titulo</th>
        <th>Direção</th>
        <th>Elenco</th>
        <th>Ano de Lançamento</th>
      </tr>
    </thead>
    <tbody>
      <tr *ngFor = "let x of filmes">
        <td>{{ x.titulo }}</td>
        <td>{{ x.direcao }}</td>
        <td>{{ x.elenco }}</td>
    
```

```

        <td>{ x.anoLancamento }</td>
    </tr>
</tbody>
</table>

</div>

```

Para melhorar o visual da tabela, insira o código abaixo dentro do arquivo **styles.css**:

### **styles.css**

```

/* estilos da table para componentes que usurão os
atributos indicados abaixo*/
.bordaTabela {
    font-family: Arial, Helvetica, sans-serif;
    border-collapse: collapse;
    width: 100%;
}
.bordaTabela td, .bordaTabela th {
    border: 1px solid #ddd;
    padding: 8px;
}
.bordaTabela tr:nth-child(even){background-color:
#f2f2f2;}
.bordaTabela tr:hover {background-color: #ddd;}
.bordaTabela th {
    padding-top: 12px;
    padding-bottom: 12px;
    text-align: left;
    background-color: #581845;
    color: white;
}

```

Execute o projeto e observe o resultado:

#### 5 FILMES SENSACIONAIS!

Título	Direção	Elenco	Ano de Lançamento
O poderoso Chefão	Francis Ford Coppola	Marlon Brando, Al Pacino, James Caan	10 de setembro, 1972
Um Estranho no Ninho	Milos Forman	Jack Nicholson, Louise Fletcher, Michael Berryman	26 de maio, 1976
A lista de Schindler	Steven Spielberg	Liam Neeson, Ralph Fiennes, Ben Kingsley	11 de março, 1993
Forrest Gump - O contador de histórias	Robert Zemeckis	Tom Hanks, Robin Wright, Gary Sinise	7 de dezembro, 1994
Laranja Mecânica	Stanley Kubrick	Malcolm McDowell, Patrick Magee, Michael Bates	04 de setembro, 1971

## Diretivas de atributo

Uma attribute directive ou estilo pode alterar a aparência ou o comportamento de um elemento.

### **ngClass**

A diretiva *ngClass* é usada para adicionar ou remover as classes CSS de um elemento HTML. Usar *ngClass* pode criar estilos dinâmicos em páginas HTML.

### Implementação

Agora, crie um novo projeto chamado *directive* ou continue usando o projeto angular-alpha para implementar a diretiva *ngClass*.

Para esse fim, crie um novo componente e nomeie-o como ***ng-class*** dentro da pasta app do seu projeto – para criar este novo componente - via angular cli – é necessário usar o comando abaixo:

***ng generate component ng-class***

Abra seu projeto e, no arquivo *ng-class.component.ts*, crie uma variável chamada *cssalteradoViaVar* – essa variável será inicializada com duas propriedades ‘color’ e ‘size’. Além de criar essa variável, fora da classe principal – AppComponent – crie uma nova classe chamada *UmaClasseCss* e, na sequencia, crie sua instância dentro da classe principal. Observe o código abaixo e implemente-o como se segue:

### ***ng-class.component.ts***

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

@Component({
  selector: 'app-ng-class',
  templateUrl: './ng-class.component.html',
  styleUrls: ['./ng-class.component.css']
})
export class NgClassComponent implements OnInit {

  constructor() { }

  ngOnInit(): void {
  }
}
```

```

}

tituloComponente: string = 'Implementação NgClass'

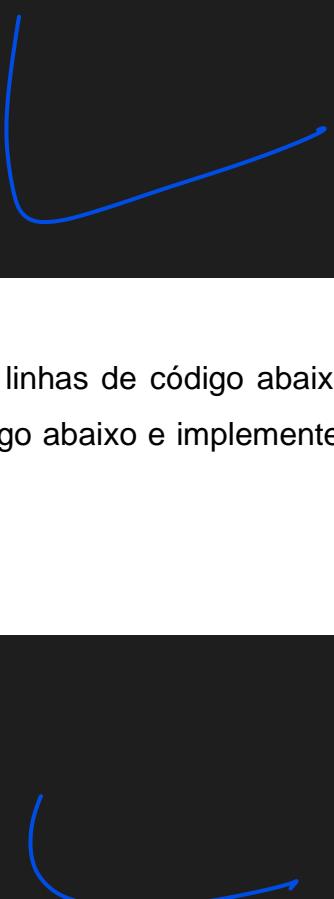
// criar uma propriedade que será vinculada à view -
posteriormente
estiloAlterViaVar: string = 'color size'

// patricular a instância da classe PropriedadesCss
objCSS: PropriedadesCss = new PropriedadesCss()

}

// criando uma classe externa
class PropriedadesCss{
  color: boolean = true
  size: boolean = true
}

```



Agora, é necessário implementar as linhas de código abaixo dentro do arquivo .css do componente. Observe o código abaixo e implemente-o como se segue:

#### ***ng-class.component.css***

```

/*estilos para ngClass*/
.color{
  color: #6e0707;
}
.size{
  font-size: 35px;
}

```



Implemento o código abaixo dentro do arquivo *app.component.html* como se segue:

#### ***ng-class.component.html***

```

<div class = "content">
  <h2>Implementação ngClass</h2>
  <div [ngClass] = "'color size'">
    <p>

```



```

Cor e fonte alterados pelas propriedades CSS -
descritas no arquivo .css - com atribuição de propriedade
de string.
</p>
</div>

<div [ngClass] = "cssAlteradoViaVar">
  <p>
    Cor e fonte alterados pelas propriedades CSS -
    descritas no arquivo TS - com atribuição de propriedade
    (property binding - uso direto)
  </p>
</div>

<div [ngClass] = "objCSS">
  <p>
    Cor e fonte alterados pelas propriedades CSS -
    descritas no arquivo TS - atribuição de propriedade de
    instancia da PropsCss (property binding)
  </p>
</div>
</div>
```

Acima, é possível observar as diferentes formas de implementação das propriedades de estilo configuradas no arquivo .css. A diretiva `ngClass` é aplicada em todas as modalidades com diferentes implementações.

### ***app.component.html***

```

<!-- estes são os links de navegação entre
componentes/telas-->
<nav>
  <a routerLink = "/home">Home</a>
  <a routerLink = "/interpolation">Inter</a>
  <a routerLink = "/p-binding">PBinding</a>
  <a routerLink = "/e-binding">EBinding</a>
  <a routerLink = "/two-way">Two Way</a>
  <a routerLink = "/input-output">Input/output</a>
  <a routerLink = "/ng-if">ng-if</a>
  <a routerLink = "/ng-for">ng-for</a>
  <a routerLink = "/ng-class">ng-class</a>
</nav>
```



Salve o projeto. Observe o resultado no browser. Deve ser semelhante ao indicado abaixo:

### Implementação NgClass

Cor e fonte alterados pelas CSS - descritas no arquivo .css do componente - com atribuição de propriedades, aqui, definidas como string.

Cor e fonte alterados pelas propriedades CSS - descritas no arquivo TS. Com atribuição de propriedade (property binding) de uso e vinculação diretos.

Cor e fonte alterados pelas propriedades CSS - descritas no TS. Com atribuição de instância do objeto gerado a partir da classe PropriedadesCss()

### ngStyle

ngStyle é usado para alterar as várias propriedades de estilo de nossos elementos HTML. Também podemos vincular essas propriedades a valores que podem ser atualizados pelo usuário ou por nossos componentes. Observe o snippet abaixo:

```
<div [ngStyle]="{'color': 'blue', 'font-size': '24px',
'font-weight': 'bold'}">
    Algum texto
</div>
```

### Implementação

Agora, crie um novo projeto chamado *directive* ou continue usando o projeto angular-alpha para implementar a diretiva *ngStyle*.

Para esse fim, crie um novo componente e nomeie-o como ***ng-style*** dentro da pasta app do seu projeto – para criar este novo componente - via angular cli – é necessário usar o comando abaixo:

***ng generate component ng-style***

Abra seu projeto e, no arquivo `app.component.ts`, crie três variáveis: tamanho, color, estiloClasse – essas variáveis serão inicializadas com os valores indicados no código abaixo. Além de criar as variáveis, fora da classe principal – AppComponent – crie uma nova classe chamada EstiloClasse, na sequência, crie sua instância dentro da classe principal. Observe o código abaixo e implemente-o como segue:

### ***ng-style.component.ts***

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

@Component({
  selector: 'app-ng-style',
  templateUrl: './ng-style.component.html',
  styleUrls: ['./ng-style.component.css']
})
export class NgStyleComponent{

  tituloComp: string = 'Implementação ngStyle'

  // criando as propriedades para vincular via ngStyle
  cores: string = '#380F6C'
  tamanho: number = 12
  // instancia da classe
  objStyle: EstiloCss = new EstiloCss()

}

// criando a classe para vincular a diretiva ngStyle
class EstiloCss{
  'color': string = '#33C9FF'
  'font-size.%': number = 80
  'font-weight': string = 'bold'
}
```

Agora, implemento o código abaixo – como se segue – dentro do arquivo `app.component.html`

### ***ng-style.component.html***

```
<div class = "content">
  <h2>{{ tituloComp }}</h2>
  <br /><br />
  <!-- primeira implementação da diretiva-->
  <input type = "color" [(ngModel)] = "cores" />
  <div [ngStyle] = "{ 'color': cores}">
```

```

        Mudar a cor do texto!
</div>
<br />

<!-- segunda implementação da diretiva -->
<input type = "number" [(ngModel)] = "tamanho" />
<div [ngStyle] = "{ 'font-size.px': tamanho }">
    Mude o tamanho deste texto!
</div>
<br />

<!-- terceira implementação da diretiva -->
<div [ngStyle] = "objStyle">
    Muda a cor e o tamanho da fonte do texto - via
propriedade de instância de classe - descrita no arquivo TS
</div>
</div>
```

Acima, o código demonstra as diferentes implementações da diretiva `ngStyle`. Dessa forma é possível alterar o estilo dos componentes do projeto Angular.

### ***app.component.html***

```

<!-- estes são os links de navegação entre
componentes/telas-->
<nav>
    <a routerLink = "/home">Home</a>
    <a routerLink = "/interpolation">Inter</a>
    <a routerLink = "/p-binding">PBinding</a>
    <a routerLink = "/e-binding">EBinding</a>
    <a routerLink = "/two-way">Two Way</a>
    <a routerLink = "/input-output">Input/output</a>
    <a routerLink = "/ng-if">ng-if</a>
    <a routerLink = "/ng-for">ng-for</a>
    <a routerLink = "/ng-class">ng-class</a>
    <a routerLink = "/ng-style">ng-style</a>
</nav>
```

Salve o projeto e execute. O resultado no browser deve ser semelhante ao indicado abaixo:



## Diretivas de componente

Componentes (Components) são directives especiais em Angular. Eles são as directives com um template (exibição)

### Criando directives personalizadas

É possível criar directives personalizadas no Angular. O processo é criar uma classe JavaScript e aplicar o atributo **@Directive** a essa classe. Você pode escrever o comportamento desejado na classe.

### Como criar directives personalizadas?

Diretivas personalizadas são criadas por nodes e não são padrão.

Vamos ver como criar a diretiva personalizada. Criaremos a diretiva usando a linha de comando. O comando para criar a diretiva usando a linha de comando é o seguinte:

```
ng g directive nomeadirectiva
→ ng g directive alterar-texto
```

Ele aparece na linha de comando, conforme indicado no código abaixo:

```
Projetos-Angular\angular-alpha>ng g directive alterar-
texto
CREATE src/app/alterar-texto.directive.spec.ts (249
bytes)
CREATE src/app/alterar-texto.directive.ts (153 bytes)
UPDATE src/app/app.module.ts (487 bytes)
```

```

CREATE src/app/alterar-texto.directive.spec.ts (249 bytes)
CREATE src/app/alterar-texto.directive.ts (153 bytes)
UPDATE src/app/app.module.ts (487 bytes)

```

Os arquivos acima, ou seja, `alterar-texto.directive.spec.ts` e `alterar.directive.ts`, são criados e o arquivo `app.module.ts` é atualizado.

### **app.module.ts**

```

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { AlterarTextoDirective } from './alterar-
texto.directive';

@NgModule({
  declarations: [
    AppComponent,
    AlterarTextoDirective
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

A classe `AlterarTextoDirective` está incluída nas declarações do arquivo acima. A classe também é importada do arquivo fornecido abaixo:

### **alterar-texto.directive.ts**

```

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

@Directive({
  selector: '[appAlterarTexto]'
})
export class AlterarTextoDirective {

  constructor() { }

}

```

O arquivo acima tem uma diretiva e também possui uma propriedade seletora. Tudo aquilo que for definido na propriedade `selector` deve corresponder na `view` onde a diretiva customizada for atribuida

Na view `app.component.html`, adicione a diretiva como se segue abaixo:

### **`app.component.html`**

```
<h1 appAlterarTexto></h1>
```

Abaixo, serão implementadas as alterações `alterar-texto.directive.ts`. Implemente o código como se segue:

### **`alterar-texto.directive.ts`**

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

@Directive({
  selector: '[appAlterarTexto]'
})
export class AlterarTextoDirective {

  constructor(elementoHTML: ElementRef) {
    console.log(elementoHTML);
    elementoHTML.nativeElement.innerText = "Este é o texto inserido a partir da diretiva customizada!";
  }
}
```

No arquivo acima, a classe principal se chama `AlterarTextoDirective` e um construtor, que utiliza o elemento do tipo `ElementRef` - obrigatório. O elemento possui todos os detalhes aos quais a diretiva `appAlterarTexto` é aplicada.

Ao adicionar a instrução `console.log` a saída de `ElementRef` pode ser observada no console do navegador. O texto atribuído a `Element` é inserido.

Agora, o navegador mostrará o seguinte:

```

    ElementRef {
      nativeElement: h1 {
        accessKey: ""
        align: ""
        ariaAtomic: null
        ariaAutoComplete: null
        ariaBusy: null
        ariaChecked: null
        ariaColCount: null
        ariaColIndex: null
      }
    }
  
```

Acima, observa-se todas as propriedades do elemento nativo html para o qual o seletor da diretiva – `appAlterarTexto` – foi referenciado. Como a diretiva `appAlterarTexto` foi adicionada a uma tag `<h1></h1>`, os detalhes do elemento estão disposto no console do browser.

## Pipes

Neste próximo passo será possível observar o trabalho com *pipes* Angular. Os pipes foram anteriormente chamados de filtros - na primeira versão do angular - e renomeados para pipes – a partir da segunda versão.

O caractere | é usado para aplicar transformações em dados. Observe a sintaxe de um pipe *built-in*:

```
 {{ Olá Angular | lowercase }}
```

É possível trabalhar com pipe aplicando qualquer tipo de dado -por exemplo: números inteiros, sequências de caracteres, matrizes, entre outros. Usa-se o caractere | (barra vertical) para converter no formato conforme necessário e exibi-lo no navegador.

Aqui, o objetivo exibir o texto fornecido em maiúsculas. Isso pode ser feito usando pipes da seguinte maneira:

Aqui estão alguns pipes embutidos disponíveis com angular:

- Lowercasepipe
- Uppercasepipe
- Datepipe
- Currencypipe

- Jsonpipe
- Percentpipe
- Decimalpipe
- Slicepipe

Agora, observe a aplicação do pipe.

Para esse fim, crie um novo componente e nomeie-o como **pipe** dentro da pasta app do seu projeto – para criar este novo componente - via angular cli – é necessário usar o comando abaixo:

**ng generate component pipe**

Abra a pasta do componente e edite os arquivos conforme indicado abaixo. As seguintes linhas de código nos ajudará a definir as variáveis necessárias no arquivo *pipe.component.ts*:

### **pipe.component.ts**

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

@Component({
  selector: 'app-pipe',
  templateUrl: './pipe.component.html',
  styleUrls: ['./pipe.component.css']
})
export class PipeComponent {

  novoTitulo: string = 'Implementação custom Pipe'

  // implmentação das propriedades Pipe
  numFloat: number = 678.94
  dataHoje: Date = new Date()
  numDecimal: number = 7893547.9563
  objLiteral = {
    nome: 'Dexter',
    endereco:{
      rua: 'Rua da Casinha',
      numero: '1'
    }
  }

  mesesAno: Array<string> = ['Jan', 'Fev', 'Mar', 'Abr',
  'Mai', 'Jun', 'Jul', 'Ago', 'Set', 'Out', 'Nov', 'Dez']
}
```

```

    numPercent: number = 0.29
}

```

Os pipes serão implementados no arquivo *pipe.component.html*, como mostrado abaixo:

### ***pipe.component.html***

```

<div class = "content">
    {{ novoTitulo }}
</div>
<table class = "bordaTabela">
    <thead>
        <tr>
            <th>UpperCase Pipe</th>
            <th>LowerCase Pipe</th>
            <th>Currency Pipe - pipe usado para conversão de moedas</th>
            <th>Date Pipe - pipe usado para conversão de datas</th>
            <th>Decimal Pipe - formatação de dados numéricos a partir de casas decimais</th>
            <th>Json Pipe - transforma um objeto literal em json format</th>
            <th>Slice Pipe - "fatia" um array de elementos na view</th>
            <th>Percent Pipe - transformação de valor em escala de porcentagem</th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <td>{{ tituloComp | uppercase }}</td>
            <td>{{ umTexto | lowercase }}</td>
            <td>
                {{ numFloat | currency: "USD" }}
                <br />
                {{ numFloat | currency: "BRL" }}
            </td>
            <td>
                {{ dataHoje | date: "d/M/y" }}
                <br />
                {{ dataHoje | date: "shortTime" }}
            </td>
        </tr>
    </tbody>
</table>

```

```

<td>{{ numDecimal | number: "3.4-6" }}</td>
<td>{{ objLiteralDados | json }}</td>
<td>{{ mesesAno | slice: 2:6 }}</td>
<td>{{ numPercent | percent }}</td>
</tr>
</tbody>
</table>
```

Para que o uso pipe aplicado a conversão de moedas – neste cenário, convertendo o número para a moeda BRL (real brasileiro) – deve ser implementado no arquivo *app.module.ts* algumas dependências Angular. Implemente o código abaixo como indicado:

### ***app.module.ts***

```

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { AlterarTextoDirective } from './alterar-
texto.directive';

import { LOCALE_ID, DEFAULT_CURRENCY_CODE } from '@angular/co
re';
import localePt from '@angular/common/locales/pt';
import { registerLocaleData } from '@angular/common';

registerLocaleData(localePt, 'pt');

@NgModule({
  declarations: [
    AppComponent,
    AlterarTextoDirective,
    ComponenteFilhoComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [ {
    provide: LOCALE_ID,
    useValue: 'pt'
  },
  {
    provide: DEFAULT_CURRENCY_CODE,
```

```

        useValue: 'BRL'
    } , ],
bootstrap: [AppComponent]
)
export class AppModule { }

```

### **app.component.html**

```

<!-- estes são os links de navegação entre
componentes/telas-->
<nav>
    <a routerLink = "/home">Home</a>
    <a routerLink = "/interpolation">Inter</a>
    <a routerLink = "/p-binding">PBinding</a>
    <a routerLink = "/e-binding">EBinding</a>
    <a routerLink = "/two-way">Two Way</a>
    <a routerLink = "/input-output">Input/output</a>
    <a routerLink = "/ng-if">ng-if</a>
    <a routerLink = "/ng-for">ng-for</a>
    <a routerLink = "/ng-class">ng-class</a>
    <a routerLink = "/ng-style">ng-style</a>
    <a routerLink = "/pipe">pipe</a>
</nav>

```

A captura de tela a seguir mostram a saída para cada pipe:



UpperCase Pipe	LowerCase Pipe	Currency Pipe - Pipe para conversão de moedas	Date Pipe - Pipe para conversão de datas	Decimal Pipe - Formatação a partir de casas numéricas
ANGULAR-ALPHA	angular-alpha	US\$ 678,94 R\$ 678,94	31/1/2022 12:06	7.893.547,9563

### **Pipe personalizado**

Para criar um pipe personalizado, será necessário criar um novo arquivo `.ts`. Com este pipe personalizado será possível calcular a raiz quadrada de algum número fornecido e lido pelo pipe. Para criar o pipe, clique com o botão direito do mouse em cima da pasta `app` e escolha *New File*; nomeie-o como `pipe-raiz-quadrada.ts`. Na sequencia, abra o arquivo recém-criado e implemente o código abaixo como indicado:

→ `new file > pipe-raiz-quadrada.ts`

### **pipe-raiz-quadrada.ts**

```
import {Pipe, PipeTransform} from '@angular/core';
```

```

@Pipe ({
  name : 'raizquadrada'
})
export class RaizQuadrada implements PipeTransform {
  transform(numero : number) : number {
    return Math.sqrt(numero);
  }
}

```

Para criar um pipe personalizado, foi necessário importar a dependência *Pipe* e *PipeTransform* de `@angular/core`. Na diretiva `@Pipe`, foi indicado um nome para o pipe, que será referenciado dentro do arquivo `app.component.html`.

Na sequencia foi criada a classe `RaizQuadrada`. Esta classe implementará o *PipeTransform*.

O método de `transform()` definido na classe terá o argumento *número* e retornará o valor depois de obter a raiz quadrada.

Agora que o pipe foi criado e implementado, será necessário adicioná-lo e registrá-lo no arquivo `app.module.ts`. Isso é feito da seguinte maneira:

### **app.module.ts**

```

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { AlterarTextoDirective } from './alterar-
texto.directive';
import { ComponenteFilhoComponent } from './componente-
filho/componente-filho.component';

import { RaizQuadrada } from './pipe-raiz-quadrada';

@NgModule({
  declarations: [
    AppComponent,
    AlterarTextoDirective,
    ComponenteFilhoComponent,
    RaizQuadrada
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
}

```

```

    providers: [],
    bootstrap: [AppComponent]
})
export class AppModule { }

```

No arquivo *pipe.component.ts* é necessário criar as novas propriedade indicadas abaixo:

### ***pipe.component.ts***

```

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

@Component({
  selector: 'app-pipe',
  templateUrl: './pipe.component.html',
  styleUrls: ['./pipe.component.css']
})
export class AppComponent{

  // propriedades do Custom Pipe
  raizUm: number = 25
  raizDois: number = 150
  raizTres: number = 1458

}

```

Agora, implemente a chamada feita para o *pipe* raiz-quadrada no arquivo *pipe.component.html*.

### ***pipe.component.html***

```

<div class = "content">
  <h2>Implementação de Pipes</h2>

  <table class = "bordaTabela">
    <thead>
      <tr>
        <th>Raiz Um</th>
        <th>Raiz Dois</th>
        <th>Raiz Tres</th>
      </tr>
    </thead>
    <tbody>
      <tr>
        <td>
          <b>Raiz quadrada de {{ raizUm }} é:</b>
          <br />
          {{ raizUm | pipeRaizQuadrada }}
        </td>
      </tr>
    </tbody>
  </table>
</div>

```

```

        <td>
            <b>Raiz quadrada de {{ raizDois }}</b>
é:</b>
        <br />
        {{ raizDois | pipeRaizQuadrada }}
    </td>
    <td>
        <b>Raiz quadrada de {{ raizTres }}</b>
é:</b>
        <br />
        {{ raizTres | pipeRaizQuadrada }}
    </td>
</tr>
</tbody>
</table>
</div>

```

A seguir está a saída:

Raiz Um	Raiz Dois	Raiz Tres
Raiz quadrada de 25 é: 5	Raiz quadrada de 150 é: 12.24744871391589	Raiz quadrada de 1458 é: 38.18376618407357

## Forms

Neste passo, veremos como os formulários são usados no Angular. Discutiremos duas maneiras de trabalhar com formulários:

- Template Driven Form
- Model Driven Form

### Template Driven Form

Com um Template Driven Form (formulário orientado a modelo), a maior parte do trabalho é feita no *template*. Com o formulário controlado pelo *model*, a maior parte do trabalho é realizada na classe de componentes (camada lógica).

Vamos, agora, trabalhar com formulário orientado *template*.

Para esse fim, crie um novo componente e nomeie-o como **formulario** dentro da pasta app do seu projeto – para criar este novo componente - via angular cli – é necessário usar o comando abaixo:

```
ng generate component formulario
```

Abra a pasta do componente e edite os arquivos conforme indicado abaixo. Será implementado o código para a criação de um formulário simples. Neste formulário será adicionado um campo para inserção de um endereço de email, a senha e o botão enviar estes dados. Para começar, será necessário importar o módulo de dependencia *FormsModule* a partir de `@angular/forms`, o que é feito em *app.module.ts* da seguinte maneira:

### ***app.module.ts***

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
import { FormsModule } from '@angular/forms';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    BrowserAnimationsModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Portanto, em *app.module.ts*, importamos o *FormsModule* e seu registro é adicionado ao array *imports*, conforme mostrado no código destacado.

Vamos agora criar o formulário no arquivo *app.component.html*

### ***app.component.html***

```
<div class = "content">
  <form #capturaDados = "ngForm" (ngSubmit) =
"recebedorDados(capturaDados.value)">
    <input type = "email" name = "email" placeholder =
"Informe seu email" ngModel />
    <br /><br />
    <input type = "password" name = "senha" placeholder =
"Informe sua senha" ngModel/>
```

```

        <br /><br />
        <input type = "submit" value = "Enviar Dados" />
    </form>
</div>

```

Criamos um formulário simples com tags de input para email, senha e o botão enviar.

Nos formulários controlados por *template*, é necessário criar os controles do formulário adicionando a diretiva *ngModel* e o atributo *name*. Portanto, onde quisermos que o Angular acesse os dados a partir de formulários, basta adicionar a diretiva *ngModel* a essa tag, como mostrado acima. Agora, se for necessário ler os atributos *email* e *senha*, será preciso adicionar o *ngModel*.

Observando o código, é possível constatar que a diretiva *ngForm* ao elemento identificador *#capturarDados*. A diretiva *ngForm* precisa ser adicionada ao *template*. Também foi vinculada a função *recebedorDados* (que será *implementada no arquivo formulario.component.ts*) e foi atribuído o valor *capturaDados.value* a ela.

Agora, é preciso criar a função *recebedorDados* em no *formulario.component.ts* para que os valores inseridos na view sejam devidamente recebidos na camada lógica do componente. Implemente o código abaixo – como indicado:

#### **formulario.component.ts**

```

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

@Component({
  selector: 'app-formulario',
  templateUrl: './formulario.component.html',
  styleUrls: ['./formulario.component.css']
})
export class FormularioComponent{

  // criando a função para receber os dados
  recebedorDados(dadosRecebidos:any) {
    alert('O email recebido foi: ' + dadosRecebidos.email)
  }

}

```

No arquivo *formulario.component.ts*, acima, a função *recebedorDados* foi definida. Quando o usuário clicar no botão de envio do formulário, o controle chegará à função acima.

O css do formulário é adicionado em *formulario.component.css*:

#### **formulario.component.css**

```
input[type = email], input[type = password] {
```

```

        width: 100%;
        padding: 12px 20px;
        margin: 8px 0;
        display: inline-block;
        border: 1px solid #B3A9A9;
        box-sizing: border-box;
    }
    input[type = submit] {
        width: 100%;
        padding: 12px 20px;
        margin: 8px 0;
        display: inline-block;
        border: 1px solid #B3A9A9;
        box-sizing: border-box;
    }
}

```

### **app.component.html**

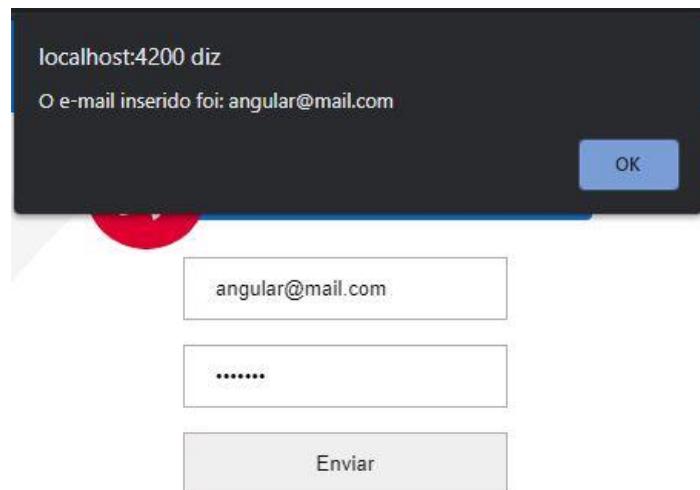
```

<!-- estes são os links de navegação entre
componentes/telas-->
<nav>
    <a routerLink = "/home">Home</a>
    <a routerLink = "/interpolation">Inter</a>
    <a routerLink = "/p-binding">PBinding</a>
    <a routerLink = "/e-binding">EBinding</a>
    <a routerLink = "/two-way">Two Way</a>
    <a routerLink = "/input-output">Input/output</a>
    <a routerLink = "/ng-if">ng-if</a>
    <a routerLink = "/ng-for">ng-for</a>
    <a routerLink = "/ng-class">ng-class</a>
    <a routerLink = "/ng-style">ng-style</a>
    <a routerLink = "/pipe">pipe</a>
    <a routerLink = "/formulario">formulario</a>
</nav>

```

O navegador exibirá nossa tela como mostrado abaixo:

Inserindo os dados no formulário, na função de envio, o email é exibido na janela de alerta, como mostrado abaixo:



## Model Driven Form

No Model Driven Form (formulário controlado pela camada lógica do componente), é necessário importar a dependencia `ReactiveFormsModule` de `@angular/forms` e registrá-lo no array `imports`:

Há uma alteração em `app.module.ts`. Observe e implemente como se segue:

### `app.module.ts`

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    BrowserAnimationsModule,
    FormsModule,
    ReactiveFormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
```

```
export class AppModule { }
```

Pode ser usado o mesmo componente já criado para template driven form – basta desabilitar ou excluir os códigos anteriormente implementados.

No arquivo *formulario.component.ts*, será preciso importar alguns módulos para o formulário controlado pela camada lógica do componente - por exemplo, *import {FormGroup, FormControl} from '@angular/forms'*. Observe o código abaixo e implemente-o como se segue:

### ***formulario.component.ts***

```
import { Component } from '@angular/core';
import { FormGroup, FormControl } from '@angular/forms';

@Component({
  selector: 'app-formulario',
  templateUrl: './formulario.component.html',
  styleUrls: ['./formulario.component.css']
})
export class FormularioComponent{

  // criando as propriedades para controlar o formulario
  dadosForm: any
  propEmail: any

  // chamando o hook para priorizar a instancia da classe
  ngOnInit() {
    this.dadosForm = new FormGroup({
      email: new FormControl('angular@mail.com')

      senha: new FormControl('@#$%')
    })
  }
  // criar uma função para exibir o resultado do "controle"
  // do form pela camada lógica
  exibidoraDados(umDado:any){
    this.propEmail = umDado.email
  }
}
}
```

As propriedades que serão usadas para operar com o formulário são indicadas no início da classe e, posteriormente, inicializadas com *FormGroup*, como mostrado acima. As variáveis *email* e *senha* são inicializadas com definição de tipos *any* e alguns valores iniciais atribuídos à elas. Somente para observação do comportamento do componente essas variáveis recebem valores iniciais – também é possível deixá-las sem qualquer atribuição de valor.

Será usada a propriedade `dadosForm` para inicializar os valores do formulário na view; será necessário usar o mesmo no formato no arquivo **`formulario.component.html`**.

### **`formulario.component.html`**

```
<div class = "content">
    <form [formGroup] = "dadosForm" (ngSubmit) =
"exibidoraDados(dadosForm.value)">
        <input type = "email" name = "email" placeholder =
"Informe seu email" formControlName = "email" />
        <br /><br />
        <input type = "password" name = "senha" placeholder =
"Informe sua senha" formControlName = "senha" />
        <br /><br />
        <input type = "submit" value = "Enviar Dados" />
    </form>
    <!--exibir o valor da propriedade email-->
    <p>O email informado é: {{ propEmail }}</p>
</div>
```

É dessa forma, conforme indicado abaixo, que os valores serão vistos no formulário da interface do usuário.

angular@mail.com

\*\*\*\*\*

Enviar

O email informado foi :

angular@mail.com

\*\*\*\*\*

Enviar

O email informado foi : angular@mail.com

No arquivo `formulario.component.html`, foi usado `formGroup` entre colchetes para o formulário - `[formGroup] = "dadosForm"`. No envio, a função é chamada `exibidoraDados` para a qual `dadosForm.value` é passado.

A marcação de entrada `FormControlName` também foi usada. À este input foi dado um valor implementado no arquivo `formulario.component.ts`.

Ao clicar em enviar, o controle passará para a função `exibidoraDados`, definida no arquivo `formulario.component.ts`.

## Validação de formulário

É possível usar a validação de formulário *built-in* (nativa Angular) ou também a abordagem de validação personalizada. Serão observadas e implementadas as duas abordagens. Ainda seguindo com o mesmo componente para este estudo.

Neste passo, será necessário importar a dependência `validators (validadores)` de `@angular/forms` como mostrado abaixo:

```
import { FormGroup, FormControl, Validators } from
'@angular/forms'
```

Angular Framework oferece alguns validadores interessantes tais como: *campo de preenchimento obrigatório* (`require validator`), também `minlength`, `maxlength` entre outros .

É possível apenas adicionar validadores ou um array de validadores necessários para informar ao Angular se um determinado campo é obrigatório. Agora, o objetivo é implementar validadores em um dos `inputs`, ou seja, a caixa para inserção de email. Para email, serão adicionados os seguintes parâmetros de validação:

- `required` e correspondência de padrões (**Validators.pattern**)

Observe o código abaixo e implemente-o como indicado:

### **formulario.component.ts**

```
import { Component, OnInit } from '@angular/core';
import { FormGroup, FormControl, Validators } from
'@angular/forms';

@Component({
  selector: 'app-formulario',
  templateUrl: './formulario.component.html',
  styleUrls: ['./formulario.component.css']
})
export class FormularioComponent implements OnInit {
  // criando propriedades para auxiliar no controle do form
  dadosForm: any
  propEmail: any
  // chamando o hook para priorizar a instancia de classe
  ngOnInit(): void {
    this.dadosForm = new FormGroup({
```

```

        email: new FormControl(' ', Validators.compose([
            Validators.required,
            Validators.pattern('[^ @]*@[^ @]*')
        ]),
        senha: new FormControl(' ')
    )
}

// criar uma função para exibir o resultado do "controle"
do form pela camada lógica
exibidoraDados(umDado:any){
    this.propEmail = umDado.email
}
}

```

A partir do elemento `Validators.compose`, é possível adicionar a lista de itens que se deseja validar no campo de entrada. Até aqui, foram adicionados os parâmetros **`required`** e de **`Validators.pattern`** para receber apenas emails válidos.

No arquivo `formulario.component.html` o botão enviar será desativado se alguma das entradas do formulário não for válida. Isso é feito da seguinte maneira:

### **`formulario.component.html`**

```

<div class = "content">
    <form [formGroup] = "dadosForm" (ngSubmit) =
"exibidoraDados(dadosForm.value)">
        <input type = "email" name = "email" placeholder =
"Informe seu email" formControlName = "email" />
        <br /><br />
        <input type = "password" name = "senha" placeholder =
"Informe sua senha" formControlName = "senha" />
        <br /><br />
        <input type = "submit" [disabled] =
"!dadosForm.valid" value = "Enviar Dados" />
    </form>
    <!--exibir o valor da propriedade email-->
    <p>O email informado é: {{ propEmail }}</p>
</div>

```

Para o botão enviar, foi adicionada e a instrução `[disabled]` no colchete, que recebe o valor da expressão lógica de negação conforme indicado abaixo:

```

<input type = "submit" [disabled] = "dadosForm.valid"
value = "Enviar Dados" />

```

Portanto, se a verificação de `dadosForm.valid` não for válido, o botão permanecerá desativado e o usuário não poderá enviá-lo.

O resultado no navegador é indicado abaixo:

The form is composed of three input fields:

- A top field labeled "Insira seu email" (Insert your email).
- A middle field labeled "Insira sua senha" (Insert your password).
- A bottom field containing the word "Enviar" (Send) which is highlighted with a thick dark blue border.

O email informado foi :

No caso acima, não foi inserido qualquer email, portanto, o botão de login está desativado. Agora, ao tentar inserir o email válido é possível perceber a diferença.

The form is composed of two input fields:

- An upper field containing the email address "angular@mail.com".
- A lower field containing the text ".....".

Below the fields is a large button labeled "Enviar" (Send) which is highlighted with a thick dark blue border.

O email informado foi : angular@mail.com

O email inserido é válido. Assim, é possível ver que o botão de login está ativado e o usuário poderá enviá-lo. Com isso, o email inserido é exibido na parte inferior.

Neste próximo passo, será possível implementar uma validação personalizada com o mesmo formulário. Para validação personalizada, é possível definir nossa própria função e adicionar os detalhes necessários. Observe o código abaixo e implemente como indicado:

#### **formulario.component.ts**

```
import { Component, OnInit } from '@angular/core';
import { FormGroup, FormControl, Validators } from '@angular/forms';

@Component({
  selector: 'app-formulario',
  templateUrl: './formulario.component.html',
  styleUrls: ['./formulario.component.css']
})
export class FormularioComponent implements OnInit {
  // criando propriedades para auxiliar no controle do form
  dadosForm: any
  propEmail: any
```

```
// chamando o hook para priorizar a instancia de classe
ngOnInit(): void {
    this.dadosForm = new FormGroup({
        email: new FormControl('', Validators.compose([
            Validators.required,
            Validators.pattern('[^ @]*@[^ @]*')
        ])),
        senha: new FormControl('', this.validacaoSenha)
    })
}
// criar uma função para validação de senha
validacaoSenha(valoresSenha: any) {
    if(valoresSenha.value.length < 5) {
        return {senha:true}
    }
    return null
}
// criar uma função para exibir o resultado do "controle"
do form pela camada lógica
exibidoraDados(umDado:any){
    this.propEmail = umDado.email
}
}
```

No exemplo acima, foi criada a função *validacaoSenha* e usada em *FormControl*:

```
senha: new FormControl('', this.validacaoSenha)
```

Na função criada, será possível verifica o “comprimento” (quantidade de caracteres) dos caracteres digitados dentro do input. Se os caracteres forem menores que cinco, ele retornará com a senha true, como mostrado acima - retorno *{"senha": true}*; Se a senha inserida for composta por mais de cinco caracteres, será considerado válido e o login será ativado. Caso contrario o botão *Enviar* permanecerá desativado.

O código do arquivo *formulario.componet.html* permanece sem alterações:

### **formulario.componet.html**

```
<div class = "content">
    <form [formGroup] = "dadosForm" (ngSubmit) =
"exibidoraDados(dadosForm.value)">
        <input type = "email" name = "email" placeholder =
"Informe seu email" formControlName = "email" />
        <br /><br />
        <input type = "password" name = "senha" placeholder =
"Informe sua senha" formControlName = "senha" />
        <br /><br />
```

```

        <input type = "submit" [disabled] =
"!dadosForm.valid" value = "Enviar Dados" />
</form>
<!--exibir o valor da propriedade email-->
<p>O email informado é: {{ propEmail }}</p>
</div>

```

Vamos agora ver como isso é exibido no navegador:



O email informado foi :

Se apenas quatro caracteres forem inseridos dentro da caixa referente a inserção da senha e o botão de *Enviar* permanecerá desativado. Para habilitá-lo, será necessário inserir cinco caracteres ou mais. Agora, inserindo um tamanho válido de caracteres é possível perceber a diferença – o botão será habilitado. Observe a imagem abaixo:



O botão enviar foi ativado, pois o email e a senha são válidos – se encaixam nos padrões descritos no arquivo *formulario.component.ts*.

## Services

Nossos componentes precisam acessar dados (conteúdo ou funcionalidade). É possível escrever o código de acesso a dados em cada componente, mas isso, por vezes, é ineficiente e pode quebrar a regra da

“responsabilidade única”. O componente deve se concentrar em apresentar dados ao usuário. A tarefa de obter dados do servidor *back-end* deve ser delegada para outra classe. Essa classe é chamada de classe de service (service class). Porque proíbe ao service fornecer dados para todos os componentes que precisam.

### O que é um service

O service é um pedaço de código reutilizável com um objetivo focado. Um código que você o usará em muitos componentes em seu aplicativo

### Para que services são utilizados

1. Recursos independentes de componentes como services de log
2. Compartilhe lógica ou dados entre componentes
3. Encapsula interações externas, como acesso a dados

### Vantagens ao se usar services

1. Os services são mais fáceis de testar.
2. Os services são mais fáceis de depurar.
3. Você pode reutilizar o service.

### Como criar um service no Angular

Um service Angular é simplesmente uma função Javascript. Tudo o que precisamos fazer é criar uma classe e adicionar métodos e propriedades. Em seguida, podemos criar uma instância dessa classe em nosso componente e chamar seus métodos.

Um dos melhores usos dos services é obter os dados de uma fonte interna ou externa. Neste primeiro, será criado um service simples, que obtém os dados de produtos e os transmite ao componente.

### Implementação

Usando o projeto *angular-alpha* para implementar o Service será necessário criar um novo arquivo que servirá como modelo para fazer o tratamento dos dados dispostos na aplicação. Para criar este novo arquivo, clique com o botão direito do mouse na pasta app e crie uma pasta chamada **model** – agora, clique com o botão direito do mouse e crie um novo arquivo dentro da

pasta *model* do seu projeto – escolha New File e dê o nome ao arquivo de *product.ts*.

→ new file > **product.ts**

### **Product (produtos)**

O arquivo criado na pasta **src/app/model** e nomeado como *product.ts* deverá conter o código abaixo:

#### **product.ts**

```
// criar o model domain para tratar os dados que serão
operados pelo service
export class Produto{
    // disponibilizar as propriedades - referenciadas e
inicializadas no construtor - para o acesso e uso da
aplicação
    idProduto:number
    nomeProduto: string
    precoProduto: number
    /*
        criar um construtor da classe e
referenciar/inicializar cada uma das estruturas
de porpriedades que serão disponibilizadas para o
us/manipulação dos dados dentro da aplicação
    */
    constructor(idProduto:number, nomeProduto: string,
    precoProduto: number){
        // inicialização das propriedades
        this.idProduto = idProduto
        this.nomeProduto = nomeProduto
        this.precoProduto = precoProduto
    }

}
```

A classe *Product* acima é o nosso *domain model* (modelo de domínio).

### **Product Service (Service do produto)**

Em seguida, crie uma nova pasta dentro de `src/app` chamada **service**. Este service angular retornará a lista a partir do *model products*. Para criar o novo service, abra o prompt de comando – via `vs code` ou pelo próprio computador – e, dentro da pasta **service** de seu projeto (*angular-alpha*), insira a seguinte instrução:

```
ng g service nomedoservice
→ ng g service product

CREATE  src/app/service/product.service.spec.ts   (362
bytes)
CREATE  src/app/service/product.service.ts (136 bytes)

Após a criação do service, implemente o código abixo como se segue:
```

### **product.service.ts**

```
import { Produto } from './product'

export class ProductService {

    // criar uma função para retornar um lista de produtos
    public getProdutos() {

        // propriedade que será a lista de produtos
        let listaProdutos: Produto[]

        // criando os itens da lista de produtos
        listaProdutos = [
            new Produto(1, 'Quadro Baby Yoda', 199),
            new Produto(2, 'Mascara Darth Vader', 159),
            new Produto(3, 'LightSaber', 89),
            new Produto(4, 'Estrela da Morte Miniatura', 39),
            new Produto(5, 'Boneco Storm Trooper', 59),
            new Produto(6, 'Miniatura Princesa Leia', 119.9),
            new Produto(7, 'Miniatura Hans Solo', 259)

        ]
        return listaProdutos
    }
}
```

Primeiro, foi importado para o arquivo `product.service.ts` a class `Product` (a partir de ser arquivo de origem `product.ts`) - criado para estabelecer o model domain.

A classe *ProductService* foi criada e exportada. Foi criado, também, um método chamado *getProdutos()* que retorna a coleção *listaProdutos[ ]*. Aqui, o conjunto de dados está disposto dentro da aplicação, no entanto, é possível utilizar procedimento semelhante para obter dados de uma fonte externa a partir do envio de uma solicitação HTTP para uma API de back-end para obter os dados.

O service está pronto. Observe que a classe acima é uma função JavaScript simples. Não há nada angular nisso.

### **Invocando o *ProductService***

A próxima etapa é chamar o *ProductService* do componente.

Para esse fim, crie um novo componente e nomeie-o como **c-service** dentro da pasta app do seu projeto – para criar este novo componente - via angular cli – é necessário usar o comando abaixo:

```
ng generate component c-service
```

Abra a pasta do componente e edite os arquivos conforme indicado abaixo.

#### **c-service.component.ts**

```
import { Component } from '@angular/core';
// importando os recursos necessários
import { Produto } from '../product';
import { ProductService } from '../product.service';

@Component({
  selector: 'app-c-service',
  templateUrl: './c-service.component.html',
  styleUrls: ['./c-service.component.css']
})

export class CServiceComponent{
  // casting das propriedades que serão usadas
  cestaProdutos!:Produto[]
  objDoService

  //chamando o construtor
  constructor() {
    this.objDoService = new ProductService()
```

```
}

// criando uma função para - através do uso do objeto
criado - acessar a lista de produtos
acessandoListaProdutos() {
    this.cestaProdutos = this.objDoService.getProdutos()
}
}
```

Primeiro, foram importadas as classes *Produto* & *ProductService* – a partir de seus arquivos de origem.

No construtor do *CServiceComponet*, foi criada instânciada classe *ProductService()*.

O método `acessandoListaProdutos()` chama o método `getProdutos()` implementado dentro do arquivo `product.service.ts`. A lista retornada de `Produtos` é armazenada na variável local `cestaProdutos`.

## Criando a view

O próximo passo é exibir *Produtos* para o usuário. Abra o arquivo **c-service.component.html** e adicione o seguinte código abaixo, como se segue:

## **c-service.component.html**

```
<!--criar a tabela para exibir os dados -->





```

No código acima foi adicionado um botão chamado "Obter Produtos", que é vinculado ao método `acessandoListaProdutos()` da classe do componente por meio de *event binding* (associação de eventos).

Os produtos são exibidos/retornados via diretiva ***ngFor***.

### ***app.component.html***

```
<nav>
  <a routerLink = "/home">Home</a>
  <a routerLink = "/interpolation">Inter</a>
  <a routerLink = "/p-binding">PBinding</a>
  <a routerLink = "/e-binding">EBinding</a>
  <a routerLink = "/two-way">Two Way</a>
  <a routerLink = "/input-output">Input/output</a>
  <a routerLink = "/ng-if">ng-if</a>
  <a routerLink = "/ng-for">ng-for</a>
  <a routerLink = "/ng-class">ng-class</a>
  <a routerLink = "/ng-style">ng-style</a>
  <a routerLink = "/pipe">Pipe</a>
  <a routerLink = "/formulario">Form</a>
  <a routerLink = "/service">Service</a>
</nav>
```

Por fim, execute o código e clique no botão “Obter Produtos” e observe o resultado no browser. Deve ser semelhante ao indicado abaixo:

IMPLEMENTAÇÃO SERVICES		
<a href="#">Ver Lista Produtos</a>		
Identificação do Produto	Nome do Produto	Preço do Produto
1	Quadro Mestre Yoda	R\$199.00
2	Mascara Darth Vader	R\$159.00
3	LightSaber	R\$89.00
4	Estrela da Morte Miniatura	R\$39.00
5	Boneco Storm Trooper	R\$59.00
6	Miniatura Princesa Leia	R\$119.90
7	Miniatura Hans Solo	R\$259.00

### **Injetando services no componente**

No projeto Services implementado acima, `productService` foi instanciado no componente diretamente, como mostrado abaixo:

```
constructor () {
  this.objDoService = new ProductService();
```

}

A instanciação direta do service, como mostrado acima, tem muitas desvantagens

1. O *ProductService* está firmemente acoplado ao componente. Se a definição da classe *ProductService* for alterada, será necessário atualizar todos os códigos em que o service é usado
2. Se, por exemplo, a classe *ProductService* tiver seu nome alterado para para *NovoProductService*, será necessário procurar o nome “antigo” *ProductService* e, manualmente, alterá-lo
3. Acoplar a instanciação da classe *ProductService* diretamente no componente faz com que os testes sejam mais difíceis para configuração e execução.

Esse problema pode ser resolvido por injeção de dependência.

## Angular Dependency Injection

A Angular Dependency Injection agora é parte central do Angular e permite que as dependências sejam injetadas no componente ou classe. Neste tutorial, aprenderemos o que é Angular Dependency Injection e como injetar dependência em um componente ou classe usando um exemplo

### O que é dependência

O service *ProductService* foi construído utilizando Angular Services. O *CServiceComponent* é dependente do *ProductService* para fornecer a lista de produtos que será exibido em tela para o usuário.

Em suma, *CServiceComponent* tem uma dependência em *ProductService*.

### Angular Dependency Injection -definição

Injeção de Dependência (DI) é uma técnica na qual é fornecida uma instância de um objeto para outro objeto, que depende dele. Essa técnica também é conhecida como "Inversão de controle" (IoC)

Observe o service – *ProductService* – criado anteriormente: *ProductService* retorna os produtos dispostos dentro do array

quando o método `getProdutos` é chamado. Observe, novamente, a estrutura de código abaixo:

```
import { Produto } from './product'

export class ProductService {

    // criar uma função para retornar um lista de produtos
    public getProdutos() {

        // propriedade que será a lista de produtos
        let listaProdutos: Produto[]

        // criando os itens da lista de produtos
        listaProdutos = [
            new Produto(1, 'Quadro Baby Yoda', 199),
            new Produto(2, 'Mascara Darth Vader', 159),
            new Produto(3, 'LightSaber', 89)
        ]

        // retorno da lista montada/criada
        return listaProdutos
    }
}
```

**productService** foi instaciado diretamente dentro do arquivo `app.component.ts`. Observe, novamente, o código abaixo:

```
import { Component } from '@angular/core';
// importando os recursos necessários
import { Produto } from '../../product';
import { ProductService } from '../../product.service';

@Component({
    selector: 'app-c-service',
    templateUrl: './c-service.component.html',
    styleUrls: ['./c-service.component.css']
})

export class CServiceComponent{
    // casting das propriedades que serão usadas
    cestaProdutos!: Produto[]
    objDoService

    //chamando o construtor
```

```

constructor() {
    this.objDoService = new ProductService();
}

// criando uma função para - através do uso do objeto
criado - acessar a lista de produtos
acessandoListaProdutos() {
    this.cestaProdutos = this.objDoService.getProdutos()
}

}

```

A instância *ProductService* possui o escopo de “instância local” para o componente. *CServiceComponent*, agora, está fortemente acoplado ao ***ProductService***. Este acoplamento rígido pode trazer problemas – como visto no texto acima.

Idealmente, é necessário criar um service ***ProductServiceSingleton*** para que possa ser usado na aplicação.

Para minimizar os problemas enfrentados com o escopo local do service criado basta mover a instanciação de ***ProductService*** para o construtor da classe ***CServiceComponent***, como mostrado abaixo:

```

//chamando o construtor
constructor(private objDoService: ProductService) {
}

```

Agora, *CServiceComponent* não cria a instância do *ProductService*. Ele apenas “pede” em seu Construtor. *AppComponent* passa a ser dissociado do *ProductService*; “não sabe nada” sobre o *ProductService*. Apenas funciona com o *ProductService* passado para ele. Dessa forma, é possível passar qualquer alteração que seja feita em qualquer service, por exemplo: *ProductService*, *BetterProductService*, *MockProductService*.

O padrão acima é conhecido como ***Dependency Injection Pattern*** (*injection pattern de dependência*).

### Importância de se usar injeção de dependência

O componente agora está fracamente acoplado ao *ProductService*. *CServiceComponent* “não sabe” como criar o *ProductService*.

*CServiceComponent* não depende mais de uma implementação particular *ProductService*. Funcionará com qualquer implementação *ProductService* transmitida a ele. Para, por exemplo, criar teste unitários basta, apenas, criar uma classe *mock ProductService* e passá-la durante o teste.

A reutilização do componente fica mais fácil. O componente agora funcionará com qualquer ***ProductService***, desde que a interface seja respeitada.

O dependency injection pattern tornou nosso ***CServiceComponent*** “testável” (passível de testes), e passível de manutenção, etc.

Ainda, nesse momento, o problema todo não foi resolvido -sómente minimizado. O problema foi movido do *Component* para o Criador do *Component*.

Se o projeto for salvo e executado com a alteração feita acima, nada será exibido em tela.

Como criamos uma instância ***ProductService*** e a passamos para o ***CServiceComponent***? É isso que ***Angular Dependency Injection*** faz.

### Estrutura da Angular Dependency Injection

A estrutura de Angular Dependency Injection implementa o Injection pattern de Dependência em Angular. Ele cria e mantém as Dependências e as “injeta” nos Components ou Services que solicitam.

### Partes da estrutura de Angular Dependency Injection

Existem cinco partes principais da estrutura de Angular Dependency Injection.

#### Consumer (Consumidor)

É o componente que precisa da dependência. No exemplo acima, o CServiceComponent é o Consumidor

### **Dependency (Dependência)**

O serviço que está sendo “injetado”. No exemplo acima, *ProductService* é a Dependency

### **DI Token**

O DI Token identifica exclusivamente uma Dependência. Usamos o DI Token quando registramos dependência

### **Provider (Fornecedor)**

Os providers (fornecedores) mantém a lista de dependências junto com seus tokens . O DI Token é usado para identificar a Dependência.

### **Injector**

O *Injector* é o mantenedor dos *Providers* e é responsável por resolver as dependências e injetar a instância da Dependency (dependência) para o Consumer (Consumidor)

## **Como funciona a injeção de dependência no Angular**

### **Observando o conceito de Inject, Injector e Injectable**

As dependências são registradas com o *Provider*. Isso é feito nos *Providers* metadados do *Injector*.

O Angular fornece uma instância de *Injector* & *Provider* para todo *consumer* (consumidor).

O *consumer* (consumidor), quando instanciado, declara as dependências de que precisa em seu construtor.

O *Injector* lê as dependências do construtor do *consumer* (consumidor) e procura a dependência no *provider* (provedor). O provider (provedor) fornece a instância e o injector e injeta no consumer (consumidor). Se a instância da Dependency já existir, ela será reutilizada, tornando a dependência única.

## Como usar a dependency Injection (injeção de dependência)

Nós criamos um simples **ProductService** em nosso último passo-a-passo. Vamos atualizá-lo agora para usar a dependency injection (Injeção de Dependência).

Primeiro, precisamos registrar as dependências no provider. Isso é feito no array de metadados *providers* do decorator `@Component`.

```
providers: [ProductService]
```

Observe o código abaixo. Implemente-o como se segue:

### **app.module.ts**

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { FormsModule } from '@angular/forms';
import { AlterarTextoDirective } from './alterar-
texto.directive';

import { ProductService } from './product.service';

@NgModule({
  declarations: [
    AppComponent,
    AlterarTextoDirective,
    ComponenteFilhoComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule
  ],
  providers: [ProductService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Assim, com o registro do service dentro do array providers, é possível referenciar a *dependency injection* no construtor de *AppComponent*. Salve o

projeto e execute. Veja o resultado no browser. Deve ser semelhante ao indicado abaixo:

IMPLEMENTAÇÃO SERVICES		
<a href="#">Ver Lista Produtos</a>		
Identificação do Produto	Nome do Produto	Preço do Produto
1	Quadro Mestre Yoda	R\$199.00
2	Mascara Darth Vader	R\$159.00
3	LightSaber	R\$89.00
4	Estrela da Morte Miniatura	R\$39.00
5	Boneco Storm Trooper	R\$59.00
6	Miniatura Princesa Leia	R\$119.90
7	Miniatura Hans Solo	R\$259.00

Em seguida, precisamos informar angular que nosso componente precisa de “injeção” de dependência. Isso é feito usando o decorator **@Injectable()**.

O decorator **@Injectable()** não é necessário, se a classe já tem outros decorators. O Angular utiliza: **@Component**, **@pipe** ou **@directive**, entre outros. Porque todos estes são um subtipo de **Injectable**.

Como nosso **AppComponent** já está “decorado” com **@Component**, não precisamos “decorar” com o **@Injectable**

*Nota: @Injectable também não é necessário se a classe não tiver nenhuma dependência a ser injetada. No entanto, a melhor prática é “decorar” todas as classes de service @Injectable(), mesmo aquelas que não têm dependências .*

*Obs.: Os services geralmente não são adicionados ao array Providers do componente, mas ao array Providers do @NgModule . Em seguida, eles estarão disponíveis para serem usados em todos os componentes no aplicativo. No entanto, é possível registrar qualquer service dentro do array providers de um componente. Veremos este uso a frente.*

Quando **CServiceComponent** é instanciado, obtém sua própria instância **Injector**. O **Injector** “sabe” que o **CServiceComponent** faz a requisição de **ProductService** para o construtor. Na sequência, ele “olha” para o **Providers** para encontrar e, então, “prover”, uma instância de **ProductService** para o **CServiceComponent**

Observe, agora que o arquivo `c-service.component.ts` deve ser composto com o seguinte código:

### **`c-service.component.ts`**

```
import { Component } from '@angular/core';
// importando os recursos necessários
import { Produto } from '../product';
import { ProductService } from '../product.service';

@Component({
  selector: 'app-c-service',
  templateUrl: './c-service.component.html',
  styleUrls: ['./c-service.component.css']
})

export class CServiceComponent{
  // casting das propriedades que serão usadas
  cestaProdutos!:Produto[]

  //chamando o construtor
  constructor(private objDoService: ProductService){

  }

  // criando uma função para - através do uso do objeto
  criado - acessar a lista de produtos
  acessandoListaProdutos(){
    this.cestaProdutos = this.objDoService.getProdutos()
  }

}
```

### **Injetando Service em Service**

Até este momento, foi observado como injetar `ProductService` a um componente. Agora, observe como injetar um service em outro service.

Crei um novo service e nomeie-o como `testando`, que registra todas as operações em uma janela do console e as injeta em `ProductService`. Para criar o novo service, abra o prompt de comando – via `vs code` ou pelo próprio computador – e, dentro da pasta de seu projeto (`angular-alpha`), insira a seguinte instrução:

```
ng g service nomedoservice
```

→ **ng g service testando**

```
angular-alpha>ng g service testando
CREATE src/app/testando.service.spec.ts (357 bytes)
CREATE src/app/testando.service.ts (135 bytes)
```

Agoram abra o novo arquivo service criado – *testando.service.ts* – e implemente o código abaixo como se segue:

### **testando.service.ts**

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

@Injectable()
export class TestandoService {
    // função para verificar se ProductService está sendo
    // executado corretamente
    unitario(mensagemTeste:any) {
        console.log(mensagemTeste)
    }
}
```

A classe **TestandoService** possui apenas um método nomeado como *log()*, que captura o valor atribuído ao argumento *mensagemTeste* e exibe no console.

Também está sendo usado o decorator, com os metadados, `@Injectable` para “decorar” a classe *TestandoService*. Tecnicamente, isso não é necessário aqui, pois a classe não possui nenhuma dependência externa. É uma boa prática adicionar metadados `@Injectable` pelos seguintes motivos:

- **provas futuras:** Não é necessário lembrar `@Injectable()` quando adicionamos uma dependência posteriormente.
- **Consistência:** todos os services seguem as mesmas regras e não precisamos nos perguntar por que um decorator está ausente.

### **Inject na classe ProductService**

Agora, o próximo passo é injetar a classe **TestandoService** na classe **ProductService**

**ProductService** necessita que *TestandoService* seja injetado. Portanto, a classe requer metadados `@Injectable`. Observe a instrução abaixo e

implemente-a como se segue – faça a importação, também, de *TestandoService* e *Injectable*:

```
import { Produto } from './product'
import { TestandoService } from './testando.service';
import { Injectable } from '@angular/core';

@Injectable()
```

Na sequência, no construtor de **ProductService** é feita a solicitação de **TestandoService**. Crie o construtor da classe e implemente o código abaixo como se segue:

```
// implementando o construtor
constructor(private resultTeste: TestandoService) {
  this.resultTeste.unitario('Construtor
chamado/construido com sucesso!')
}
```

Altere o método **GetProdutos** para usar o objeto **resultTeste**.

```
/ criar uma função para retornar um lista de produtos
public getProdutos() {

  this.resultTeste.unitario('Método/função getProdutos()
chamado superultra top!')

  // propriedade que será a lista de produtos
  let listaProdutos: Produto[]

  // criando os itens da lista de produtos
  listaProdutos = [
    new Produto(1, 'Quadro Baby Yoda', 199),
    new Produto(2, 'Mascara Darth Vader', 159),
    new Produto(3, 'LightSaber', 89)
  ]

  this.resultTeste.unitario(listaProdutos)

  // retorno da lista montada/criada
```

```

        return listaProdutos
    }
}

```

Finalmente, será necessário registrar **TestandoService** com os metadados em **Providers**. Abra o *app.module.ts* para atualizar o providers e incula o registro de **TestandoService**. Observe o código abaixo e implemente-o como indicado:

### **app.module.ts**

```

import { ProductService } from './product.service';
import { TestandoService } from './logger.service';

```

```

@NgModule({
  declarations: [
    AppComponent,
    AlterarTextoDirective,
    ComponenteFilhoComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule
  ],
  providers: [ProductService, TestandoService],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

Ao clicar no botão “Obter Produtos”, é possível observar pelo console do browser com as respectivas mensagens de log. Salve o projeto e observe o resultado. Deve ser semelhante ao indicado abaixo:

IMPLEMENTAÇÃO SERVICES		
<a href="#">Ver Lista Produtos</a>		
Identificação do Produto	Nome do Produto	Preço do Produto
1	Quadro Mestre Yoda	R\$199.00
2	Mascara Darth Vader	R\$159.00
3	LightSaber	R\$89.00
4	Estrela da Morte Miniatura	R\$39.00
5	Boneco Storm Trooper	R\$59.00
6	Miniatura Princesa Leia	R\$119.90
7	Miniatura Hans Solo	R\$259.00

```
testando
▼ Array(7) ①
▶ 0: Produto {idProduto: 1, nomeProduto: 'Quadro Mestre Yoda', precoProduto: 199}
▶ 1: Produto {idProduto: 2, nomeProduto: 'Mascara Darth Vader', precoProduto: 159}
▶ 2: Produto {idProduto: 3, nomeProduto: 'LightSaber', precoProduto: 89}
▶ 3: Produto {idProduto: 4, nomeProduto: 'Estrela da Morte Miniatura', precoProduto: 39}
▶ 4: Produto {idProduto: 5, nomeProduto: 'Boneco Storm Trooper', precoProduto: 59}
▶ 5: Produto {idProduto: 6, nomeProduto: 'Miniatura Princesa Leia', precoProduto: 119.9}
▶ 6: Produto {idProduto: 7, nomeProduto: 'Miniatura Hans Solo', precoProduto: 259}
  length: 7
  [[Prototype]]: Array(0)
```

O fornecimento do service no módulo raiz cria uma instância única e compartilhada desse service e injetará em qualquer classe que solicitar.

O código acima funciona porque, o Angular, cria a Árvore dos Injetores com o relacionamento pai-filho semelhante à Árvore dos Componentes.

**Obs.: Os serviços injetados no nível do módulo têm escopo no aplicativo, o que significa que eles podem ser acessados de todos os componentes / serviços do aplicativo. Qualquer serviço fornecido no Módulo filho está disponível em todo o aplicativo.**

**Os serviços são fornecidos em um módulo lento**, com escopo definido no módulo e disponíveis apenas para o módulo carregado lento.

**Os serviços fornecidos no nível do componente** estão disponíveis apenas para o componente e para os componentes filho.

Onde você registra sua dependência, define o escopo da dependência. A dependência registrada no Módulo usando o decorator @NgModule é anexada ao Provedor Root (Provedor anexado ao InjectorRoot). Essa dependência está disponível para todo o aplicativo.

A dependência registrada com o componente está disponível para esse componente e para qualquer componente filho desse componente.

**ProductService** possui uma relação de dependência do **LoggerService**. Por isso, é “decorado” com o decorator @Injectable. Remova @Injectable() de **ProductService** será indicado a seguinte exceção:

```
Uncaught Error: Can't resolve all parameters for ProductService: (?)
```

Isso porque, sem DI Angular não é possível saber como injetar **TestandoService** em **ProductService**.

Remover `@Injectable()` de `TestandoService` não resultará em nenhum erro, pois não há dependência.

Os *components* e *directives* já estão “decorados” com `@Component` & `@Directive`. Esses decorators também dizem ao Angular para usar o DI, portanto você não precisa adicionar o `@Injectable()`.

## Referencias:

adaptado do original que pode ser obtido pelo link abaixo

Alexandre Beato: <https://medium.com/@alexandrebbeato/pt-br-arquitetura-para-projeto-em-angular-com-f%C3%A1cil-desenvolvimento-e-manuten%C3%A7%C3%A3o-817a15e0d10c>  
<https://blog.bitsrc.io/data-binding-in-angular-cbc433481cec>  
<https://www.c-sharpcorner.com/article/learn-angular-8-step-by-step-in-10-days-data-binding>  
<https://www.c-sharpcorner.com/article/learn-angular-8-step-by-step-in-10-days-data-binding-day-3/>

Traduzido e adaptado do original que pode ser obtido através do link abaixo

<https://www.tektutorialshub.com/angular/angular-directives/>

Traduzido e adaptado do original que pode ser obtido em:

Traduzido e adaptado do original que pode ser obtido em:

<https://www.tektutorialshub.com/angular/angular-services/>

Traduzido e adaptado do original que pode ser obtido pelo link abaixo

<https://www.tektutorialshub.com/angular/angular-dependency-injection/>

Traduzido e adaptado do original que pode ser obtido pelo link abaixo:

<https://www.tektutorialshub.com/angular/angular-injector-injectable-inject/>

Traduzido e adaptado do original que pode ser obtido pelo link abaixo:

<https://www.tekpasso-a-passoshub.com/angular/angular-providers/>

Traduzido e adaptado do original que pode ser obtido pelo link abaixo:

<https://www.tekpasso-a-passoshub.com/angular/angular-hierarchical-dependency-injection/>

Traduzido e adaptado do original que pode ser obtido através do link abaixo:

<https://medium.com/@jinalshah999/reactjs-step-by-step-tutorial-series-for-absolute-beginners-part-1-9f727d72d93>

<https://medium.com/@jinalshah999/reactjs-step-by-step-tutorial-series-part-2-reactjs-components-7b6a5078f824>

<https://medium.com/@jinalshah999/reactjs-step-by-step-tutorial-series-part-3-reactjs-components-communication-2c0707e90ad4>