# PROGRAMA STARTER
# FULL STACK WEB DEVELOPER

ReactJS

## Termo de uso



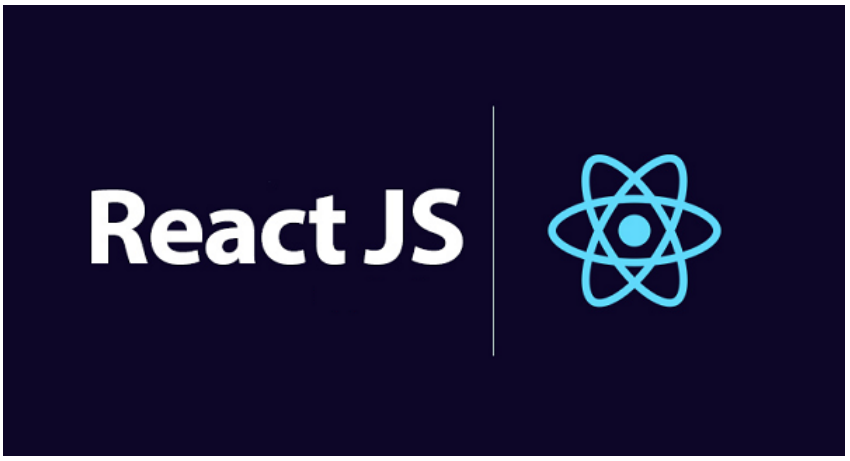
## Objetivo do documento

Este material tem como objetivo apresentar a biblioteca React.js, suas principais características e aplicações. Além disso, a configuração de um projeto React é demonstrada de forma prática, explorando de forma conceitual os alguns dos recursos da biblioteca.

# ReactJS, o que é?

A era da Internet mudou completamente a forma com que os sites e aplicações web são construídos. Muitas mudanças aconteceram nos últimos anos, de modo que tecnologias novas estão surgindo com bastante frequência. Uma das tecnologias com maior destaque atualmente é o **ReactJS**.

O ReactJS é uma **biblioteca de front-end** para construção de aplicações web com interface gráfica usando Javascript. Foi criado pela equipe do Facebook como uma biblioteca interna, mas em 2013 teve seu código aberto para a comunidade – tornando-se **open-source**.




(Disponível em: Aprenda sobre os conceitos do ReactJS - The Xcodes)

Com o ReactJS, é possível construir uma aplicação front-end de forma eficiente e ágil. Por ser baseado em Javascript, podemos usar todo o poder dessa linguagem que vem mostrando uma grande evolução. Por outro lado, a biblioteca oferece soluções para construção de aplicações usando menos código e com maior reaproveitamento de componentes.

É importante mencionar a diferença entre **ReactJS**, **React Native** e apenas **React**. O React é a biblioteca que oferece o desenvolvimento de interfaces gráficas e componentes, de modo que o ReactJS é basicamente o React

voltado para aplicações web. Por sua vez, o React Native usa a biblioteca React para a construção de aplicativos mobile. Neste documento, abordaremos apenas o **ReactJS**.

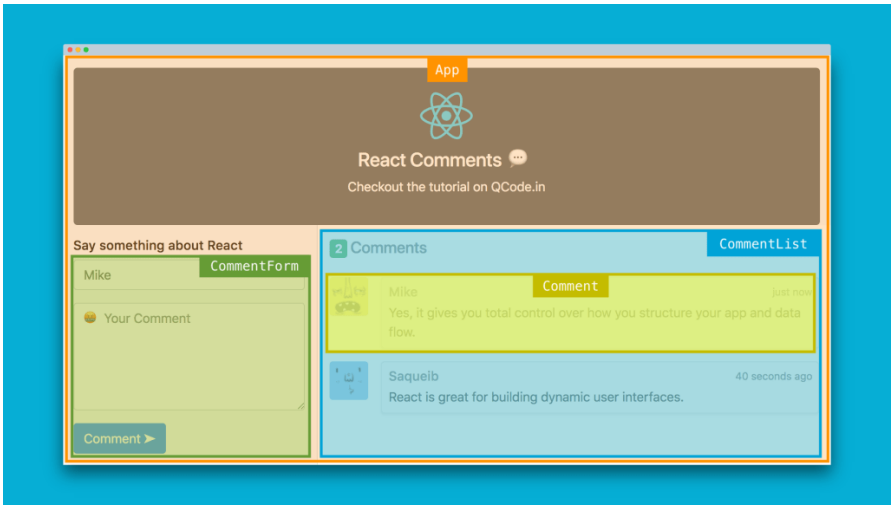
## O que o React JS veio resolver?

O ReactJS foi criado como forma de **organizar** e tornar aplicações front-end mais **eficientes** dentro do Facebook. O principal objetivo da biblioteca é facilitar o desenvolvimento de interfaces reutilizáveis e interativas – uma demanda cada vez maior no contexto da web atual.

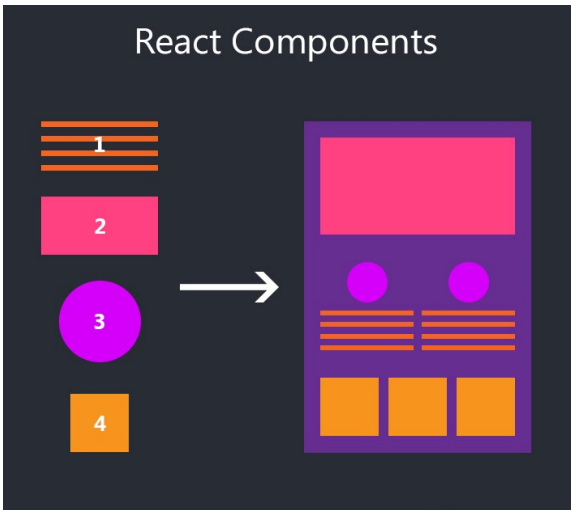
### Reutilização de componentes

Um dos principais problemas de frameworks e bibliotecas mais antigas é a falta de reutilização de componentes. Neste caso, um mesmo código é replicado em diversos pontos do projeto, causando dificuldade de manutenção e um crescimento desnecessário de código fonte.

O React resolve este problema ao dividir a aplicação em **componentes** que são **reaproveitados** em diversos lugares. Este conceito é o coração do React, já que a biblioteca é **baseada em componentes**. Uma página web é um componente que pode conter outros componentes. A divisão em componentes é chamada de **componentização**.




(Disponível em: https://thexcodes.com/conceitos-reactjs/ )




(Disponível em: Components in React)

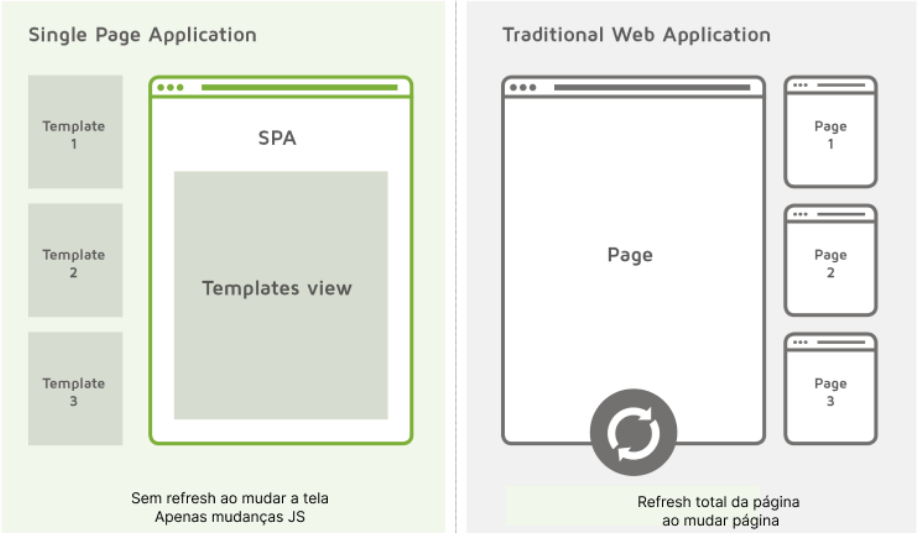
Podemos pensar cada componente como um **bloco** da aplicação. Por exemplo, uma página comum pode contar com um componente para o header, outro para o menu lateral e um outro para o conteúdo da página. A reutilização é facilitada pois um mesmo componente pode ser criado de forma independente e ser usado em diversas páginas ou outros componentes.

Imagine que um botão seja criado com algumas regras de CSS e ações JS específicas. Sem reutilização, o botão será replicado no código sempre que precisar ser usado. Com React o componente botão é criado apenas uma vez e reutilizado onde for necessário. Caso uma alteração do botão seja necessária, a modificação é feita apenas uma vez no próprio componente.

## Single Page Application (SPA)

As **aplicações de única página** – Single Page Applications (SPAs) – se tornaram populares com o ReactJS. O objetivo de uma SPA é concentrar toda uma aplicação em um **único documento** HTML que é entregue ao navegador.

Toda mudança de página em uma aplicação do estilo SPA é feita usando Javascript e não requer, necessariamente, chamadas ao lado servidor. Assim as mudanças são mais rápidas e eficientes, mas requer um maior processamento no navegador.




(Adaptado de: Single Page Applications - Why they make sense)

Portanto, todas as páginas de uma SPA são carregadas na requisição, mas apenas uma é mostrada por vez. Quando há navegação entre páginas, o Javascript se encarrega pela troca visual. Enquanto aplicações tradicionais (ou MPA – Multi Page Application) precisam requisitar toda a página novamente ao servidor quando há navegação.

## Virtual DOM

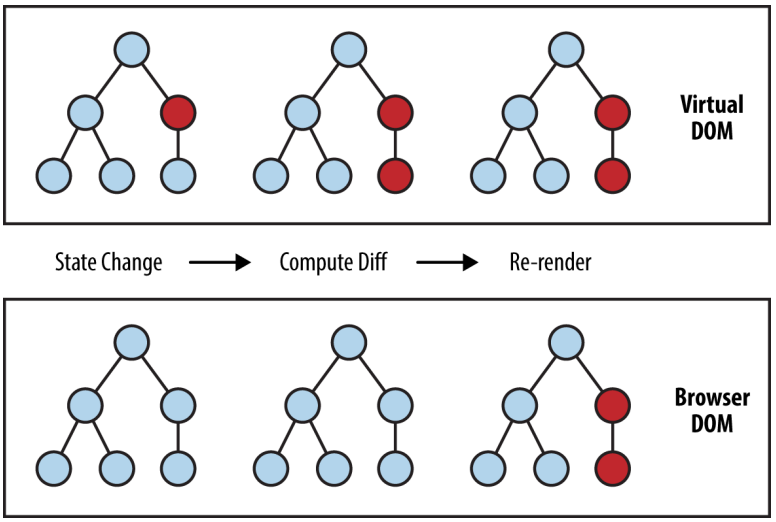
A renderização dos componentes na página conta com a ajuda de um recurso muito interessante do React: o **DOM Virtual**. O DOM é a representação da página que é interpretada pelo navegador como uma estrutura hierárquica (árvore). Já o DOM Virtual cria uma **cópia** dessa estrutura e fornece uma série de melhorias.

A atualização de um elemento dentro do DOM necessita que a árvore de elementos seja percorrida desde o seu início até achar o nó específico.

Alterações frequentes, principalmente em grandes aplicações dinâmicas, causam uma queda de desempenho da aplicação.

O DOM Virtual corrige este problema mirando nas aplicações que possuem **alterações frequentes** na sua estrutura. O objetivo deste recurso é atualizar a cópia do DOM com maior frequência e, só depois de todas as alterações feitas, o resultado é passado de forma única ao DOM real.

Assim, enquanto as modificações são construídas, o DOM real não é influenciado e não sofre com quedas de desempenho. Quando o conjunto de alterações é finalizado, o DOM real recebe uma **única atualização** (re-renderização) de forma simplificada e otimizada.




(Disponível em: React Virtual DOM Explained in Simple English)

## Projeto em React

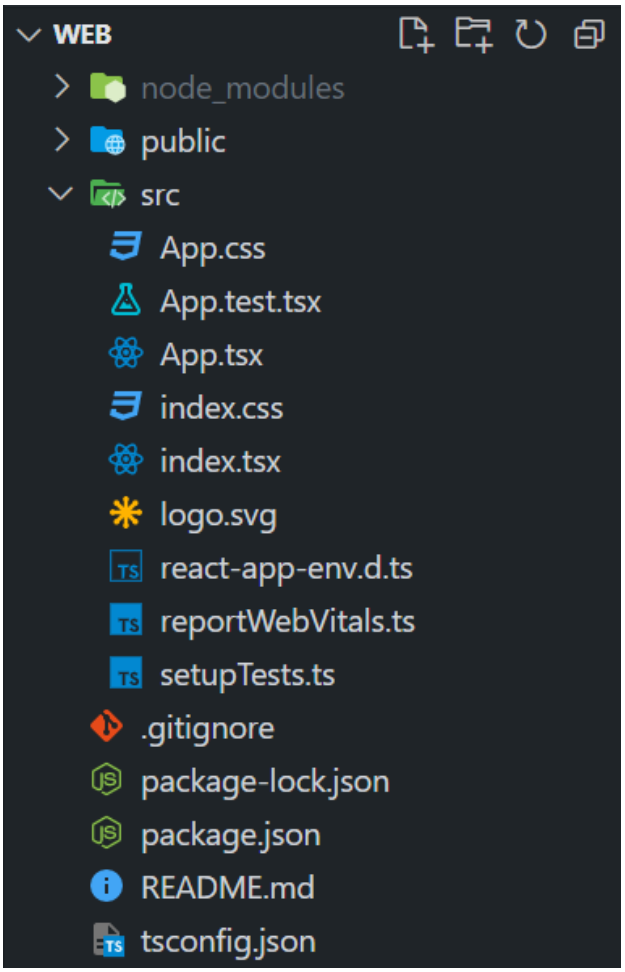
Para criarmos um projeto **ReactJS**, é necessário que o **Node.js** esteja instalado na máquina, bem como o gerenciador de pacotes **NPM**[1]. A próxima dependência que deve ser usada é o pacote **create-react-app**. Em versões mais recentes do NPM, este pacote já vem instalado.

[1] Para maiores informações acerca da instalação do Node.js, consulte o link: https://nodejs.org/en/

O **create-react-app** é uma ferramenta que auxilia na **criação de aplicações SPA com o React**. Para iniciarmos o projeto usando também o Typescript como linguagem principal[2], o comando abaixo deve ser executado no diretório desejado. No lugar de `<nome-do-projeto>`, mude para o nome que deseja definir ao projeto.

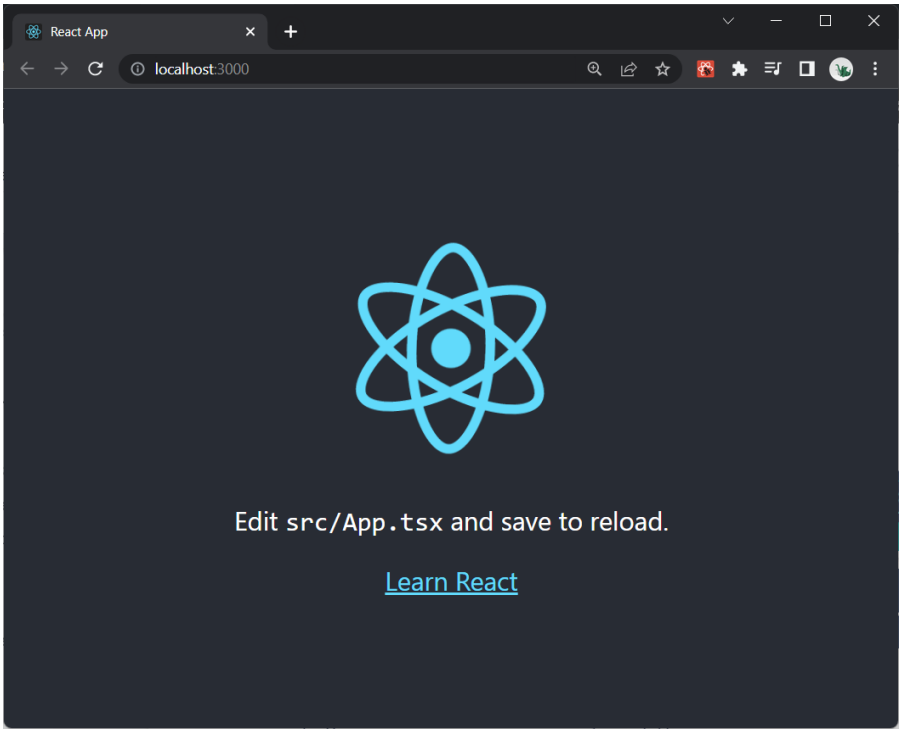
```
npx create-react-app <nome-do-projeto> --template typescript
```

Este comando deverá estabelecer um projeto Node.js dentro de uma nova pasta com o nome escolhido e, automaticamente, irá instalar as dependências mínimas para um projeto React. Ao acessar a pasta com o VSCode, a estrutura deve ser parecida com a imagem abaixo.




[2] Originalmente, o React foi concebido apenas para Javascript. Com a popularização do Typescript, projetos React usando o superset tem se tornado cada vez mais comuns.

Para **rodar o projeto** na sua máquina, basta rodar o comando **`npm start`** no terminal de sua preferência e uma página será aberta no endereço `http://localhost:3000`. Este é o endereço padrão para os testes locais do projeto. A página que se abrirá terá o layout da imagem abaixo.




Note que o comando `npm start` **mantém a aplicação React rodando** até que um cancelamento seja providenciado. Caso algum arquivo do projeto seja modificado, o comando é reinicializado automaticamente. Essa é uma ferramenta bastante útil para o desenvolvimento.

## Estrutura

Um projeto padrão do ReactJS, criado pelo create-react-app e com o template do Typescript, possui algumas seções importantes. Abaixo são listadas cada uma das partes do projeto e seus objetivos.

| Pasta/arquivo | Objetivo |
|---|---|
| . (pasta raiz) | Na pasta inicial do projeto, além das demais pastas existem os arquivos de **configuração do projeto**, como o tsconfig.json e o package.json |
| ./public | Em public é possível encontrar os **arquivos estáticos** onde a aplicação deverá ser disponibilizada. |
| ./public/index.html | O arquivo index.html dentro de public é a **página do projeto**. Todo o JS/TS criado na pasta src será introduzido a esta página, sendo que a aplicação atuará como uma SPA. |
| ./src | Na pasta src estão todos os **códigos-fonte da aplicação**: páginas, componentes e demais arquivos. |
| ./src/index.tsx | Por padrão, o **arquivo inicial** do projeto é o index.tsx dentro de src. Nele, consta um código Typescript responsável por inserir toda a aplicação dentro do index.html de public. |

Por se tratar de uma SPA, o projeto que criamos possui apenas uma página HTML (`index.html` em public). Dentro de *src*, todos os códigos fontes podem ser desenvolvidos e anexados ao HTML único. O arquivo *index.tsx* dentro de *src* faz essa "mágica" acontecer.

## Componentes

No React, (quase) tudo é considerado um componente. Para criarmos os componentes do nosso projeto, precisamos criar os códigos dentro da pasta *src*. Um exemplo de componente é o *App.tsx*, que já vem criado após configuração padrão do create-react-app.

```tsx
import React from "react";
import logo from "./logo.svg";
import "./App.css";

function App() {
    return (
        <div className="App">
            <header className="App-header">
                <img src={logo} className="App-logo" alt="logo" />
                <p>
                    Edit <code>src/App.tsx</code> and save to reload.
                </p>
                <a className="App-link" href="https://reactjs.org">
                    Learn React
                </a>
            </header>
        </div>
    );
}

export default App;
```

Observe a estrutura de um componente React: além de código Typescript/Javascript, há a **adição de HTML** e, eventualmente, de **CSS** também. A integração entre interface gráfica e código de programação facilita o desenvolvimento de aplicações dinâmicas.

O código HTML escrito dentro de um componente é quase totalmente fiel ao HTML puro. Porém, **algumas mudanças** são necessárias. Por exemplo: a palavra-chave *`class`*, usada com frequência no HTML, é uma palavra reservada no Typescript e possui significado diferente. Por isso, no TSX a classe de um elemento HTML é chamada de *`className`*.

Outro exemplo é o atributo *`for`* de um *`label`*. Quando usamos o HTML dentro de um componente React, devemos usar *`htmlFor`* já que *`for`* representa um laço de repetição. Maiores detalhes sobre as diferenças entre o HTML puro e aquele presente nos componentes React podem ser encontrados em: HTML vs JSX – What's the Difference?.

## TSX

Desenvolvedores que já tiveram contato com o Typescript puderam notar, pelas imagens acima, que a extensão de alguns arquivos no projeto não é .ts, mas sim **.tsx**. A variação do arquivo com um "x" a mais também é vista no React com Javascript puro, onde o .js vira .jsx.

O TSX é considerado um **syntax-sugar**: variação de uma linguagem (neste caso, o Typescript) para tornar o desenvolvimento mais natural para uma determinada abordagem. No TSX, a abordagem que queremos melhorar é a **construção de interfaces gráficas** dentro do código TS.

Visto que um componente React é dotado de código HTML e CSS, é importante que o desenvolvimento seja natural tanto para um contexto quanto para o outro ao mesmo tempo. Isto é, ao usar o TSX, temos o ganho quanto aos editores – principalmente o VSCode – que entendem o objetivo do código e apresentam sugestões tanto para Typescript quanto para HTML ou CSS.

Além disso, o TSX é uma forma semântica de separar códigos Typescript que possuem HTML/CSS daqueles que não são voltados para a construção de páginas. Scripts de configuração que não precisam de HTML, por exemplo, podem receber a extensão .ts dentro do projeto.

## Tipos

No React existem duas variações de componentes que remetem a forma como que são codificados. Os **componentes funcionais** (functional componentes) são criados como funções, enquanto os **componentes classe** (class components) são desenvolvidos usando classes e orientação a objetos.

★ Exemplo de um functional component:

```
import React from "react";

function MyComponent(props: any) {
    const hello = "Hello, growdevers!";

    return <p>{hello}</p>;
}

export default MyComponent;
```

★ Exemplo de um class component:

```
import React from "react";

class MyComponent extends React.Component {
    public hello: string;

    constructor(props: any) {
        super(props);
        this.hello = "Hello, growdevers!";
    }

    render() {
        return <p>{this.hello}</p>;
    }
}

export default MyComponent;
```

Note que a abordagem de class no Typescript exige que o componente seja estendido de `React.Component`. Para isso, o construtor precisa chamar o super antes de criar as propriedades da classe.

Já no componente funcional, não há esquemas de heranças nem propriedades. Ainda assim, é possível definir variáveis e outras funções dentro da função que origina o componente. Sintaticamente, esta abordagem é mais simples.

Uma importante diferença entre class e function é a **renderização** da interface gráfica. Em um componente de classe, o **método** `render()` é responsável por retornar o elemento HTML. Já na function, o **próprio retorno** da função realiza este papel.

A abordagem de class component foi historicamente mais usada por oferecer as vantagens da programação orientada a objetos, apesar de ter uma **sintaxe mais verbosa**. Além disso, a possibilidade de **componentes stateful** – com a adição de states (estados) – só era possível usando classes[3].

Componentes funcionais são **stateless**: não possuem estado. Porém, features (funcionalidades) adicionadas nas versões mais recentes do React – principalmente os Hooks e o Context API – tornaram possível o **controle de estado** nesse tipo de componente.

Porém, a abordagem mais famosa atualmente é uma variação das funções: as **arrow functions**. Componentes criados desta maneira possuem uma sintaxe menor e mais limpa, proporcionando a mesma funcionalidade de uma função normal.

[3] Stateless / Stateful Components | Medium

★ Exemplo de um functional component usando arrow function:

```
import React from "react";

const MyComponent: React.FC = (props: any) => {
    const hello = "Hello, growdevers!";

    return <p>{hello}</p>;
};

export default MyComponent;
```

## Propriedades

Um componente pode receber **parâmetros** quando é renderizado, de modo que seu conteúdo se torne **dinâmico** e possa ser aproveitado para mais de uma finalidade – fortalecendo o conceito de reaproveitamento de componente. Os parâmetros são chamados de **propriedades** (**props**).

Na definição de um componente, as props são passadas como **parâmetro da função** (em caso de componente funcional) ou do **construtor** (quando class component). Nos exemplos acima, as props foram definidas com o tipo `any`.

No Typescript, porém, é recomendado que as propriedades sejam definidas usando a **tipagem** proporcionada pelo superset. Portanto, uma prop eficiente é definida com tipo e geralmente associada a uma interface ou type.

```tsx
import React from "react";

interface MyComponentProps {
    curso: string;
    turma: number;
}

const MyComponent: React.FC<MyComponentProps> =
(props: MyComponentProps) => {
    return (
        <div>
            <p>Curso: {props.curso} </p>
            <p>Turma: {props.turma} </p>
        </div>
    );
};

export default MyComponent;
```

## Renderização

Após criarmos os componentes – seja ele funcional ou de classe, com props ou sem – devemos definir onde serão **renderizados**. É importante lembrar onde o projeto React inicia a renderização: por padrão, no método `render` do arquivo `index.tsx`.

A partir de então, podemos renderizar componentes que contém código HTML e/ou outros componentes. Para renderizar um componente, devemos **criar uma tag** com o respectivo **nome** da função/classe. Considere o exemplo do componente `MyComponent`: para renderizá-lo, devemos importá-lo na página e usar a tag `<MyComponent>`.

```
import React from "react";
import MyComponent from "./MyComponent";

function App() {
    return (
        <div>
            <h1>Renderização do componente:</h1>
            <MyComponent />
        </div>
    );
}

export default App;
```

No exemplo acima, o componente chamado `App` está renderizando um código HTML que contém uma `<div>`, um `<h1>` e outro componente React: o `MyComponent` que estamos usando como exemplo.

Note, porém, que não estamos passando parâmetros para o componente. Considere agora o exemplo de props acima em que o componente requisita duas propriedades: curso (`string`) e turma (`number`). A renderização deverá informar esses dados como atributos na tag do elemento.

```
<MyComponent curso={"Web Full-Stack"} turma={8} />
```

# Ferramentas úteis

O ReactJS é uma biblioteca criada para possibilitar a construção de aplicações web usando o Node.js. Porém não é considerado um framework. Os frameworks – como o Angular ou o Next.js – controlam o fluxo das aplicações e possuem uma série de ferramentas pré-definidas.

Por ser uma biblioteca, os projetos ReactJS oferecem uma maior liberdade para o desenvolvedor escolher as dependências do projeto. Abaixo são detalhados alguns dos principais pacotes usados pela comunidade React.

## React Router DOM

Outra dependência essencial a quase todos os projetos ReactJS é o React Router DOM. Com este pacote, podemos criar **navegação** entre as páginas da nossa SPA sem necessidade de carregamento de toda a página em cada mudança.

O roteamento no React Router possui duas versões: o React Router Native para o React Native (mobile) e o **React Router DOM** que, por sua vez, é usado no ReactJS e tem como objetivo a navegação usando o DOM: https://reactrouter.com/docs/en/v6/getting-started/overview [4].


[4] Cuidado! O React Router DOM está atualmente na versão 6, mas a documentação encontrada ao pesquisar por "react router dom" pode apontar para a versão 5. Algumas diferenças entre as versões são importantes e podem causar confusão.

A instalação do React Router DOM segue o script abaixo.

```
npm install react-router-dom@6
```

Para utilizar ele no projeto, devemos criar um componente de roteamento que segue algumas regras e precisa usar componentes do pacote.

- `<BrowserRouter>`: Define um esquema de roteamento para navegadores (usando o DOM);
- `<Routes>`: Wrapper que contém a definição das rotas;
- `<Route>`: Representa cada uma das rotas.

Definição das rotas em `Router.tsx`:

```tsx
import React from "react";
import { BrowserRouter, Routes, Route } from "react-router-dom";

import About from "./About";
import Home from "./Home";
import Login from "./Login";

const Router = () => {
    return (
        <BrowserRouter>
            <Routes>
                <Route path="/" element={<Home />} />
                <Route path="/about" element={<About />} />
                <Route path="/login" element={<Login />} />
            </Routes>
        </BrowserRouter>
    );
};

export default Router;
```

Cada rota em `<Route>` recebe dois atributos essenciais: `path` e `element`. O `path` indica qual deve ser o **caminho** na URL para a rota. Por exemplo, a rota `/home` deverá ser renderizada quando o usuário acessar a URL `http://meudominio.com.br/home`. Já `element` é o **componente** que deve ser renderizado quando a rota for acessada.

Usando o Router em `App.tsx`:

```
import React from "react";
import Router from "./Router";

function App() {
    return <Router />;
}

export default App;
```

Dentro da aplicação, podemos criar **links** de modo que as rotas sejam trocadas sem necessidade de recarregamento total. Para isso, usamos o componente `Link` do React Router DOM informando qual a rota deve ser acessada.

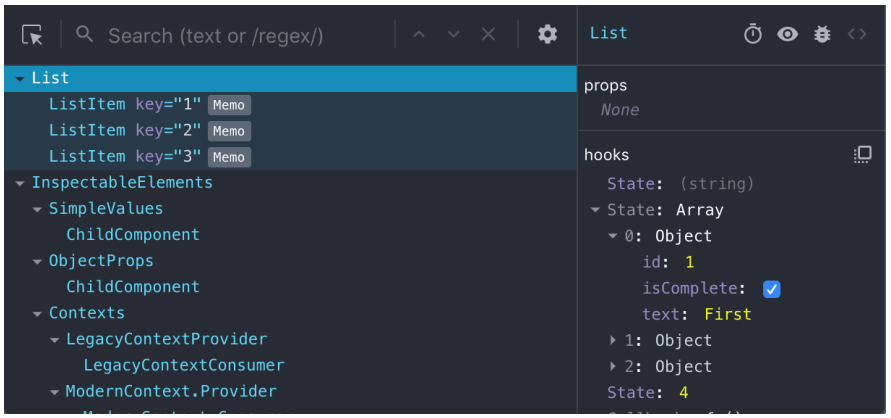
```
import React from "react";
import { Link } from "react-router-dom";

const Home = () => (
    <div>
        <h1>Home Page!</h1>
        <br />
        <Link to={"/login"}>Go to Login Page</Link>
    </div>
);

export default Home;
```

## React DevTools

Os principais navegadores – como o Chrome e o Firefox – oferecem uma seção de desenvolvedores onde podemos verificar os elementos do DOM, os códigos CSS e Javascript, além de um console Javascript e outras inúmeras funcionalidades.

O React também possui uma ferramenta de "DevTools" disponível para estes navegadores. Trata-se do **React Developer Tools**: Apresentando o novo React DevTools. Com ele é possível monitorar a aplicação com uma visão React.

Nesta ferramenta, podemos visualizar e alterar a **árvore de componentes** criados. Dentro de cada componente, também podemos visualizar **estados**, **propriedades** e demais **características** importantes. Além disso, existem recursos para avaliação de **desempenho** usando o React Profiler.



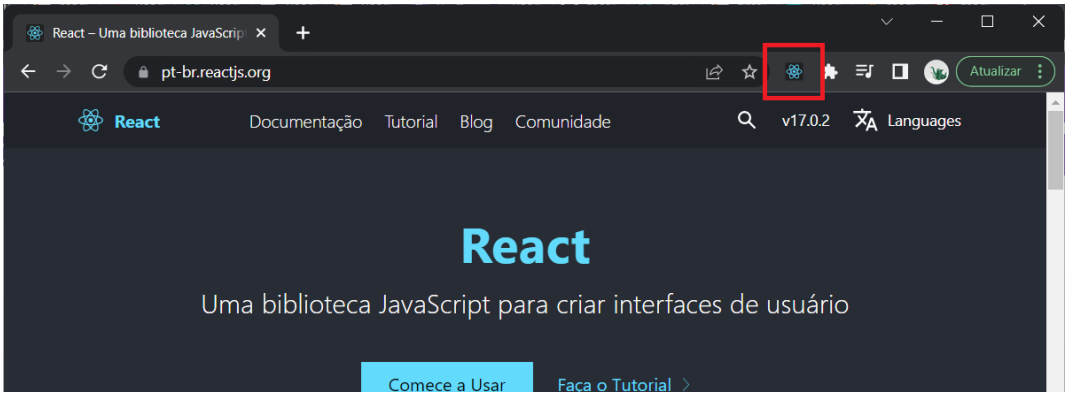

(Disponível em: Apresentando o novo React DevTools)

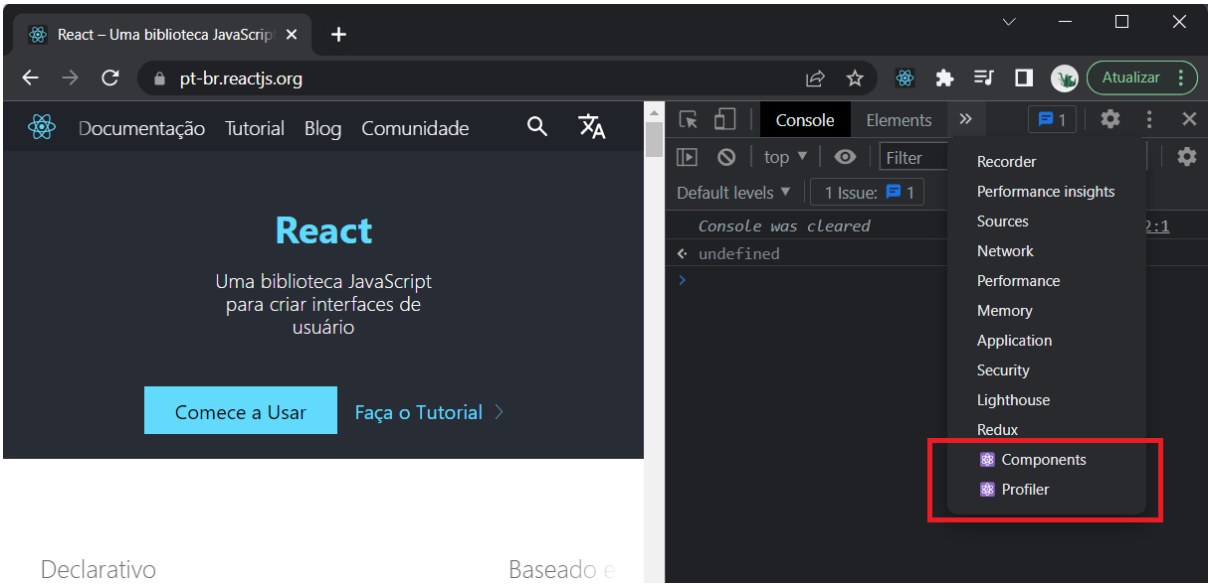
O React DevTools funciona como uma **extensão** dos navegadores Chrome, Chromium e Firefox, por isso sua instalação deve ser feita através das lojas de extensões. Os links abaixo direcionam para a loja de cada navegador:

- Chrome/Chromium: React Developer Tools - Chrome Extension

- Firefox: [React Developer Tools – Firefox Add-on]

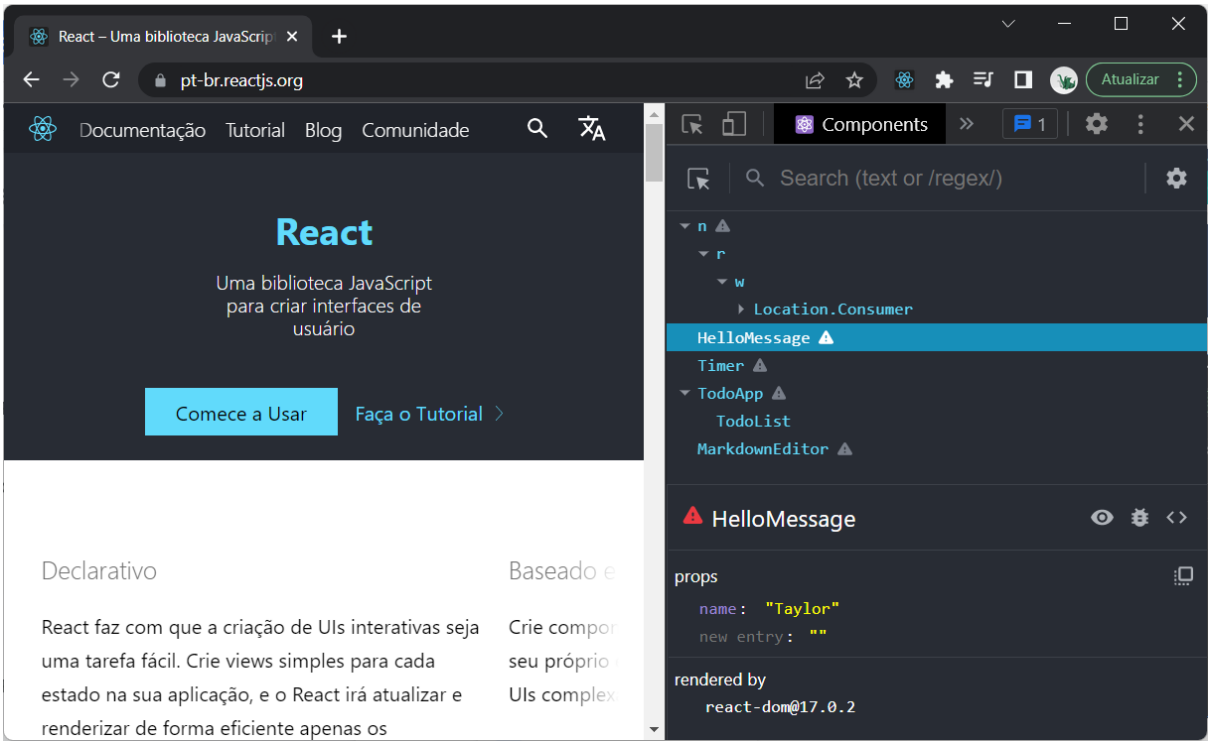
Para toda a aplicação React rodando no navegador, a extensão – se ativada – irá mostrar um ícone na barra de navegação. Ao abrir o DevTools do navegador (F12 no Chrome), as seções de Components e Profiler do React irão aparecer dentre as opções de menu.




Declarativo

Baseado e


React – Uma biblioteca JavaScrip
pt-br.reactjs.org
Atualizar
Documentação Tutorial Blog Comunidade
React
Uma biblioteca JavaScript para criar interfaces de usuário
Comece a Usar
Faça o Tutorial
Declarativo
React faz com que a criação de UIs interativas seja uma tarefa fácil. Crie views simples para cada estado na sua aplicação, e o React irá atualizar e renderizar de forma eficiente apenas os
Components
Search (text or /regex/)
Location.Consumer
HelloMessage
Timer
TodoApp
TodoList
MarkdownEditor
HelloMessage
props
name: "Taylor"
new entry: ""
rendered by
react-dom@17.0.2

# Referências

1. React
2. Crie um novo React App
3. Create React App
4. GitHub - facebook/create-react-app: Set up a modern web app by running one command.
5. React TypeScript Cheatsheets
6. HTML vs JSX – What's the Difference?
7. TypeScript: Documentation - JSX
8. TypeScript e React usando create-react-app: um guia passo-a-passo para configurar seu primeiro aplicativo - Agatetepê
9. Iniciando com React – #2 Criando a estrutura do projeto | iMasters
10. Um guia para usar React JS
11. React Virtual DOM Explained in Simple English - Programming with Mosh
12. React Native Basics: Componentes Funcionais vs Classes
13. The Best Guide to Know What Is React [Updated]
14. react - npm
15. GitHub - facebook/react: A declarative, efficient, and flexible JavaScript library for building user interfaces.
16. Reactjs
17. Aprenda sobre os conceitos do ReactJS - The Xcodes
18. Componentes e Props – React
19. Components in React
20. The Art Of Reusable Components
21. Quick Start Overview
22. Apresentando o novo React DevTools
23. React DevTools: 5 things you didn't know you could do
24. React Developer Tools – Acervo Lima
25. styled-components
26. Blog do Matheus Castiglioni | Criando Styled Components Com React

# PROGRAMA STARTER
# FULL STACK WEB DEVELOPER

ReactJS