



---

# FRAMEWORKS FRONT END

**Anderson da Silva Marcolino**

# **FRAMEWORKS FRONT END**

1ª edição

São Paulo  
Platos Soluções Educacionais S.A  
2021

© 2021 por Platos Soluções Educacionais S.A.

Todos os direitos reservados. Nenhuma parte desta publicação poderá ser reproduzida ou transmitida de qualquer modo ou por qualquer outro meio, eletrônico ou mecânico, incluindo fotocópia, gravação ou qualquer outro tipo de sistema de armazenamento e transmissão de informação, sem prévia autorização, por escrito, da Platos Soluções Educacionais S.A.

**Diretor Presidente Platos Soluções Educacionais S.A**

Paulo de Tarso Pires de Moraes

**Conselho Acadêmico**

Carlos Roberto Pagani Junior  
Camila Braga de Oliveira Higa  
Camila Turchetti Bacan Gabiatti  
Giani Vendramel de Oliveira  
Gislaine Denisale Ferreira  
Henrique Salustiano Silva  
Mariana Gerardi Mello  
Nirse Ruscheinsky Breternitz  
Priscila Pereira Silva  
Tayra Carolina Nascimento Aleixo

**Coordenador**

Henrique Salustiano Silva

**Revisor**

Paulo Henrique Santini

**Editorial**

Alessandra Cristina Fahl  
Beatriz Meloni Montefusco  
Carolina Yaly  
Mariana de Campos Barroso  
Paola Andressa Machado Leal

Dados Internacionais de Catalogação na Publicação (CIP)

---

M321f      Marcolino, Anderson da Silva  
Frameworks front end / Anderson da Silva Marcolino, –  
São Paulo: Platos Soluções Educacionais S.A., 2021.  
44 p.  
ISBN 978-65-89965-07-7

1.Vue. 2. Angular. 3. React. I. Título.

CDD 006

---

Evelyn Moraes – CRB-8 SP-010289/O

2021  
Platos Soluções Educacionais S.A  
Alameda Santos, nº 960 – Cerqueira César  
CEP: 01418-002 — São Paulo — SP  
Homepage: <https://www.platosedu.com.br/>

## FRAMEWORKS FRONT END

### SUMÁRIO

Diferenciando e escolhendo um Framework Front End	05
Compreendendo e utilizando o Framework Vue.js	21
Angular: utilizando o TypeScript com o Framework da Google	40
React: desenvolvimento de componentes com o Framework do Facebook	65

# Diferenciando e escolhendo um Framework Front End.

Autoria: Anderson da Silva Marcolino

Leitura crítica: Paulo Henrique Santini



## Objetivos

- Conhecer os *frameworks* Vue.js, Angular e React e seu contexto histórico.
- Apresentar as diferenças entre os *frameworks* Vue.js, Angular e React em relação a suas vantagens e desvantagens e sua sintaxe inicial.
- Identificar e selecionar um dos *frameworks* de acordo com a necessidade de desenvolvimento.

## 1. Introdução

O desenvolvimento de aplicações web tem se popularizado ainda mais com a ampliação, o barateamento e o acesso à internet e aos serviços de processamento e armazenamento na nuvem. Para agilizar o processo de desenvolvimento de tais aplicações, reduzindo custos e esforços e agilizando o processo de entrega de soluções de software, arcabouços ou comumente chamados *Frameworks*, tornam-se preferência na hora de se desenvolver. Porém, devido aos diferentes e diversos tipos de tecnologias e opções, há sempre pontos positivos e negativos na hora de escolher qual utilizar.

Considerando a linguagem JavaScript – uma das linguagens mais utilizadas no contexto do desenvolvimento de aplicações web – e a escolha de um *framework* para tal linguagem, é necessário avaliar alguns pontos-chaves (FERREIRA, 2018, p. 112):

- Abstração ou generalização de operações longas e complexas para garantir compatibilidade entre navegadores.
- Desenvolvimento rápido.
- Alta qualidade de código.
- Boa performance.

Nessa perspectiva e considerando tais pontos, passaremos a analisar e em seguida comparar os três *frameworks* JavaScript mais utilizados do mercado: Vue.js, Angular e React (FERREIRA, 2018, p. 112).

### 1.1 Vue.js

Vue (pronuncia-se /vju:/, como *view*, em inglês) ou Vue.js é um *framework* progressivo para a construção de interfaces de usuário. Ao contrário

de outros *frameworks* monolíticos, ou seja, *frameworks* que não podem ser integrados com outros, foi projetado desde sua concepção para ser utilizado de modo incremental, isto é, apenas parte do *framework* pode ser adotada. A biblioteca principal é focada exclusivamente na camada visual (*view layer*), sendo fácil adotar e integrar com outras bibliotecas ou projetos existentes. Por outro lado, também é perfeitamente capaz de permitir a criação das chamadas aplicações de página única (*Single-Page Applications*), quando usado em conjunto com ferramentas modernas e bibliotecas de apoio (VUE.JS, [s.d.]).

Basicamente, o Vue mantém uma função que “observa” um objeto JavaScript e reflete qualquer mudança do seu estado no modelo de documento de objeto (*Document Object Model* – DOM) da página em linguagem de marcação de hipermídia (HTML).

**Figura 1 – Logotipo do Framework Vue.js**

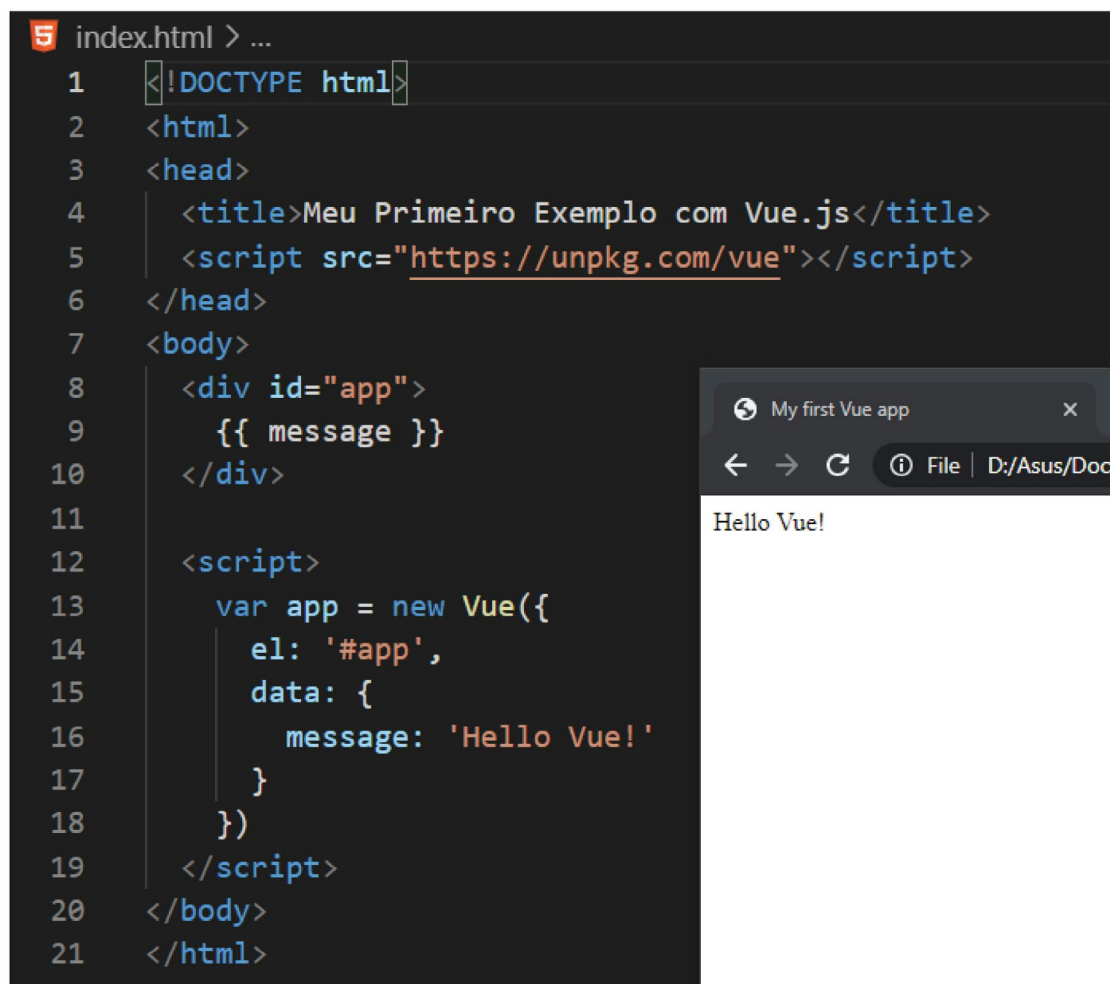


Fonte: adaptada de [https://upload.wikimedia.org/wikipedia/commons/9/95/Vue.js\\_Logo\\_2.svg](https://upload.wikimedia.org/wikipedia/commons/9/95/Vue.js_Logo_2.svg). Acesso em: 2 ago. 2021.

A origem do *framework* se deu na Google Creative Labs, por Evan You, com o objetivo de desenvolver rapidamente um protótipo que fornecesse um meio de realizar o *data binding* – sincronia entre uma fonte de dados da lógica de negócio com a interface gráfica de usuário da aplicação – reativo em componentes reusáveis (FERREIRA, 2018). Como requisitos necessários para sua utilização, indica-se ter um nível de conhecimento intermediário em HTML, folha de estilo em cascata (CSS) e JavaScript.

A Figura 2 apresenta um exemplo prático de integração facilitada do Vue.js em um projeto.

**Figura 2 – Hello World com Vue**



The image shows a screenshot of the Visual Studio Code editor. On the left, the 'index.html' file is open, displaying the following code:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <title>Meu Primeiro Exemplo com Vue.js</title>
5   <script src="https://unpkg.com/vue"></script>
6 </head>
7 <body>
8   <div id="app">
9     {{ message }}
10  </div>
11
12  <script>
13    var app = new Vue({
14      el: '#app',
15      data: {
16        message: 'Hello Vue!'
17      }
18    })
19  </script>
20 </body>
21 </html>
```

On the right, a browser preview window titled 'My first Vue app' is shown, displaying the output of the code: 'Hello Vue!'.

Fonte: captura de tela do Editor de Código Visual Studio Code (<https://code.visualstudio.com/>).

O código representado na Figura 2 corresponde a uma página HTML (index.html) que possui na sua linha 4 a integração com a biblioteca do *framework* Vue. Podem ser utilizadas duas opções de biblioteca:

- A específica para desenvolvimento, que exibe avisos úteis no console dos navegadores:  
`<script src="https://cdn.jsdelivr.net/npm/vue"></script>`



- A versão de produção, ou seja, a disponibilizada na página final que será utilizada pelos usuários:

```
<script src="https://cdn.jsdelivr.net/npm/vue"></script>
```

Pode-se optar também pela utilização do vue-cli, que requer ferramentas de construção (*build*) em Node.js.

Retornando ao código da Figura 2. Temos na sequência os elementos HTML comuns, como o *body*, que por sua vez possui uma *div* e o elemento de *script*. A *div* é o elemento em que uma mensagem (*message*) será exibida por meio do Vue. Para isso, um novo objeto do tipo Vue é instanciado na linha 13 na variável “app”. Esse objeto é então vinculado ao elemento *div* por meio do identificador inserido na marcação de abertura da *div* (“id=“app””). A propriedade “message”, que no elemento de *script* recebe um valor, é então apresentada como conteúdo da *div*. A ligação dos dados ao documento HTML agora é considerada reativo. Em outras palavras, na mudança dos valores da propriedade “message” do objeto “app”, temos a alteração do conteúdo exibido na página.

Isso é facilmente testado por meio do console de seu navegador. Para isso, basta abri-lo e digitar `app.message = “Outro texto”` e pressionar “Enter”. Na sequência, a mensagem que antes era “Hello Vue!” passará a ser o texto digitado.

No entanto, o Vue não é apenas um *framework* para renderizar um texto (*string*). É possível integrar diretivas em elementos HTML de modo a alterar suas propriedades.

```
<div id="app">
```

```
<span v-bind:title="message">
```

Pare o mouse sobre mim e veja a dica interligada dinamicamente!

```
</span>
```

```
</div>
```

No código anterior, o elemento *span* apresenta o texto que está na propriedade “message” do objeto “app”, visto na Figura 2, como uma mensagem ao permanecer com o mouse sobre o texto interno do Span. Isso se deve pelo uso de uma diretiva, no caso a “v-bind”. As diretivas são prefixadas com “v-” para indicar que são atributos especiais providos pelo Vue e aplicam comportamento especial de reatividade ao DOM renderizado. Nesse caso, basicamente está sendo dito: “mantenha o atributo *title* do elemento sempre atualizado em relação à propriedade *message* da instância Vue” (VUE.JS, [s.d.]).

Agora que temos o entendimento básico do Vue, passamos a analisar os mesmos elementos no *framework* Angular.

## 1.2 Angular

O *framework* Angular tem um histórico turbulento, tal como seu predecessor. Ele é a versão 2 do *framework* AngularJS.

Quando nos referimos ao AngularJS, estamos falando da sua primeira versão, anunciada pela Google em 2010. Esse *framework* é ainda utilizado em várias aplicações legadas, mas apresentava muitos problemas de desempenho e em sua arquitetura. Esse é um dos motivos para o AngularJS nunca ter chegado a uma segunda versão, tendo sido encerrado em 2016 com o lançamento do Angular 2, ou somente Angular.

O Angular está na versão 11 e passou pelas versões 2, 4 e seguintes até chegar à atual. A versão 1 não foi lançada e, devido a vários problemas encontrados na versão 2, que corresponde à primeira versão, pulou para a versão 4. Da versão 2 para a 4, muitos problemas e *bugs* foram

reportados, o que acabou reduzindo sua popularidade e sua adoção. Porém, nas versões mais atuais, teve corrigida grande parte dos problemas.

Outra diferença que merece destaque é a utilização da linguagem TypeScript como substituta à linguagem JavaScript do AngularJS. Essa diferença é o que impede a compatibilidade entre os projetos de AngularJS com o Angular. Contudo, a sintaxe da linguagem TypeScript é facilmente aprendida pelos desenvolvedores que já tiveram contato intermediário ou avançado com JavaScript.

O *framework* Angular é considerado parte integrante da chamada Plataforma de Desenvolvimento Angular, que inclui, além do *framework* baseado em componentes para a criação de aplicações web escaláveis, uma coleção de bibliotecas bem definidas que cobre uma variedade de funcionalidades, incluindo rotinas, gerenciamento de formulários, comunicação cliente-servidor e uma suíte de ferramentas de desenvolvimento para auxiliar a desenvolver, construir, testar e atualizar o código (ANGULAR, [s.d.]).

### Figura 3 – Logotipo do Angular



Fonte: adaptada de [https://upload.wikimedia.org/wikipedia/commons/c/cf/Angular\\_full\\_color\\_logo.svg](https://upload.wikimedia.org/wikipedia/commons/c/cf/Angular_full_color_logo.svg). Acesso em: 2 ago. 2021.

Para criar uma aplicação com Angular, é necessário instalar o Node.js, que é um construtor de aplicações JavaScript em tempo de execução baseado na *Engine* JavaScript do Chrome V8, que pode ser obtida no portal do [nodejs.org](https://nodejs.org). Após a instalação do Node.js, os seguintes

comandos deverão ser executados no terminal do seu sistema operacional para criar uma aplicação Angular:

```
ng new meu-primeiro-projeto
```

```
cd meu-primeiro-projeto
```

```
ng serve
```

O comando `ng new my-first-project` cria uma pasta e realiza o download de todas as bibliotecas complementares ao Angular, necessárias para a execução do projeto. No exemplo, o nome dado ao projeto é “meu-primeiro-projeto” e pode ser alterado de acordo com seu projeto. O segundo comando, “`cd my-first-project`”, acessa a pasta do projeto, considerando que se esteja utilizando o sistema operacional Windows. Por fim, o último comando, “`ng serve`” executa o serviço para subir o servidor e permitir a execução da aplicação. Para acessar o projeto que está sendo executado localmente, digite no navegador o endereço e a porta a seguir `http://localhost:4200`.

É possível testar projetos em Angular por meio de um link disponibilizado na documentação oficial do *framework*: `stackblitz.com/angular/akqlqboqnbo` (ANGULAR, [s.d.]).

**Figura 4 – Hello World em Angular**

```

1 <h2>Primeiro Exemplo em Angular</h2>
2 <hello-world>
3 </hello-world>

```

```

1 import { Component }
  from '@angular/core';
2
3 @Component({
4   selector:
    'hello-world',
5   template: `
6     <h2>Hello
      World</h2>
7     <p>This is my
      first
      component!</p>
8   `,
9 })
10 export class
  HelloWorldComponent {
11 }

```

```

1 import {
  Component }
  from
    '@angular/core';
2
3 @Component({
4   selector:
    'app-root',
5   templateUrl:
    './app.component.html'
6 })
7 export class
  AppComponent { }
8

```

Primeiro Exemplo em Angular

# Hello World

This is my first component!

Console 1

Fonte: captura de tela do Editor de Código on-line Stackblitz (<https://angular.io/generated/live-examples/what-is-angular/stackblitz.html>).

Um projeto em Angular é dividido em diferentes arquivos. Para que a aplicação final seja apresentada, tais arquivos precisam estar integrados

e configurados corretamente. Na Figura 4, vemos três arquivos de código e o resultado da aplicação.

No primeiro quadrante da Figura 4, no canto superior esquerdo, temos o documento HTML, nomeado “app.component.html”, que gera os elementos HTML que são inseridos na página index.html. Em especial, o elemento `<hello-world></hello-world>` é reflexo do componente angular definido no arquivo “hello-world.component.ts” localizado da pasta “app/hello-world”.

Um componente definido no arquivo do tipo “ts” indica um arquivo do tipo TypeScript que exporta, na linha 11, do segundo quadrante, no canto superior direito, a classe `HelloWorldComponent`, que depois é renderizada ao encontrar um elemento HTML que tenha o mesmo nome do *selector*, como pode ser visto na linha 5. O conteúdo que será renderizado é codificado, por sua vez, no bloco *template*, das linhas 6 a 9.

No terceiro quadrante, no canto inferior esquerdo da Figura 4, é apresentado o código do arquivo “app.component.ts”. Nele, é definido o *selector* “app-root”, que indica o componente principal que inicia a aplicação e é inserido no arquivo index.html da pasta raiz do projeto. Já na linha 5 do mesmo trecho de código, temos no `templateURL` a indicação do arquivo app.component.html, já apresentado.

Por fim, como o resultado, temos o conteúdo apresentado na Figura 4, no quarto quadrante do canto inferior direito. Logo, os componentes definidos nos arquivos TypeScript se tornam componentes que podem ser renderizados em uma página e inseridos em outras, ou seja, reutilizados.

## 1.3 React

O React, apesar de ser comumente considerado um *framework*, é na realidade uma biblioteca declarativa de interface de usuário, ou seja,

representa a camada de visão (*view*) do padrão Modelo-Visão-Controle (*Model-View-Controller* – MVC). Isso não exime a sua performance, visto que permite atualizar o DOM apenas nos elementos que cria/altera. Adicionalmente, por ser uma biblioteca declarativa, é utilizado com outras bibliotecas e *frameworks* (FERREIRA, 2018).

O React foi desenvolvido por Jordam Walke, um engenheiro de software do Facebook, sendo mantido por tal juntamente com o Instagram e outros aplicativos, bem como uma comunidade de desenvolvedores individuais. Foi concebido para enfatizar a programação funcional – os blocos de funcionalidades e a integração entre eles se dão por funções – em vez da programação orientada a objetos, levando a benefícios na área de testabilidade e desempenho (FERREIRA, 2018).

**Figura 5 – Logotipo do React**



Fonte: <https://pt-br.reactjs.org/docs/add-react-to-a-website.html>. Acesso em: 2 ago. 2021.

O React pode ser instalado e utilizado com o Node.js ou, assim como o Vue.js, pode ser adicionado no elemento de *script* do HTML. Considerando que podemos utilizar apenas algumas das funcionalidades que o React oferece, a inclusão via *script* pode ser uma boa alternativa para começar a se aventurar em tal biblioteca.

A Figura 6 apresenta um exemplo básico do uso do React (REACT, [s.d.]).



**Figura 6 – Hello World em React**

```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="UTF-8" />
5     <title>Hello World em React!</title>
6   </head>
7   <body>
8     <h2>Hello World em React!</h2>
9     <p>Exemplo do uso de React.</p>
10    <div id="like_button_container"></div>
11    <!-- Substitua "development.js"
12         por "production.min.js" de acordo
13         com o ambiente -->
14    <script src="https://unpkg.com/
15      react@16/umd/react.development.
16      js" crossorigin></script>
17    <script src="https://unpkg.com/
18      react-dom@16/umd/react-dom.
19      development.js" crossorigin></
20      script>
21    <script src="like_button.js"></
22      script>
23  </body>
24 </html>

```

```

1 'use strict';
2 const e = React.createElement;
3
4 class LikeButton extends React.
5   Component {
6     constructor(props) {
7       super(props);
8       this.state = { liked: false };
9     }
10    render() {
11      if (this.state.liked) {
12        return 'Você deu like!.';
13      }
14      return e(
15        'button',
16        { onClick: () => this.setState({
17          liked: true }) },
18        'Like'
19      );
20    }
21  }
22
23 const domContainer = document.
24   querySelector('#like_button_container');
25 ReactDOM.render(e(LikeButton),
26   domContainer);

```

Hello World em React!  
 Exemplo do uso de React.  
 Like

Fonte: captura de tela do Editor de Código *Visual Studio Code*.

Na primeira parte da Figura 6, temos um documento HTML. Os principais pontos de destaque estão nas linhas 10, 12, 13 e 14. Na linha 10, temos a criação de uma *div* com identificador “like\_button\_container”. É nesse elemento do documento HTML que será apresentado um botão, criado no arquivo “like\_button.js” e incluído no HTML na linha 14. Por fim, nas linhas 12 e 13, temos a inclusão das bibliotecas React, específicas para o ambiente de homologação (desenvolvimento). Já para disponibilizar e utilizar a biblioteca em ambiente de produção, deve-se alterar o nome da biblioteca de “react.development.js” para “react.production.min.js” e de “react-dom.development.js” para “react-dom.production.js”.



Seguindo para a análise do “like\_button.js”, que utiliza elementos da biblioteca React, temos a declaração de uma constante na linha 2, que recebe o código para criar elementos do React: “React.createElement”. Na linha 4, temos a declaração da classe “!LikeButton”, que estende o React.Component, ou seja, há a declaração de um componente React. Já na linha 5, temos o método construtor que recebe como parâmetros as propriedades (*props*) e cria um objeto, ou estado (*state*), na linha 7, com a propriedade “liked” com valor “false”.

Na linha 9, temos o método “render()”, responsável por desenhar os elementos no documento HTML. Na linha 10, temos uma estrutura de decisão “if”; assim, se a propriedade “liked” for falsa (o que ocorrerá enquanto o botão não for pressionado pelo usuário), o botão será apresentado na tela – como é visível na página HTML na parte inferior da Figura 6. Caso a propriedade tenha valor “false”, a mensagem “Você deu like!” será exibida.

Na linha 20, temos então uma constante que recebe a referência do documento HTML e um seletor comum do JavaScript, que retorna o elemento que possui como “id” o texto “like\_button\_container”. Sabemos que se trata de um “id” pelo símbolo “#” (tralha, cerquinha ou sustenido). Temos então, na linha 21, a chamada da biblioteca React e do método render, que recebe como parâmetro a criação do elemento “e”, o objeto que será criado, por meio da referência de sua classe “LikeButton” e um segundo parâmetro, que corresponde à constante que recebe o elemento do documento HTML que irá exibir o componente criado pelo React.

Qualquer alteração nos elementos ou estados (*state*) no React resultará em uma alteração nos elementos HTML, desde que as verificações no método “render()” avaliem os estados, ou assim que o desenvolvedor inserir outras possíveis verificações ou chamadas de funções.

## 1.4 Comparando Vue.js, Angular e React

Ao identificarmos os *frameworks* que possuem como base JavaScript ou TypeScript, notamos fortes semelhanças no que diz respeito à atualização das interfaces de usuário – o que faz todo sentido, já que o foco desses *frameworks* é o desenvolvimento de interfaces. Quando comparamos outros elementos e indicações destes, conseguimos perceber outras diferenças, a saber (FERREIRA, 2018; DE CAMARGOS *et al.*, 2019):

- Quanto ao tipo de padrão arquitetural indicado:
  - Angular: não segue um padrão específico em sua documentação.
  - Vue: é inspirado no padrão Modelo-Visão-VisãoModelo, em especial em VisãoModelo, já que liga os modelos diretamente com a interface do usuário (visão).
  - React: é baseado na visão da arquitetura Modelo-Visão-Controle.
- Quanto ao suporte da documentação e da comunidade:
  - Angular: destaca-se negativamente, por não possuir fórum para dúvidas.
  - Vue e React: possuem comunidade no Discord, o que facilita a interação e a solução de dúvidas.
  - Os três *frameworks* possuem repositórios no Github.
- Tamanhos de pacotes de arquivos:
  - Ainda que os tamanhos dos pacotes dependam do projeto, um projeto desenvolvido nos três *frameworks*, com as

mesmas funcionalidades, indica que o Angular é o que gera os arquivos maiores, enquanto o React é o *framework* ou, mais especificamente, a biblioteca que apresenta o menor tamanho de arquivos de projeto.

- Tempo de renderização:
  - React renderiza mais rápido seus projetos, seguido do Vue. O pior desempenho é do Angular. Aqui é importante lembrar que a comparação considera um projeto específico desenvolvido em cada um deles.

Desse modo, podemos notar que, apesar das diferenças, há poucos pontos negativos para os três *frameworks* mencionados. É importante destacar que estes são específicos para JavaScript ou TypeScript, sendo ambos muito similares. Logo, há uma série de outros *frameworks* e bibliotecas que podem trazer funcionalidades e outros benefícios. Como discutido, nenhum dos *frameworks* vistos é prescritivo, ou seja, limitam os desenvolvedores a utilizar apenas eles, o que garante a possibilidade de evoluir e criar aplicações web robustas e interessantes, considerando a integração de outras tecnologias.

Como veredito final para a escolha do melhor, notamos que React e Vue são mais robustos. Contudo, as vantagens são pequenas e o que importa é a aptidão da equipe de desenvolvimento para usar ou não uma das opções!

Bons estudos!



## Referências

ANGULAR. **Getting Started – What is Angular?**. [s.d.]. Disponível em: <https://angular.io/guide/what-is-angular>. Acesso em: 6 abr. 2021.

DE CAMARGOS, João Gabriel Colares *et al.*. Uma Análise Comparativa entre os Frameworks Javascript Angular e React. **Computação & Sociedade**, Belo Horizonte, v. 1, n. 1, 2019.

FERREIRA, Haline Kelly; ZUCHI, Jederson Donizete. Análise Comparativa entre Frameworks Frontend baseados em JavaScript para Aplicações Web. **Revista Interface Tecnológica**, Taquaritinga, v. 15, n. 2, p. 111-123, 2018.

REACT. Documentos – Introdução. [s.d.]. Disponível em: <https://pt-br.reactjs.org/docs/getting-started.html>. Acesso em: 6 abr. 2021.

VUE.JS. **Aprenda – Documentação – Guia**. [s.d.]. Disponível em: <https://br.vuejs.org/v2/guide/index.html>. Acesso em: 6 abr. 2021.

# Compreendendo e utilizando o Framework Vue.js.

Autoria: Anderson da Silva Marcolino

Leitura crítica: Paulo Henrique Santini



## Objetivos

- Conhecer mais a fundo como utilizar o *framework* Vue.js em projetos reais.
- Identificar a sintaxe e a semântica do *framework* Vue.js.
- Utilizar o Vue.js para criar soluções de página única.



## 1. Introdução

O Vue.js é um *framework* JavaScript pertencente à segunda geração de *frameworks* JavaScript junto com o Angular e o React. Um *framework* abstrai a interação do navegador com o modelo de objeto de documento (DOM). No lugar de se referenciar e manipular elementos no DOM, a interação com tais elementos ocorre declarativamente, em alto nível.

Comparativamente, é como utilizarmos uma linguagem de programação de alto nível em vez de programarmos utilizando a linguagem Assembly. Assim, a utilização de *frameworks* como o Vue.js reduz a complexidade de desenvolver aplicações web robustas, tornando o processo de desenvolvimento mais fácil. Em termos de popularidade, o Vue.js possui mais de 207 mil repositórios de projetos no Github e mais de 3.321 repositórios com mais de mil estrelas (GITHUB, [s.d.]), o que evidência sua relevância em um dos maiores repositórios de código do mundo.

Um dos principais motivos que tornam o Vue.js tão atrativo para os desenvolvedores é ser um *framework* progressivo, ou seja, não precisa ser adotado em sua completude em um projeto. Adicionalmente, há a possibilidade de se integrar por uma *tag script* da linguagem de marcação de hipertexto (HTML). É importante destacar que o Vue é um projeto considerado *indie*, ou seja, não é mantido por grandes empresas de software, sendo, portanto, atualizado e mantido pela própria comunidade, por meio de doações.

### 1.1 Vue CLI

A indicação do *framework* Vue por meio da *tag script* pode ser uma alternativa interessante para quem está começando a desbravar o *framework*. Contudo, um meio mais elegante para começar a utilizá-lo é por meio da sua interface de linha de comandos, o Vue CLI (VUE.JS,

[s.d.]), que é um sistema completo para agilização do desenvolvimento utilizando o Vue.js. Ele traz ainda: projeto base padrão, rápida prototipação, dependências em tempo de execução, uma coleção rica de plug-ins e uma interface gráfica para gerenciar os projetos em Vue.

Como pré-requisito para a instalação e a utilização do Vue, deve-se instalar o Node.js via terminal. Em seguida, execute o seguinte comando no terminal:

```
npm install -g @vue/cli
```

Para confirmar a instalação, entre com o comando:

```
vue --version
```

Como resultado, você visualizará a versão que foi instalada no seu computador. Caso deseje atualizar a versão, pode utilizar o comando:

```
npm update -g @vue/cli
```

É importante destacar algumas considerações sobre tais comandos. O “npm” indica o *node package manager*; logo, o Vue é um pacote obtido em tal repositório. O parâmetro “-g” nos comandos indica que estamos realizando uma instalação global do Vue no computador, o que é fortemente indicado. A Figura 1 exibe a execução e as saídas de cada um dos comandos indicados.

**Figura 1 – Execução de comandos de instalação no terminal (Windows 10)**

```

Prompt de Comando
> node -e "try{require('./postinstall')}catch(e){}"

Thank you for using core-js ( https://github.com/zloirock/core-js ) for polyfilling JavaScript standard library!

The project needs your help! Please consider supporting of core-js on Open Collective or Patreon:
> https://opencollective.com/core-js
> https://www.patreon.com/zloirock

Also, the author of core-js ( https://github.com/zloirock ) is looking for a good job -)

> @apollo/protobufjs@1.1.0 postinstall C:\Program Files\nodejs\node_modules\@vue\cli\node_modules\@apollo\protobufjs
> node scripts/postinstall

> ejss@2.7.4 postinstall C:\Program Files\nodejs\node_modules\@vue\cli\node_modules\ejss
> node ./postinstall.js

Thank you for installing EJS: built with the Jake JavaScript build tool (https://jakejs.com/)

+ @vue/cli@4.5.12
added 936 packages from 603 contributors in 223.252s

C:\Users\ander>vue --version
@vue/cli 4.5.12

C:\Users\ander>

```

Fonte: elaborada pelo autor.

O comando para criar uma aplicação (ou simplesmente *app*) com Vue.js utilizando o CLI é:

`vue create <nome do app>`

Por exemplo, podemos criar um projeto chamado “vueapp”. Para criá-lo, usaríamos o comando “vue create vueapp”. É importante mencionar que o nome não pode apresentar letras maiúsculas ou caracteres especiais.

O projeto criado apresentará um arquivo “package.json”, que corresponde ao arquivo de configurações do projeto e a uma estrutura inicial de arquivos (VUE.JS, [s.d.]; FILIPOVA, 2016):

- index.html.
- src/App.vue.
- src/main.js.



- src/assets/logo.png.
- src/componentes/HelloWorld.vue.

Vejamos detalhadamente o que cada arquivo possui.

- **index.html**

```
<!DOCTYPE html>

<html>

<head>

  <meta charset="utf-8">

  <meta name="viewport" content="width=device-
width,initial-scale=1.0">

  <title>CodeSandbox Vue</title>

</head>

<body>

  <div id="app"></div>

</body>

</html>
```

Documento HTML simples com apenas indicações de duas meta tags no cabeçalho do documento, a *tag* de título e uma *div* com o atributo "id=app". É nessa *div* e por meio de tal atributo que o Vue injetará seu conteúdo.

### • **src/main.js**

```
import Vue from 'vue'

import App from './App'

Vue.config.productionTip = false

new Vue({

  el: '#app',

  components: { App },

  template: '<App/>'

})
```

Este é o arquivo principal em JavaScript. Nas duas primeiras linhas, **é** importada a biblioteca do Vue.js; na seguinte, é importando o componente App.vue. O parâmetro “Vue.config.productionTip = false” foi configurado como falso para não ser exibida a mensagem que se está no modelo de desenvolvimento (*you're in development mode*). Na sequência, ocorre a criação de uma instância do Vue, por meio do elemento do DOM identificado por “#app” no documento index.html.

### • **src/App.vue**

O arquivo “App.vue” é um componente de arquivo único que contém três trechos de código: HTML, folha de estilo em cascata (CSS) e JavaScript. Essa mistura de tecnologias pode parecer estranha, mas um arquivo de componente é a maneira com que os componentes mantêm todas as suas dependências e elementos juntos.

<template>

```


<HelloWorld msg="Welcome to Your Vue.js App"/>

</template>

<script>

import HelloWorld from './components/HelloWorld.vue'

export default {

  name: 'App',

  components: {

    HelloWorld

  }

}

</script>

<style>

#app {

  font-family: Avenir, Helvetica, Arial, sans-serif;

  -webkit-font-smoothing: antialiased;

  -moz-osx-font-smoothing: grayscale;

  text-align: center;

  color: #2c3e50;
```

```
margin-top: 60px;

}
```

```
</style>
```

Temos a marcação HTML e o JavaScript, que irá interagir com tal marcação. O CSS, por sua vez, pode ter ou não escopo. Nesse caso, ele não tem, resultando em um CSS comum, como os inseridos em páginas HTML, de modo a ser aplicado em todas as páginas e elementos que seus seletores puderem referenciar.

Esse componente será referenciado por meio de uma dependência:

```
<div id="app">

  <HelloWorld/>

</div>
```

É por meio da referência "HelloWorld" que o Vue vai inserir automaticamente o componente no *app*. Ele é uma importação do arquivo "componentes/HelloWorld.vue", descrito a seguir.

- **src/components/HelloWorld.vue**

Este é o componente HelloWorld que será incluso pelo componente *app*. Ele apresentará como saída uma série de links.

Uma diferença relevante quanto ao CSS desse componente, é que seu CSS possui escopo, sendo aplicado, portanto, apenas para o componente no qual está inserido. Para facilitar a visualização, basta olhar na *tag sttyle*. Se ela possuir o atributo *scoped*, então trata-se de um CSS com escopo: `<style scoped>`.

A mensagem que será a saída do componente é armazenada na propriedade *data* da instância do Vue; ela será inserida como saída no *template* por meio do conteúdo “{{msg}}”. Qualquer conteúdo armazenado na propriedade *data* será exibido automaticamente, de acordo com seu nome. No caso, considerando o conteúdo “{{msg}}”, este não precisa ser referenciado como “data.msg”, bastando apenas usar “msg”. A seguir temos o código do componente “HelloWorld.vue”:

```
<template>

  <div class="hello">

    <h1>{{ msg }}</h1>

    <p>

      For a guide and recipes on how to configure /
      customize this project,<br>

      check out the

      <a href="https://cli.vuejs.org" target="_blank"
      rel="noopener">vue-cli documentation</a>.

    </p>

    <h3>Installed CLI Plugins</h3>

    <ul>

      <li><a href="https://github.com/vuejs/vue-cli/tree/
      dev/packages/%40vue/cli-plugin-babel" target="_blank"
      rel="noopener">babel</a></li>

      <li><a href="https://github.com/vuejs/vue-cli/tree/
      dev/packages/%40vue/cli-plugin-eslint" target="_
      blank" rel="noopener">eslint</a></li>
```

```
</ul>
```

```
<h3>Essential Links</h3>
```

```
<ul>
```

```
<li><a href="https://vuejs.org" target="_blank"
rel="noopener">Core Docs</a></li>
```

```
<li><a href="https://forum.vuejs.org" target="_blank"
rel="noopener">Forum</a></li>
```

```
<li><a href="https://chat.vuejs.org" target="_blank"
rel="noopener">Community Chat</a></li>
```

```
<li><a href="https://twitter.com/vuejs" target="_
blank" rel="noopener">Twitter</a></li>
```

```
<li><a href="https://news.vuejs.org" target="_blank"
rel="noopener">News</a></li>
```

```
</ul>
```

```
<h3>Ecosystem</h3>
```

```
<ul>
```

```
<li><a href="https://router.vuejs.org" target="_
blank" rel="noopener">vue-router</a></li>
```

```
<li><a href="https://vuex.vuejs.org" target="_blank"
rel="noopener">vuex</a></li>
```

```
<li><a href="https://github.com/vuejs/
vue-devtools#vue-devtools" target="_blank"
rel="noopener">vue-devtools</a></li>
```

```
<li><a href="https://vue-loader.vuejs.org" target="_
blank" rel="noopener">vue-loader</a></li>
```

```
      <li><a href="https://github.com/vuejs/awesome-vue"
        target="_blank" rel="noopener">awesome-vue</a></li>

    </ul>

  </div>

</template>

<script>

export default {
  name: 'HelloWorld',
  props: {
    msg: String
  }
}

</script>

<style scoped>

h3 {
  margin: 40px 0 0;
}

ul {
  list-style-type: none;
  padding: 0;
}
```

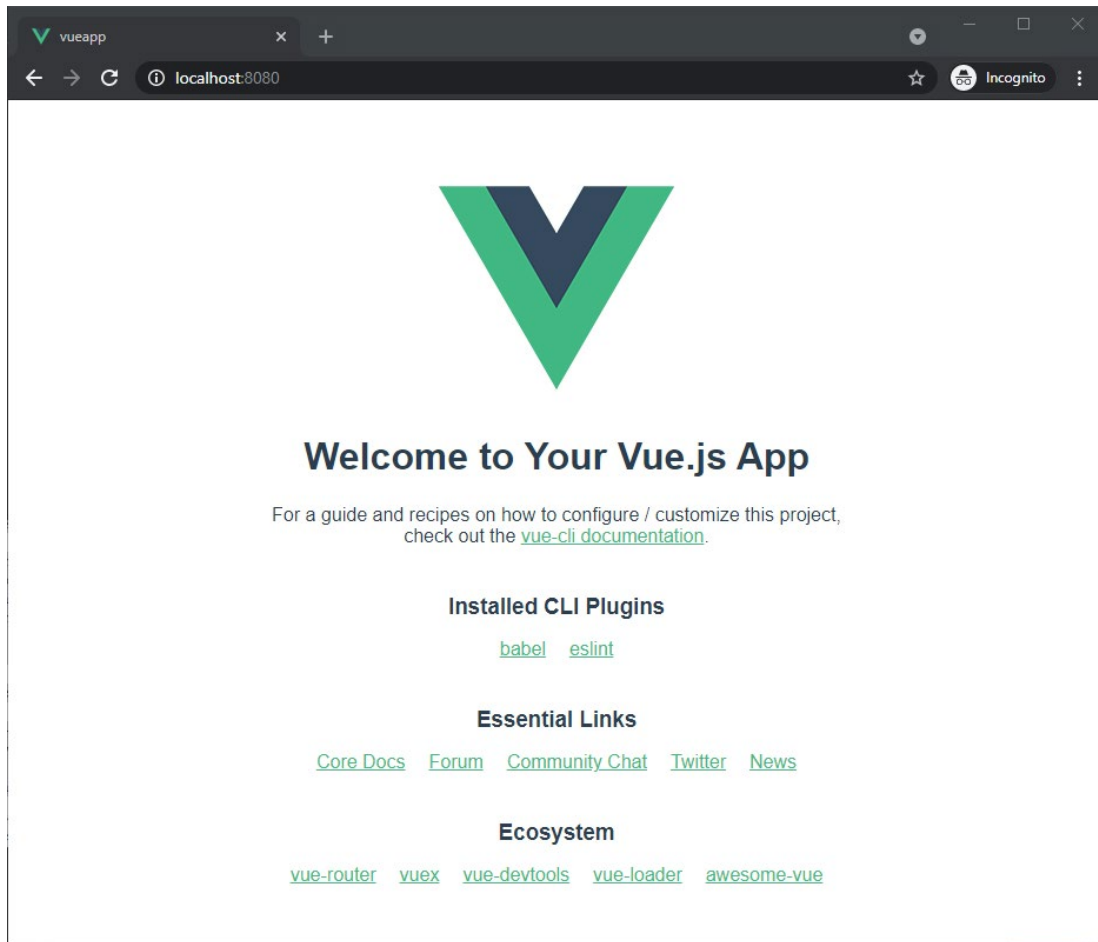
```
li {  
    display: inline-block;  
    margin: 0 10px;  
}  
  
a {  
    color: #42b983;  
}  
  
</style>
```

Para executar o projeto criado, basta seguir os comandos já apresentados na Figura 1: acessar a pasta em que o projeto foi criado, no nosso exemplo “cd vueapp”, e executar o comando “npm run serve”. Assim que os comandos forem executados no terminal, uma mensagem indicará qual endereço local deverá ser utilizado para acessar a aplicação no navegador. É importante lembrar que a aplicação está sendo executada localmente.

A Figura 2 apresenta a aplicação base que vem como exemplo ao criar um projeto com o Vue CLI. O código e a estrutura foram analisados e explicados anteriormente.



**Figura 2 – Exemplo da execução do projeto base do Vue**



Fonte: captura de tela do projeto em execução no Google Chrome v89.

A Figura 2 apresenta os elementos definidos no componente HelloWorld. vue, que, após a execução do projeto via comando no terminal, está disponível localmente via endereço “http://localhost:8080/”. É importante destacar que esse endereço pode variar em cada computador, pois pode ser que já exista outra aplicação sendo executada na mesma porta (8080), já que é muito utilizada por outros servidores web durante o desenvolvimento.

Por fim, outro comando importante para o projeto é como transformá-lo em uma versão que pode ser disponibilizada para o cliente, ou seja, “construir o projeto” ou, em inglês, executar o *build*. O comando que deve ser executado é o “npm run build”. Essa instrução é apresentada

sempre que a versão de desenvolvimento é executada, facilitando sua recordação.

Ainda nessa perspectiva, nunca devemos disponibilizar uma versão que não foi construída para produção, uma vez que ela não é otimizada e pode, em determinadas situações, apresentar mensagens que são de apoio ao desenvolvedor (nível técnico), o que pode gerar desconfortos aos usuários finais não familiarizados com tais mensagens (MACRAE, 2018).

## 1.2 Avançando na sintaxe e na semântica com Vue.js

Notamos com o exemplo criado pelo Vue CLI que os elementos criados são de fácil entendimento, visto que são criados com JavaScript e integram documentos HTML e CSS. Para auxiliar na utilização e na aprendizagem do Vue.js, recomenda-se utilizar o editor gratuito da Microsoft: o *Visual Studio Code* (VSC). Além de ser um dos editores mais utilizados atualmente, a sua flexibilidade por meio das extensões que podem ser instaladas facilita significativamente a utilização de linguagens e *frameworks*, incluindo o Vue.js. Para isso, devemos acessar no menu do VSC o item *Extensões*, que ficam no menu *Interface*. Também é possível acessar a mesma opção pelo botão na lateral esquerda, que lembra quatro blocos empilhados.

Após acessar a opção no VSC, busque pela extensão “Vetur”. Serão apresentados vários resultados, mas você deve buscar a versão mais baixada, que corresponde à criada por Pine Wu. Na sequência, clique no botão de instalar e aguarde a finalização da instalação pelo VSC. O editor será reinicializado e então você terá acesso às funcionalidades da extensão.

A funcionalidade mais perceptível é o destaque do código nos arquivos de extensão “.vue”. Porém, a mais útil são os chamados *snippets* ou

fragmentos de códigos inseridos de modo automático após digitarmos a entrada que inserirá todo o texto do código.

Os códigos do Vue.js serão inseridos de acordo com a seção e o tipo de conteúdo que o desenvolvedor está usando. Por exemplo, se deseja inserir um fragmento de código para uma **tag** específica a ser utilizada no documento HTML, você deverá digitar o nome de tal fragmento. Por exemplo “vif” irá inserir um fragmento do tipo “v-if”. Caso tenha problemas no uso dos fragmentos instalados com o Vetur, você pode instalar a extensão “Vue 3 Snippets” pelo mesmo acesso no VSC.

Preparado nosso ambiente de desenvolvimento, vamos avançar entendendo a sintaxe e a semântica da criação de componentes com o Vue.js. O primeiro passo é entender o que são tais componentes. Um componente é uma unidade de uma interface independente e pode possuir seu próprio estado, marcação, para ser inserido em um projeto e estilo.

Um componente em Vue pode ser definido por meio de quatro etapas. O **new Vue ({})** e o **Vue.component('component-name',{})** são o padrão de utilização quando se deseja construir uma aplicação de página única (*Single Page Application* – SPA). A terceira etapa se dá por meio da criação de componentes locais, que são acessíveis por componentes específicos, mas não são disponibilizados em todo o projeto, o que garante o encapsulamento. Para se criar um SPA, dá-se preferência pela criação de componentes em uma quarta e último etapa: os arquivos “.vue”, considerados componentes de arquivo único ou *Single File Components*.

Um componente para ser inserido e exibido na aplicação é instanciado em um elemento DOM, por meio do valor dado ao atributo “id” no elemento. Por exemplo: “<div id=“app”></div>”. Por sua vez, para criar o componente e relacioná-lo com o elemento DOM, deve-se indicar exatamente o mesmo nome na propriedade “el”: “new Vue({el: ‘#app’})”.

O componente indicado como “new” é usualmente o componente principal. Os demais componentes são inicializados por meio do comando “Vue.component()” e podem ser inseridos múltiplas vezes no código. A saída do componente deverá estar na propriedade “template”.

```
<div id="app">  
  
<nome-usuario nome="João"></nome-usuario>  
  
</div>
```

```
Vue.component('nome-usuario', {  
  
  props: ['nome'],  
  
  template: '<p>Olá {{ nome }}</p>'  
  
})  
  
new Vue({  
  
  el: '#app'  
  
})
```

No exemplo anterior, o componente é inicializado por meio do componente de “root” com a propriedade “el”, e, dentro desse componente (Vue.component()), temos o nome do componente “nome-usuario”. O componente aceita a passagem de propriedades, chamada *props*, utilizada para passar dados para o componente filho.

Há padrões indicados para serem utilizado na nomenclatura do componente: kebab-case ou PascalCase: “Vue.component('nome-usuario'...” ou “Vue.component('NomeUsuario'...”, respectivamente. O Vue internamente cria um apelido de um nome no padrão kebab-

case para PascalCase e vice-versa. Desse modo, indica-se utilizar “NomeUsuario” no JavaScript e “nome-usuario” no elemento DOM.

Todo componente é criado por meio do “Vue.component()” e pode ser utilizado globalmente. Contudo, é possível restringir um componente localmente, definindo um objeto como uma variável:

```
const Sidebar = {  
  
  template: '<aside>Sidebar</aside>'  
  
}
```

Dessa forma, tornamos o componente disponível para a utilização no outro componente, utilizando a propriedade *components*:

```
new Vue({  
  
  el: '#app',  
  
  components: {  
  
    Sidebar  
  
  }  
  
})
```

Adicionalmente, é possível criar um componente em um mesmo arquivo, mas em geral o melhor modo é exportar um componente em um módulo JavaScript:

```
import Sidebar from './Sidebar'  
  
export default {
```

```
el: '#app',  
  
components: {  
  
  Sidebar  
  
}  
  
}
```

Assim, considerando os componentes filhos, podemos reutilizá-lo várias vezes, sendo cada instância separada uma da outra:

```
<div id="app">  
  
<nome-usuario nome="Carlos"></nome-usuario>  
  
<nome-usuario nome="Maria"></nome-usuario>  
  
<nome-usuario nome="Gabriela"></nome-usuario>  
  
</div>
```

Para concluir a criação de componentes, temos um conjunto de propriedades que podem ser utilizadas neles:

- *El*: utilizado apenas nos componentes de *root* inicializados utilizando "new Vue({})".
- *Props*: lista todas as propriedades que podem ser passada como parâmetros para um componente filho.
- *Template*: propriedade que recebe o *template* de um componente e é responsável por definir a saída de um componente gerado.
- *Data*: o estado local do componente.

- *Watch*: os observadores dos componentes.
- *Methods*: os métodos dos componentes.
- *Computed*: as propriedades computadas associadas com um componente.

Destarte, temos os conceitos que permitirão a criação de componentes e aplicações de página única, que podem possuir uma quantidade ilimitada de componentes para formar aplicações mais robustas ou páginas. É fortemente indicado que a documentação do Vue na página do *framework* seja consultada e praticada em seus projetos! Bons estudos!



## Referências

FILIPOVA, Olga. **Learning Vue.js 2**. Birmingham: Packt Publishing Ltd, 2016.

GITHUB. **Busca Vue.js – Topics**. [s.d.]. Disponível em: <https://github.com/search?q=vue+stars%3A%3E1000&type=topics>. Acesso em: 22 abr. 2021.

MACRAE, Callum. **Vue.js – Up and Running**: Building Accessible and Performant Web Apps. USA: O'Reilly Media, 2018.

VUE.JS. **Introduction – What is Vue.js?**. [s.d.]. Disponível em: <https://v3.vuejs.org/guide/introduction.html>. Acesso em: 24 abr. 2021.

# Angular: utilizando o TypeScript com o Framework da Google.

Autoria: Anderson da Silva Marcolino

Leitura crítica: Paulo Henrique Santini



## Objetivos

- Conhecer mais a fundo como utilizar o *framework* Angular em projetos reais.
- Identificar a sintaxe e a semântica da linguagem TypeScript utilizada no *framework* Angular.
- Utilizar o Angular para criar soluções de página única.



## 1. Introdução

O Angular é uma atualização do AngularJs, isto é, uma versão reorganizada e melhorada do AngularJs, criado em 2008 na Google. Esse *framework* adota como linguagem padrão o TypeScript, que é, por sua vez, um superconjunto de JavaScript, desenvolvido pela Microsoft, que inclui novos recursos e tipagens às variáveis (ANGULAR, [s.d.]).

A linguagem TypeScript é uma linguagem de código aberto construída em JavaScript, adicionando a definição de tipos estáticos. Além disso, possibilita um caminho para descrever as formas de um objeto, criando assim melhores documentações e permitindo a validação do seu código de modo direto. Escrever em TypeScript pode ser opcional, porque seus tipos permitem que o desenvolvedor também escreva código utilizando JavaScript (TYPESCRIPT, [s.d.]).

A maior vantagem no uso de TypeScript é que um código JavaScript interpretado pode ser validado durante a sua criação, via TypeScript. Na sequência, todo o código escrito em tal linguagem é transformado em JavaScript via um compilador TypeScript, chamado Babel. Uma vez compilado, o TypeScript será convertido em JavaScript, e o código gerado, limpo e simples poderá ser executado em qualquer tipo de ambiente: seja no navegador ou em aplicações desenvolvidas com Node.js.

Apesar das vantagens do uso de TypeScript, utilizar tal linguagem não é uma escolha binária. Pode-se inicializar utilizando JavaScript e criar arquivos no mesmo projeto com TypeScript, já que todo código desenvolvido em tal linguagem, ao final, será convertido em JavaScript (TYPESCRIPT, [s.d.]).

**Figura 1 – Logos da linguagem TypeScript e do *framework* Angular**



Fontes: adaptada de [https://upload.wikimedia.org/wikipedia/commons/2/29/TypeScript\\_Logo\\_%28Blue%29.svg](https://upload.wikimedia.org/wikipedia/commons/2/29/TypeScript_Logo_%28Blue%29.svg) e <https://commons.wikimedia.org/wiki/File:Angular-logo.png> respectivamente. Acesso em: 3 ago. 2021.

O processo para criar uma aplicação utilizando o Angular consiste em fazer a instalação do Node.js e, em seguida, instalar o Angular e a ferramenta para a criação via linha de código (CLI), por meio do comando “ng install -g @angular/cli”. Após a instalação, você poderá criar um projeto completo via *prompt* de comando.

Acesse a pasta em que deseja criar o projeto e execute o comando “ng new [nomedoapp]”. Aqui o nome da aplicação (*app*) será “angularapp”; logo, o comando que deve ser executado para criar tal projeto é “ng new angularapp”. Após a criação do projeto, que demora alguns segundos, podemos acessar a pasta do projeto via terminal utilizando o comando “cd angularapp”. Ao acessar essa pasta podemos inserir novos pacotes e complementos, como o Bootstrap, “npm install bootstrap”. Uma vez instalado, o pacote será exibido em package.json. Porém, se você o remover de lá, ele não será mais considerado no projeto.

Por fim, para executar o projeto, devemos utilizar o comando “ng serve”. Ao término da compilação e da disponibilização do projeto para acesso, ainda no terminal, será exibido o endereço URL a ser digitado no navegador. Geralmente, tal endereço é <http://localhost:4200/>. Considerando esse projeto, vamos iniciar a utilização do *framework* nos familiarizando com a linguagem TypeScript.

## 1.1 A Linguagem TypeScript

Antes de entendermos os conceitos básicos do TypeScript, devemos lembrar que um projeto em Angular é sempre desenvolvido com base em componentes, os quais permitem a sua reutilização, facilitando o processo de desenvolvimento por meio do Angular (ANGULAR, [s.d.]). Um componente inclui uma classe TypeScript com um *decorator* `@Component()`, um *template* na linguagem de marcação de hipertexto (HTML), e uma folha de estilo em cascata (CSS). O *decorator* `@Component()` é um elemento textual que especifica as informações do componente:

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

@Component({
  selector: 'hello-world',
  template: `
    <h2>Hello World</h2>
    <p>Este é o meu primeiro componente!</p>
  `,
})

export class HelloWorldComponent {
  // Código do comportamento do componente.
}
```

No *selector*, define-se como o componente será utilizado no *template*. No caso, refere-se ao elemento HTML, que deverá ter o mesmo nome do atribuído no seletor. O *template* indica como o Angular deve renderizar

o componente. Pode-se incluir ainda o CSS para definir o estilo do componente.

Para utilizá-lo com componente descrito no código anterior, precisamos utilizar o componente do *template*, como segue:

```
<hello-world></hello-world>
```

Quando o componente foi renderizado, o modelo de documento do objeto aparecerá como:

```
<hello-world>
```

```
  <h2>Hello World</h2>
```

```
  <p>Este é o meu primeiro componente!</p>
```

```
</hello-world>
```

Ainda sobre o código apresentado, este incluiu todos os elementos de um componente, porém pode ser dividido. Por exemplo, o conteúdo inserido na propriedade *template* pode ter a indicação de um arquivo HTML que especifique os elementos HTML que serão renderizados em uma propriedade de nome *templateUrl*, o que também pode ser feito com a inserção de um arquivo de folha de estilos, em uma propriedade chamada *styleUrls*.

O código desenvolvido em TypeScript será inserido especificamente no corpo da classe. Como já mencionado, o TypeScript é um supergrupo do JavaScript, disponibilizando funcionalidades adicionais para a especificação da linguagem JavaScript.

Uma das principais funcionalidades e facilidades trazidas pelo TypeScript são as anotações conhecidas como *type* (do inglês, tipo), que ajudam a reduzir erros comuns de JavaScript, aplicando checagem de *types* quando o código é compilado, de um modo similar como ocorre ao

compilarmos linguagens como C# e Java. Uma vez que o JavaScript não é uma linguagem compilada, e sim interpretada, essa funcionalidade ajuda a identificar erros de modo mais rápido.

Vejamos a sintaxe e a semântica de elementos específicos do TypeScript integrados ao JavaScript que irão facilitar nossas vidas.

Em um arquivo `converteTemperatura.ts`, por exemplo, temos o seguinte código:

```
export class ConverteTemperatura {  
    static convertFtoC(temp) {  
        return ((parseFloat(temp.toPrecision(2))-32) / 1.8).  
            toFixed(1);  
    }  
}
```

No caso, temos uma função estática que converte fahrenheit para graus célsius, e para isso a função recebe uma temperatura. Essa temperatura é passada como parâmetro e é convertida em um número com duas casas decimais (`temp.toPrecision(2)`), retornando o valor em célsius no formato *string*; tal valor é arredondado e exibido em 1 casa decimal (`.toFixed(1)`).

Como a função espera receber um valor do tipo *number* e retorna, ao final, uma *string*, e o JavaScript não permite a verificação do tipo de dado que uma variável deve receber, ao chamarmos tal método, como no código a seguir, podemos ter um problema na hora de executar o código:

```
import { ConverteTemperatura } from “./ converteTemperatura “;  
let cTemp = ConverteTemperatura.convertFtoC(“25”);
```

```
console.log(`A temperatura é ${cTemp}C`);
```

O código importa o arquivo `converteTemperatura.ts` – note que não é necessário indicar a extensão do arquivo, pois o Angular consegue identificá-la, ainda que não seja exibida. Propositalmente, o parâmetro passado para a função `convertFtoC()` é a *string* `"25"`. A passagem de parâmetros incorretos pode ser comum, principalmente quando se trabalha em uma equipe com vários desenvolvedores e o projeto, bem como suas funções, não está bem documentado – como é o caso do nosso exemplo.

Por receber uma *string* e a função esperar um valor do tipo *number*, quando o código for executado, o navegador exibirá a seguinte mensagem: `temp.toPrecision is not a function`, ou seja, indica-se que o código não é uma função.

Esse erro poderia ser evitado apenas utilizando JavaScript, mas para isso teríamos que avaliar o valor dos parâmetros passados e, com base nessa verificação, retornar indicando que o código solicitado não é um número. Contudo, com o TypeScript, essa tarefa se torna mais fácil, visto que se indica o tipo de dado que é esperado tanto como parâmetro quanto em relação ao retorno da função. Isso é realizado por meio das anotações de tipos ou *types*, que são adicionadas ao código JavaScript, como pode ser visto a seguir no arquivo `converteTemperatura.ts`.

```
export class ConverteTemperatura {  
    static convertFtoC(temp: number) : string {  
        return ((parseFloat(temp.toPrecision(2))-32) / 1.8).  
            toFixed(1);  
    }  
}
```

Perceba que temos a adição das palavras *number* e *string* após o símbolo de dois-pontos (":"). Essa é uma característica típica de uma *type annotation*, ou seja, uma anotação que indica o tipo de dado esperado. Ao ser executado novamente, dessa vez o compilador TypeScript será capaz de mostrar o erro correto em relação ao código, exibindo a seguinte mensagem no compilador: *Argument of type 'string' is not assignable to parameter of type 'number'*, ou seja, um argumento do tipo *string* não pode ser atribuído ao parâmetro do tipo *number*. Para corrigir o problema, devemos alterar o parâmetro "25" para 25, ou seja, no formato especificado, por meio da anotação *types*. Desse modo, nosso código será compilado sem problemas e exibirá no console de nosso código a saída: "A temperatura é -3.8C".

Há a possibilidade de aplicar o *type annotations* também em variáveis e propriedades, para garantir que todos os tipos utilizados possam ser verificados pelo compilador. O código a seguir apresenta o uso de *types* em tais elementos:

```
export class Nome {  
    first: string;  
    second: string;  
    constructor(first: string, second: string) {  
        this.first = first;  
        this.second = second;  
    }  
    get mensagem() : string {  
        return `Olá ${this.first} ${this.second}`;  
    }  
}
```

```
}
```

Note que as variáveis *first* e *second* são marcadas com seus respectivos tipos, por meio das anotações. O mesmo ocorre para os dois parâmetros do método construtor, que, por coincidência, recebem os mesmos nomes. Outra aplicação do uso de anotações é no método mensagem, que aplica a anotação *string*.

O TypeScript também permite a atribuição de múltiplos tipos a uma variável nas assinaturas dos métodos e em seus retornos, facilitando e possibilitando comportamentos mais dinâmicos:

```
export class ConverteTemperatura {

    static convertFtoC(temp: number | string): string {

        let value: number = (<number>temp).toPrecision

        ? <number>temp : parseFloat(<string>temp);

        return ((parseFloat(value.toPrecision(2))-32) / 1.8).
        toFixed(1);

    }

}
```

Os múltiplos tipos são atribuídos por meio de uma lista cujos tipos são separados pelo caractere “|”, chamado de *pipe*. No exemplo do código anterior, a temperatura passada como parâmetro no método `convertFtoC` pode aceitar, por meio da indicação “: number | string”, ambos os tipos de valores, o que é chamado de união de tipos. Note que ainda é necessário usar o tipo como uma marcação antes da variável `temp`, no corpo do método: “<number>temp” ou “<string>temp”, para o tipo respectivo a ser utilizado. Essa marcação especial garante a conversão do valor para o tipo marcado, garantindo a execução correta do código, sem possíveis erros.



Uma alternativa para a união de tipos é o uso da palavra “any” (do inglês, qualquer um), que permite que qualquer tipo seja atribuído a uma variável, um argumento ou um retorno. No entanto, esse uso também exige que o restante do código esteja preparado para utilizar a variável de acordo com o tipo realmente esperado:

```
export class ConverteTemperatura {  
    static convertFtoC(temp: any): string {  
        let value: number;  
        if ((temp as number).toPrecision) {  
            value = temp;  
        } else if ((temp as string).indexOf) {  
            value = parseFloat(<string>temp);  
        } else {  
            value = 0;  
        }  
        return ((parseFloat(value.toPrecision(2))-32) / 1.8).  
            toFixed(1);  
    }  
}
```

Note que há uma condicional *if*, declarada como em JavaScript, que verifica se o parâmetro passado é um *number* ou uma *string*: “temp as number” ou “temp as string”, respectivamente.

Outra especificidade do TypeScript é a especificação de tipos em vetores:

...

```
let tuple: [string, string, string];
```

```
tuple = ["Brasília", "chuvoso", ConverteTemperatura.  
convertFtoC("34")]
```

```
console.log(`É ${tuple[2]} graus célsius e está ${tuple[1]}  
em ${tuple[0]}`);
```

...

Note que temos a criação de um vetor que aceita três valores, todos do tipo *string*. Tais valores podem ser acessados por meio de índices, também como em JavaScript, como é apresentado na impressão no console: `tuple[2]`, em que o índice que inicia no 0 e vai até o 3 retorna, no caso, a temperatura convertida em graus célsius.

O TypeScript também possibilita a utilização de tipos indexados, que, associados por uma chave com um valor, podem criar uma coleção tipo mapa, muito utilizada em Java, que pode ser usada para associar itens relacionados e agrupados. É o que pode ser notado no código a seguir:

...

```
let cidades: { [index: string]: [string, string] } = {};
```

```
cidades ["Londres"] = ["chuvoso", ConverteTemperatura.  
convertFtoC("34")];
```

```
cidades ["Paris"] = ["ensolarado", ConverteTemperatura.  
convertFtoC("50")];
```

```
cidades ["Berlim"] = ["nevoso", ConverteTemperatura.  
convertFtoC("25")];
```

```
for (let key in cidades) {  
  
    console.log(` ${key}: ${cidades[key][0]}, ${cidades[key]  
    [1]} ` );  
  
}
```

No código, temos a criação de uma lista chamada “cidades”, na qual temos a primeira posição do tipo *string*, e, identificadas como index, há ainda duas outras posições do tipo *string*. Note que, na atribuição de tais valores para cada uma das cidades, existe ainda a atribuição de dois valores que completam a lista de três elementos para cada cidade.

Por fim, para se percorrer o vetor e imprimir seu conteúdo, temos um laço de repetição *for* que obtém a quantidade de elementos no vetor cidades, e, com base no incremento automático desse valor (“key”), pode-se acessar cada um dos registros do vetor e seus dois outros elementos: “( ` \${key}: \${cidades[key][0]}, \${cidades[key][1]} `)”. É importante destacar que apenas *string* e *numbers* podem ser utilizados como chaves dos tipos indexados.

Uma implementação existente na linguagem TypeScript que não é suportada em JavaScript são as proteções de acesso ou os modificadores de acesso, o que significa que as classes, as propriedades e os métodos podem ser acessados por qualquer parte da aplicação. Há uma convenção, quando se usa apenas JavaScript, para se indicar que algo que deve ser restrito deve iniciar pelo caractere “\_”, porém é apenas uma convenção e não funciona de fato, como é possível ao utilizar o TypeScript.

O TypeScript disponibiliza três palavras adotadas para indicar as restrições de acesso (encapsulamento), que são utilizadas pelo compilador para verificar se estão sendo atendidas ou não no código:

- *Public*: palavra utilizada para especificar que um método pode ser acessado em qualquer lugar do projeto. É o tipo de visibilidade padrão, caso nenhuma palavra que limite o acesso seja utilizada.
- *Private*: palavra utilizada para especificar que a propriedade ou o método assim marcado poderá ser utilizada apenas pela classe que a define.
- *Protected*: palavra utilizada para indicar que o método ou as propriedades pode ser acessado apenas pelas classes que o definem ou pelas classes que estendem a classe que o definiu.

O código a seguir exemplifica a utilização de modificadores de acesso:

```
export class ConverteTemperatura {  
    static convertFtoC(temp: any): string {  
        let value: number;  
        if ((temp as number).toPrecision) {  
            value = temp;  
        } else if ((temp as string).indexOf) {  
            value = parseFloat(<string>temp);  
        } else {  
            value = 0;  
        }  
        return ConverteTemperatura.executarCalculo(value).  
            toFixed(1);  
    }  
    private static executarCalculo (value: number): number {
```

```
        return (parseFloat(value.toPrecision(2))-32) / 1.8;
    }
}
```

Note que o método `executarCalculo` está marcado como privado, o que resultará em erro pelo compilador do TypeScript, caso qualquer outra parte da aplicação tente invocar tal método.

## 1.2 Criando uma aplicação com Angular

Agora que identificamos as especificidades do TypeScript, vamos criar uma aplicação de página única que irá apresentar uma calculadora cujo resultado, após o usuário inserir valores, a aplicação deverá atualizar sem que a página seja atualizada. Vamos continuar utilizando o projeto criado por meio do Angular CLI.

Considerando que o principal recurso do Angular é desenvolver aplicações por meio de componentes e a ferramenta de linha de comando facilita tal criação, podemos gerar um componente utilizando essa ferramenta. Para isso, podemos utilizar o comando: `“ng generate component calculadora –skip-tests”`. É necessário sempre estar no diretório do projeto. O comando gera um componente com o nome “calculadora”, mas a instrução `“–skip-tests”` não gera o arquivo para teste do componente.

Como resultado da execução do comando, teremos a criação de uma pasta chamada “calculadora” no caminho `“angularapp/src/app”` com três arquivos: o `.ts`, o `.html` e o `.css`, com o desenvolvimento em TypeScript, o *template* em HTML e as folhas de estilos, respectivamente (TECHIEDIARIES, [s.d.]). A primeira etapa é identificar se nosso componente está devidamente estruturado, como segue:

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

@Component({

  selector: 'app-caculadora',

  templateUrl: './caculadora.component.html',

  styleUrls: ['./caculadora.component.css']

})

export class CaculadoraComponent implements OnInit {

  constructor() { }

  ngOnInit(): void {

  }

}
```

Nota-se que, para utilizar o componente, devemos utilizar a *tag* `<app-caculadora>`. Vamos atualizar o arquivo principal da aplicação, de modo que nosso novo componente possa ser exibido no navegador. Considerando que ainda não atualizamos o projeto base criado pelo CLI, podemos excluir todo o conteúdo do arquivo `src/app/app.component.html` e inserir o elemento `<app-caculadora></app-caculadora>`. Após essa modificação, podemos executar o comando `ng serve`. Como resultado, devemos conseguir enxergar a frase `calculadora Works!` no navegador. Seguimos então para a criação da interface (*template* e CSS).

No arquivo `src/app/calculadora/calculadora.component.html`, vamos inserir o conteúdo que segue. Note que ele já possui as referências aos métodos que serão implementados no arquivo TypeScript (.ts):

```
<div class="calculadora">

<input type="text" class="visor-calculadora"
[value]="numeroAtual" disabled />

<div class="teclas">

<button type="button" (click) = "getOperador('+')"
class="operador" value="+">+</button>

<button type="button" (click) = "getOperador('-')"
class="operador" value="-">-</button>

<button type="button" (click) = "getOperador('*')"
class="operador" value="*">&times;</button>

<button type="button" (click) = "getOperador('/')"
class="operador" value="/">&divide;</button>

<button type="button" (click) = "getNumero('7')" value="7">7</
button>

<button type="button" (click) = "getNumero('8')" value="8">8</
button>

<button type="button" (click) = "getNumero('9')" value="9">9</
button>

<button type="button" (click) = "getNumero('4')" value="4">4</
button>

<button type="button" (click) = "getNumero('5')" value="5">5</
button>

<button type="button" (click) = "getNumero('6')" value="6">6</
button>

<button type="button" (click) = "getNumero('1')" value="1">1</
button>
```

```
<button type="button" (click) = "getNumero('2')" value="2">2</button>
```

```
<button type="button" (click) = "getNumero('3')" value="3">3</button>
```

```
<button type="button" (click) = "getNumero('0')" value="0">0</button>
```

```
<button type="button" (click) = "getDecimal()" class="decimal" value=".">.</button>
```

```
<button type="button" (click) = "limpar()" class="limpar" value="limpar">AC</button>
```

```
<button type="button" (click) = "getOperador('=')" class="sinal-igual" value="=">=</button>
```

```
</div>
```

```
</div>
```

No arquivo "src/app/calculadora/calculadora.component.css", vamos inserir as propriedades a seguir:

```
.calculadora {  
  
border: 1px solid #ccc;  
  
border-radius: 5px;  
  
position: absolute;  
  
top: 50%;  
  
left: 50%;  
  
transform: translate(-50%, -50%);
```



```
width: 400px;

}

.visor-calculadora {

width: 100%;

font-size: 5rem;

height: 80px;

border: none;

background-color: #252525;

color: #fff;

text-align: right;

padding-right: 20px;

padding-left: 10px;

}

button {

height: 60px;

background-color: #fff;

border-radius: 3px;

border: 1px solid #c4c4c4;

background-color: transparent;

font-size: 2rem;
```

```
color: #333;

background-image: linear-gradient(to
bottom,transparent,transparent 50%,rgba(0,0,0,.04));

box-shadow: inset 0 0 0 1px rgba(255,255,255,.05),
inset 0 1px 0 0 rgba(255,255,255,.45), inset 0 -1px 0 0
rgba(255,255,255,.15), 0 1px 0 0 rgba(255,255,255,.15);

text-shadow: 0 1px rgba(255,255,255,.4);

}

button:hover {

background-color: #eaeaea;

}

.operador {

color: #337cac;

}

.limpar {

background-color: #f0595f;

border-color: #b0353a;

color: #fff;

}

.limpar:hover {

background-color: #f17377;

}
```

```
.sinal-igual {  
background-color: #2ec03a;  
border-color: #47ac33;  
color: #fff;  
height: 100%;  
grid-area: 2 / 4 / 6 / 5;  
}  
  
.sinal-igual:hover {  
background-color: #4ed476;  
}  
  
.teclas {  
display: grid;  
grid-template-columns: repeat(4, 1fr);  
grid-gap: 20px;  
padding: 20px;  
}
```

Precisamos adicionar também o código adicional no estilo global da aplicação, localizado em “src/styles.css”:

```
html {  
  
font-size: 62.5%;
```

```
box-sizing: border-box;

}

*, *::before, *::after {

margin: 0;

padding: 0;

box-sizing: inherit;

}
```

Agora partimos para a criação das variáveis e dos métodos em TypeScript, no arquivo “src/app/calculadora/calculadora.component.ts”. Para facilitar o entendimento do código, comentários foram adicionados:

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

@Component({

  selector: 'app-calculadora',

  templateUrl: './calculadora.component.html',

  styleUrls: ['./calculadora.component.css']

})

export class CalculadoraComponent implements OnInit {

  numeroAtual = '0'; //armazena o número que será exibido
  no resultado
```

```
primeiroOperando = null; //recebe o valor do primeiro  
operando
```

```
operador = null; //recebe o tipo de operação
```

```
aguardarSegundoNumero = false; //estado que indica se o  
usuário terminou de digitar o primeiro operando
```

```
//Método que obtém o número atual
```

```
public getNumero(v: string){
```

```
    console.log(v);
```

```
    if(this.aguardarSegundoNumero)
```

```
    {
```

```
        this.numeroAtual = v;
```

```
        this.aguardarSegundoNumero = false;
```

```
    }else{
```

```
        this.numeroAtual === '0'? this.numeroAtual = v: this.  
        numeroAtual += v;
```

```
    }
```

```
}
```

```
//Método que acrescenta o ponto de decimal
```

```
getDecimal(){
```

```
    if(!this.numeroAtual.includes('.')){
```

```
        this.numeroAtual += '.';

    }

}

//Método que efetua o cálculo

private efetuarCalculo(op , segundoOperando){

    switch (op){

        case '+':

            return this.primeiroOperando += segundoOperando;

        case '-':

            return this.primeiroOperando -= segundoOperando;

        case '*':

            return this.primeiroOperando *= segundoOperando;

        case '/':

            return this.primeiroOperando /= segundoOperando;

        case '=':

            return segundoOperando;

    }

}

//Define o método para executar a operação
```

```
public getOperador(op: string){  
    console.log(op);  
    if(this.primeiroOperando === null){  
        this.primeiroOperando = Number(this.numeroAtual);  
    }else if(this.operador){  
        const result = this.efetuarCalculo(this.operador ,  
        Number(this.numeroAtual))  
        this.numeroAtual = String(result);  
        this.primeiroOperando = result;  
    }  
    this.operador = op;  
    this.aguardarSegundoNumero = true;  
    console.log(this.primeiroOperando);  
}  
  
//Define o método para limpar a calculadora  
public limpar(){  
    this.numeroAtual = '0';  
    this.primeiroOperando = null;  
    this.operador = null;
```

```
        this.aguardarSegundoNumero = false;

    }

    constructor() { }

    ngOnInit(): void {

    }

}
```

Como resultado, teremos uma calculadora que realizará as operações básicas. Para um melhor entendimento, recomendamos a execução do projeto (arquivos disponibilizados junto com o tema que devem ser executados após o comando “npm install”) para que todos os elementos e linhas de código sejam reconhecidos. Adicionalmente, indica-se a leitura das referências deste Tema, que são as documentações oficiais do Angular, as quais fornecem um excelente referencial para o aprofundamento dos conhecimentos. Bons estudos!



## Referências

ANGULAR. What is Angular?. [s.d.]. Disponível em: <https://angular.io/guide/what-is-angular>. Acesso em: 4 maio 2021.

TECHIEDIARIES. **Angular 10/9 Tutorial and Example:** Build your First Angular App. Disponível em: <https://www.techiediaries.com/angular/angular-9-tutorial-and-example/>. Acesso em: 7 maio 2021.

TYPESCRIPT. **What is TypeScript?**. [s.d.]. Disponível em: <https://www.typescriptlang.org/>. Acesso em: 4 maio 2021.



# React: desenvolvimento de componentes com o Framework do Facebook.

Autoria: Anderson da Silva Marcolino

Leitura crítica: Paulo Henrique Santini



## Objetivos

- Conhecer o *framework* React e como ele pode ser utilizado em projetos reais.
- Identificar elementos sintáticos e semânticos do framework React com JSX.
- Utilizar o framework React para criar uma aplicação de página única.



## 1. Introdução

O React é um *framework* que pode ser adotado de modo incremental. Em outras palavras, você pode ter um projeto que pode utilizar mais ou menos elementos do *framework*. Adicionalmente, ele oferece o conhecimento básico para que possa adotar alguns conjuntos de ferramentas, de acordo com sua demanda, como (REACT, [s.d.]):

- *Create React App*: para aprender React ou criar um *single-page app*.
- Next.js: para criar um site renderizado no servidor (SSR) com Node.js.
- Gatsby: para criar um site estático orientado a conteúdo.
- Outras como Neutrino, Nx, Parcel e Razzle para desenvolvedores mais experientes.

Seguindo tais indicações, o conjunto de ferramentas mais indicado para a aprendizagem do React e para a criação de uma aplicação do tipo página única, do inglês *single-page app*, é a ferramenta *Create React App*, que, em tradução direta, significa Criar uma Aplicação React.

O *Create React App* permite criar um ambiente de desenvolvimento já pré-configurado, pronto com as funcionalidades mais recentes do JavaScript – já que o React é um *framework* baseado em tal linguagem. Entre tais funcionalidades, temos a otimização da aplicação para produção, ou seja, para o ambiente de execução final da aplicação. Para a utilizar essa ferramenta, será necessária a instalação prévia do Node.js versão 10.16 ou superior e do *Node Package Manager* (npm) em versão 5.6 ou superior.

Há a possibilidade de iniciar um novo projeto sem o download e a instalação local do React, via comando no terminal:

```
npx create-react-app meu-app
```

O nome “meu-app” poderá ser escolhido para a aplicação a ser criada. Em seguida, para acessar o projeto criado, basta entrar na pasta deste via terminal com o comando “`cd meu-app`” e, na sequência, com o comando “`npm start`”. É importante destacar que o *Create React App* não lida com a lógica de *back end*, o que faz todo o sentido, já que se trata de um *framework front end*. A ferramenta atua criando um projeto que pode ser utilizado com qualquer *back end*. Atuam ainda, por trás do projeto, o Babel e o webpack.

O Babel é um transcompilador de código aberto que atua convertendo código ECMAScript mais atual em versões antigas para serem executadas sem erros em meios que não aceitam as versões mais atuais. Vale lembrar que ECMAScript é o nome correto para JavaScript. Já o webpack (que se escreve com o “w” minúsculo) é uma ferramenta responsável por dinamizar, melhorar e empacotar arquivos, principalmente códigos, de um projeto de acordo com as pré-configurações e otimizações definidas da própria ferramenta.

Retornado ao React, outra consideração importante a ser introduzida é o uso do JSX, como pode ser visualizado na declaração de uma variável a seguir:

```
const element = <h1>Olá Mundo!</h1>;
```

O JSX não é uma marcação HTML nem de uma *string*; ele é uma extensão de sintaxe para o JavaScript. É basicamente uma linguagem para a definição da interface do usuário (UI), mas atrelada a funcionalidades e dinamicidades do JavaScript.

Nesse contexto, o React acaba por basear-se na lógica de renderização interligada com a lógica de manipulação de eventos, como a alteração de dados (chamada de *state*), que muda à medida que as aplicações

são utilizadas e como os dados são exibidos. Isso implica integrar esses diferentes elementos em unidades fracamente acopladas denominadas componentes.

É possível não adotar o JSX. No entanto, seu uso é indicado, já que permite exibir mensagens mais úteis de erros e avisos.

Seguimos então para entender os elementos essenciais do JSX para, na sequência, criarmos uma pequena aplicação de página única.

## 1.1 JSX e o React

Iniciemos por um exemplo com JSX (REACT, [s.d.]):

```
const nome = 'João das Neves';

const element = <h1>Olá, {nome}</h1>;

ReactDOM.render(

  element,

  document.getElementById('root')

);
```

Nas duas primeiras linhas de código, temos a declaração de duas constantes. A primeira delas ("nome") recebe uma cadeia de caracteres (*string*) e a outra ("elemento") recebe o elemento HTML e a própria constante "nome". É aí que temos a utilização do JSX, que ocorre ao envolvermos a constante com chaves. Além do uso de variáveis, podemos adicionar também, dentro das chaves, expressões JavaScript, como `3 + 4`, `cliente.primeiroNome`, ou `formatarNome(cliente)`.

No código a seguir, há a incorporação de uma função para tratar e formatar o nome do cliente:

```
function formatarNome(cliente) {  
    return cliente.primeiroNome + ' ' + cliente.ultimoNome;  
}  
  
const cliente = {  
    primeiroNome: 'João',  
    ultimoNome: 'Neves'  
};  
  
const element = (  
    <h1>  
        Olá, {formatarNome(cliente )}!  
    </h1>  
);  
  
ReactDOM.render(  
    element,  
    document.getElementById('root')  
);
```

No trecho de código anterior temos a função `formatarNome()`, que formata o nome, concatenando o `primeiroNome` e `ultimoNome` presentes no objeto `cliente`.

Como as expressões JSX são convertidas em funções JavaScript, podemos utilizar o JSX dentro de blocos `if` e `for`, bem como atribuir tais valores em variáveis e utilizá-las como funções:

```
function getGreeting(user) {  
  
    if (user) {  
  
        return <h1>Hello, {formatName(user)}!</h1>;  
  
    }  
  
    return <h1>Hello, Stranger.</h1>;  
  
}
```

Já no contexto de especificação de atributos com JSX, temos a possibilidade de declarar *strings* como atributos:

```
const element = <div tabIndex="0"></div>;
```

Adicionalmente, também é possível incluir uma expressão JavaScript em um atributo:

```
const element = <img src={cliente.figuraUrl}></img>;
```

Assim como no HTML, podemos apresentar elementos simples e compostos. Os elementos simples não possuem marcação de fechamento, apenas de abertura, sendo o fechamento destes realizado por meio de `"/>"`. Já os elementos compostos possuem marcação de abertura e de fechamento. Adicionalmente, o JSX pode integrar elementos:

```
const element = (  
  <div>  
    <h1>Olá!</h1>  
    <h2>Good to see you here.</h2>  
  </div>  
)
```

O JSX também garante a segurança, prevenindo injeção de conteúdos indesejados e ataques do tipo *Cross-site-scripting* (XSS), já que tudo é verificado pelo Modelo de Documento de Objetos React (DOM React). Ao final, tudo é convertido em cadeia de caracteres (*strings*).

O XSS é uma vulnerabilidade de aplicações web que permite que código JavaScript sejam injetados, em especial em páginas comuns aos usuários, como páginas de log-in ou iniciais de um site. O ataque ocorre por meio de formulários alterados por scripts JavaScript de modo a redirecionar dados dos usuários para os atacantes. A fim de evitar tais ataques, como os presentes no JSX, **é necessário** controlar e tratar os dados de entrada e saída tanto de usuários quanto de aplicações.

Uma última característica importante do JSX é a capacidade do Babel de compilar objetos representados por meio do `React.createElement()`. Os trechos de código a seguir, por exemplo, resultam na mesma apresentação:

```
const elemento = (  
  <h1 className="saudacoes">  
    Olá, Mundo!
```

```
</h1>

);

const elemento = React.createElement(

  'h1',

  {className: 'saudacao'},

  'Olá, Mundo!'

);
```

O método `createElement()` cria e retorna um novo elemento React de tipo específico. No caso do primeiro trecho de código, por utilizar o JSX, o método `React.createElement()` não é exibido, mas este é chamado para transformar o trecho em um elemento React. Já no seguinte trecho de código, escrito sem JSX, o método é chamado diretamente.

Adicionalmente, o método `React.createElement()` realiza a validação do código, de modo a evitar erros, gerando um objeto similar ao apresentado a seguir:

```
const element = {

  type: 'h1',

  props: {

    className: 'greeting',

    children: 'Hello, world!'

  }
```



```
};
```

Tais objetos são chamados de elementos React e resultam no que será renderizado na tela do navegador. O React utiliza tais objetos para construir o DOM. Essa renderização ocorre por meio do ReactDOM.`render()`. Aplicações puramente desenvolvidas considerando React possuem apenas uma chamada do `ReactDOM.render()`, como é possível observar nos dois primeiros trechos de código desta seção. Contudo, aplicações desenvolvidas com a integração de partes do React podem apresentar quantas chamadas forem necessárias ao método.

## 1.2 Criando uma aplicação React

Vamos criar uma aplicação para registrar atividades de página única com React. A primeira etapa é criar a aplicação por meio do comando `npx create-react-app listadeatividades`. Na sequência, espere a criação do projeto e acesse a pasta criada por meio do comando `cd listadeatividades`. Por fim, para testar a aplicação, execute `npm start`. Um projeto básico com o símbolo do React será apresentado, indicando que a criação foi realizada com êxito (KIRUPA, [s.d.]).

Na sequência, abra seu projeto com seu editor favorito. Para a realização desse projeto, utilizei o Visual Studio Code e alguns plug-ins específicos para o React.

O primeiro passo é substituir o conteúdo do arquivo “index.html” localizado na pasta *public* pelo código HTML que segue:

```
<!DOCTYPE html>
```

```
<html lang="pt-br">
```

```
<head>
```

```
<title>Lista de Atividades</title>

</head>

<body>

  <div id="container">

</div>

</body>

</html>
```

Nesse trecho, temos a definição de um código HTML básico. O elemento mais importante é a *div* com atributo de identificador *container*; é nesse elemento que a aplicação React será integrada de fato.

Vamos modificar também o código do arquivo “index.css”, localizado na pasta *src*, como segue:

```
body {

  padding: 50px;

  background-color: # 7bda4f;

  font-family: sans-serif;

}

#container {

  display: flex;

  justify-content: center;
```

```
}
```

No trecho anterior, temos duas regras de folha de estilos em cascata, uma para o elemento *body* e outra para a *div*. Ambas estão localizadas no arquivo `index.html` na pasta *public*.

Por fim, altere o conteúdo do arquivo “`index.js`”, mantido também na pasta *src*, como segue:

```
import React from “react”;

import ReactDOM from “react-dom”;

import “./index.css”;

var destination = document.querySelector(“#container”);

ReactDOM.render(

  <div>

    <p>Olá!</p>

  </div>,

  destination

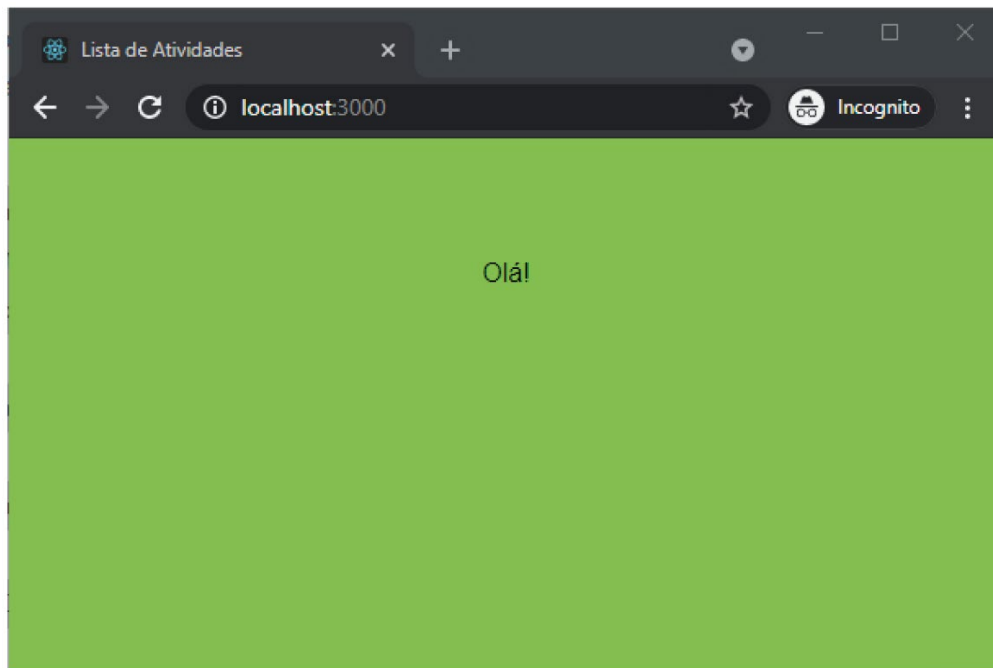
);
```

Nesse código, temos a criação do ponto de renderização do React. Nas três primeiras linhas, temos a importação do React, a importação do ReactDOM e a folha de estilos criada anteriormente. A variável *destination* recebe o seletor do DOM da página, que busca o elemento com identificador *container*, que condiz com a *div*. Na sequência, temos a chamada do `ReactDOM.render()`, que realiza a renderização

propriamente dita do React, que, nesse primeiro, momento apresenta apenas uma *div* com um parágrafo de conteúdo “Olá!”.

O resultado inicial da modificação dos arquivos `index.html`, `index.js` e `index.css` é o apresentado na Figura 1.

**Figura 1 – Página inicial do projeto**



Fonte: captura de tela do Google Chrome v90.

Com base na estrutura inicial, vamos criar o componente React. Para isso, temos que criar um arquivo na pasta `src` chamado “`ListaDeAtividades.js`”, com o código que segue:

```
import React, { Component } from “react”;

class ListaDeAtividades extends Component {

  render() {

    return (

      <div className=”ListaDeAtividadesMain”>
```

```

    <div className="header">

      <form>

        <input placeholder="Entre com sua atividade.">

          </input>

          <button type="submit">Adicionar</button>

        </form>

      </div>

    </div>

  );

}

}

```

```
export default ListraDeAtividades;
```

Agora, podemos incluir no elemento *div* do arquivo “index.js” na pasta *src* o componente *ListraDeAtividades*, resultando no seguinte código:

```

import React from “react”;

import ReactDOM from “react-dom”;

import “./index.css”;

import ListraDeAtividades from “./ListaDeAtividades”

var destination = document.querySelector(“#container”);

```

```
ReactDOM.render(  
  <div>  
    <ListaDeAtividades/>  
  </div>,  
  destination  
);
```

Assim, podemos passar para a implementação da adição de tarefas, a apresentação das tarefas adicionadas. Para isso, vamos criar um componente responsável por gerar a lista de atividades a serem exibidas. Esse componente recebe os dados enviados do componente “ListaDeAtividades.js” e se chamará “ListaAtividades.js”:

```
import React, { Component } from “react”;  
  
class ListaAtividades extends Component {  
  criaAtividade(atividade) {  
    return <li key={atividade.key}>{atividade.text}</li>  
  }  
  
  render() {  
    var atividadesEntrada = this.props.entradas;  
  
    var listaAtividades = atividadesEntrada.map(this.  
      criaAtividade);  
  
    return (  

```

```
    <ul className="theList">

        {listaAtividades}

    </ul>

);

}

};
```

```
export default ListaAtividades;
```

A atualização do componente “ListaDeAtividades.js” recebe a implementação do método `adicionaAtividade()`, responsável por atualizar os valores da lista das atividades. Essa atualização ficará como segue:

```
import React, { Component } from “react”;

import ListaAtividades from “./ListaAtividades”

class ListaDeAtividades extends Component {

    constructor(props) {

        super(props);

        this.state = {

            atividades: []

        };

        this.adicionaAtividade = this.adicionaAtividade.
        bind(this);
```

```
}

adicionaAtividade(e) {

  if (this._inputElement.value !== "") {

    var novaAtividade = {

      text: this._inputElement.value,

      key: Date.now()

    };

    this.setState((prevState) => {

      return {

        atividades: prevState.atividades.
          concat(novaAtividade)

      };

    });

    this._inputElement.value = "";

  }

  console.log(this.state.atividades);

  e.preventDefault();

}

render() {

  return (
```



```

<div className="ListaDeAtividadesMain">

  <div className="header">

    <form onSubmit={this.adicionaAtividade}>

      <input ref={(a) => this._inputElement = a}
        placeholder="Entre com sua atividade.">

        </input>

        <button type="submit">Adicionar</button>

      </form>

    </div>

    <ListaAtividades entradas={this.state.atividades}/>

  </div>

);

}

}

export default ListaDeAtividades;

```

A chamada do método `onSubmit={this.adicionaAtividade}` no formulário envia as informações para o método `adicionaAtividade`, que envia os dados para atualizar o estado `this.state` e registrar as atividades na lista, as quais são então renderizadas pelo seu mapeamento em `var listaAtividades = atividadesEntrada.map(this.criaAtividade);`, retornando, na sequência, a lista atualizada de elementos:

```
return (  
  
  <ul className="theList">  
  
    {listaAtividades}  
  
  </ul>  
  
);
```

Assim, temos uma aplicação que exibe uma lista de tarefas básica, por meio de dois componentes React. O componente ListaDeAtividades importa o componente ListaAtividades, de modo a exibir o conteúdo deste.

Para aprofundamento dos estudos, recomendamos criar tal projeto e modificá-lo de modo a garantir o seu entendimento, bem como consultar a documentação no site oficial do React (REACT, [s.d.]). Como atividade complementar, sugerimos implementar a exclusão das atividades da lista, identificando seu índice e o excluído do `this.state`. Bons estudos!



## Referências

KIRUPA. **Building an Awesome Todo List App in React**. [s.d.]. Disponível em: [https://www.kirupa.com/react/simple\\_todo\\_app\\_react.htm](https://www.kirupa.com/react/simple_todo_app_react.htm). Acesso em: 20 maio 2021.

REACT. **Introdução**. [s.d.]. Disponível em: <https://pt-br.reactjs.org/docs/getting-started.html>. Acesso em: 23 maio 2021.



---

BONS ESTUDOS!