



React

APOSTILA





É proibida a duplicação ou reprodução deste volume, no todo ou em parte, em quaisquer formas ou por quaisquer meios (eletrônicos, mecânico, gravação, fotocópia, distribuição pela internet ou outros), sem permissão prévia do Instituto da Oportunidade Social.

São Paulo
2022

Créditos

Organização e Produção

Davi Domingues

Douglas de Oliveira

Fernando Esquírio Torres

Rafael Lopes dos Santos

Atenção

Em caso de dúvidas, sugestões ou reclamações. Entre em contato com a equipe de conteúdo.

educacional@ios.org.br

ESSA É SUA
APOSTILA!



Ela vai te acompanhar durante todo o período do curso.

Cuide dela com carinho e responsabilidade.

Para que ela chegasse toda cheia de estilo do jeito que você está vendo, muita gente quebrou a cabeça para te entregar a melhor experiência de aprendizagem.

Ótimos estudos!

SUMÁRIO

Introdução ao React.....	5
Instalação do React	17
Componentes funcionais	29
Componentes com Classe	44
Projeto Agenda de Compromissos – Parte 01	54
Projeto Agenda de Compromissos – Parte 02	67
Projeto Agenda de Compromissos – Parte 03	78
Projeto Agenda de Compromissos – Parte 04	88



Introdução ao React

Os objetivos desta aula são:

- Apresentar o React como uma biblioteca para construção de interfaces
- Discutir sobre renderização de elementos React
- Conceituar componentes e props

Bons estudos!

React

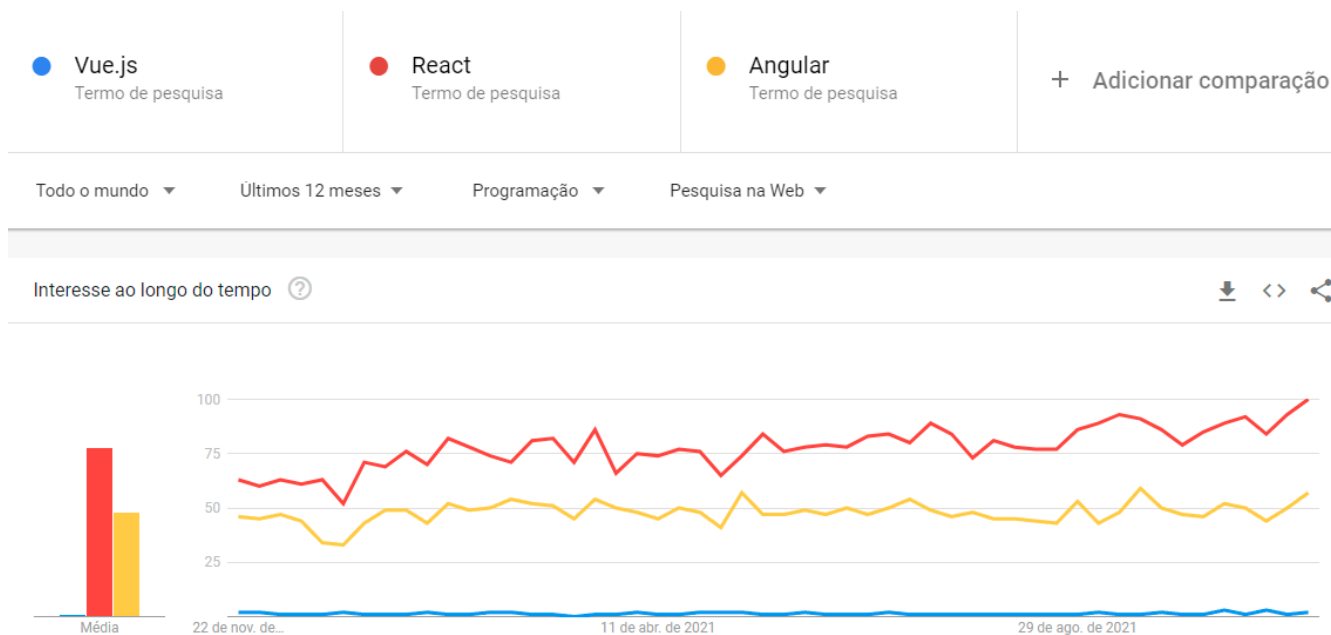
React é uma **biblioteca** usada para a **construção de interfaces**. Ou seja, ele é utilizado para o desenvolvimento de aplicações do **lado do cliente**. O React é frequentemente chamado de um “**framework**” para Frontend, pois possui capacidade semelhantes e é comparado com os frameworks **Angule** e **Vue**.

Uma aplicação construída com o React utiliza o conceito de **Single Page Application (SPA, aplicação de página única)**. SPA é uma aplicação web ou um site de **uma única página**. Ele interage com o usuário **reescrevendo dinamicamente a página atual com o novo dado solicitado**. Isso quer dizer que esse tipo de aplicação **recarrega apenas parte da interface** do usuário (**UI, User Interface**), que **precisa ser atualizada**. O conceito **oposto ao SPA** é o de **múltiplas páginas (MPA)**, que busca a **página inteira no servidor** e **recarrega a nova página completa a cada atualização necessária**.

Utilizar **SPA** no **desenvolvimento de aplicações web** é **vantajoso**, pois o gerenciamento da interface é realizado do **lado do cliente** e as **consultas no servidor só são necessárias para buscas novos dados e não páginas inteiras**. Por isso, as aplicações desenvolvidas com **SPA** fazem as **transições entre páginas mais rápidas** e o usuário tem a sensação de estar navegando por um **aplicativo nativo**.

O **React** é uma biblioteca criada e mantida pelo **Facebook** (**trocou de nome em 2021 para Meta**). Atualmente é a ferramenta **mais popular usada para o desenvolvimento de SPA** de acordo com o **Google Trends**. Disponível no link:

<https://trends.google.com/trends/explore?cat=31&q=Vue.js,React,Angular>.



Tendências do Google.

JSX

O React utiliza uma sintaxe chamada **JSX (JavaScript Syntax Extension)**, que é uma **extensão** da **sintaxe do JavaScript** e a documentação oficial (<https://reactjs.org/docs>) recomenda a utilização dessa linguagem para desenvolver a sua aplicação.

A **vantagem de desenvolver** a sua aplicação **React utilizando o JSX** ao **invés de JavaScript puro** é por ela parecer uma **linguagem de marcação** que possui todo o poder de desenvolvimento **provido pela JavaScript**.

A **sintaxe JSX** é formada de algo que chamamos de **elementos React**:

```
const mensagem = <h1>Hello, world!</h1>;
```

A expressão acima **não é uma string** e **nem uma expressão válida do JavaScript**. Ela é simplesmente uma **instrução em JSX** que armazena na **variável mensagem** o **elemento React** **<h1>Hello, world!</h1>**. Esse elemento parece uma marcação de HTML, mas a diferença é que **podemos mesclar com expressões do JavaScript**. Por exemplo:

```
const nome = 'Irmão do Jorel';
const mensagem = <h1>Olá, {nome}</h1>;
```

Nesse exemplo, o **conteúdo da variável nome** é **incorporado na criação do elemento React** **<h1>**, assim esse elemento pode **mudar o conteúdo exibido dinamicamente**. Falamos dinamicamente porque quando o conteúdo da variável **nome** é **alterado**, o **elemento React** na variável **mensagem também muda**. Observe que para **incorporar a expressão JavaScript** é necessário que você a **insira entre chaves**.

Vejamos outro exemplo:

```
const user = {
  firstName: 'Irmão',
  lastName: 'do Jorel',
};

function formataNome(user) {
  return user.firstName + ' ' + user.lastName;
}

const mensagem = <h1>Hello, {formataNome(user)}!</h1>;
```

Nesse outro exemplo, **incorporamos uma invocação de função em JavaScript**. A função **formataNome** é **chamada**. Então, ela **constrói o nome completo** (**firstName** junto com o **lastName**) e aí ela passa o valor completo para a **construção do elemento React** na variável **mensagem**.

Você pode usar atributos nos elementos React da mesma forma que no HTML, por exemplo:

```
const elemento01 = <div tabIndex="0"></div>;
const elemento02 = <img src={user.avatarUrl}></img>;
```

Note que para **especificar o valor do atributo**, você deve usar o **literal entre aspas** (por exemplo **tabIndex="0"**) e quando você usar um **valor de um objeto do JavaScript**, o mesmo deve ser colocado **entre chaves e sem aspas** (por exemplo **src={user.avatarUrl}**).



Importante: Como o JSX está mais próximo do JavaScript do que do HTML, o **React DOM usa a convenção camelCase de nomenclatura de propriedade** em vez de nomes de atributos HTML. Sendo assim, o atributo **class** do HTML torna-se **className** em JSX e **tabindex** torna-se **tabIndex**.

Outro ponto importante é que o **elemento React** pode ter apenas **um elemento React pai e vários filhos**. Isto é: para ser possível renderizar o elemento React **só podemos ter um elemento na hierarquia mais superior do elemento**.

Por exemplo, o **código (a)** mostra uma **sintaxe inválida para a criação de um elemento React**, pois as marcações **<h1>** e **<h2>** estão no **primeiro nível da hierarquia**. Ou seja, **temos dois elementos pais**. Já o **código (b)** mostra uma **sintaxe válida** com um elemento **pai <div>** e dois elementos **filhos <h1>** e **<h2>**.

Sintaxe inválida

```
// Código (a)
const mensagem = (
  <h1>Olá!</h1>
  <h2>Como é bom ver vocês.</h2>
);
```

Sintaxe válida

```
// Código (b)
const mensagem = (
  <div>
    <h1>Olá!</h1>
    <h2>Como é bom ver vocês.</h2>
  </div>
);
```


A **sintaxe JSX** também **permite criar uma marcação vazia como pai de várias marcações filhas**. O código abaixo mostra **outra sintaxe válida** para a criação do elemento **React**.

```
const mensagem = (
  <>
    <h1>Olá!</h1>
    <h2>Como é bom ver vocês.</h2>
  </>
);
```

Sintaxe válida com marcação vazia.

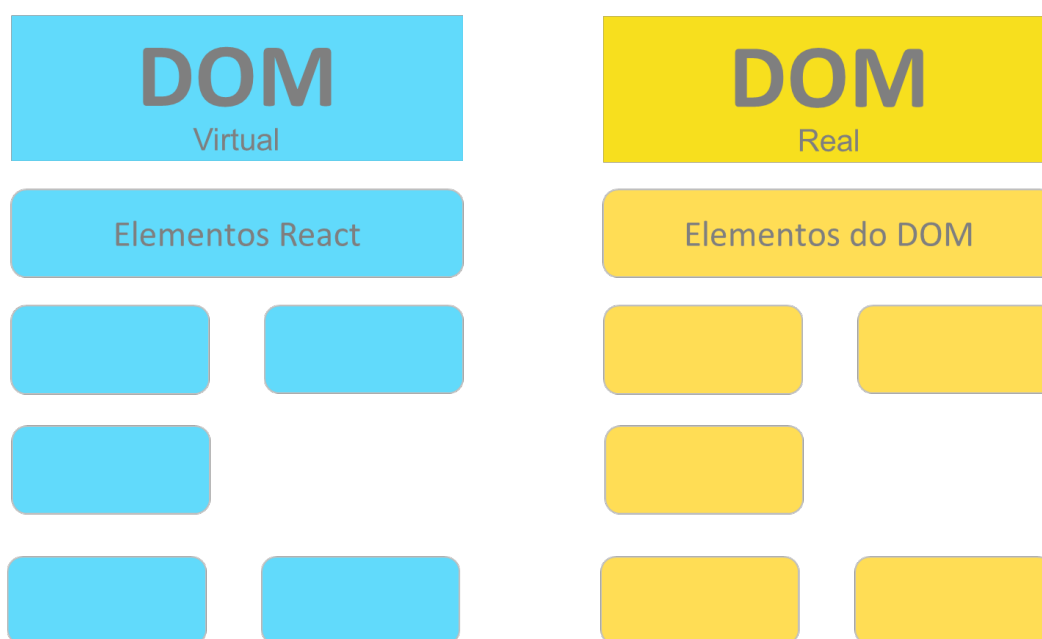
Renderização de elementos React

Um elemento **React** descreve **o que você visualiza na interface que está sendo desenvolvida**. Diferente de elementos **DOM do navegador**, elementos **React** são **objetos simples** e utilizam **menos recursos para serem criados e gerenciados**.

O React possui um **DOM Virtual**, que é formado pelos elementos **React** e é **responsável por atualizar o DOM do navegador (DOM Real)** para exibir os elementos **React renderizados como marcações HTML** na página web.

O **DOM Virtual** é **simples e barato de criar (requer menos recursos)**, pois sempre que um **elemento React é criado** ou tem **seu estado alterado**, ele se **encarrega de atualizar o DOM Real**.

Isso faz com que economizamos **muitas linhas de código**, necessárias se estivéssemos **usando o JavaScript**, por exemplo. Esse é o motivo da biblioteca se chamar **React (Reagir)**, pois ela **“reage” a mudanças de estado e atualiza do DOM**.



Dom Virtual e DOM Real.

Todo **elemento React** é **renderizado na página web**. Renderizar algo, nesse contexto, é **transformar uma sintaxe do JSX em algo que vai ser exibido na página web**. Por exemplo, é comum que no arquivo **index.html** de uma aplicação **React** exista a marcação **<div>** da seguinte forma:

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

Essa marcação é **chamada de nó raiz do DOM**, pois **toda a aplicação React** será **renderizada e gerenciada dentre dessa marcação**. Para renderizar um elemento React em um nó raiz como o mostrado anteriormente, deve-se utilizar o método **render()**. Por exemplo:

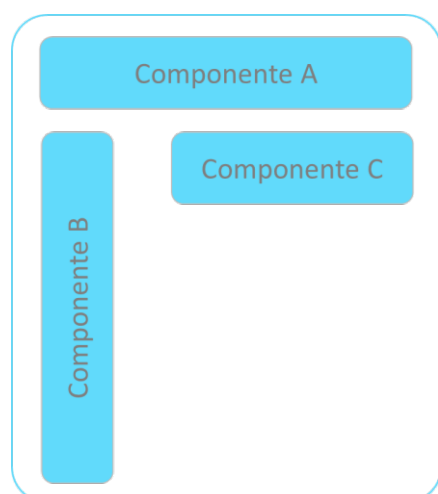
```
const mensagem = <h1>Hello, world</h1>;
ReactDOM.render(mensagem, document.getElementById('root'));
```

O método **render()** é **responsável por descrever como a interface do usuário deverá ser apresentada no navegador web**. O resultado é a exibição de **"Hello World"** na página web. Ou seja, o **elemento React que está armazenado na variável mensagem** é renderizado como uma marcação **HTML <h1>** com o texto **Hello World**.

Esse é um **exemplo simples** e a princípio pode parecer que estamos **gastando muitas linhas de código para fazer escrever** um **"Olá mundo"** em uma página web. Mas é importante você entender como funciona o React para que quando estivermos nos exemplos mais avançados ficará mais fácil de você compreendê-los.

Componentes

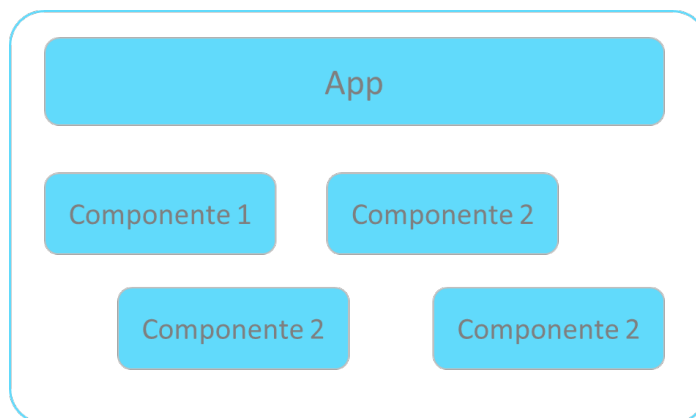
O **coração de toda aplicação** desenvolvida em **React** são os **componentes**. Os componentes são essencialmente uma **parte da interface do usuário** e podem ser **formados por um ou mais elementos React**. Quando estamos construindo **aplicação com React** estamos construindo **vários componentes independentes, isolados e reutilizáveis**. E esses componentes juntos são usados para compor uma interface do usuário complexa.



Interface do usuário com diversos componentes

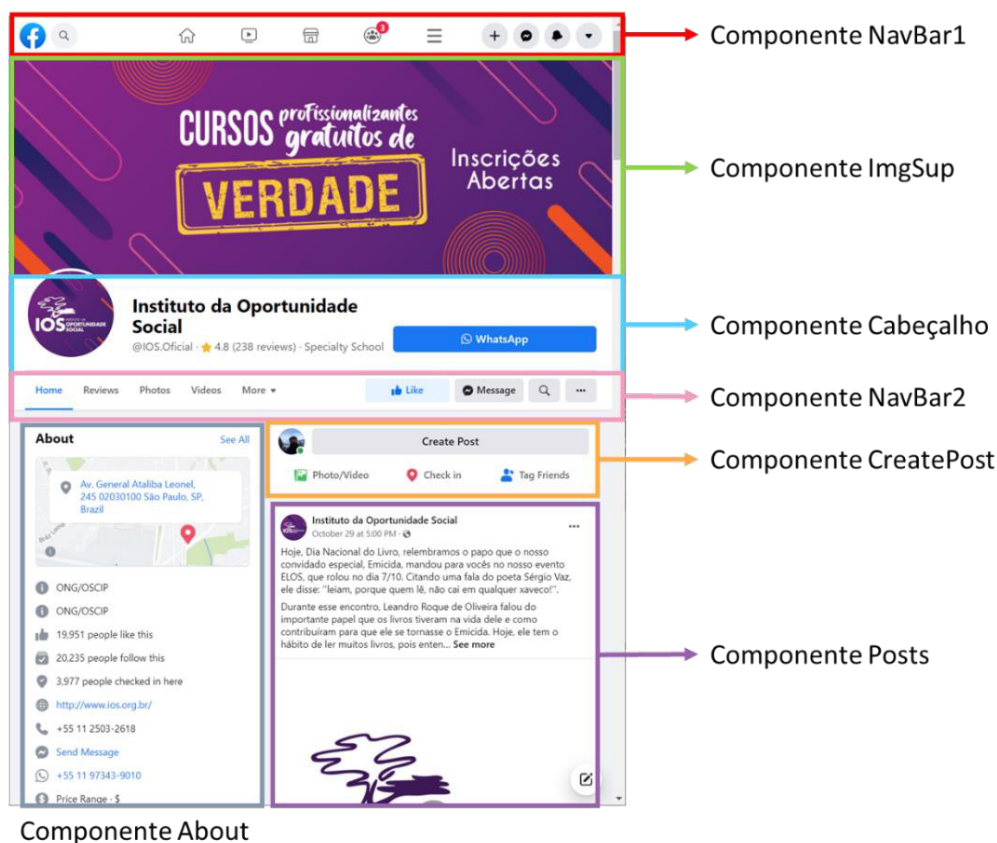
Todas as aplicações **React** têm pelo menos **um componente**, que é referenciado como **componente raiz** e na **maioria das aplicações que você encontrar** o nome desse componente é **App**. Esse componente representa a **aplicação interna** e contém outros **componentes filhos**.

Então, toda aplicação React é essencialmente uma **árvore de componentes**, onde existe um **único componente raiz (App)**, que está no **topo da hierarquia dos componentes**.



Árvore de componentes.

Vamos ver um exemplo, podemos dividir a página do **Facebook do IOS** em **diversos componentes** como mostrado na imagem abaixo. E esses componentes juntos constroem a interface da página do Facebook.



Página do Facebook do IOS dividida em diversos componentes.

Em termos de implementação, o **componente** pode ser implementado como uma **classe** ou como uma **função do JavaScript**. **Componentes funcionais** foram introdução no React a partir da versão **16.8** no final de **2019** com a inserção do conceito de **hooks**, **até então** os componentes eram criados **apenas utilizando classes**. Atualmente, é bem provável que você irá encontrar em uma empresa uma **implementação híbrida**, que utiliza a **abordagem de componentes com classe e componentes funcionais**.

```
// Código (a)
function Mensagem() {
  return <h1>Olá, pessoal!</h1>;
}
```

```
// Código (b)
class Mensagem extends React.Component {
  render() {
    return <h1>Olá, pessoal!</h1>;
  }
}
```

Componente Mensagem implementado com a abordagem (a) de função e (b) de classe.

Nesse primeiro, material vamos utilizar a **abordagem mais recente** e implementar **os nossos componentes utilizando funções**. No exemplo, mostrado nos códigos acima, o **componente Mensagem** renderiza um cabeçalho **<h1>** com o conteúdo **“Olá, pessoal”**. Ou seja, ele cria uma marcação HTML **<h1>** na página web e **exibe esse conteúdo na página**.

O conteúdo dentro do **return** é o elemento do **React** que **será criado na interface gráfica**, quando o componente for renderizado.



Importante: Observe que quando o componente é implementado utilizando a **abordagem de classe**, precisamos colocar o método **render()** para indicar a parte que será renderizada.

Props

Componentes podem ter **props** (**Abreviação de propriedades**), que é a maneira de **passar uma informação** para um **componente**. Props também são usadas para **passar dados** de um **componente pai** para um **componente filho** na hierarquia de componentes.

Por exemplo, o componente **Mensagem** mostrado no código abaixo possui uma **props** chamada **name**, que é passada na criação do elemento React (**const elemento = <Mensagem name="Irmão do Jorel" />**).

Na instrução **<Mensagem name="Irmão do Jorel" />**, a **propriedade é passada** como se fosse um **atributo de uma marcação em HTML**.

```
function Mensagem(props) {
  return <h1>Olá, {props.name}</h1>;
}


const elemento = <Mensagem name="Irmão do Jorel" />;

ReactDOM.render(elemento, document.getElementById('root'));
Componente Mensagem com uma propriedade name.
```

Observe que o **valor da propriedade name** é um **literal**, portanto deve ser **passado entre aspas**. Note que o nome da propriedade **deve ser o mesmo quando for acessado dentro do componente**. No código acima, quando chamamos o método **ReactDOM.render()** com o elemento, o React chama o componente **Mensagem** e **passa {name: "Irmão do Jorel"}** como **propriedade para o componente**.

O componente **Mensagem** retorna um elemento **<h1>Hello, Irmão do Jorel</h1>** como resultado (instrução **return <h1>Olá, {props.name}</h1>;**). Pois o valor passado por meio da **props** para o componente é inserido na parte **{props.name}**, quando o elemento é criado.

Não importa o nome que você der para a props, desde que você siga as **regras de criação de variáveis e funções** de toda linguagem de programação e use o **nome corretamente** quando quiser **acessar a propriedade** dentro do **componente**.

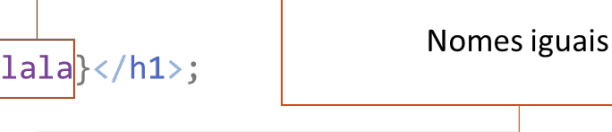


```
function Mensagem(props) {
  return <h1>Olá, {props.name}</h1>;
}

const elemento = <Mensagem name="Irmão do Jorel" />;

ReactDOM.render(elemento, document.getElementById('root'));
```

Nomes iguais



```
function Mensagem(props) {
  return <h1>Olá, {props.lala}</h1>;
}

const elemento = <Mensagem lala="Irmão do Jorel" />;

ReactDOM.render(elemento, document.getElementById('root'));
```

Nomes iguais

Nomes das propriedades em um componente.

Estado de um componente

Os componentes no **React não são apenas marcações estáticas**, eles têm um **conteúdo dinâmico** chamado **estado**. O estado é basicamente um **objeto**, que determina como um **componente será renderizado**. Ou seja, seu **comportamento dentro da UI**. Um estado criado no **componente raiz (App)** é considerado **global**. Ou seja, pode ser visto por todos os **componentes da aplicação** e não somente pelo componente raiz.

Se o **estado** de um componente for **modificado**, isso irá provocar a **renderização do componente** e **atualização automática da parte da UI** que esse componente representa. Essa é a “**magia**” por trás do React, **reagir atualizando** automaticamente as partes que compõe da interface do usuário.

O bloco de código (a) mostra um componente funcional chamado **Exemplo** com o estado **contador** e a utilização do **hook useState**, que permite trabalhar com estados em componentes implementados com **função**. E no bloco de código (b), temos o código de um componente **Exemplo** utilizando **classe** com o **estado (state)** contendo o campo contador.

```
// Código (a)
import { useState } from 'react';

function Exemplo() {
  // Declara uma nova variável de estado, a qual chamaremos de "contador"
  const [contador, setContador] = useState(0);

  return (
    <div>
      <p>Você clicou {contador} vezes</p>
      <button onClick={() => setContador(contador + 1)}>
        Clique aqui
      </button>
    </div>
  );
}
```

```
// Código (b)
class Exemplo extends Component {
  // Declare uma nova variável de estado, a qual chamaremos de "contador"
  // e é um campo do objeto state
  constructor(props) {
    super(props);
    this.state = {
      contador: 0,
    };
  }

  render() {
    return (
```

```

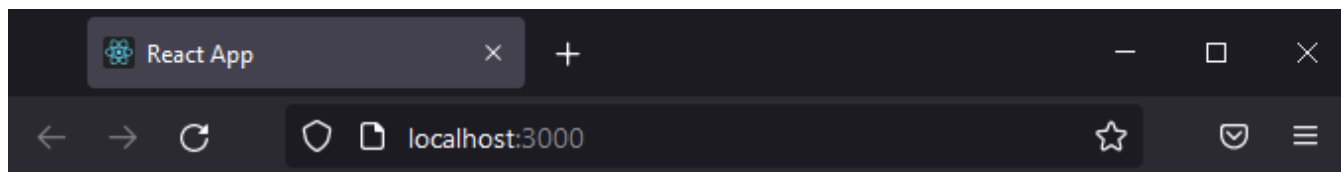
    <div>
      <p>Você clicou {this.state.contador} vezes</p>
      <button
        onClick={() =>
          this.setState({ contador: this.state.contador + 1 })
        }
      >
        Clique aqui
      </button>
    </div>
  );
}
}

```

Estado de um componente (a) na abordagem de componentes funcionais e (b) na abordagem de componentes com classe.

Em ambos os códigos mostrados acima, o componente **Exemplo**, quando renderizado na página web, mostra um **parágrafo** com uma frase mostrando o **valor do contador** e um **botão** para **incrementar o contador** e atualizar a informação mostrada no parágrafo.

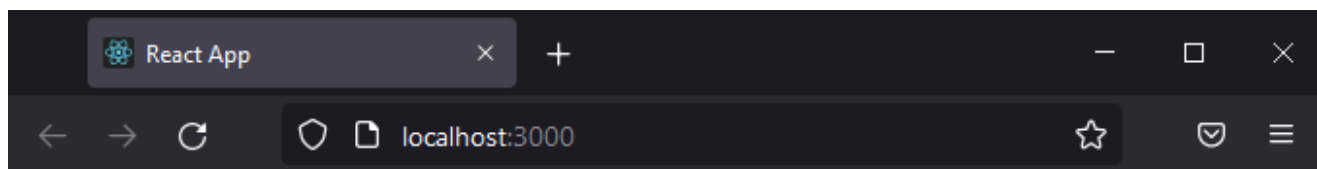
Aplicação iniciado no navegador.



Você clicou 0 vezes

Clique aqui

Aplicação React depois de ter clicado por duas vezes no botão.



Você clicou 2 vezes

Clique aqui

Aplicação com o componente Exemplo executando no navegador.

Por que você deve usar o React?

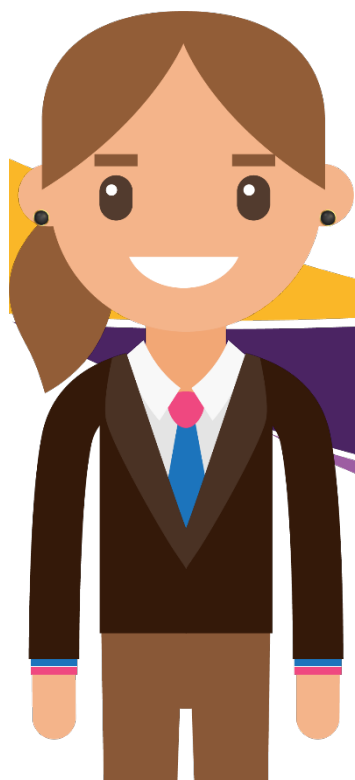
Existem diversos motivos para **utilizar o React** no desenvolvimento de suas aplicações, podemos citar alguns:

- O React estrutura a parte visual da sua aplicação. Se você tentar construir uma aplicação grande e complexa utilizando apenas Vanilla JavaScript, a sua implementação tem grandes chances de se tornar uma bagunça com os códigos HTML, estilos CSS e JavaScript por todo lugar que for possível.
- O React permite construir uma interface do usuário utilizando componentes reutilizáveis. Desse modo, cada parte da interface é representada por um componente com estado, dados e métodos.
- O React utiliza a sintaxe JSX, que é uma linguagem de marcação que não necessita separar a parte de marcação e a parte lógica como acontece com o HTML e JavaScript. A linguagem é basicamente o JavaScript, porém é formatada como se fosse o HTML.
- As aplicações construídas com o React são bem interativas, pois elas usam o DOM Virtual, que permite atualizar parte da página sem a necessidade de recarregá-la completamente. O gerenciamento da página sendo feito pelo DOM Virtual, torna a aplicação mais rápida, dinâmica e interativa.
- O React traz benefícios de performance e para a realização de testes da sua aplicação.

O que você deve saber antes de aprender React?

Você deve saber bem trabalhar com a linguagem JavaScript, principalmente os seguintes assuntos:

- Tipos de dados, variáveis, funções, loop, etc.
- Métodos de alto nível para a manipulação de arrays, tais como: `forEach()`, `map()`, `filter()`, etc.



Instalação do React

Os objetivos desta aula são:

- Exercitar a criação da primeira aplicação em React
- Apontar formas de otimizar arquivos de projetos em React
- Discutir sobre estrutura dos arquivos do projeto
- Apresentar a Extensão React Developer Tools

Bons estudos!

Criando uma aplicação em React

Antes de criar qualquer aplicação em React, você precisa ter o software **Node.js** instalado no seu computador. O **Node.js** é um **software de código aberto** que permite executar códigos **JavaScript fora de um navegador web**. Ele pode ser instalado nos sistemas operacionais **Windows, Linux** e **MacOS**.

Precisamos do **Node.js** para utilizarmos os gerenciadores pacotes (**bibliotecas**) **NPM (Node Package Manager)** e **NPX (Node Package Execute)** e também para executarmos os códigos **JavaScript** necessários para rodar uma aplicação React. O software Node.js está disponível para download em: <https://nodejs.org/pt-br/>.

O **NPX** será utilizado para criar a **aplicação React**, enquanto o **NPM** será utilizado para **instalar bibliotecas de terceiros**, que podem ser necessárias e **não são instaladas por padrão** ao criar a aplicação React.



Importante! O fato de você desenvolver a sua aplicação React e executá-la em um ambiente Node.js, faz-se necessário o uso de muito recurso computacional. Isto é, a execução e atualização de uma aplicação React no navegador pode ficar lenta em computadores com pouco poder computacional.

Instalando a aplicação React

Para **instalar/criar** uma aplicação **React** você deve **escolher um diretório de workspace** no **VS Code**. Por exemplo, no meu caso todas as aplicações serão criadas no meu diretório **React** e esse diretório. No meu caso é um **repositório local do meu GitHub**:



```

TERMINAL  PROBLEMAS  SAÍDA  CONSOLE DE DEPURACÃO
Rafael Lopes@NB-IOS-CZB8ZH3 MINGW64 /c/react (main)
$ 
  
```

Escolhido o diretório local, você deve usar o **seguinte comando** no terminal da sua **IDE** para criar a sua aplicação React:

```
npx create-react-app my-app
```

Onde: my-app deve ser **substituído** pelo **nome da sua aplicação** que será criada. O nome da aplicação deve **conter apenas letras minúsculas** e **não pode conter espaços em branco** ou **caracteres especiais**. Assim podemos executar no terminal podemos por exemplo executar o comando:

```
npx create-react-app tema_02_exemplo_01
```

Esse comando **cria um diretório** com o **nome da sua aplicação**, instala os **módulos** (**pacote** ou **bibliotecas**) necessárias para **executar a aplicação React**. Na próxima secção, falaremos da organização dos arquivos criadas ao instalarmos uma aplicação React.

Após finalizada a instalação, a seguinte instrução é apresentada no terminal:

```

TERMINAL  PROBLEMAS  SAÍDA  CONSOLE DE DEPURAÇÃO
bash + -  [ ] [ ] ^ x

npm test
Starts the test runner.

npm run eject
Removes this tool and copies build dependencies, configuration files
and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

cd tema_02_exemplo_01
npm start

Happy hacking!

Rafael Lopes@NB-IOS-CZB8ZH3 MINGW64 /c/react (main)
$ 

```

A primeira **cd tema_02_exemplo_01** deve ser usada para **entrar no diretório da aplicação criada** e a segunda **npm start** deve ser usada para **inicializar a aplicação** e mostrá-la executando no navegador.

Importante: o comando **npm start** é usado **sempre que quisermos executar** e visualizar a aplicação **React** no navegador. Esse comando deve ser **sempre executado dentro do diretório, onde a aplicação está instalada**.

Sendo assim, podemos executar no terminal:

```

TERMINAL  PROBLEMAS  SAÍDA  CONSOLE DE DEPURAÇÃO
bash + -  [ ] [ ] ^ x

Rafael Lopes@NB-IOS-CZB8ZH3 MINGW64 /c/react (main)
$ cd tema_02_exemplo_01/

Rafael Lopes@NB-IOS-CZB8ZH3 MINGW64 /c/react/tema_02_exemplo_01 (main)
$ npm start

```

Então o terminal mostra o servidor node local criado para a execução da aplicação

```

TERMINAL  PROBLEMAS  SAÍDA  CONSOLE DE DEPURAÇÃO
bash + -  [ ] [ ] ^ x

Compiled successfully!

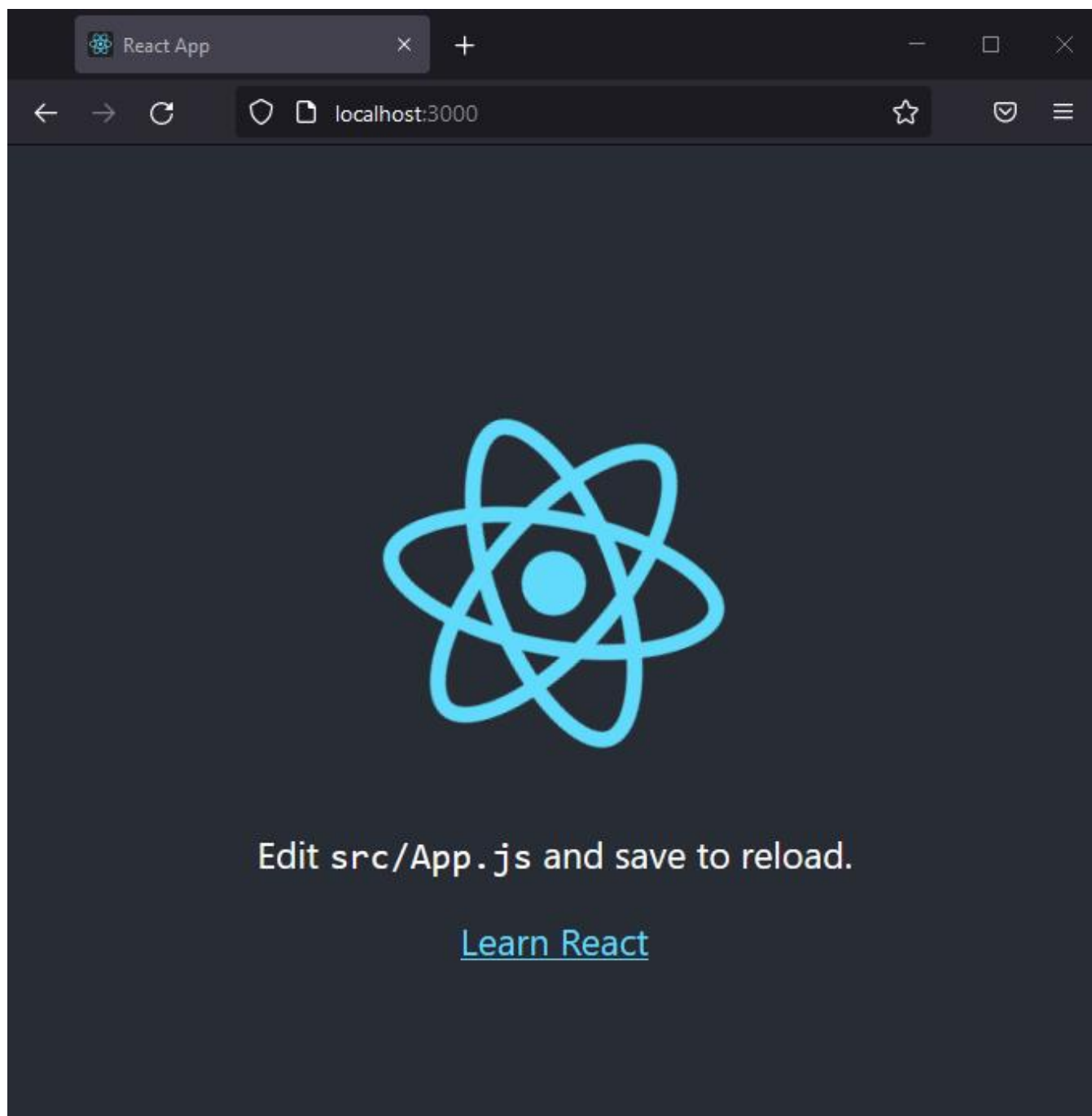
You can now view tema_02_exemplo_01 in the browser.

Local:      http://localhost:3000
On Your Network: http://192.168.0.169:3000

Note that the development build is not optimized.
To create a production build, use npm run build.

```

No navegador web você pode visualizar a aplicação padrão sendo executada.

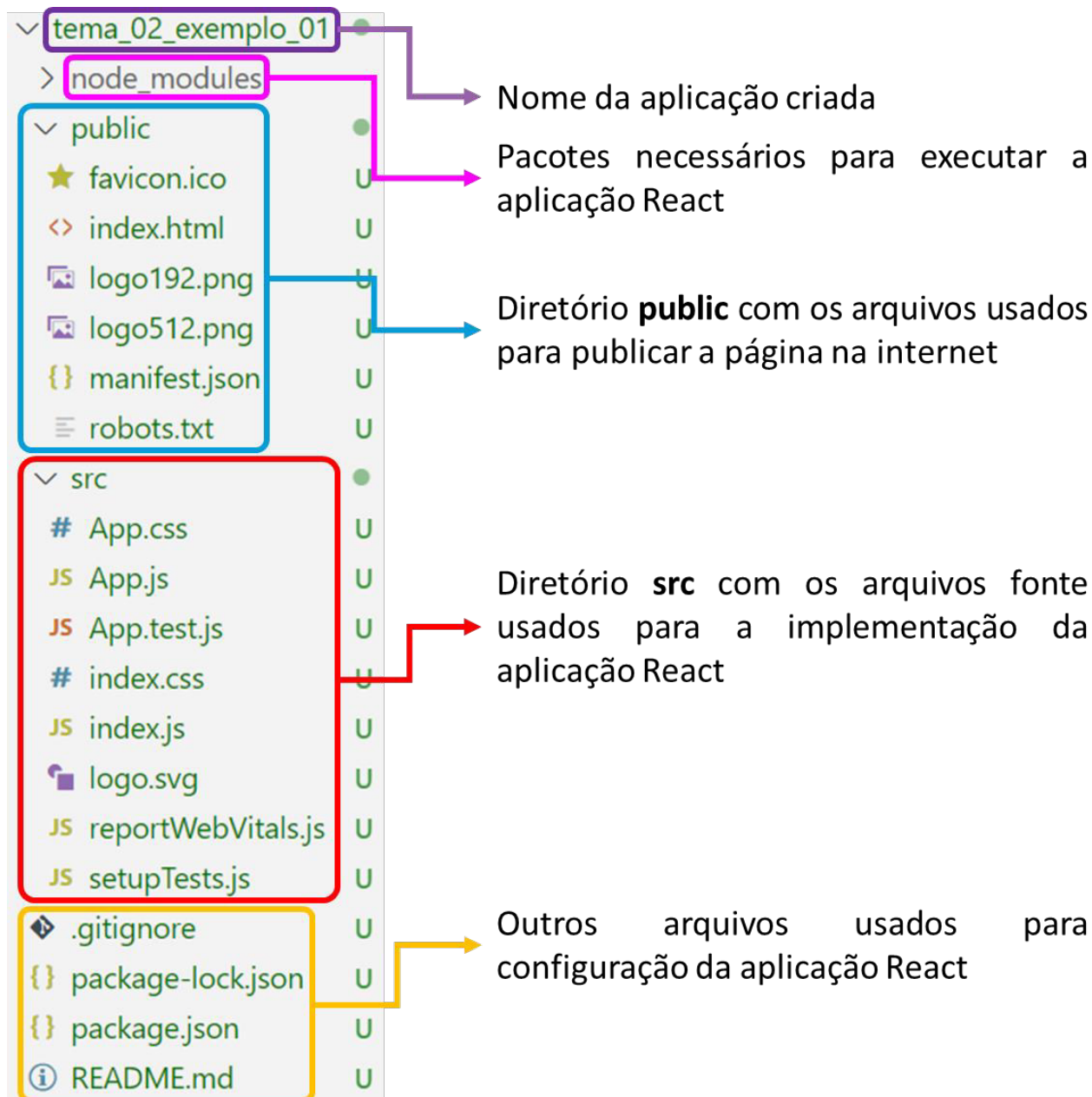


A página mostrada no navegador web é uma **aplicação pré-instalada** quando criamos nosso projeto **React** com o comando **npx create-react-app**. Observe que a aplicação React executa na porta **3000** de um **servidor local** criado no seu computador com o comando **npm start**.

Caso você queria terminar a execução da aplicação React no navegador você deve aperta as teclas **Ctrl+C** no terminal da **IDE** e fechar o navegador.

Organização dos arquivos do projeto

Ao instalar uma aplicação React, podemos visualizar a seguinte organização dos arquivos:



Primeiro, podemos notar o **nome do diretório raiz da aplicação** é o mesmo nome que **demos na instalação do React (tema_02_exemplo_01)**. Em seguida, podemos ver o diretório **node_modules**, que contém diversas **bibliotecas usadas pela aplicação React**.



Importante! Se você tiver que instalar algum pacote extra (por exemplo: Bootstrap, react-router, redux, etc.) utilizando o gerenciador de pacotes **NPM**, ele será armazenado no diretório **node_modules**. Além disso, se você abrir o arquivo **.gitignore** verá que o diretório **node_modules** está na lista de ignorados para serem armazenados no repositório online. Isso faz sentido, porque é um diretório com biblioteca que devem estar instaladas no ambiente de desenvolvimento e não no repositório.

```
.gitignore U x
tema_02_exemplo_01 > .gitignore
1 # See https://help.github.com/articles/ignoring-files/ for more about ignoring files.
2
3 # dependencies
4 /node_modules
5 /.pnp
6 .pnp.js
7
8 # testing
9 /coverage
10
11 # production
12 /build
13
14 # misc
15 .DS_Store
16 .env.local
17 .env.development.local
18 .env.test.local
19 .env.production.local
20
21 npm-debug.log*
22 yarn-debug.log*
23 yarn-error.log*
```

Continuando a análise da organização dos arquivos, temos o diretório **public**, que contém os **arquivos usados para colocar a aplicação em um ambiente de produção**. Quando a implementação estiver **finalizada** você pode executar o comando **npm build**. Ele gerará os códigos (**HTML, CSS e JavaScript**) e **arquivos da aplicação**, que devem ser colocados no **servidor de hospedagem** do site.

Se você abrir o arquivo **index.html** e **apagar os comentários**, poderá ver a **estrutura básica da aplicação React**.

```
1. <!DOCTYPE html>
2. <html lang="pt-br">
3.   <head>
4.     <meta charset="utf-8" />
5.     <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
6.     <meta name="viewport" content="width=device-width, initial-scale=1" />
7.     <meta name="theme-color" content="#000000" />
8.     <meta
9.       name="description"
10.      content="Web site created using create-react-app"
11.    />
12.     <link rel="apple-touch-icon" href="%PUBLIC_URL%/logo192.png" />
13.     <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
14.     <title>React App</title>
15.   </head>
16.   <body>
17.     <noscript>You need to enable JavaScript to run this app.</noscript>
18.     <div id="root"></div>
19.   </body>
20. </html>
```

Nessa estrutura você pode alterar o **título** do site na **linha 14**. Observe na **linha 18** a instrução **<div id="root"></div>**, local onde **toda a aplicação React será renderizada**.

Em seguida, temos o diretório **src**, que onde **criaremos** nossos **arquivos** colocaremos nossos **códigos** e faremos toda a **implementação necessária** dentro da aplicação React. Se você abrir

o arquivo **index.js**, poderá visualizar o método **render()** usado para renderizar o componente raiz **App** no elemento **<div>** do arquivo **index.html** que contém a **ID root**.

```
1. import React from 'react';
2. import ReactDOM from 'react-dom';
3. import './index.css';
4. import App from './App';
5. import reportWebVitals from './reportWebVitals';
6.
7. ReactDOM.render(
8.   <React.StrictMode>
9.     <App />
10.   </React.StrictMode>,
11.   document.getElementById('root')
12. );
```

Além do arquivo **index.js**, temos os arquivos:

- **index.css**: contém os estilos CSS que podem ser acessados por todos os componentes da nossa aplicação.
- **App.js**: componente raiz da aplicação React que é renderizado na linha 9 do código do arquivo **index.js**.
- **App.css**: contém os estilos CSS que podem ser acessados pelo componente **App** e pelos seus componentes filhos.

Por fim, temos os arquivos extras:

- **configuração do git (.gitignore)**. Se você não tiver um repositório local configurado no seu workspace ou o git configurado no VS Code esse arquivo não irá aparecer.
- **package.json**: contém a configuração da aplicação React, onde é possível visualizar as dependências, os scripts e outras informações do projeto.

```
1. {
2.   "name": "tema_02_exemplo_01",
3.   "version": "0.1.0",
4.   "private": true,
5.   "dependencies": {
6.     "@testing-library/jest-dom": "^5.15.0",
7.     "@testing-library/react": "^11.2.7",
8.     "@testing-library/user-event": "^12.8.3",
9.     "react": "^17.0.2",
10.    "react-dom": "^17.0.2",
11.    "react-scripts": "4.0.3",
12.    "web-vitals": "^1.1.2"
13.  },
14.  "scripts": {
15.    "start": "react-scripts start",
16.    "build": "react-scripts build",
17.    "test": "react-scripts test",
18.    "eject": "react-scripts eject"
19.  },
20.  "eslintConfig": {
21.    "extends": [
22.      "react-app",
23.      "react-app/jest"
24.    ]
25.  },
26.  "browserslist": {
```

```

27.   "production": [
28.     ">0.2%",
29.     "not dead",
30.     "not op_mini all"
31.   ],
32.   "development": [
33.     "last 1 chrome version",
34.     "last 1 firefox version",
35.     "last 1 safari version"
36.   ]
37. }
38. }

```

Estrutura dos arquivos do projeto

Não existe uma forma única de criar a estrutura de diretórios e arquivos no seu projeto React, o que existe é uma recomendação na documentação oficial do React (<https://pt-br.reactjs.org/docs/faq-structure.html>) para se usar um dos dois tipos de agrupamentos de arquivos: por funcionalidades/rotas ou por tipos de arquivos:

Agrupamento por funcionalidades/rotas	Agrupamento por tipo de arquivos
common/ Avatar.js Avatar.css APIUtils.js APIUtils.test.js feed/ index.js Feed.js Feed.css FeedStory.js FeedStory.test.js FeedAPI.js profile/ index.js Profile.js ProfileHeader.js ProfileHeader.css ProfileAPI.j	api/ APIUtils.js APIUtils.test.js ProfileAPI.js UserAPI.js components/ Avatar.js Avatar.css Feed.js Feed.css FeedStory.js FeedStory.test.js Profile.js ProfileHeader.js ProfileHeader.css

Nos nossos projetos, usaremos o agrupamento por tipo de arquivos por ser mais simples de ser compreendido.

Vamos praticar

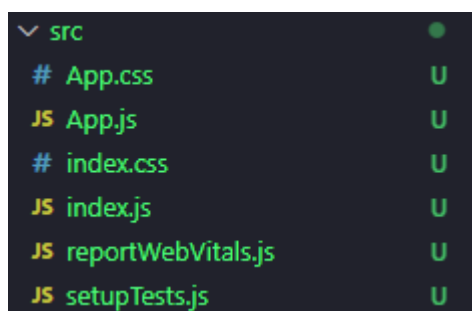
Vamos **modificar** o **código da aplicação** padrão do **React** e **visualizar o resultado no navegador web**. Se sua aplicação **não estiver inicializada** ou se **você fechou o navegador**

web, você deve executar novamente o comando **npm start**, para visualizar as mudanças realizadas no código. Siga os passos para realizarmos as alterações:

Primeiramente, apague alguns arquivos do diretório **src**, que **não usaremos nessa aplicação**:

- App.test.js
- logo.svg

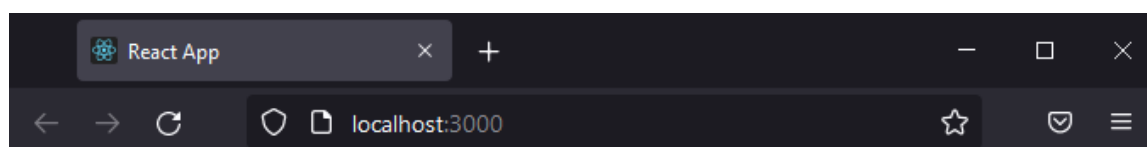
Seu diretório, deve ficar como esse:



Em seguida, podemos **limpar** o conteúdo do arquivo **App.js** e atualizá-lo como mostrado abaixo:

```
1. import './App.css';
2.
3. function App() {
4.   return (
5.     <div className="App">
6.       <h1>Hello World</h1>
7.     </div>
8.   );
9. }
10.
11. export default App;
```

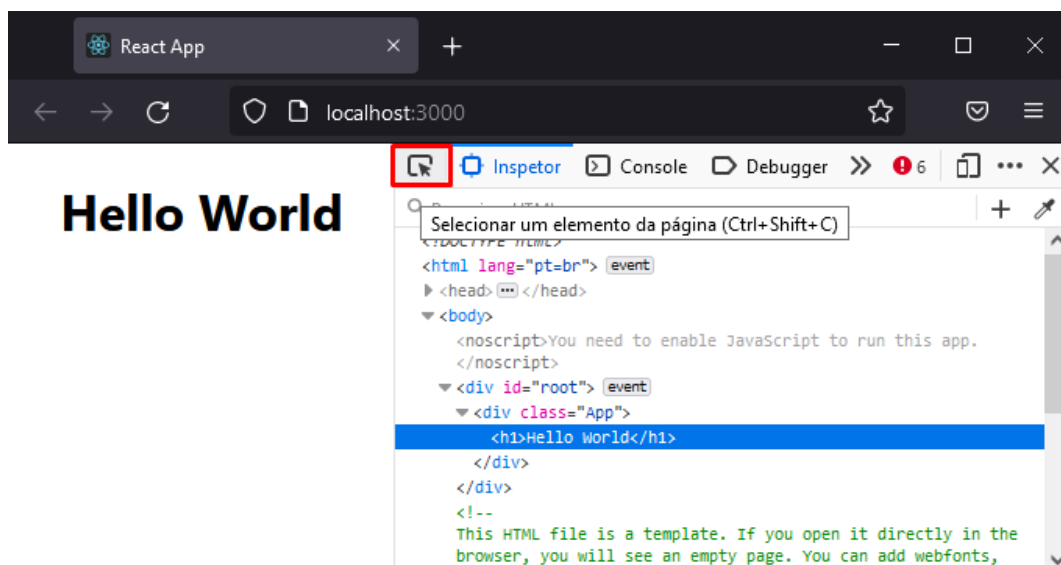
O resultado pode ser visualizado no navegador web.



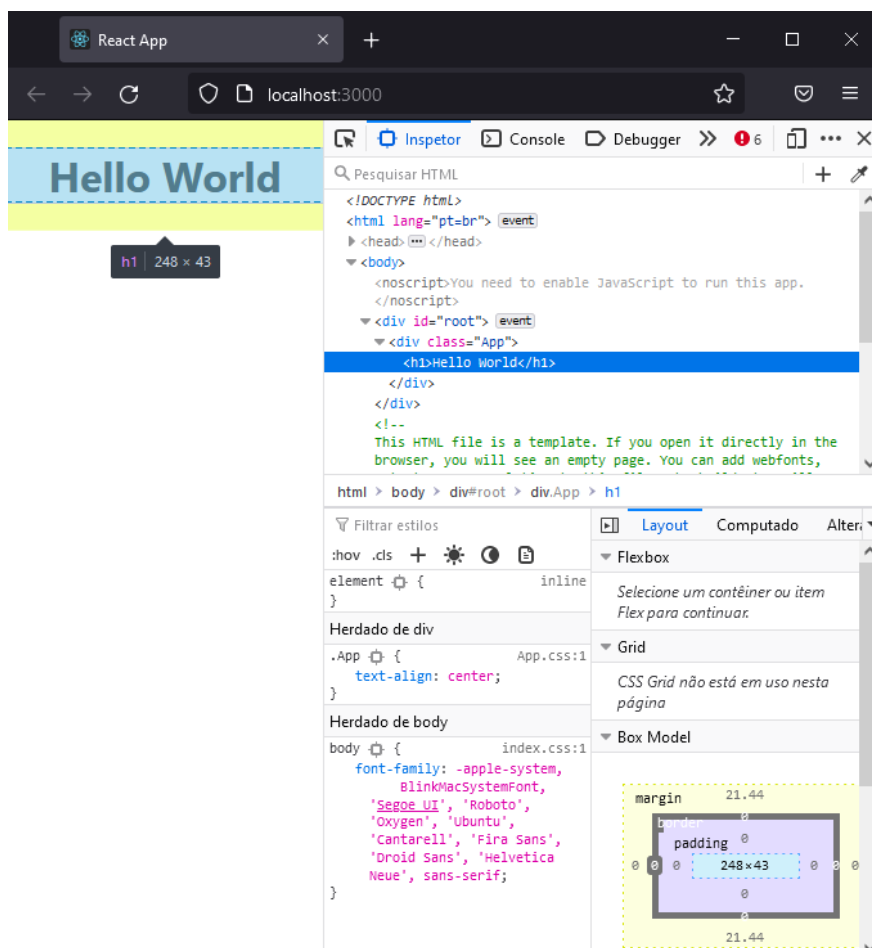
Hello World

Observe que o código do **componente App** está utilizando a sintaxe **JSX**, muito **semelhante ao HTML**. Mas você notará mais a diferença quando começarmos a incorporar o **JavaScript** e criarmos os **nossos componentes da aplicação**.

Na **linha 6** do código do componente **App** inserimos uma instrução para criar uma marcação **heading <h1>** com o conteúdo **Hello World**. E foi exatamente isso que aconteceu. Se abrirmos as ferramentas de **debug** do **navegador web** (**tecla de atalho F12**) e utilizarmos a ferramenta para selecionar **um elemento da página**.



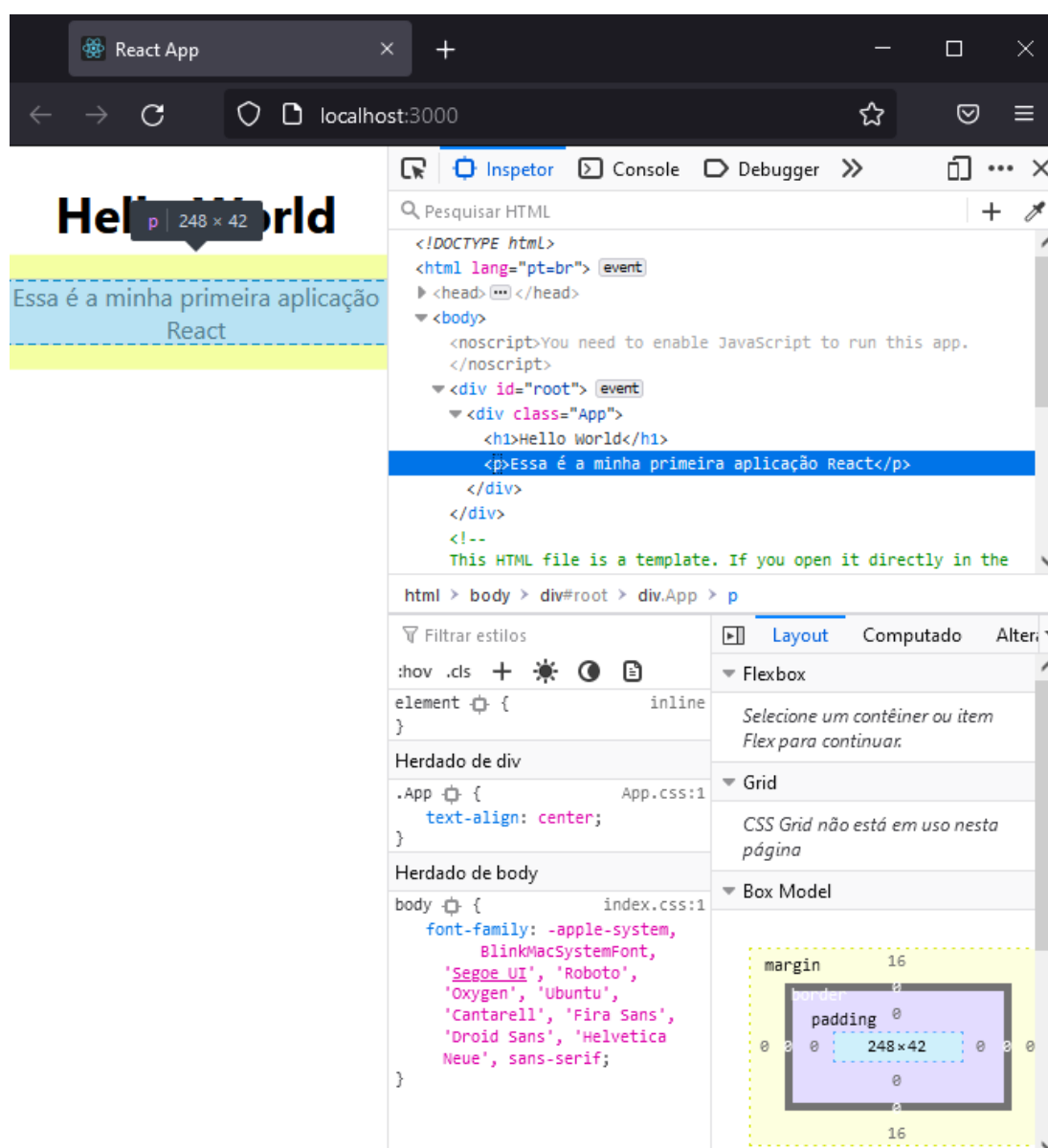
Podemos ver a marcação HTML **<h1>**, que foi criada **dentro da marcação <div>** com **ID root** (**local onde a aplicação React é renderizada na página web**).



Podemos **atualizar** o código do **componente App** e inserir uma instrução para criar um elemento parágrafo:

```
1. import './App.css';
2.
3. function App() {
4.   return (
5.     <div className="App">
6.       <h1>Hello World</h1>
7.       <p>Essa é a minha primeira aplicação React</p>
8.     </div>
9.   );
10. }
11.
12. export default App;
```

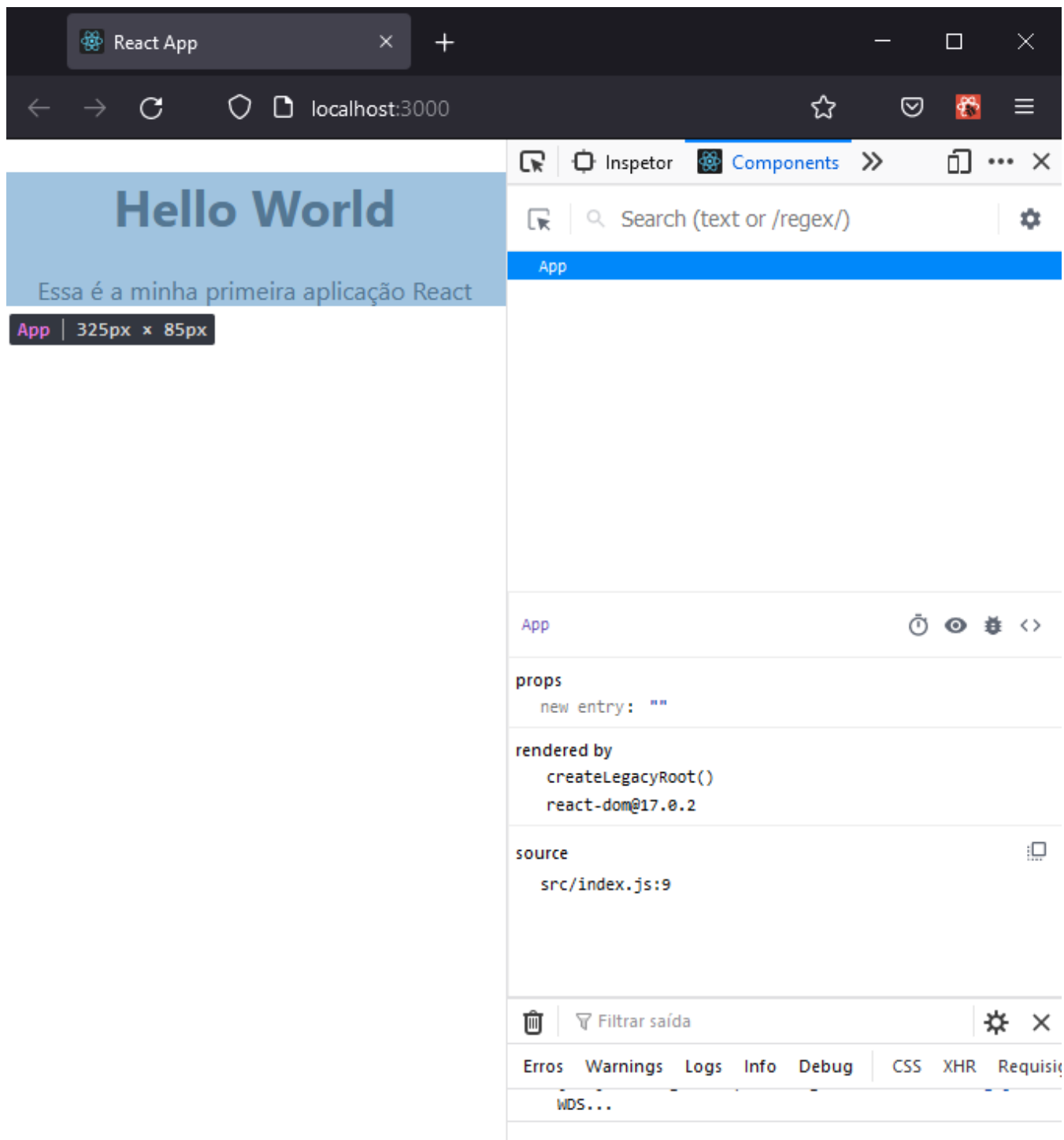
O resultado pode ser visualizado no navegador web.



Agora temos também um parágrafo na aplicação web.

Extensão React Developer Tools

Se abrirmos a opção **Components** da extensão **React Developer Tools**, podemos ver o único **componente da aplicação chamado App**, esse componente não mostra nenhum estado (**state** ou **hooks**) ou propriedades (**props**).





Componentes funcionais

Os objetivos desta aula são:

- Apresentar as funções React Hooks
- Exercitar a criação de componentes funcionais

Bons estudos!

Hooks

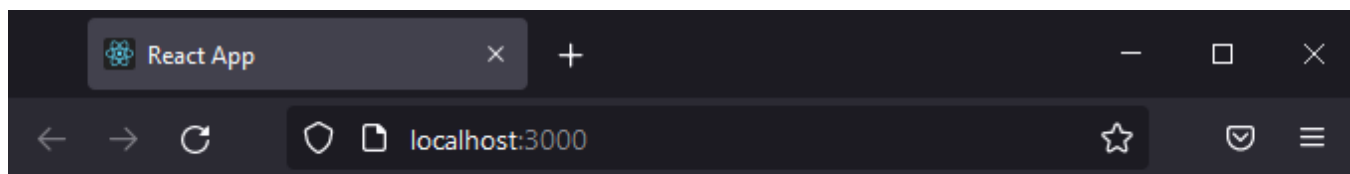
React Hooks são **funções** que nos permitem usar **estado (state)** e **recursos do ciclo de vida de um componente** quando utilizamos componentes funcionais. Os **hooks não funcionam dentro de classes**. Os principais hooks são:

- **useState**: usado para criar o estado de um componente, atribuir um valor inicial para o estado e, também, uma função para que possamos atualizar esse estado.
- **useEffect**: usado para produzir efeitos colaterais em componentes funcionais.

Além desses dois existem o `useContext`, `useReducer`, `useRef`, `useLocation`, entre outros. Para saber mais: Você pode acessar a página de Introdução aos Hooks da documentação oficial disponível em: <https://pt-br.reactjs.org/docs/hooks-intro.html>

Criando componentes funcionais

Vamos fazer o exemplo do tema 01 com o **componente funcional** que possui um **parágrafo** e um **botão**, onde o **parágrafo mostra quantas você clicou no botão da interface**.



Instalando a aplicação React

Escolha um diretório de trabalho, acesse esse diretório pelo **terminal do VS Code** e execute o seguinte comando.

```
npx create-react-app tema_03_conta_cliques
```

Criando o componente Exemplo

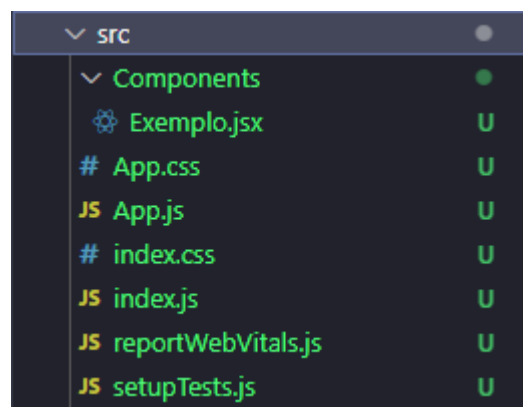
Com a aplicação instalada, vamos criar um diretório chamado **Components**, que deve estar dentro da pasta **src**. Dentro desse novo diretório **Components**, vamos **criar** o arquivo do nosso componente com o nome **Exemplo.jsx**.

Primeiramente, apagar alguns arquivos do diretório **src**, que **não usaremos nessa aplicação**.

São os arquivos:

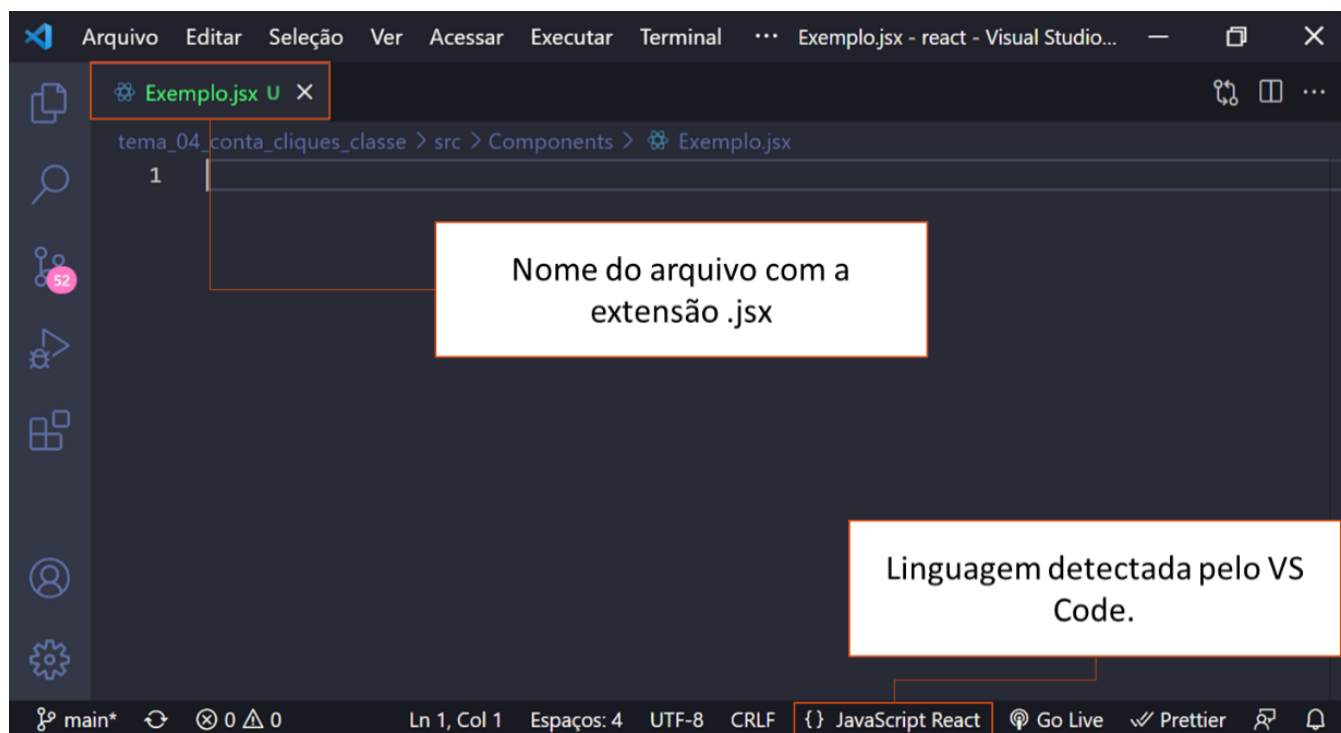
- App.test.js
- logo.svg

Seu diretório, deve ficar como esse:



Antes de continuarmos, precisamos fazer algumas considerações importantes:

- O nome do componente deve **obrigatoriamente** começar com **letra maiúscula e não pode conter espaços em branco** ou **caracteres especiais**.
- A **extensão** do arquivo pode ser **.js** (de arquivo **JavaScript**) ou **.jsx** (de arquivo **JSX**). A vantagem de usar a extensão **.jsx** é que o **VS Code** já interpreta como um arquivo com a **sintaxe JSX**. Ela é usada no **React** e faz as sugestões de **correções para essa linguagem**. Se você criar o arquivo com a extensão **.js**, você terá que **mudar manualmente a linguagem selecionada**.



Extensão do arquivo com a linguagem detectada pelo VS Code.

Agora insira o código abaixo no arquivo **Exemplo.jsx**:

```
1. import { useState } from 'react';
2.
3. const Exemplo = () => {
4.   const [contador, setContador] = useState(0);
5.
6.   return (
7.     <div>
8.       <p>Você clicou {contador} vezes</p>
9.       <button onClick={() => setContador(contador + 1)}>
10.        Clique aqui
11.      </button>
12.    </div>
13.  );
14. };
15.
16. export default Exemplo;
```

Observe que esse componente possui um **estado** chamado **contador**, que é **iniciado** com o valor **zero**, **useState(0)**, e uma função **setContador** usada para atualizar o valor do estado, **setContador(contador + 1)**. Você só pode atualizar o **estado de um componente** com a função gerada pelo **hook useState**, que nesse exemplo é a **setContador**. Para utilizar o **hook useState** é necessário **importá-lo no componente** através da instrução na **linha 1**.

O valor do **estado do componente** é mostrado no **parágrafo** da **linha 8**, note que estamos inserindo uma **expressão JavaScript** em uma **instrução JSX**. Por isso é necessário colocar a **expressão entre chaves**.



Importante! Note, também, que você só pode ter um elemento React pai dentro do método **return**. Desse modo precisamos colocar o elemento **<div>** para envolver os outros dois elementos, **parágrafo** e **botão**. Dessa forma, a aplicação funcionará corretamente.

Isso não é permitido

```
return (
  <p>Você clicou {contador} vezes</p>
  <button onClick={() => setContador(contador + 1)}>
    Clique aqui
  </button>
);
```

Isso está correto

```
return (
  <div>
    <p>Você clicou {contador} vezes</p>
    <button onClick={() => setContador(contador + 1)}>
      Clique aqui
    </button>
  </div>
);
```


Após a inserção do código, **salve o seu arquivo**.



Importante! O VS Code possui uma extensão para agilizar na criação do código de um componente React: **ES7 React/Redux/GraphQL/React-Native snippets**



ES7 React/Redux/GraphQL/React-Native snippets v3.1.1

dsznajder | 3.829.441 | ★★★★★ (40)

Simple extensions for React, Redux and GraphQL in JS/TS with ES7 syntax

[Disable](#) [Uninstall](#) [Refresh](#) [Settings](#)

This extension is enabled globally.

Veja um exemplo ao digitar **rafce** e pressionar a tecla **Enter** ou **tab**.

```

1 rafce
  _rafce
  RTCDTMFToneChangeEvent
  RTCertificate
  ReadableStreamDefaultController

Creates a React Arrow Function Component with ES7 module system (ES7 React/Redux/GraphQL/React-Native snippets)

import React from 'react'

const = () => {
  return (
    <div>

    </div>
  )
}

export default
  
```

Esse **snippet** gerará um **código automático** de um **componente funcional** (utiliza **Arrow function**) e com o **mesmo nome dado ao arquivo do componente**.

```

1 import React from 'react'
2
3 const Exemplo = () => {
4   return (
5     <div>
6
7     </div>
8   )
9 }
10
11 export default Exemplo
  
```

Nas versões atuais da biblioteca React, a instrução **import React from 'react'** não é **obrigatória**.

Inserindo o componente criado no componente raiz da aplicação React

Até agora, nós apenas criamos o arquivo e inserimos o **código do componente**. Ainda precisamos inserir esse componente na **árvore de componentes da aplicação**. Como todo projeto **React**, temos um **componente raiz**, esse nosso componente criado será um **componente filho do componente raiz App**.

Então, no arquivo **App.js**, devemos atualizar o código da seguinte forma:

```
1. import './App.css';
2. import Exemplo from './Components/Exemplo';
3.
4. function App() {
5.   return (
6.     <div className="App">
7.       <Exemplo />
8.     </div>
9.   );
10. }
11.
12. export default App;
```

Na **linha 2** do código mostrado, temos a **instrução para importar** o componente **filho** no componente **pai App** e, desse modo, ele ser **reconhecido como um componente válido**. Caso você esqueça de colocar essa instrução, a aplicação acusará erro na linha 7.

A **linha 7** contém o nosso **componente criado**, indicando o local (ordem) que ele deverá ser renderizado. Note que devemos sempre usar a notação:

Abre o componente com o símbolo de menor que	Nome do componente	Espaço	Fechar o componente com barra e o sinal de maior que
<Exemplo />			

Vão ter casos que iremos colocar propriedade dentro da instrução do componente, mas por enquanto vamos entender de forma simples a instrução para renderização do componente. Salve o **arquivo App.js**

Inicializando a aplicação e visualizando o resultado no navegador web

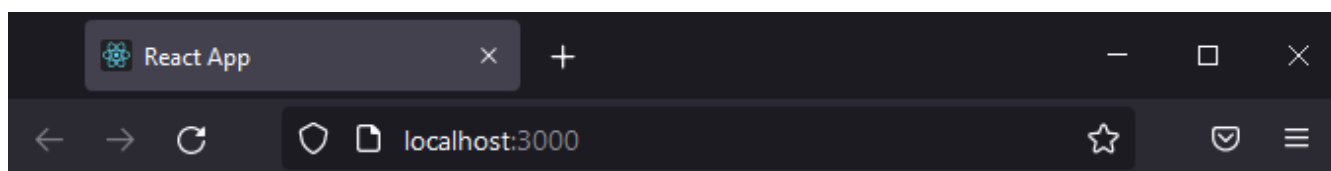
Vamos ver o resultado a nossa aplicação React. Para isso siga os passos. No terminal do VS Code, entre no **diretório da aplicação**, caso ainda não tenha entrada, e execute o comando **npm start**.

```

TERMINAL  PROBLEMAS  SAÍDA  CONSOLE DE DEPURACÃO
Rafael Lopes@NB-IOS-CZB8ZH3 MINGW64 /c/react (main)
$ cd tema_03_conta_cliques/

Rafael Lopes@NB-IOS-CZB8ZH3 MINGW64 /c/react/tema_03_conta_cliques (main)
$ npm start
  
```

Você pode ver o resultado no navegador web.



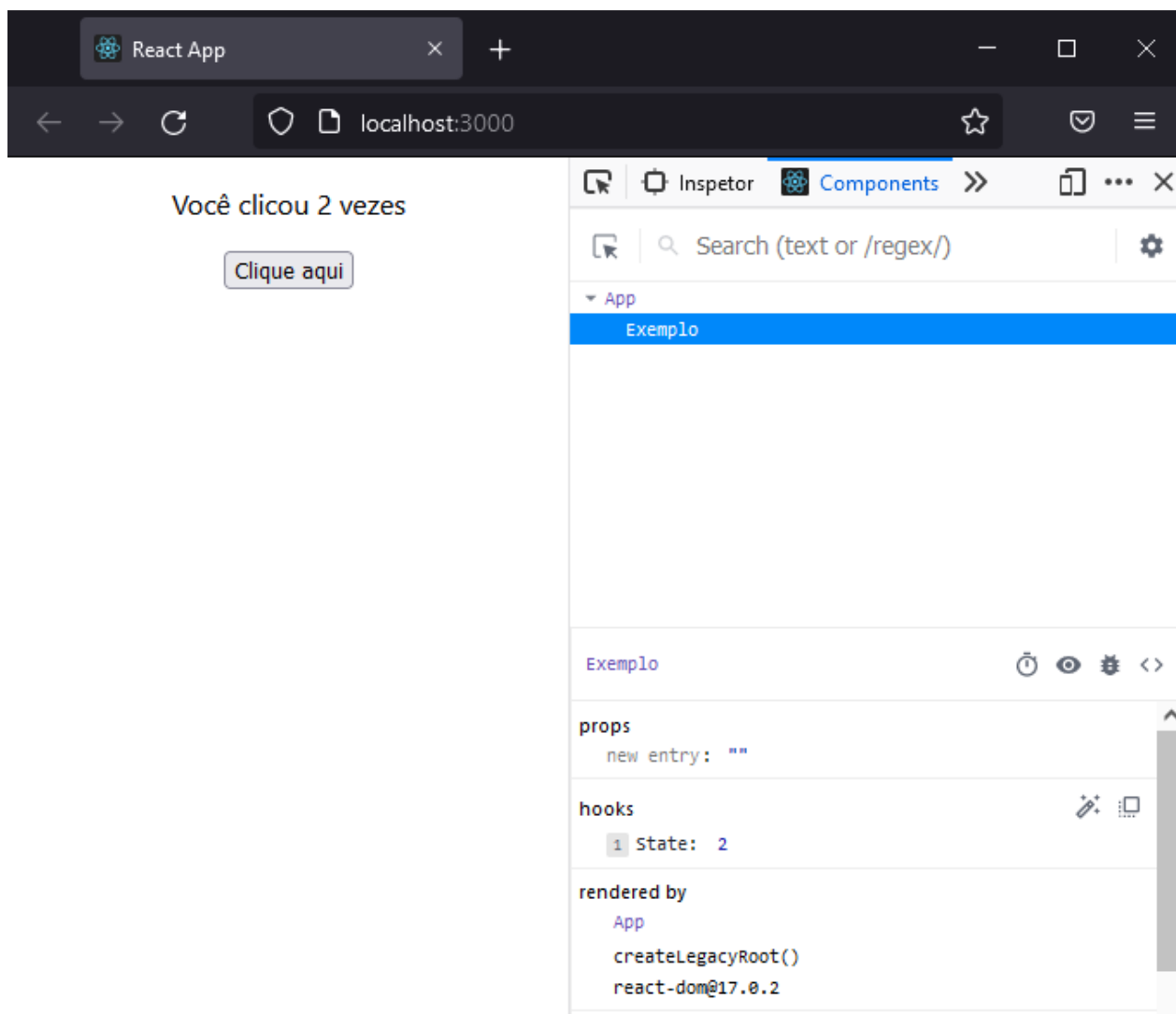
Você pode **clicar algumas vezes no botão** para ver como a **aplicação atualiza o valor mostrado** na tela sem a necessidade de **recarregar toda a página novamente**.



Importante! Todas essas atualizações da interface do usuário (**contagem do número de cliques**) estão sendo realizadas no **Frontend**, portanto se você atualizar a página (**recarregar a página novamente**), ela sempre voltará a exibir os **valores iniciais**. Isso porque, **não existe um Backend** para armazenar os dados de forma **permanente**.

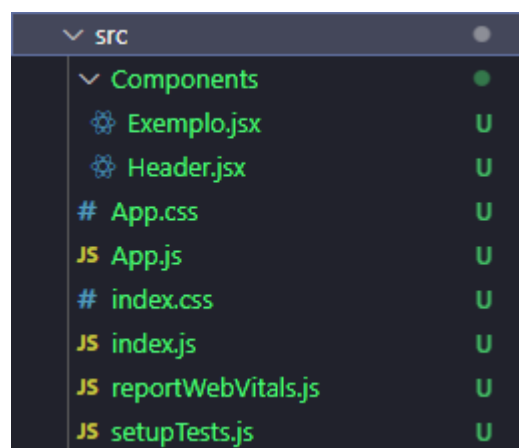
Extensão React Developer Tools

Se abrirmos a opção **Components** da extensão **React Developer Tools**, podemos, agora, dois componentes na árvore de componentes sendo que se clicarmos no componente Exemplo, podemos ver seu **estado** controlado pelo **hook**.



Inserindo um novo componente na hierarquia

Vamos criar um componente chamado **Header** que irá conter o **cabeçalho da nossa aplicação**. Crie o arquivo **Header.jsx** dentro do diretório **Components**.



Insira o seguinte código no arquivo **Header.jsx**.

```

1. const Header = (props) => {
2.   return (
3.     <div>
4.       <h1>Olá, {props.nome}, seja bem-vindo!</h1>
5.     </div>
6.   );
7. };
8.
9. export default Header;

```

Observe que esse componente espera pelo menos **uma propriedade** (linha 1 do código), que será **passada** pelo componente **pai**. Um componente pode receber **uma ou mais propriedades** simplesmente indicando com a instrução **props** entre parênteses.

Indica que o componente pode receber dados através de propriedades

`const Header = (props) => {`

Na **linha 4**, acessamos o conteúdo passado na propriedade chamada **nome**. Com o **componente criado**, devemos atualizar o componente **raiz App.js** com o seguinte código:

```

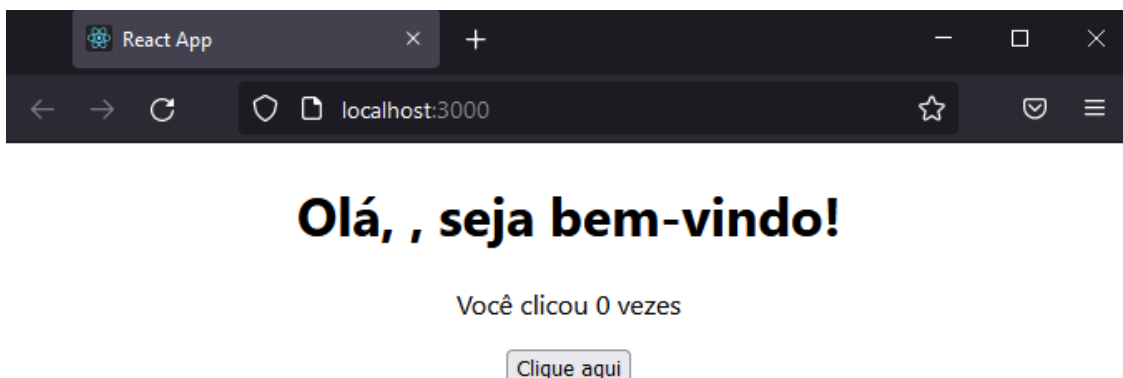
1. import './App.css';
2. import Exemplo from './Components/Exemplo';
3. import Header from './Components/Header';
4.
5. function App() {
6.   return (
7.     <div className="App">
8.       <Header />
9.       <Exemplo />
10.    </div>
11.  );
12. }
13.
14. export default App;

```



Importante! Lembre-se que você só pode ter **um componente pai**.

Observe que se **não passarmos nenhum dado** na instrução do componente na **linha 8**, a aplicação não mostra **nenhuma informação na página web**.



Se sua aplicação não estiver inicializada, execute o comando **npm start**. Agora, se atualizarmos o **código** do **componente App.js** na **linha 8** passando o valor da propriedade **nome**:

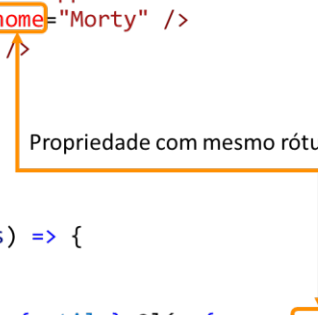
```
1. import './App.css';
2. import Exemplo from './Components/Exemplo';
3. import Header from './Components/Header';
4.
5. function App() {
6.   return (
7.     <div className="App">
8.       <Header nome="Morty" />
9.       <Exemplo />
10.    </div>
11.  );
12. }
13.
14. export default App;
```

Note que o nome da propriedade deve ser o mesmo que o esperado/acessado pelo componente:

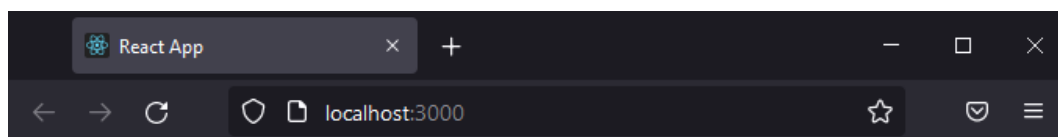
```
function App() {
  return (
    <div className="App">
      <Header nome="Morty" />
      <Exemplo />
    </div>
  );
}

const Header = (props) => {
  return (
    <div>
      <h1 style={estilo}>Olá, {props.nome}, seja bem-vindo!</h1>
    </div>
  );
};
```

Propriedade com mesmo rótulo/nome



O navegador mostra a informação passada de um componente pai para um componente filho.

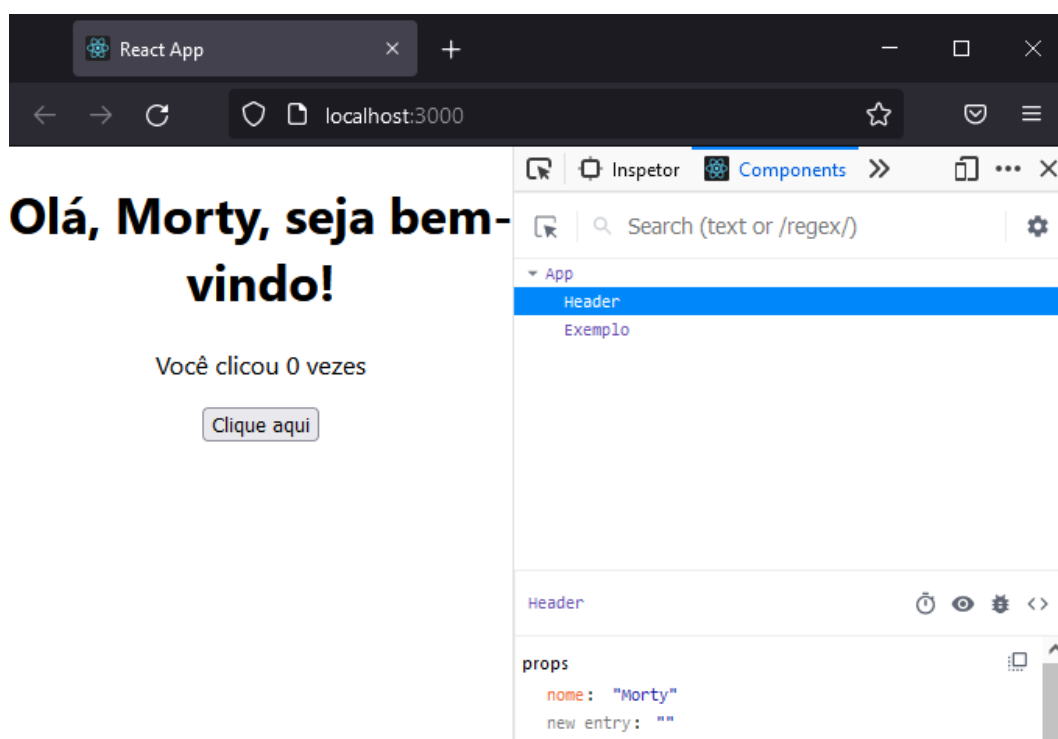


Olá, Morty, seja bem-vindo!

Você clicou 0 vezes

Clique aqui

Você pode visualizar na extensão **React Developer Tools** a nova árvore de componente e se você clicar no componente **Header** poderá ver a propriedade **nome** do **componente**.



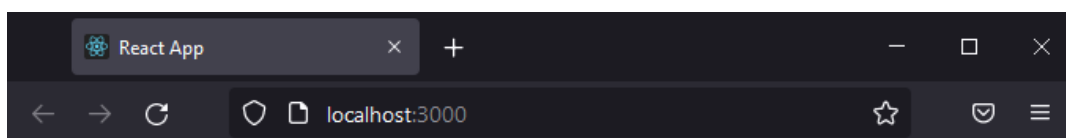
Valor padrão para uma propriedade

Para evitar esse problema de **não passar o valor desejado** e não **aparecer nenhuma informação**, você pode configurar um valor padrão para ser exibido no **Header.jsx**

```
1. const Header = (props) => {
2.   return (
3.     <div>
4.       <h1>Olá, {props.nome}, seja bem-vindo!</h1>
5.     </div>
6.   );
7. };
8.
9. Header.defaultProps = {
10.   nome: 'Nome padrão',
11. };
12.
13. export default Header;
```

A instrução **Header.defaultProps** (linhas 9 a 11) configura valores padrão para as propriedades esperadas pelo **componente**. Desse modo, se o valor da propriedade **não for passado** pelo componente **pai**, teremos a **exibição de valor padrão**. Atualize a função **App()** arquivo **App.js**

```
1. import './App.css';
2. import Exemplo from './Components/Exemplo';
3. import Header from './Components/Header';
4.
5. function App() {
6.   return (
7.     <div className="App">
8.       <Header />
9.       <Exemplo />
10.    </div>
11.  );
12. }
13.
14. export default App;
```



Olá, Nome padrão, seja bem-vindo!

Você clicou 0 vezes

[Clique aqui](#)

Checagem de tipos de uma propriedade

Outra informação importante ao trabalhar com **propriedades** é que você pode definir o **tipo de dado** que deverá ser **passado na propriedade**. Isso é feito pelo recurso **PropTypes** do React. Você pode utilizar os seguintes validadores de tipo de dados:

Validador	Descrição
PropTypes.array,	Verifica se é um array
PropTypes.bool,	Verifica se é um booleano
PropTypes.func,	Verifica se é uma função
PropTypes.number,	Verifica se é um número
PropTypes.object,	Verifica se é um objeto
PropTypes.string,	Verifica se é uma string
PropTypes.symbol,	Verifica se é um símbolo

Por exemplo, na propriedade do **componente Header** estamos **esperando uma string**. Então podemos **atualizar o código do** componente **Header.jsx** com a seguinte instrução:


```

1. import PropTypes from 'prop-types';
2.
3. const Header = (props) => {
4.   return (
5.     <div>
6.       <h1>Olá, {props.nome}, seja bem-vindo!</h1>
7.     </div>
8.   );
9. };
10.
11. Header.defaultProps = {
12.   nome: 'Nome padrão',
13. };
14.
15. Header.propTypes = {
16.   nome: PropTypes.string,
17. }
18.
19. export default Header;

```

Observe na **linha 1** a necessidade de **importar o recurso no componente** e nas **linhas 15 a 17** é onde **definimos que o valor passado pela propriedade nome deve ser uma string**. Se na renderização do componente Header dentro do componente App passarmos um valor que não seja string, por exemplo: `<Header nome=1 />`

O terminal mostrará erro de compilação:



```

Failed to compile.

./src/App.js
SyntaxError: C:\react\tema_03_conta_clicques\src\App.js: JSX value should be either an expression or a quoted JSX text. (8:25)

   6 |     return (
   7 |       <div className="App">
>  8 |         <Header nome=1 />
      |                             ^
   9 |       <Exemplo />
  10 |     </div>
  11 |   );

```

E o navegador também mostrará o mesmo erro:



```

Failed to compile

./src/App.js
SyntaxError: C:\react\tema_03_conta_clicques\src\App.js: JSX value should be either an expression or a quoted JSX text. (8:25)

   6 |     return (
   7 |       <div className="App">
>  8 |         <Header nome=1 />
      |                             ^
   9 |       <Exemplo />
  10 |     </div>
  11 |   );

```

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

Vamos desfazer a alteração `<Header nome=1 />` para `<Header />`

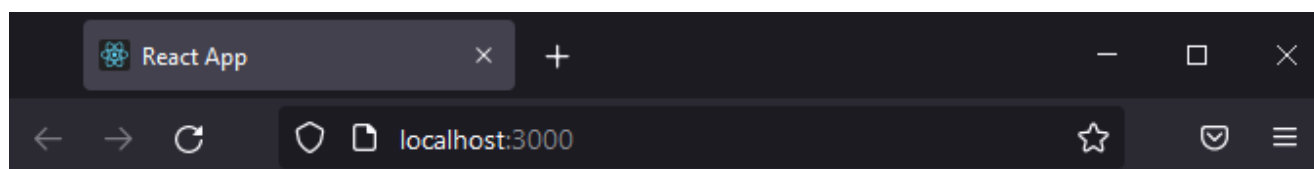
Adicionando um estilo a um componente

Existem várias formas de **configurar estilos** em um componente **React**: **inline**, pelo arquivo **App.css** ou **index.css**, através de **Bootstrap**, etc. Vamos ver um primeiro exemplo de aplicar estilos **CSS inline no componente**.

No componente Header podemos utilizar o atributo **style** e definir a cor da fonte do texto que aparecer no heading `<h1>`.

```
1. import PropTypes from 'prop-types';
2.
3. const Header = (props) => {
4.   return (
5.     <div>
6.       <h1 style={{ color: 'red' }}>Olá, {props.nome}, seja bem-vindo!</h1>
7.     </div>
8.   );
9. };
10.
11. Header.defaultProps = {
12.   nome: 'Nome padrão',
13. };
14.
15. Header.propTypes = {
16.   nome: PropTypes.string,
17. };
18.
19. export default Header;
```

O resultado pode ser visualizado no navegador web.



Olá, Nome padrão, seja bem-vindo!

Você clicou 0 vezes

Clique aqui

Observe que, o atributo **style={{ color: 'red' }}** na **linha 6** configura a **cor da fonte como vermelha**. Note que, quando utilizamos a propriedade **CSS** devemos colocá-la entre **duas aberturas e fechamento de chaves**.



Importante! Note que o **valor da propriedade** deve ser colocado **entre aspas**, diferentemente de como fazemos no arquivo CSS.

Para ficar mais organizado, podemos **criar uma variável** com os **estilos que queremos aplicar** e colocar **somente a variável dentro de uma abertura e fechamento de chaves do atributo style**.

```

1. import PropTypes from 'prop-types';
2.
3. const estilo = { color: 'red' };
4.
5. const Header = (props) => {
6.   return (
7.     <div>
8.       <h1 style={estilo}>Olá, {props.nome}, seja bem-vindo!</h1>
9.     </div>
10.   );
11. };
12.
13. Header.defaultProps = {
14.   nome: 'Nome padrão',
15. };
16.
17. Header.propTypes = {
18.   nome: PropTypes.string,
19. };
20.
21. export default Header;

```

O resultado é o mesmo, porém o **código fica mais organizado**. Veremos outras formas de configurar estilos CSS na aplicação React.



Componentes com Classe

Os objetivos desta aula são:

- Apresentar o conceito de State
- Criar componentes utilizando classe
- Inserir e adicionar estilos aos componentes utilizando React

Bons estudos!

State

Em componentes implementados com classe, utilizamos o conceito de **state**, ao invés de **hooks** como em **componentes funcionais**. Desse modo, o componente criado com **classe** terá o seu **state**, que poderá possuir **um** ou **mais campos**.

```
class NomeComponente extends Component {
  state = {
    campo01: valor01;
    campo02: valor02;
    ...
    campoN: valorN;
  }
  render() {
    return (
      ...
    );
  }
}
```

E para acessar esse estado devemos utilizar **this**:

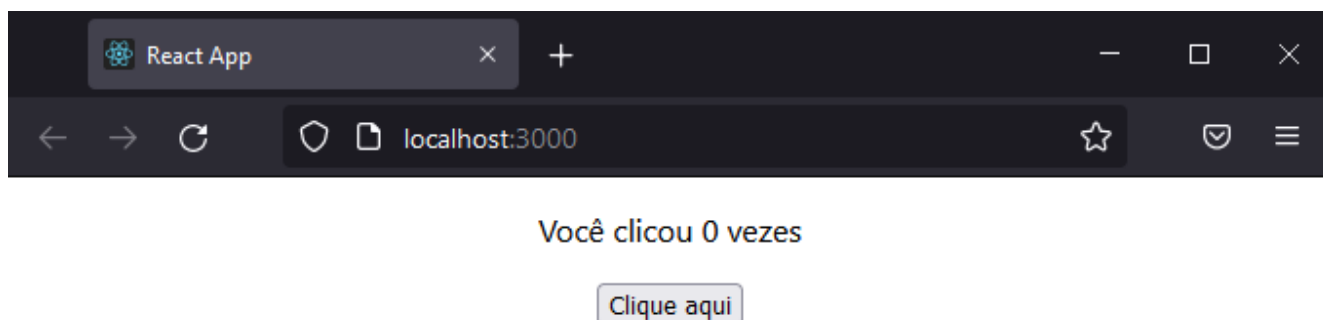
```
this.state.campoX
```

E para atualizar o estado utilizamos **this.setState()**:

```
this.setState(campoX: novo_valor)
```

Criando componentes utilizando classe

Vamos fazer o exemplo do **tema 01** com o componente utilizando **classe** que possui um **parágrafo** e um **botão**, onde o parágrafo mostra **quantas você clicou no botão da interface**.



Instalando a aplicação React

Escolha um diretório de trabalho, acesse esse diretório pelo terminal do **VS Code** e execute o seguinte comando.

```
npx create-react-app tema_04_conta_cliques_classe
```

Criando o componente Exemplo

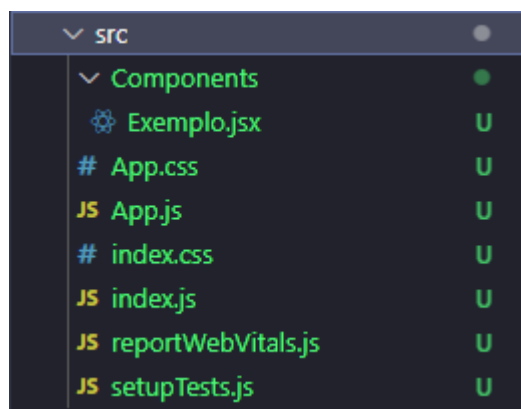
Com a aplicação instalada, vamos criar um diretório chamado **Components**, que deve estar dentro da pasta **src**. Dentro desse novo diretório **Components**, vamos criar o arquivo do nosso componente com o nome **Exemplo.jsx**.

Primeiramente, apagar alguns arquivos do diretório **src**, que **não usaremos nessa aplicação**.

São os arquivos:

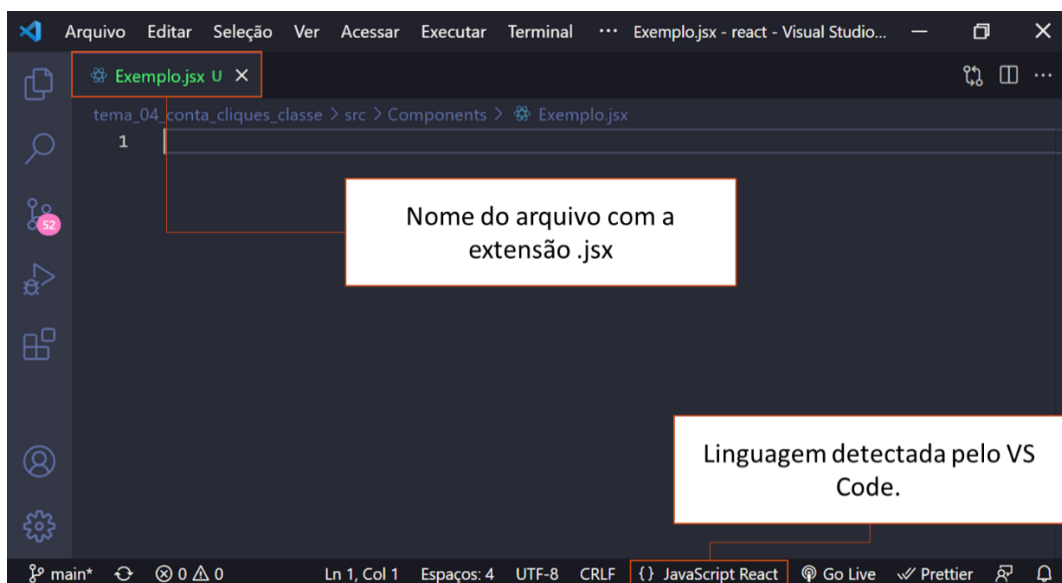
- App.test.js
- logo.svg

A organização do seu diretório deve ficar como mostrado abaixo:



Antes de continuarmos, precisamos fazer algumas considerações importantes:

- O nome do componente deve **obrigatoriamente** começar com **letra maiúscula e não pode conter espaços em branco** ou **caracteres especiais**.
- A **extensão** do arquivo pode ser **.js** (de arquivo **JavaScript**) ou **.jsx** (de arquivo **JSX**). A vantagem de usar a extensão **.jsx** é que o **VS Code** já interpreta como um arquivo com a **sintaxe JSX**, que é a usada no **React**, e faz as sugestões de **correções para essa linguagem**. Se você criar o arquivo com a extensão **.js**, você terá que **mudar manualmente a linguagem selecionada**.



Extensão do arquivo com a linguagem detectada pelo VS Code.

Agora insira o código abaixo no arquivo **Exemplo.jsx**:

```
1. import { Component } from 'react';
2.
3. class Exemplo extends Component {
4.   // Declare uma nova variável de estado, a qual chamaremos de "contador"
5.   // e é um campo do objeto state
6.   constructor(props) {
7.     super(props);
8.     this.state = {
9.       contador: 0,
10.    };
11.  }
12.
13.  render() {
14.    return (
15.      <div>
16.        <p>Você clicou {this.state.contador} vezes</p>
17.        <button
18.          onClick={() =>
19.            this.setState({ contador: this.state.contador + 1 })
20.          }
21.        >
22.          Clique aqui
23.        </button>
24.      </div>
25.    );
26.  }
27. }
28.
29. export default Exemplo;
```

Observe que esse **componente** possui um **estado** chamado **state** com um **campo contador**, que é **iniciado** com o **valor zero** (linhas 8 a 10). E como padrão de classes o método **setState** deve ser usado para **atualizar** o valor do campo (linha 19).

O valor do **estado do componente** é mostrado no **parágrafo** da **linha 16**. Note que estamos **inserindo uma expressão JavaScript em uma instrução JSX**. Por isso é necessário colocar a expressão **entre chaves**.

Note a necessidade do **this** para acessar o campo do **estado** (**linha 16**) ou para **atualizá-lo** (**linha 19**). Isso é uma **necessidade** quando estamos trabalhando com **classe**. Componentes criados **utilizando classe** devem “**herdar**” funcionalidades do **objeto Component** do react. Por isso, é necessário **importar** esse recurso **na linha 1** do código.



Importante! Você **só pode ter um elemento React pai** dentro do método **return**. Desse modo precisamos colocar o elemento **<div>** para **envolver** os outros dois elementos, **parágrafo** e **botão**. Dessa forma, a aplicação funcionará corretamente.

Isso não é permitido

```
return (
  <p>Você clicou {this.state.contador} vezes</p>
  <button
    onClick={() =>
      this.setState({ contador: this.state.contador + 1 })
    }
  >
    Clique aqui
  </button>
);
```

Isso está correto

```
return (
  <div>
    <p>Você clicou {this.state.contador} vezes</p>
    <button
      onClick={() =>
        this.setState({ contador: this.state.contador + 1 })
      }
    >
      Clique aqui
    </button>
  </div>
);
```

Após a inserção do código, salve o seu arquivo.

Inserindo o componente criado no componente raiz da aplicação React

Nós apenas **criamos o arquivo** e inserimos o **código do componente**. Ainda precisamos **inserir** esse **componente** na **árvore de componentes da aplicação**. Como todo projeto React, temos um **componente raiz**. Esse nosso componente criado será um **componente filho** do componente **raiz App**.

Então, no arquivo **App.js**, devemos atualizar o código da seguinte forma:

```
1. import './App.css';
2. import { Component } from 'react';
3. import Exemplo from './Components/Exemplo';
4.
5. export class App extends Component {
6.   render() {
7.     return (
8.       <div className="App">
9.         <Exemplo />
10.      </div>
11.    );
12.  }
13. }
14.
15. export default App;
```

Na **linha 3** do código mostrado, temos a instrução para **importar** o componente **filho** no componente **pai App** e, desse modo, ele ser reconhecido como um **componente válido**. Caso você esqueça de colocar essa instrução, a aplicação acusará erro na linha 9.

A linha 9 contém o nosso **componente criado**, indicando o **local (ordem)** que ele deverá ser **renderizado**. Note que devemos sempre usar a notação:

Abre o componente com o símbolo de menor que	Nome do componente	Espaço	Fechar o componente com barra e o sinal de maior que
<Exemplo />			

Note que alteramos o componente **App** e o **implementamos** utilizando o **conceito de classe**. Isso foi feito só para mantermos a **consistência** na implementação do projeto. Ou seja, para utilizarmos a mesma abordagem em todo o projeto.



Importante! Na vida real, você deverá aprender a implementar e dar manutenção em sistemas híbridos, não faça alterações de abordagem se não forem necessárias e/ou solicitadas. Deixe todas as abordagens conviverem em paz no sistema já em funcionamento.

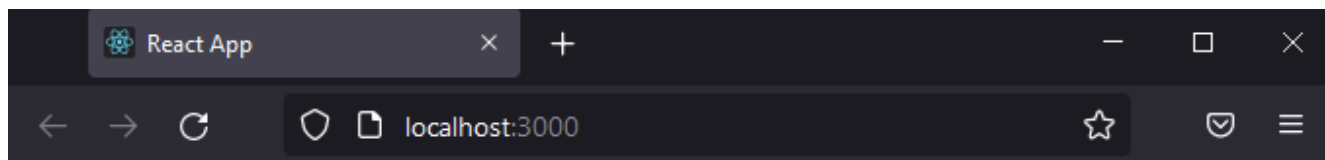
Salve o arquivo **App.js**

Inicializando a aplicação e visualizando o resultado no navegador web

Vamos ver o resultado a nossa **aplicação React**. Para isso siga os passos:

No terminal do **VS Code**, entre no diretório da aplicação, caso ainda não tenha entrado, e execute o comando **npm start**.

Você pode ver o resultado no navegador web.



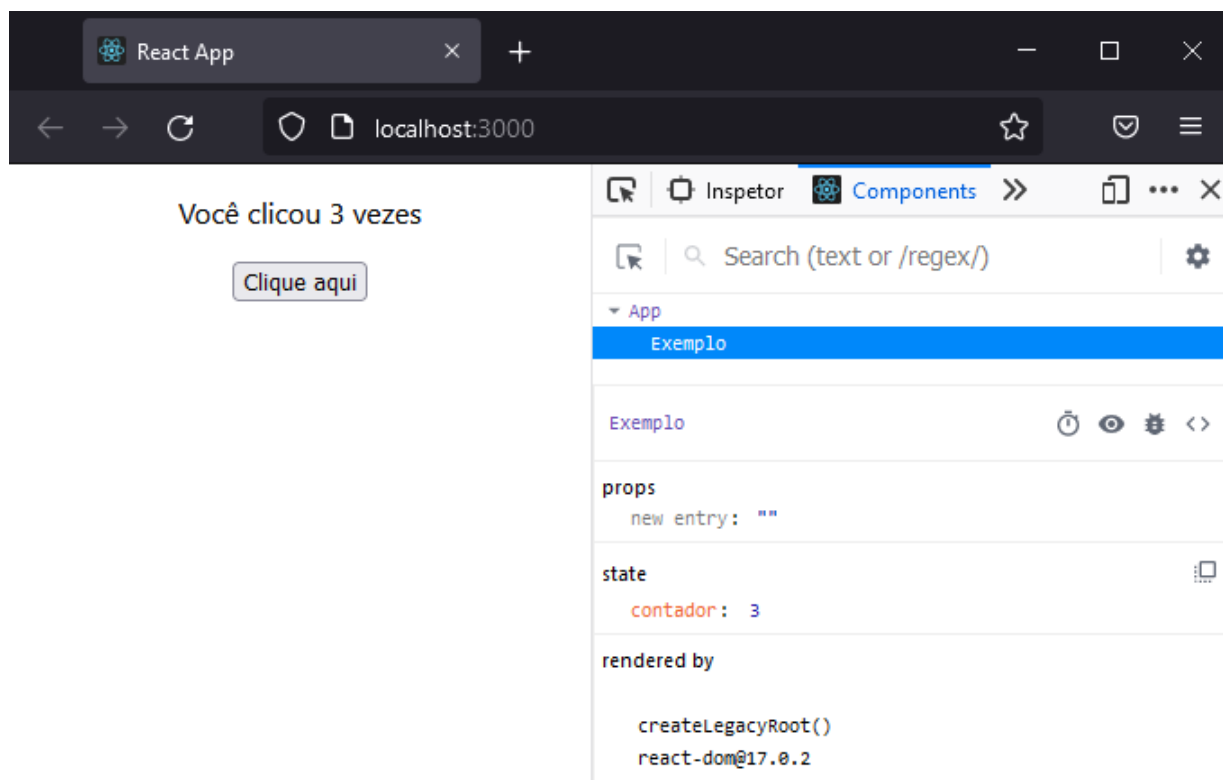
Você clicou 0 vezes

Clique aqui

Você pode clicar algumas vezes no botão para ver como a aplicação atualiza o valor mostrado na tela sem a necessidade de recarregar toda a página novamente.

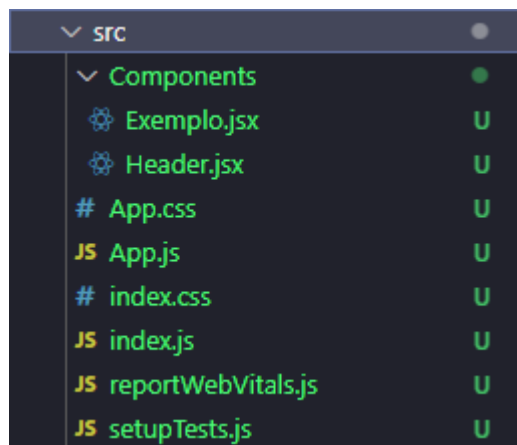
Extensão React Developer Tools

Se abrirmos a opção **Components** da extensão **React Developer Tools**, podemos, agora, **dois componentes** na árvore de componentes sendo que se clicarmos no componente Exemplo, podemos ver o campo **contador** do **state** com o valor do **estado do componente**.



Inserindo um novo componente na hierarquia

Vamos criar um componente chamado **Header**, que irá conter o cabeçalho da nossa aplicação. Crie o arquivo **Header.jsx** dentro do diretório **Components**.



Insira o seguinte código no arquivo **Header.jsx**.

```
1. import { Component } from 'react';
2. import PropTypes from 'prop-types';
3.
4. export class Header extends Component {
5.   render() {
6.     return (
7.       <div>
8.         <h1>Olá, {this.props.nome}, seja bem-vindo!</h1>
9.       </div>
10.    );
11.  }
12. }
13.
14. Header.defaultProps = {
15.   nome: 'Nome padrão',
16. };
17.
18. Header.propTypes = {
19.   nome: PropTypes.string,
20. };
21.
22. export default Header;
```

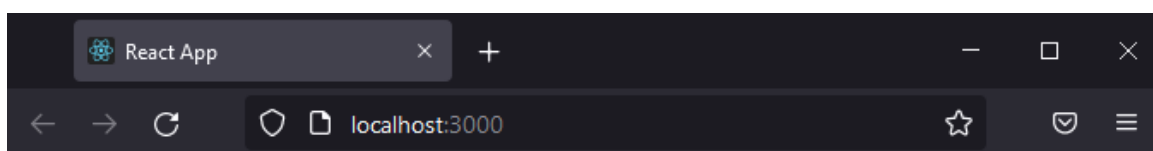
Observe que **componente implementados** com **classe** não precisamos fazer como **componentes funcionais** e indicar que estamos **esperamos uma propriedade** basta colocar o **acesso** como mostrado na **linha 7**.

Note que já colocamos o **valor padrão da propriedade** e o **validador** para verificar se o conteúdo passado **é uma string**. Note também que para acessar a propriedade devemos utilizar o **this**.

Com o componente criado devemos **atualizar o componente** raiz **App.js** com o seguinte código:

```
1. import './App.css';
2. import { Component } from 'react';
3. import Exemplo from './Components/Exemplo';
4. import Header from './Components/Header';
5.
6. export class App extends Component {
7.   render() {
8.     return (
9.       <div className="App">
10.        <Header nome="Morty" />
11.        <Exemplo />
12.      </div>
13.    );
14.  }
15. }
16.
17. export default App;
```

O resultado pode ser visto no navegador web.

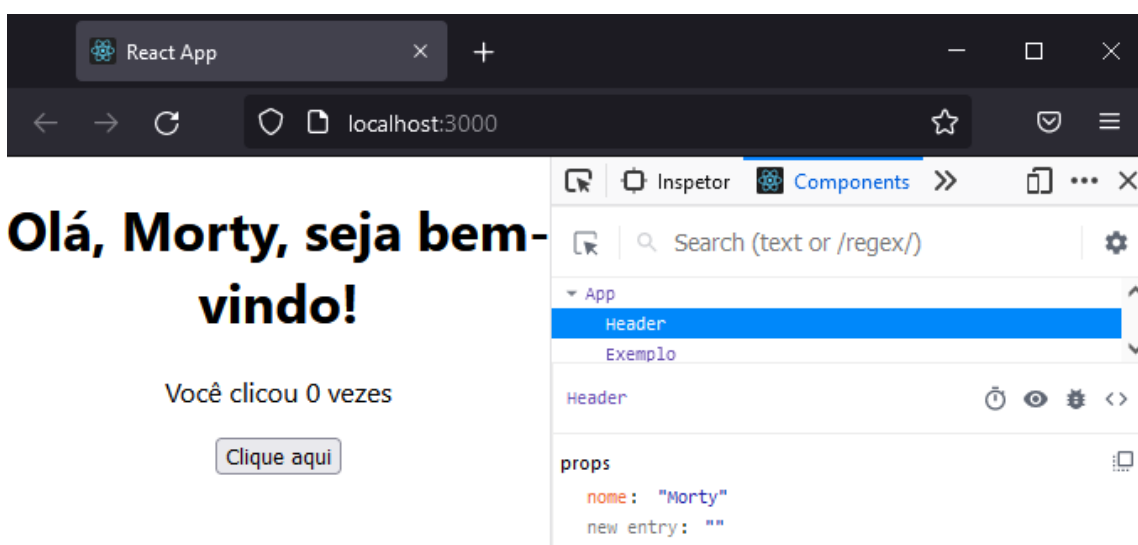


Olá, Morty, seja bem-vindo!

Você clicou 0 vezes

Clique aqui

Se sua aplicação não estiver inicializada, execute o comando **npm start**. Você pode visualizar na extensão **React Developer Tools** a **nova árvore de componente** e se você clicar no componente **Header** poderá ver a propriedade **nome** do componente.



Adicionando um estilo a um componente

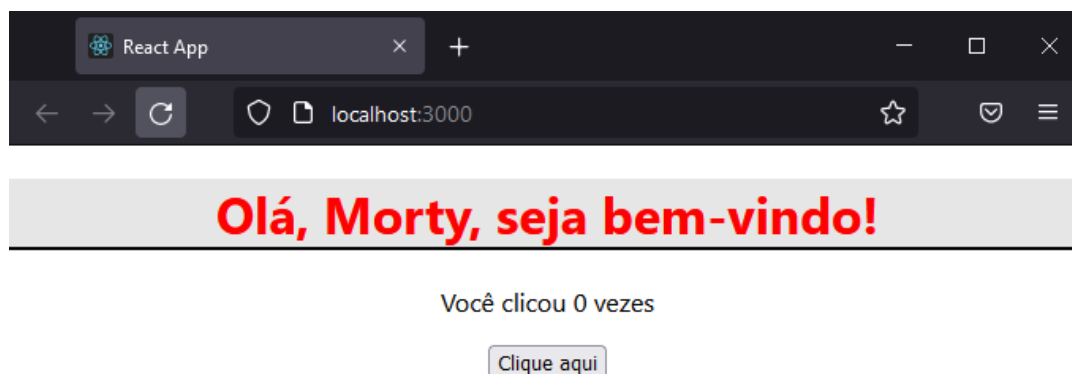
Existem várias formas de configurar **estilos** em um componente React: **inline**, pelo arquivo **App.css** ou **index.css**, através de **Bootstrap**, etc. Vamos ver um primeiro exemplo de aplicar estilos **CSS inline** no componente.

No componente **Header.jsx** podemos utilizar o atributo **style** e definir alguns estilos para o cabeçalho.

```

1. import { Component } from 'react';
2. import PropTypes from 'prop-types';
3.
4. const estilo = {
5.   color: 'red',
6.   borderBottom: 'black solid 2px',
7.   backgroundColor: '#E6E6E6',
8. };
9.
10. export class Header extends Component {
11.   render() {
12.     return (
13.       <div>
14.         <h1 style={estilo}>Olá, {this.props.nome}, seja bem-vindo!</h1>
15.       </div>
16.     );
17.   }
18. }
19.
20. Header.defaultProps = {
21.   nome: 'Nome padrão',
22. };
23.
24. Header.propTypes = {
25.   nome: PropTypes.string,
26. };
27.
28. export default Header;
  
```

O resultado pode ser visualizado no navegador web.



A variável possui **três propriedade de estilos** que devem ser **separadas** por **vírgula** e propriedades como **border-bottom** no **CSS** são usadas com **camelCase** **borderBottom** no **JSX**, como também fizemos com **background-color** do **CSS** agora é **backgroundColor** no **JSX**.



Projeto Agenda de Compromissos - Parte 01

Os objetivos desta aula são:

- Exercitar a utilização do React por meio da criação de uma Agenda
- Discutir sobre os elementos e componentes necessários para o projeto

Bons estudos!

Criando nosso projeto Inicial

Escolha um **diretório de trabalho**, acesse esse diretório pelo **terminal do VS Code** e execute o seguinte comando:

```
npx create-react-app tema_05_agenda
```

Ao final da instalação, você deve acessar o diretório criado pelo terminal com o seguinte comando:

```
cd tema_05_agenda
```

Vamos abrir o arquivo **index.html** no diretório **public** e alterar o **título** da nossa página:

```
<title>Agenda de compromissos</title>
```

Primeiramente, apagar alguns arquivos do diretório **src**, que **não usaremos nessa aplicação**.

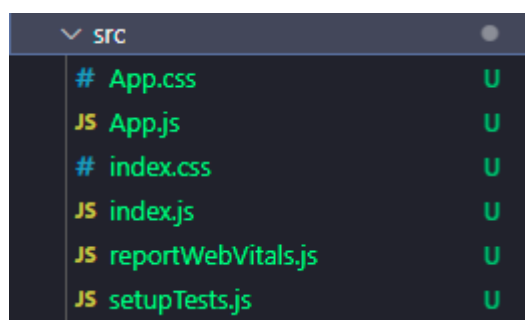
São os arquivos:

- App.test.js
- logo.svg



Importante! Esses arquivos não serão utilizados nesse projeto, mas podem ser importantes em outra implementação.

A organização do seu diretório deve ficar como mostrado abaixo:



Para configurar o **estilo CSS** da aplicação, vamos utilizar o arquivo **index.css**. Vamos **limpar** e **atualizar** o arquivo **App.js** no diretório **src**.

```

1. function App() {
2.   return (
3.     <div className="container">
4.       <h1>Olá Pessoal!</h1>
5.       <p>Vamos criar uma agenda de compromissos.</p>
6.     </div>
7.   );
8. }
9.
10. export default App;

```



Importante! Não se esqueça que o React só permite ter **um elemento React pai** dentro do **método return**, por isso precisamos inserir o **cabeçalho <h1>** e o **parágrafo <p>** dentro do elemento **<div>**.

Isso não é permitido

```

return (
  <h1>Olá, Pessoal!</h1>
  <p>Agenda de compromissos.</p>
);

```

```

return (
  <div className="container">
    <h1>Olá, Pessoal!</h1>
  </div>
  <p>Agenda de compromissos.</p>
);

```

Isso está correto

```

return (
  <div className="container">
    <h1>Olá, Pessoal!</h1>
    <p>Agenda de compromissos.</p>
  </div>
);

```

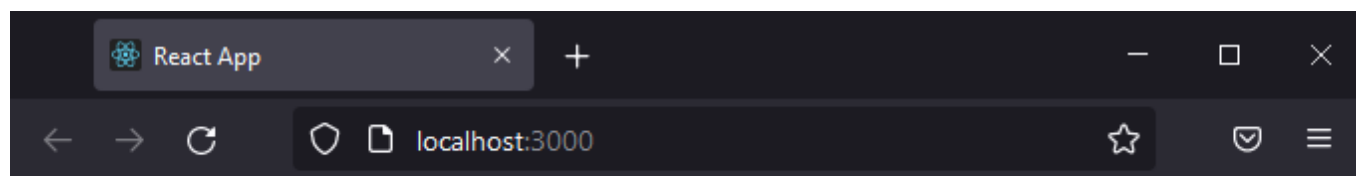
```

return (
  <>
    <h1>Olá, Pessoal!</h1>
    <p>Agenda de compromissos.</p>
  </>
);

```


No terminal do **VS Code**, entre no diretório da aplicação, caso ainda não tenha entrado, e execute o comando **npm start**.

No navegador web, podemos ver a barra de **título** da página com o valor que nós editamos e um cabeçalho “**Olá, Pessoal!**” e um **parágrafo**.

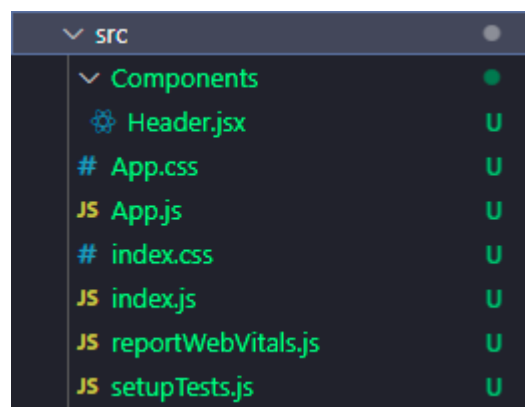


Olá Pessoal!

Vamos criar uma agenda de compromissos.

Criando o componente Header

Vamos criar um **diretório** chamado **Components**, que deve estar dentro da pasta **src**. Dentro desse novo diretório **Components**, vamos **criar o arquivo** do nosso componente com o nome **Header.jsx**. A organização do seu diretório deve ficar como mostrado na figura abaixo:



Agora insira o código abaixo no arquivo **Header.jsx**:

```
1. const Header = () => {
2.   return (
3.     <header>
4.       <h1>Agenda de compromissos</h1>
5.     </header>
6.   );
7. };
8.
9. export default Header;
```

Então, queremos agora **renderizar esse componente** na página web. No arquivo **App.js**, devemos atualizar o código da seguinte forma:

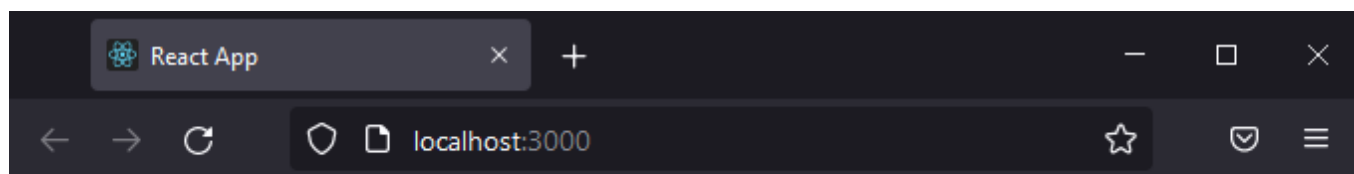
```
1. import Header from './Components/Header';
2.
3. function App() {
4.   return (
5.     <div className="container">
6.       <Header />
7.     </div>
8.   );
9. }
10.
11. export default App;
```

Na **linha 1** do código mostrado, temos a instrução para **importar** o componente **filho** no componente **pai App** e, desse modo, ele ser reconhecido como um **componente válido**. Caso você esqueça de colocar essa instrução, a aplicação acusará erro na **linha 6**.

A **linha 6** contém o nosso componente criado, indicando o **local** (ordem) que ele deverá ser **renderizado**. Note que devemos sempre usar a notação:

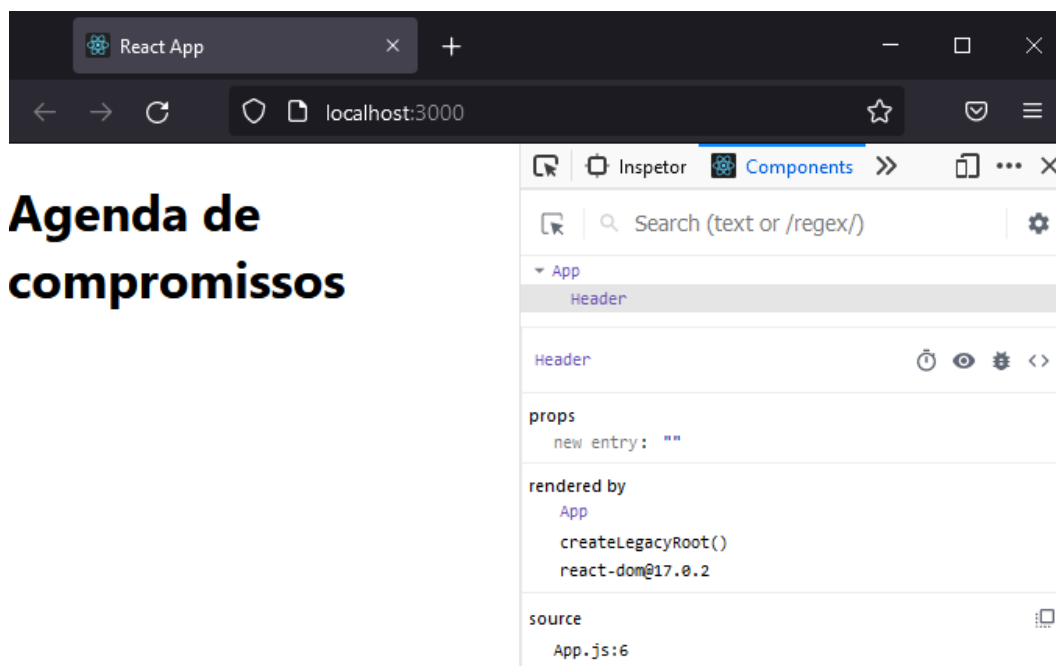
Abre o componente com o símbolo de menor que	Nome do componente	Espaço	Fechar o componente com barra e o sinal de maior que
<Exemplo />			

Salve o arquivo **App.js** e você pode ver o resultado no navegador web.



Agenda de compromissos

Se abrirmos a opção **Components** da extensão **React Developer Tools**, podemos, agora, dois componentes na árvore de componentes.



Utilizando propriedades em um componente

Vamos passar uma informação através de uma propriedade do componente pai App para o componente filho Header. Siga os passos para atualizar o projeto.

No arquivo **App.js** vamos **atualizar o código** inserindo na **linha 6** a propriedade **title**. Atualize o código como mostrado abaixo.

```
1. import Header from './Components/Header';
2.
3. function App() {
4.   return (
5.     <div className="container">
6.       <Header title="tarefas" />
7.     </div>
8.   );
9. }
10.
11. export default App;
```

No componente **Header**, podemos **atualizar o código** para receber a **propriedade** passada pelo componente pai **App**.

```
1. const Header = ({ title }) => {
2.   return (
3.     <header>
4.       <h1>Agenda de {title}</h1>
5.     </header>
6.   );
7. };
8.
9. export default Header;
```

Observe, na **linha 1**, que nesse exemplo o componente **recebe a propriedade de uma forma diferente**, aqui estamos utilizando **atribuição via desestruturação**, que você já viu como

funciona no JavaScript. Dessa forma, não precisamos usar a palavra **props** e acessar o **conteúdo da propriedade** através de **props.nome_propriedade**. Nós simplesmente precisamos usar **chaves** colocar o **nome da propriedade que querem receber**. Se houver mais de uma propriedade, devemos colocá-las **separadas por vírgula dentro das chaves**. Compare o código utilizando props e utilizando atribuição vai desestruturação:

Utilizando props

```
const Header = (props) => {
  return (
    <header>
      <h1>Agenda de {props.title}</h1>
    </header>
  );
};

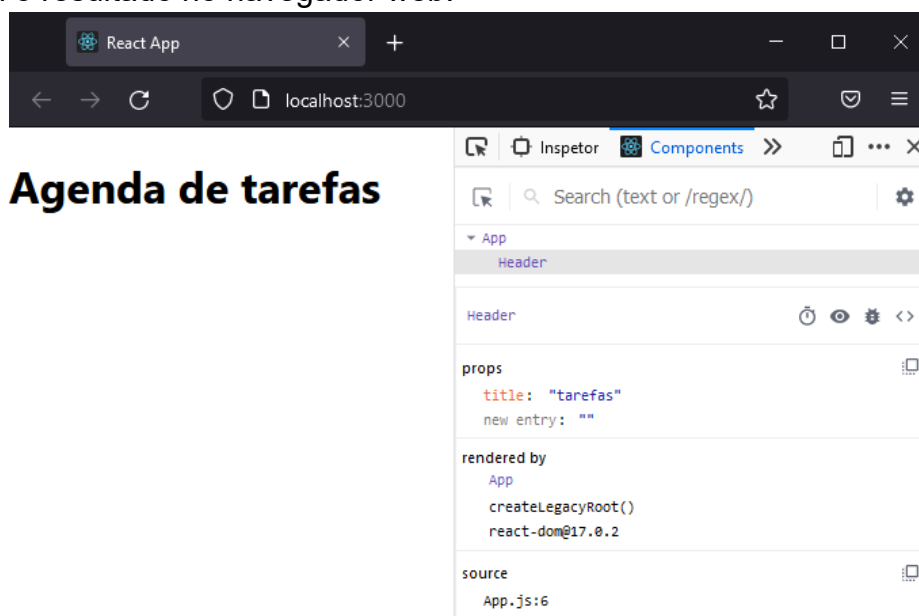
export default Header;
```

Atribuição vai desestruturação

```
const Header = ({ title }) => {
  return (
    <header>
      <h1>Agenda de {title}</h1>
    </header>
  );
};

export default Header;
```

Utilizando a atribuição via desestruturação o código fica mais **simples** e **intuitivo**. Salve os arquivos e veja o resultado no navegador web.



defaultProps e PropTypes

Para ficar completinho o nosso componente, vamos configurar o valor **padrão da propriedade**, caso ela **não seja passada** pelo componente **pai** e também inserir o **validador** de tipo da **propriedade**. Atualize o código do arquivo **Header.jsx** com o seguinte código.

```
1. import PropTypes from 'prop-types';
2.
3. const Header = ({ title }) => {
4.   return (
5.     <header>
6.       <h1>Agenda de {title}</h1>
7.     </header>
8.   );
9. };
10.
11. Header.defaultProps = {
12.   title: 'compromissos',
13. };
14.
15. Header.propTypes = {
16.   title: PropTypes.string.isRequired,
17. };
18.
19. export default Header;
```

Configurando o estilo CSS da aplicação

Vamos **atualizar** o arquivo **index.css** para inserir o **estilo CSS** de toda a nossa aplicação. Iremos configurar o estilo todo de uma vez. Portanto, se você tiver alguma dúvida sobre alguma classe utilizado no projeto, você deve abrir esse arquivo e consultar para ver o que ela está definindo com estilo do elemento.

Abra o arquivo **index.css**, apague qualquer código que tenho dentro dele e insira o seguinte código.

```
1. @import url('https://fonts.googleapis.com/css2?family=Poppins:wght@300;400&display=swap');
2.
3. * {
4.   box-sizing: border-box;
5.   margin: 0;
6.   padding: 0;
7. }
8.
9. body {
10.   font-family: 'Poppins', sans-serif;
11. }
12.
13. .container {
14.   max-width: 500px;
15.   margin: 30px auto;
16.   overflow: auto;
17.   min-height: 300px;
18.   border: 1px solid steelblue;
19.   padding: 30px;
20.   border-radius: 5px;
21. }
22.
```

```

23. .header {
24.     display: flex;
25.     justify-content: space-between;
26.     align-items: center;
27.     margin-bottom: 20px;
28. }
29.
30. .btn {
31.     display: inline-block;
32.     background: #000;
33.     color: #fff;
34.     border: none;
35.     padding: 10px 20px;
36.     margin: 5px;
37.     border-radius: 5px;
38.     cursor: pointer;
39.     text-decoration: none;
40.     font-size: 15px;
41.     font-family: inherit;
42. }
43.
44. .btn:focus {
45.     outline: none;
46. }
47.
48. .btn:active {
49.     transform: scale(0.98);
50. }
51.
52. .btn-block {
53.     display: block;
54.     width: 100%;
55. }
56.
57. .task {
58.     background: #f4f4f4;
59.     margin: 5px;
60.     padding: 10px 20px;
61.     cursor: pointer;
62. }
63.
64. .task.reminder {
65.     border-left: 5px solid green;
66. }
67.
68. .task h3 {
69.     display: flex;
70.     align-items: center;
71.     justify-content: space-between;
72. }
73.
74. .add-form {
75.     margin-bottom: 40px;
76. }
77.
78. .form-control {
79.     margin: 20px 0;
80. }
81.
82. .form-control label {
83.     display: block;
84. }
85.
86. .form-control input {
87.     width: 100%;
88.     height: 40px;

```

```

89.   margin: 5px;
90.   padding: 3px 7px;
91.   font-size: 17px;
92. }
93.
94. .form-control-check {
95.   display: flex;
96.   align-items: center;
97.   justify-content: space-between;
98. }
99.
100.   .form-control-check label {
101.     flex: 1;
102.   }
103.
104.   .form-control-check input {
105.     flex: 2;
106.     height: 20px;
107.   }
108.
109.   footer {
110.     margin-top: 30px;
111.     text-align: center;
112.   }

```

Vamos dar uma rápida olhadinha na classe **.container**:

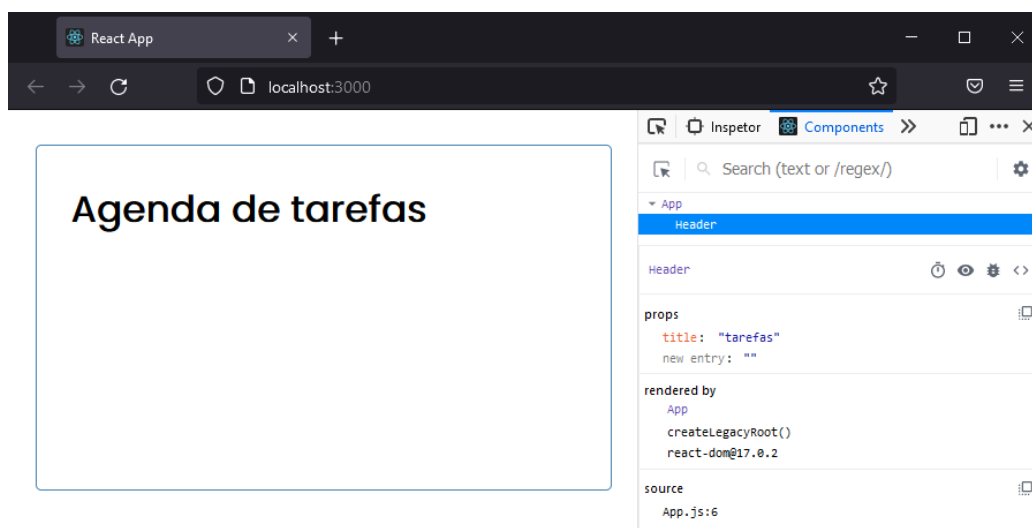
```

.container {
  max-width: 500px;
  margin: 30px auto;
  overflow: auto;
  min-height: 300px;
  border: 1px solid steelblue;
  padding: 30px;
  border-radius: 5px;
}

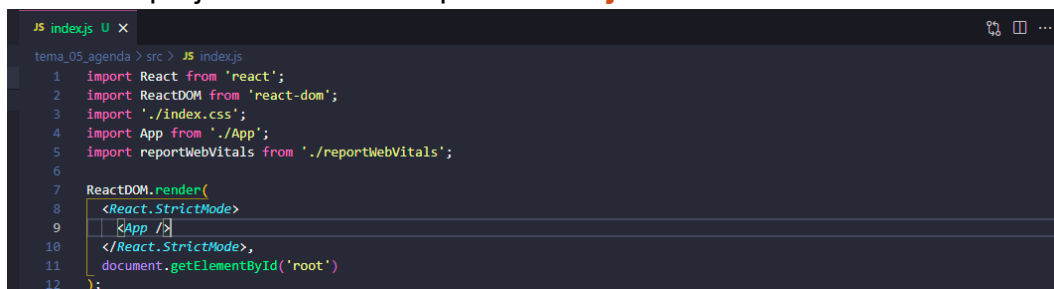
```

- Largura máxima da agenda
- Margem da agenda
- Overflow
- Altura mínima da agenda
- Borda da agenda
- Espaçamento
- Bordas arredondadas da agenda

Essa classe configura **diversas propriedades CSS** para nossa agenda tais como: **largura máxima, margem, altura mínima, borda**, etc. Ao salvar o arquivo **index.css** você verá nossa agenda estilizada assim:



Você não precisa importar o arquivo com o **estilo CSS** no componente **App** porque esse arquivo já está vinculado no projeto dentro do arquivo **index.js**.



```

1 import React from 'react';
2 import ReactDOM from 'react-dom';
3 import './index.css';
4 import App from './App';
5 import reportWebVitals from './reportWebVitals';
6
7 ReactDOM.render(
8   <React.StrictMode>
9     <App />
10   </React.StrictMode>,
11   document.getElementById('root')
12 );

```

É nesse arquivo que o componente **raiz App** e seus componentes **filhos** são renderizados para serem exibidos na página web. Por isso todos os **arquivos**, **recursos**, **funcionalidades** importadas nesse arquivo podem ser acessadas por **todos os componentes da aplicação**.

Vamos atualizar o arquivo **Header.jsx** e inserir a classe **.header** no elemento **<header>**:

```

1. import PropTypes from 'prop-types';
2.
3. const Header = ({ title }) => {
4.   return (
5.     <header className="header">
6.       <h1>Agenda de {title}</h1>
7.     </header>
8.   );
9. };
10.
11. Header.defaultProps = {
12.   title: 'compromissos',
13. };
14.
15. Header.propTypes = {
16.   title: PropTypes.string.isRequired,
17. };
18.
19. export default Header;

```

Criar um componente botão (Evento em React)

Vamos criar um **componente botão** para colocarmos no **cabeçalho**. Esse componente será usado para **inserir novas tarefas na agenda**. Siga os passos para criar o componente botão:

No diretório **Components**, crie um arquivo **Button.jsx** e insira o seguinte código:

```

1. import PropTypes from 'prop-types';
2.
3. const Button = ({ bgColor, text, onClick }) => {
4.   return (
5.     <button
6.       onClick={onClick}
7.       style={{ backgroundColor: bgColor }}
8.       className="btn"
9.     >
10.       {text}
11.     </button>
12.   );

```



```

13. };
14.
15. Button.defaultProps = {
16.   bgColor: 'steelblue',
17. };
18.
19. Button.propTypes = {
20.   text: PropTypes.string.isRequired,
21.   bgColor: PropTypes.string,
22.   onClick: PropTypes.func,
23. };
24.
25. export default Button;

```

Observe que esse **botão** tem um **evento React** chamado **onClick**. Eventos no React nos permitem **executar ações por meio de funções do JavaScript**. Essas ações são de acordo com a **interação do usuário** e podem ser **movimento do mouse**, **clique do mouse**, **foco em um elemento**, etc.

No nosso exemplo o evento é o **onClick**, que é disparado **sempre o botão for clicado pelo usuário**. A função desse evento é passada por meio de propriedade do componente pai **Header** para o componente filho **Button**.

O componente espera **três** propriedades **bgColor**, **text** e **onClick**, que estão na **linha 3** separadas por **vírgula** e entre as chaves (**{ bgColor, text, onClick }**). Estamos usando a **atribuição** via **desestruturação** do JavaScript.

Além disso, também definimos o **valor padrão** para a propriedade **bgColor** (**linhas 18 a 20**) e criamos os validadores das três propriedades (**linhas 22 a 26**).

Para ver o **botão na interface** devemos atualizar o código do componente **Header** e inserir a renderização do componente na **linha 11**.

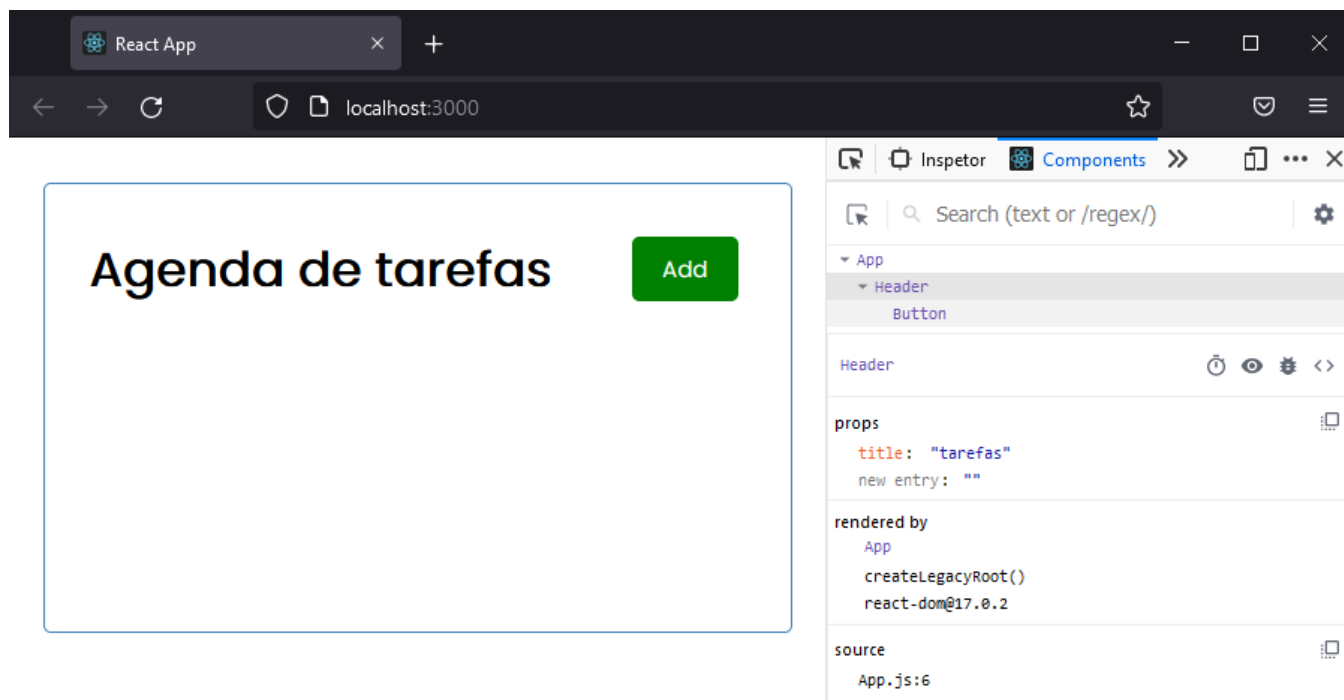
```

1. import PropTypes from 'prop-types';
2. import Button from './Button';
3.
4. const Header = ({ title }) => {
5.   const onClick = () => {
6.     alert('Clicou');
7.   };
8.   return (
9.     <header className="header">
10.      <h1>Agenda de {title}</h1>
11.      <Button bgColor="green" text="Add" onClick={onClick} />
12.    </header>
13.  );
14. };
15.
16. Header.defaultProps = {
17.   title: 'compromissos',
18. };
19.
20. Header.propTypes = {
21.   title: PropTypes.string.isRequired,
22. };
23.
24. export default Header;

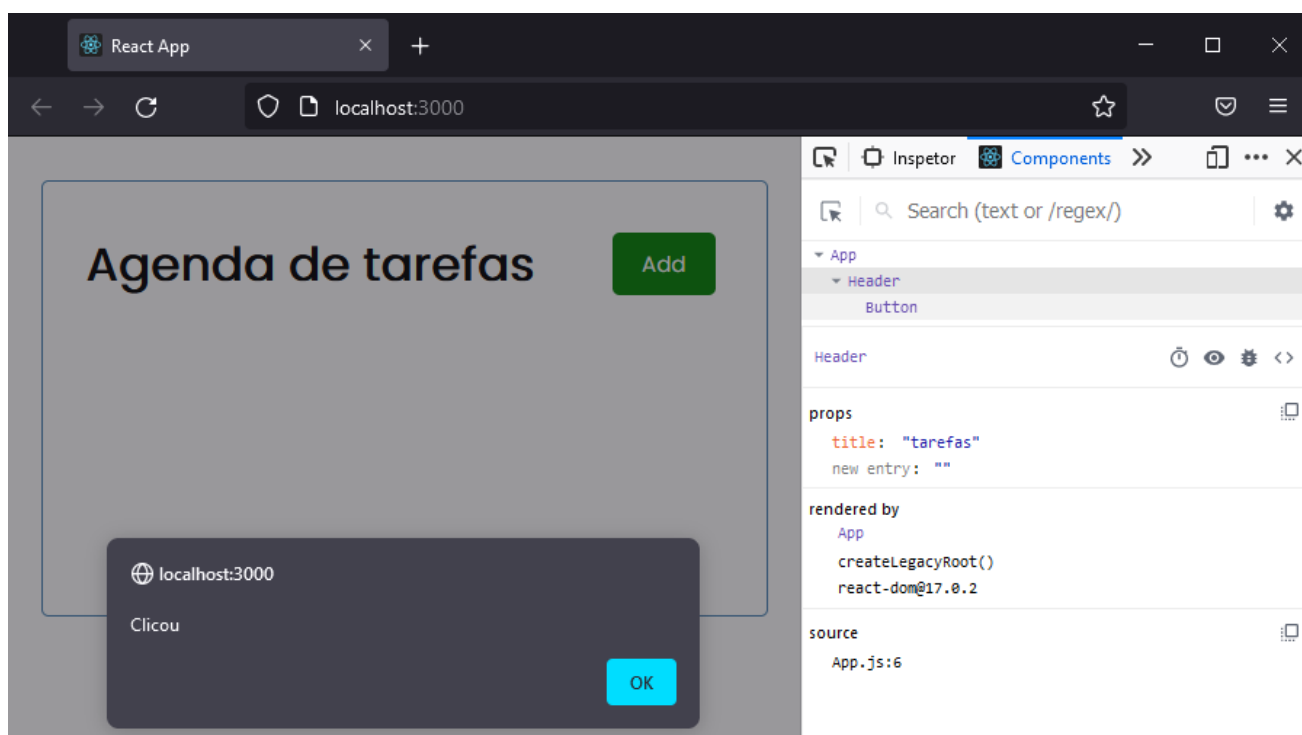
```

Na **linha 11** a instrução passa **três** propriedades **bgColor**, **text** e **onClick**, que são esperadas pelo componente **Button**. Lembre-se que a propriedade **text** é **obrigatória**. Por enquanto, a função do evento **onClick** vai estar no componente **Header**, apenas para fins didáticos. Mais pra frente iremos alterar o **local dessa função**.

Não se esqueça que precisamos **importar o componente Button** como foi feito na **linha 1**. O resultado pode ser visualizado no navegador web.



Se você **clicar no botão** o evento **onClick** será disparado e exibirá a **mensagem de alerta**.





Projeto Agenda de Compromissos - Parte 02

Os objetivos desta aula são:

- Apontar como criar o componente Tasks
- Exercitar a criação do componente TaskItemxxx
- Realizar a instalação de um módulo no React

Bons estudos!



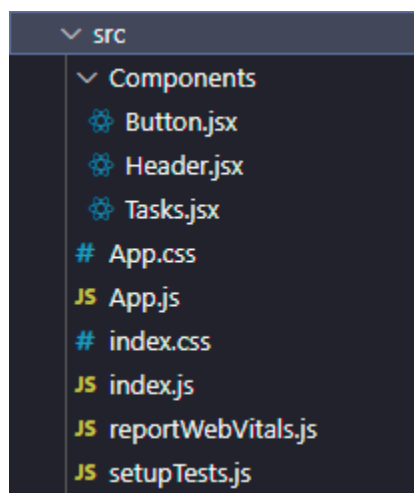
Importante!

Continue a implementação a partir do mesmo ponto que paramos no tema passado.

Criar o componente Tasks

Vamos agora criar o componente **Tasks**, responsável por exibir a **lista de tarefas da aplicação**. Siga os seguintes passos para fazer a implementação.

No diretório **Components**, crie um arquivo **Tasks.jsx**. A organização do seu diretório deve ficar como mostrado na figura abaixo:



Agora insira o código abaixo no arquivo **Tasks.jsx**:

```
1.  const tasks = [
2.    {
3.      id: 1,
4.      text: 'Consulta médica',
5.      day: '5 de Fev as 14:30',
6.      reminder: true,
7.    },
8.    {
9.      id: 2,
10.     text: 'Reunião na Escola',
11.     day: '6 de Fev as 13:30',
12.     reminder: true,
13.    },
14.    {
15.      id: 3,
16.      text: 'Compras no Supermercado',
17.      day: '7 de Fev as 8:30',
18.      reminder: false,
19.    },
20.  ];
21.
22.  const Tasks = () => {
23.    return (
24.      <>
25.        {tasks.map((task) => (
26.          <h3>{task.text}</h3>
27.        ))}
28.      </>
29.    );

```

```
30. };
31.
32. export default Tasks;
```

Observe que esse componente tem uma variável chamada **tasks** com as tarefas que queremos exibir na interface do usuário (**linhas 1 a 20**). Inserir os dados em um componente torna-os **visível** apenas para **esse componente** e seus **filhos**. Portanto, esses dados ficarão no componente **Tasks** temporariamente. Mais pra frente atualizaremos eles para serem o estado da aplicação **React**.

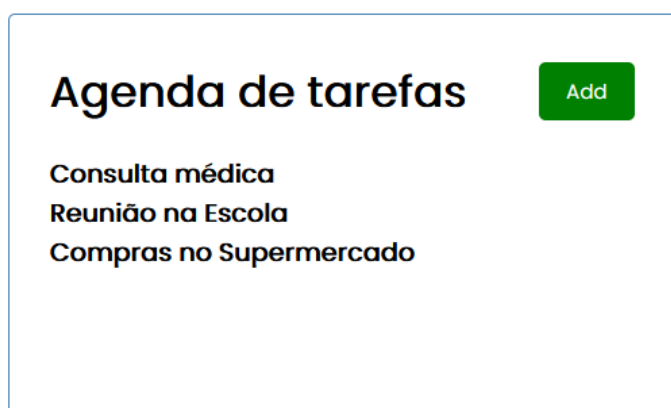
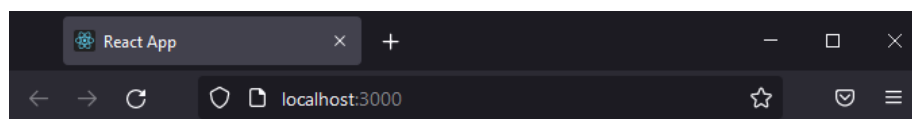
Note também que utilizamos o método de **alto nível** para trabalhar com **array** e criar um a um cada texto das tarefas para serem exibidos na interface do usuário.

Então, agora precisamos exibir o componente **Tasks** na interface do usuário. Para isso, precisamos fazer com que o componente seja renderizado na página web. No arquivo **App.js**, devemos atualizar o código da seguinte forma:

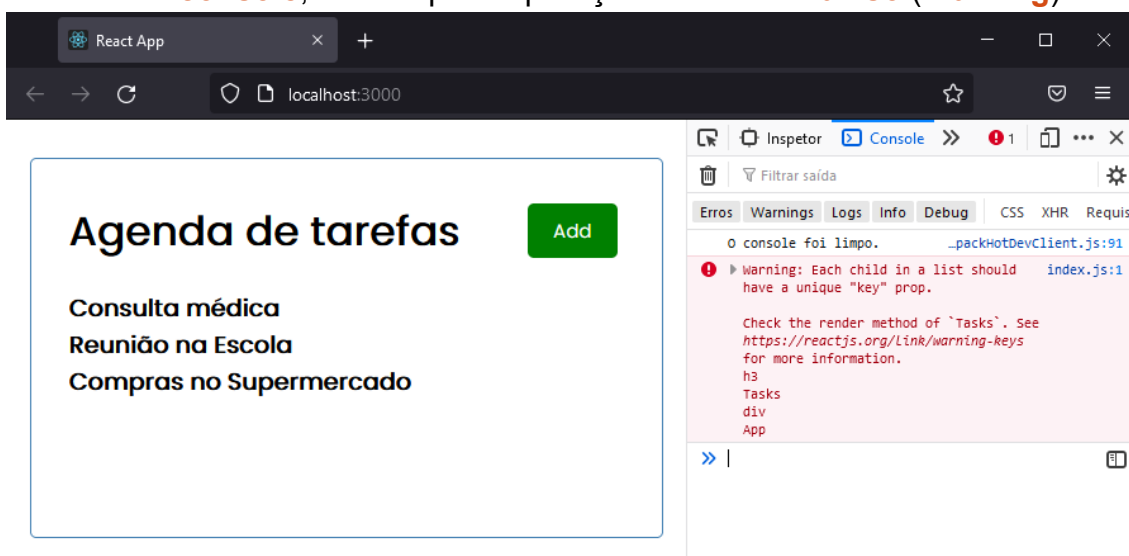
```
1. import Header from './Components/Header';
2. import Tasks from './Components/Tasks';
3.
4. function App() {
5.   return (
6.     <div className="container">
7.       <Header title="tarefas" />
8.       <Tasks />
9.     </div>
10.   );
11. }
12.
13. export default App;
```

Na **linha 2** do código mostrado, temos a instrução para importar o componente filho **Tasks** no componente pai **App**. Desse modo ele ser reconhecido como um componente válido. Caso você esqueça de colocar essa instrução, a aplicação acusará erro na **linha 8**.

A **linha 8** contém o nosso componente indicando que ele deve ser renderizado e colocado após o componente **Tasks**. Salve o arquivo **App.js** e você pode ver o resultado no navegador web.



Mas se você abrir o **console**, notará que a aplicação contém um **aviso (Warning)**.



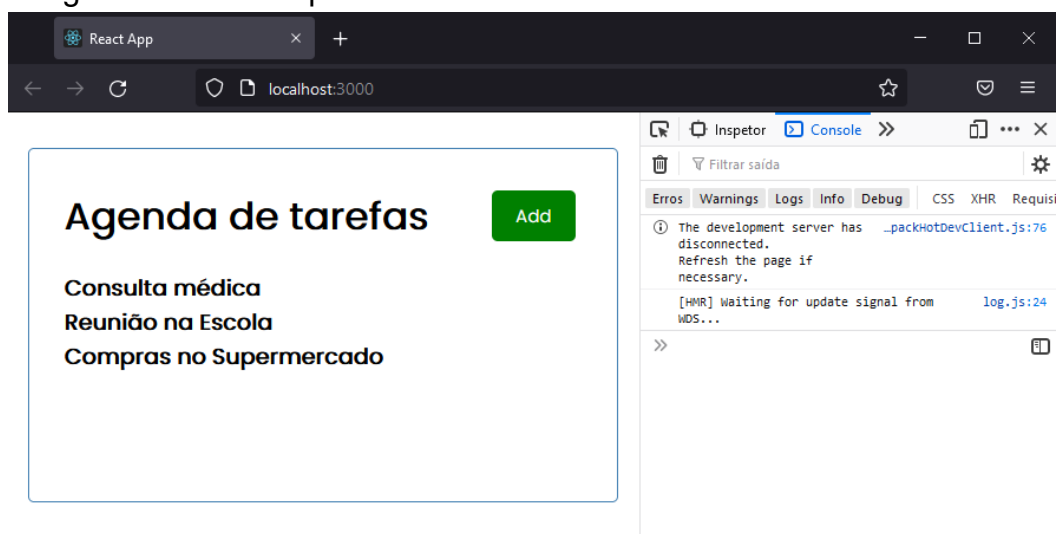
Esse aviso indica que cada **filho criado em uma lista** deve conter uma chave (**key**) **única**. No nosso exemplo estamos criando uma **lista de elemento <h3>** através de método **map()**. Portanto devemos passar **uma key única a cada elemento criado**. Para remover esse aviso, você deve **atualizar o código como mostrado na linha 26**. Nessa linha usamos o campo **ID** de cada uma das tarefas, pois esse campo não possui valor igual e é único para cada tarefa.

```

1. const tasks = [
2.   {
3.     id: 1,
4.     text: 'Consulta médica',
5.     day: '5 de Fev as 14:30',
6.     reminder: true,
7.   },
8.   {
9.     id: 2,
10.    text: 'Reunião na Escola',
11.    day: '6 de Fev as 13:30',
12.    reminder: true,
13.  },
14.  {
15.    id: 3,
16.    text: 'Compras no Supermercado',
17.    day: '7 de Fev as 8:30',
18.    reminder: false,
19.  },
20. ];
21.
22. const Tasks = () => {
23.   return (
24.     <>
25.       {tasks.map((task) => (
26.         <h3 key={task.id}>{task.text}</h3>
27.       ))}
28.     </>
29.   );
30. };
31.
32. export default Tasks;

```

Observe que agora o aviso não parece mais.



Estado da aplicação

Como citamos anteriormente, vamos fazer as tarefas serem um **estado na aplicação** e vamos **transformá-lo em um estado global**, que poderá ser acessado por todos os componentes da aplicação. Para isso, devemos realizar as seguintes atualizações no nosso código:

No arquivo **App.js**, vamos atualizar o código.

```
1. import Header from './Components/Header';
2. import Tasks from './Components/Tasks';
3. import { useState } from 'react';
4.
5. function App() {
6.   const [tasks, setTasks] = useState([
7.     {
8.       id: 1,
9.       text: 'Consulta médica',
10.      day: '5 de Fev as 14:30',
11.      reminder: true,
12.    },
13.    {
14.      id: 2,
15.      text: 'Reunião na Escola',
16.      day: '6 de Fev as 13:30',
17.      reminder: true,
18.    },
19.    {
20.      id: 3,
21.      text: 'Compras no Supermercado',
22.      day: '7 de Fev as 8:30',
23.      reminder: false,
24.    },
25.  ]);
26.
27.  return (
28.    <div className="container">
29.      <Header title="tarefas" />
30.      <Tasks tasks={tasks} />
31.    </div>
32.  );
33. }
```

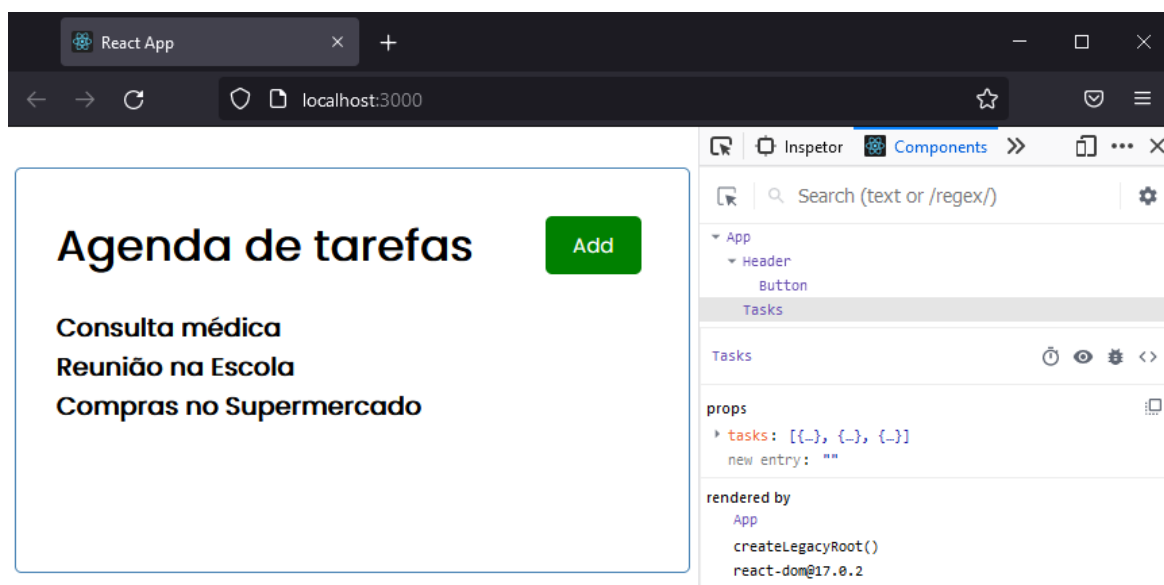
```
34.
35. export default App;
```

Aqui estamos usando o **hook useState** (linha 6) para definir **tasks como um estado da aplicação React**. Isso é importante nesse projeto, pois mais a diante iremos precisar que outros componentes acessem as informações contidas nesse estado. Além disso, na **linha 30**, estamos passando o conteúdo do estado por meio da propriedade **tasks**.

No arquivo **Task.jsx**, precisamos agora **modificar o código** para receber a propriedade e manipulá-la para exibir corretamente na interface do usuário. Desse modo, atualize o código do componente **Tasks** conforme mostrada abaixo.

```
1. const Tasks = ({ tasks }) => {
2.   return (
3.     <>
4.       {tasks.map((task) => (
5.         <h3 key={task.id}>{task.text}</h3>
6.       ))}
7.     </>
8.   );
9. };
10.
11. export default Tasks;
```

O resultado visual no navegador é o mesmo, mas essa atualização é interessante pois, além de utilizarmos um recurso importante do React, será possível que outros componentes acessem o estado da aplicação. O que muda é que agora o componente **Tasks** possui uma propriedade **tasks** por onde vem o conteúdo do estado global da aplicação.



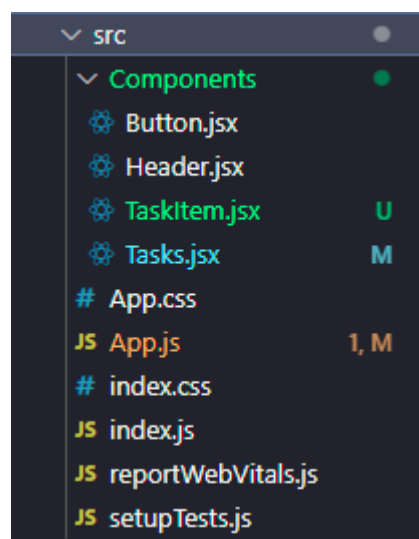
Criando o componente TaskItem

Em programação sempre devemos “**dividir para conquistar**”, isto é, não devemos criar uma **função, elemento, componente Chimera** (pronuncia-se Quimera), que possui **todas as partes do código e realiza todas as ações**, etc. Desse modo, vamos criar um elemento **filho** do

componente **Tasks**, que será responsável por criar a exibição de cada agendamento e, assim, dividir a responsabilidade com o componente **Pai**.

Siga os passos para criar o componente **TaskItem**.

No diretório **Components**, crie um arquivo **TaskItem.jsx**. A organização do seu diretório deve ficar como mostrado na figura abaixo:



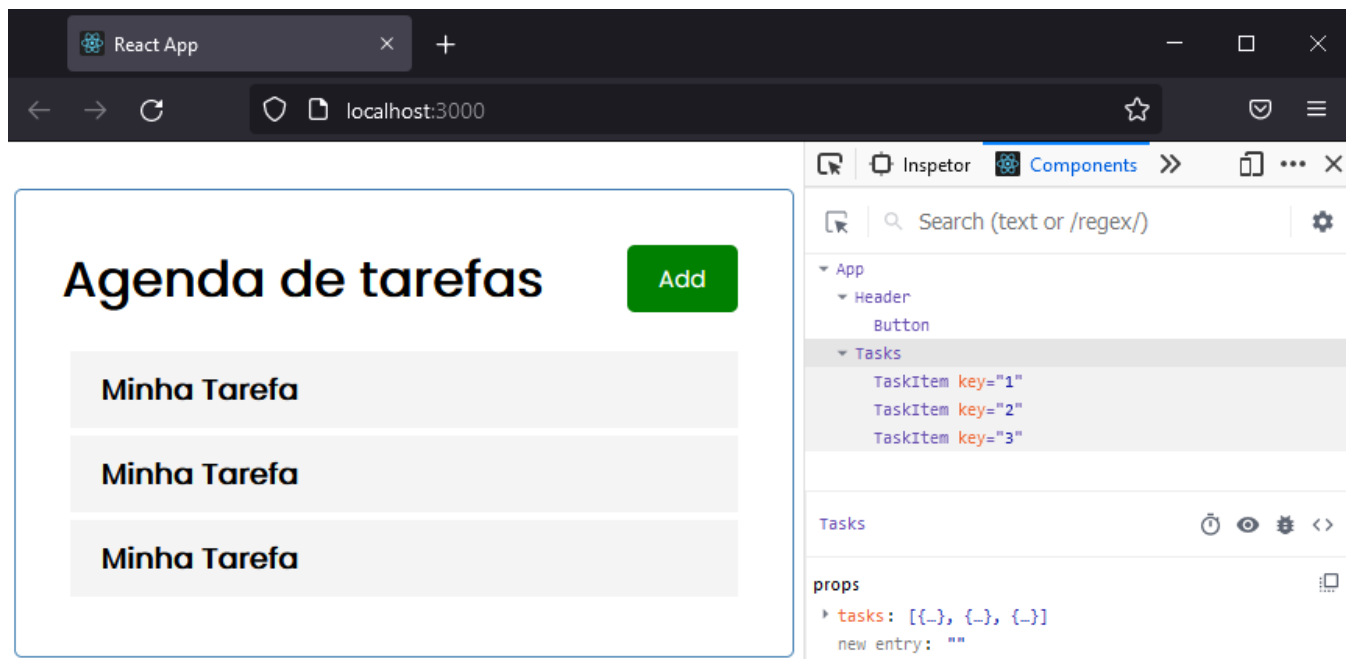
Agora insira o código abaixo no arquivo **TaskItem.jsx**:

```
1. const TaskItem = () => {
2.   return (
3.     <div className="task">
4.       <h3>Minha Tarefa</h3>
5.     </div>
6.   );
7. };
8.
9. export default TaskItem;
```

No arquivo **Tasks.js**, vamos modificar o código para incluir o componente **TaskItem** e passar as informações por meio de propriedades para esse componente. **Atualize o código** como mostrado abaixo:

```
1. import TaskItem from './TaskItem';
2.
3. const Tasks = ({ tasks }) => {
4.   return (
5.     <>
6.       {tasks.map((task) => (
7.         <TaskItem key={task.id} task={task} />
8.       ))}
9.     </>
10.   );
11. };
12.
13. export default Tasks;
```

Se você visualizar a aplicação no navegador web verá a seguinte interface.



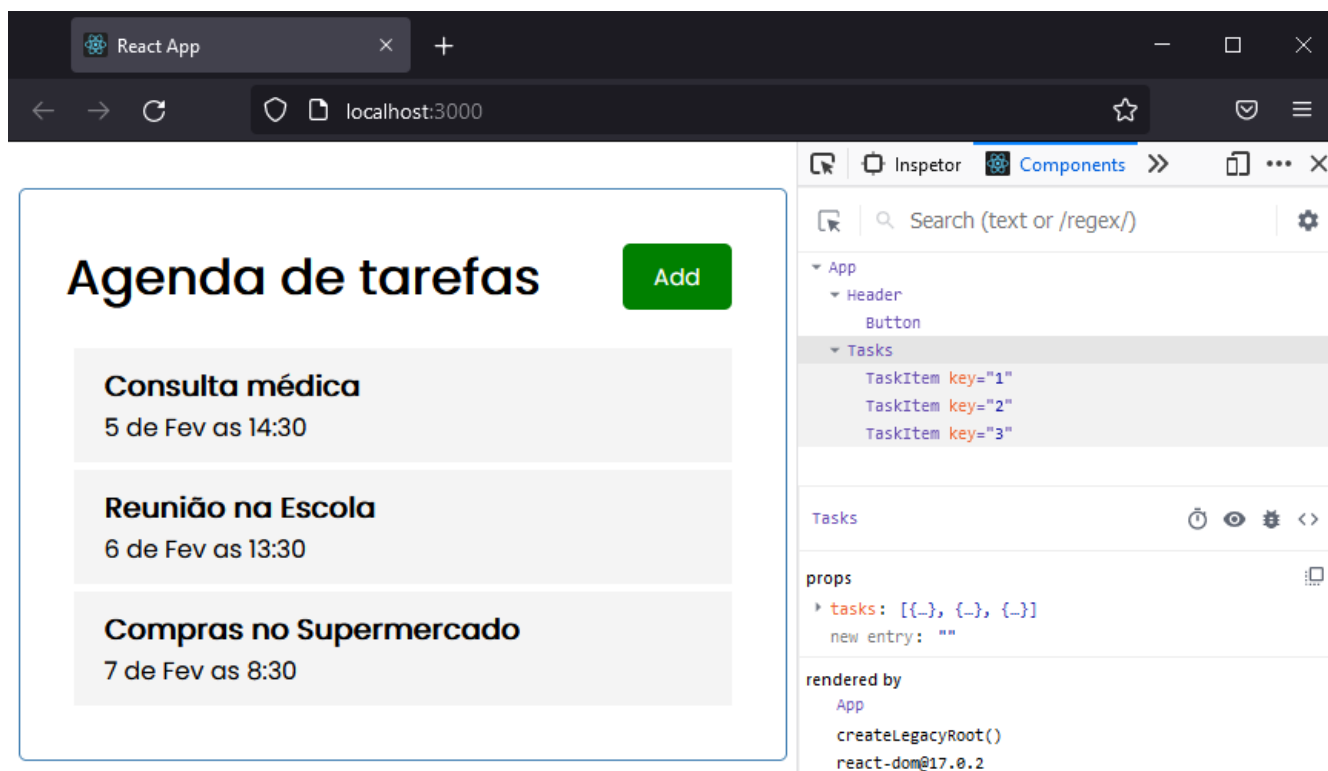
Isso porque o nosso código do componente **TaskItem** possui um **valor estático** com o elemento **<h3>** e o texto **minha tarefa**.

A informação aparece três vezes porque o **método map()** percorre os três elementos dentro do array de objeto **tasks**. Desse modo, são criados três elementos **TaskItem com Keys diferentes, como podemos ver na extensão React Developer Tools**.

Vamos **modificar o código** do componente **TaskItem** para que ele atualize o conteúdo dinamicamente. Atualize o código como mostrado abaixo:

```
1. const TaskItem = ({ task }) => {
2.   return (
3.     <div className="task">
4.       <h3>{task.text}</h3>
5.       <p>{task.day}</p>
6.     </div>
7.   );
8. };
9.
10. export default TaskItem;
```

Agora podemos ver no navegador web a nossa aplicação funcionando.



Se você clicar em cada um dos componentes **TaskItem** na extensão **React Developer Tools**, poderá ver que cada um tem o conteúdo de um item do **array de objetos tasks**, que é nosso estado global da aplicação.



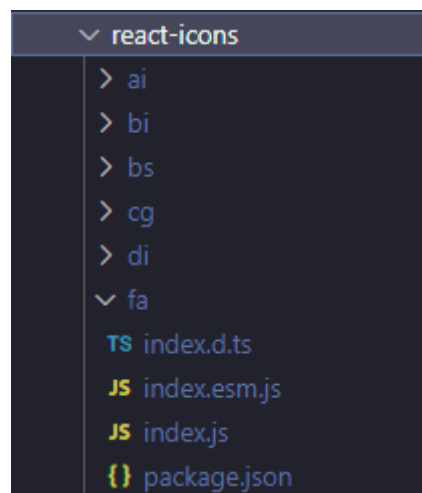
Instalando e utilizando um módulo no React

Uma das vantagens do **React** é que podemos utilizar o gerenciador de pacotes **NPM** para **instalar módulos de terceiros**. Assim podemos utilizar na aplicação. Vamos fazer isso para instalar o pacote **react-icons**. Esse pacote contém diversas **fontes de ícones** para serem utilizados em uma página web. No nosso caso, estamos interessados nos ícones do **Font Awesome**. Siga os passos para instalar a biblioteca react-icons.

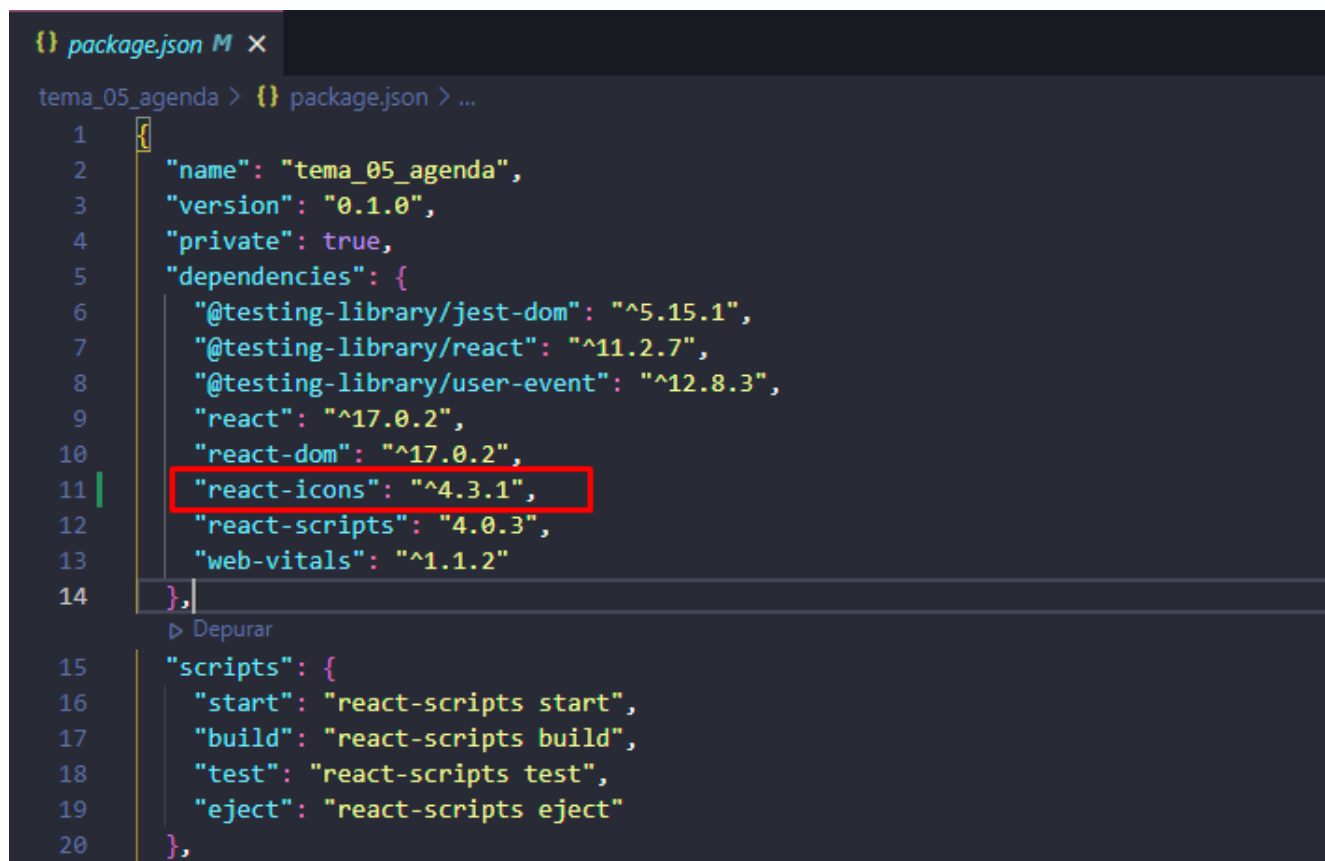
No terminal do VS Code, tenha certeza de que **você está no diretório do projeto** e execute o seguinte comando:

```
npm i react-icons
```

Com a instalação realizada com sucesso, você pode ver o pacote instalado dentro da pasta **node-modules**, dentro desse pacote, temos a pasta **fa** de **Font Awesome**:



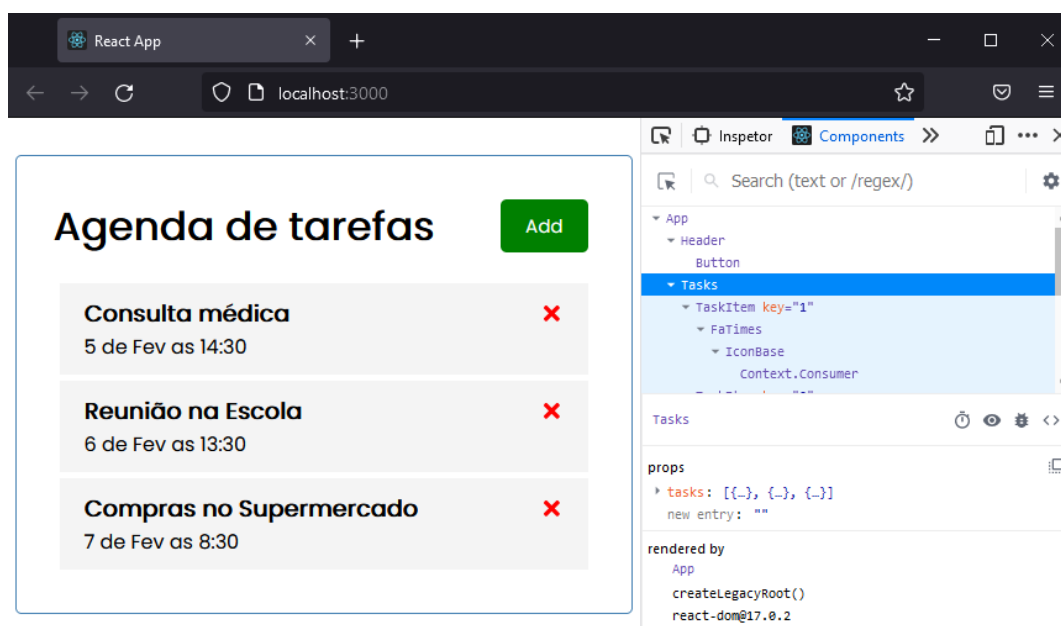
Você não precisa expandir esse diretório, pois ele é muito grande e pode se perder na hierarquia dos arquivos do projeto depois. Quando voltarmos com a implementação. Você pode conferir de uma forma mais fácil pelo arquivo **package.json**



Se o pacote não estiver instalado, a aplicação acusará erro nos próximos passos e você deverá fazer a instalação corretamente para funcionar. Então, devemos agora **atualizar o código** do componente **TaskItem** como mostrado abaixo.

```
1. import { FaTimes } from 'react-icons/fa';
2.
3. const TaskItem = ({ task }) => {
4.   return (
5.     <div className="task">
6.       <h3>
7.         {task.text}
8.         <FaTimes style={{ color: 'red', cursor: 'pointer' }} />
9.       </h3>
10.      <p>{task.day}</p>
11.    </div>
12.  );
13. };
14.
15. export default TaskItem;
```

Agora podemos ver no navegador web o ícone **X** que será usado para “**deletar**” o agendamento. No momento, esse ícone não faz nada.





Projeto Agenda de Compromissos - Parte 03

Os objetivos desta aula são:

- Exercitar a criação de eventos para automatizar funções

Bons estudos!



Importante!

Continue a implementação a partir do mesmo ponto que paramos no tema passado.

Evento para deletar uma tarefa agendada

Vamos agora criar o evento para **apagar uma tarefa exibida na aplicação**, quando clicamos no ícone **X**. A função desse evento vai estar implementada no componente App e será passada para os componentes **filhos** através de **props**. Siga os passos para **atualizar o código**.

No arquivo **App.js**, vamos modificar o código como mostrada abaixo.

```
1. import Header from './Components/Header';
2. import Tasks from './Components/Tasks';
3. import { useState } from 'react';
4.
5. function App() {
6.   const [tasks, setTasks] = useState([
7.     {
8.       id: 1,
9.       text: 'Consulta médica',
10.      day: '5 de Fev as 14:30',
11.      reminder: true,
12.    },
13.    {
14.      id: 2,
15.      text: 'Reunião na Escola',
16.      day: '6 de Fev as 13:30',
17.      reminder: true,
18.    },
19.    {
20.      id: 3,
21.      text: 'Compras no Supermercado',
22.      day: '7 de Fev as 8:30',
23.      reminder: false,
24.    },
25.  ]);
26.  // Deletar tarefa
27.  const deletaTarefa = (id) => {
28.    setTasks(tasks.filter((task) => task.id !== id));
29.  };
30.
31.  return (
32.    <div className="container">
33.      <Header title="tarefas" />
34.      <Tasks tasks={tasks} onDelete={deletaTarefa} />
35.    </div>
36.  );
37. }
38.
39. export default App;
```

Observe que nas **linhas 27 a 29** criamos a função **deletaTarefa**, que utiliza o método de alto nível **filter** para **manipular arrays**, que filtrará a tarefa que desejamos apagar e exibirá somente as outras tarefas na interface do usuário.

Além disso, queremos passar essa função como propriedade para o componente **TaskItem**, então temos que primeiro passar para o componente **Tasks** e depois para o componente

TaskItem. Pois temos que respeitar a hierarquia da árvore de componentes. Não podemos saltar componentes na árvore.

Desse modo, na **linha 34**, passamos a função **deletaTarefa** através da propriedade **onDelete** para o componente **Tasks**.

No arquivo **Tasks.jsx**, devemos receber a propriedade e passá-la para o componente **filho TaskItem**. Para isso, atualize o código como mostrado abaixo.

```
1. import TaskItem from './TaskItem';
2.
3. const Tasks = ({ tasks, onDelete }) => {
4.   return (
5.     <>
6.       {tasks.map((task) => (
7.         <TaskItem key={task.id} task={task} onDelete={onDelete} />
8.       ))}
9.     </>
10.   );
11. };
12.
13. export default Tasks;
```

Note que a função é recebida como uma propriedade pelo componente na **linha 3** (**{ tasks, onDelete }**) e ela é repassada para o componente **TaskItem** como uma propriedade para o componente filho **TaskItem** na **linha 7** **<TaskItem key={task.id} task={task} onDelete={onDelete} />**.

Por fim, devemos receber e trabalhar com a função no componente **TaskItem** conforme o código mostrado abaixo.

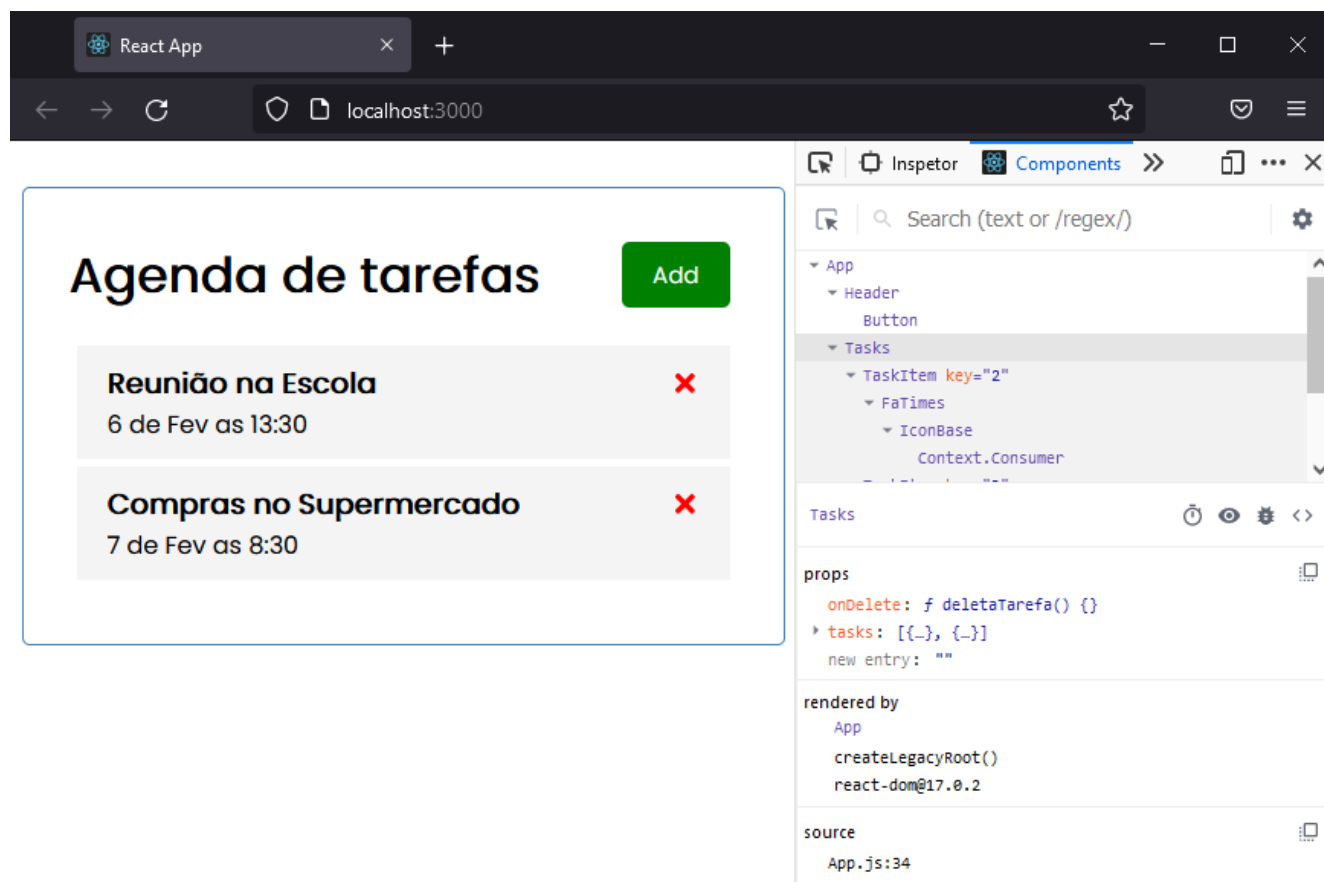
```
1. import { FaTimes } from 'react-icons/fa';
2.
3. const TaskItem = ({ task, onDelete }) => {
4.   return (
5.     <div className="task">
6.       <h3>
7.         {task.text}
8.         <FaTimes
9.           style={{ color: 'red', cursor: 'pointer' }}
10.          onClick={() => onDelete(task.id)}
11.        />
12.       </h3>
13.       <p>{task.day}</p>
14.     </div>
15.   );
16. };
17.
18. export default TaskItem;
```

Agora o componente **TaskItem** está recebendo a propriedade **onDelete**, na linha 3.

```
const TaskItem = ({ task, onDelete })
```

E como essa propriedade é uma **função**, devemos utilizá-la como tal. Por isso, na **linha 10** foi necessário usar o evento React **onClick**, para que a função seja chamada quando clicarmos no

ícone **X**. Além de usar uma **Arrow Function** para invocar a função passada na propriedade **onDelete()**. Também precisamos passar a **ID** da tarefa que estamos clicando. Se você agora clicar no ícone **X** de alguma tarefa agendada, ela não será mais exibida.



Atenção: Essa ação só acontece no **Frontend** se você **carregar a página** novamente tudo voltará para a exibição inicial, pois não temos um **Backend** para **armazenar** as **alterações**. Siga os seguintes passos para fazer a implementação.

Vamos colocar uma mensagem para caso todas as tarefas sejam apagadas. Para isso, você deve **atualizar o código** do componente **App.js**, dentro do método **return** como mostrado abaixo.

```
1. import Header from './Components/Header';
2. import Tasks from './Components/Tasks';
3. import { useState } from 'react';
4.
5. function App() {
6.   const [tasks, setTasks] = useState([
7.     {
8.       id: 1,
9.       text: 'Consulta médica',
10.      day: '5 de Fev as 14:30',
11.      reminder: true,
12.    },
13.    {
14.      id: 2,
15.      text: 'Reunião na Escola',
16.      day: '6 de Fev as 13:30',
17.      reminder: true,
18.    },
19.  ]),
20. }
```

```

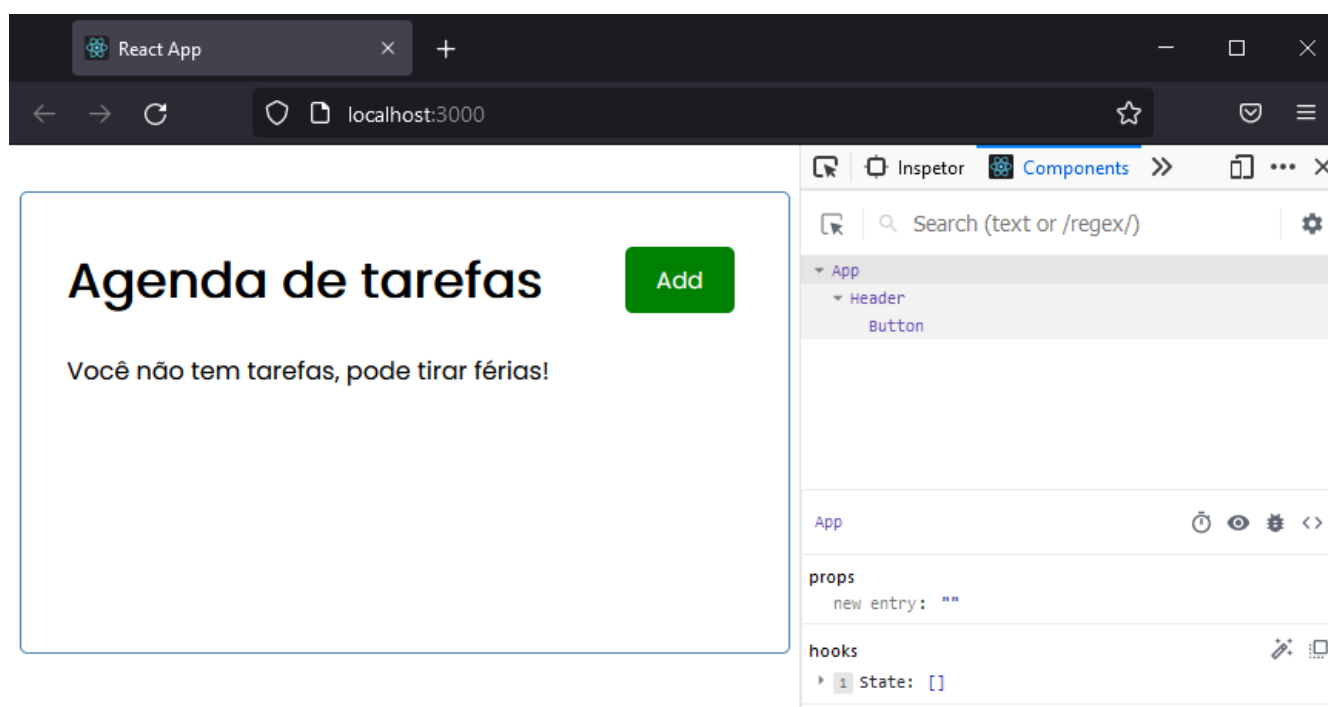
19.      {
20.        id: 3,
21.        text: 'Compras no Supermercado',
22.        day: '7 de Fev as 8:30',
23.        reminder: false,
24.      },
25.    ]);
26.    // Deletar tarefa
27.    const deletaTarefa = (id) => {
28.      setTasks(tasks.filter((task) => task.id !== id));
29.    };
30.
31.    return (
32.      <div className="container">
33.        <Header title="tarefas" />
34.        {tasks.length > 0 ? (
35.          <Tasks tasks={tasks} onDelete={deletaTarefa} />
36.        ) : (
37.          'Você não tem tarefas, pode tirar férias!'
38.        )}
39.      </div>
40.    );
41.  }
42.
43. export default App;

```

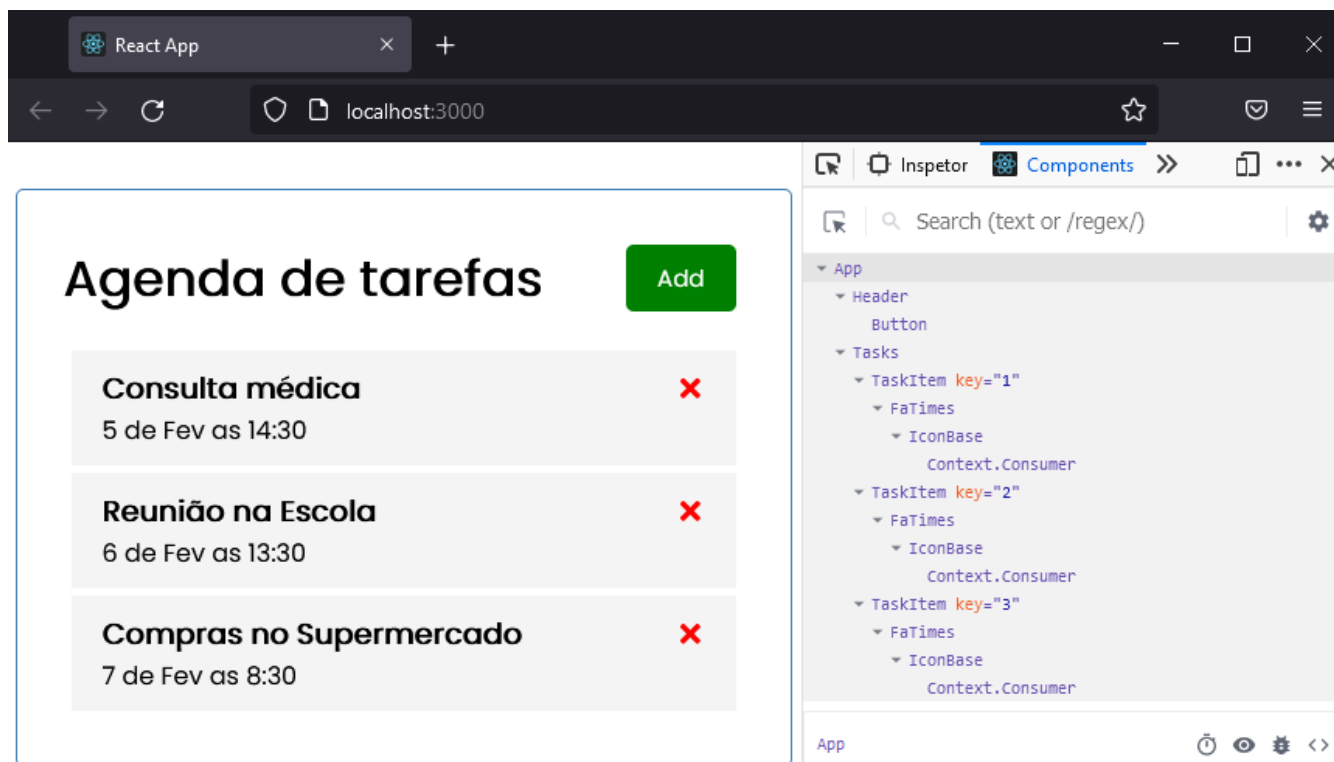
O código mostrado nas **linhas 34 a 38** é o que chamamos de **renderização condicional** de um componente. Nesse exemplo, utilizamos o operador ternário para verificar se o tamanho do **array de objetos** de tarefas tem alguma informação. Caso contenha elementos, a condição `tasks.length > 0` é verdadeira e então o componente é **renderizado**. Caso contrário, a mensagem será exibida na tela.

Detalhe: essa verificação é feita em tempo de execução e não no estado com valor original.

Veja o que acontece quando você clica no ícone **X** de todas as tarefas e a interface não tem tarefa para ser exibida.



Observe que o hook **State** está **vazio**, mas se você **recarregar a página novamente tudo volta ao normal**. Porque toda essa interação está acontecendo em tempo de execução no **Frontend**, não existe um **Backend** para tornar as alterações permanentes.



Evento para alterar o reminder do agendamento

Vamos criar o evento para mudar o valor booleano (de **true** para **false** ou de **false** para **true**) do campo **reminder** de cada item no array de objetos. Esse evento será disparado quando clicarmos duas vezes sobre a tarefa agendada. A função desse evento vai estar implementada no componente App e será passada para os componentes filhos através de **props**. Siga os passos para atualizar o código.

No arquivo **App.js**, vamos **modificar o código** como mostrada abaixo.

```
1. import Header from './Components/Header';
2. import Tasks from './Components/Tasks';
3. import { useState } from 'react';
4.
5. function App() {
6.   const [tasks, setTasks] = useState([
7.     {
8.       id: 1,
9.       text: 'Consulta médica',
10.      day: '5 de Fev as 14:30',
11.      reminder: true,
12.    },
13.    {
14.      id: 2,
15.      text: 'Reunião na Escola',
16.      day: '6 de Fev as 13:30',
17.      reminder: true,
```

```

18.     },
19.     {
20.         id: 3,
21.         text: 'Compras no Supermercado',
22.         day: '7 de Fev as 8:30',
23.         reminder: false,
24.     },
25.   ]);
26.   // Deletar tarefa
27.   const deletaTarefa = (id) => {
28.     setTasks(tasks.filter((task) => task.id !== id));
29.   };
30.   // Alterar o reminder
31.   const mudarReminder = (id) => {
32.     setTasks(
33.       tasks.map((task) =>
34.         task.id === id ? { ...task, reminder: !task.reminder } : task
35.       )
36.     );
37.     console.log(id);
38.   };
39.
40.   return (
41.     <div className="container">
42.       <Header title="tarefas" />
43.       {tasks.length > 0 ? (
44.         <Tasks
45.           tasks={tasks}
46.           onDelete={deletaTarefa}
47.           onToggle={mudarReminder}
48.         />
49.       ) : (
50.         'Você não tem tarefas, pode tirar férias!'
51.       )}
52.     </div>
53.   );
54. }
55.
56. export default App;

```

Observe que nas **linhas 31 a 38** criamos a função **mudarReminder**, que utiliza o método de alto nível **map** para buscar a tarefa que queremos alterar o **valor booleano** do campo **reminder**. Utilizamos operador **rest** (...) na **linha 34** para atualizar o array **tasks** com o objeto que teve o **reminder** alterado.

Além disso, queremos passar essa função como propriedade para o componente **TaskItem**. Então temos que primeiro passar para o componente **Tasks** e depois para o componente **TaskItem**. Pois é necessário respeitar a hierarquia da árvore de componentes. Não podemos saltar componentes na árvore.

Desse modo, na **linha 47** passamos a função **mudarReminder** através da propriedade **onToggle** para o componente **Tasks**.

No arquivo **Tasks.jsx**, devemos receber a propriedade e passá-la para o componente filho **TaskItem**. Para isso, atualize o código como mostrado abaixo.

```

1. import TaskItem from './TaskItem';
2.
3. const Tasks = ({ tasks, onDelete, onToggle }) => {
4.   return (
5.     <>

```

```

6.      {tasks.map((task) => (
7.          <TaskItem
8.              key={task.id}
9.              task={task}
10.             onDelete={onDelete}
11.             onToggle={onToggle}
12.         )/>
13.     ))}
14. </>
15. );
16. };
17.
18. export default Tasks;

```

Note que a função é recebida como uma propriedade pelo componente na **linha 3** ({ `tasks`, `onDelete` , `onToggle` }) e ela é repassada para o componente **TaskItem** como uma propriedade para o componente filho **TaskItem** nas **linhas 6 a 13**.

```

{tasks.map((task) => (
  <TaskItem
    key={task.id}
    task={task}
    onDelete={onDelete}
    onToggle={onToggle}
  />
))}

```

Por fim, devemos receber e trabalhar com a função no componente **TaskItem.jsx** conforme o código mostrado abaixo.

```

import { FaTimes } from 'react-icons/fa';

const TaskItem = ({ task, onDelete, onToggle }) => {
  return (
    <div
      className={`task ${task.reminder ? 'reminder' : ''}`}
      onDoubleClick={() => onToggle(task.id)}
    >
      <h3>
        {task.text}
        <FaTimes
          style={{ color: 'red', cursor: 'pointer' }}
          onClick={() => onDelete(task.id)}
        />
      </h3>
      <p>{task.day}</p>
    </div>
  );
};

export default TaskItem;

```

Agora o componente **TaskItem** está recebendo a propriedade **onToggle**, na **linha 3**

```
const TaskItem = ({ task, onDelete, onToggle })
```

E como essa propriedade também é uma **função**, devemos utilizá-la como tal. Por isso, na **linha 7** foi necessário usar a **Arrow Function** para invocar a **função** e passar a **ID do agendamento** que estamos clicando.

Além disso, colocarmos a chamada dessa função no evento React **onDoubleClick** no elemento **<div>**. Pois assim podemos clicar em qualquer parte da tarefa agendada para alterar o **reminder** (**linhas 5 a 8**).

Observe também que na **linha 6** colocamos uma condição para o atributo **className**:

```
className={`task ${task.reminder ? 'reminder' : ''}`}
```

Caso o **reminder** esteja com o valor **true**, além da classe **task** o elemento **<div>** também conterá a classe **reminder**. Caso contrário, o elemento **<div>** conterá somente a classe **task**. A classe **reminder** coloca uma barra verde no início da tarefa agendada.

Com a classe reminder

Consulta médica

5 de Fev as 14:30

×

Sem a classe reminder

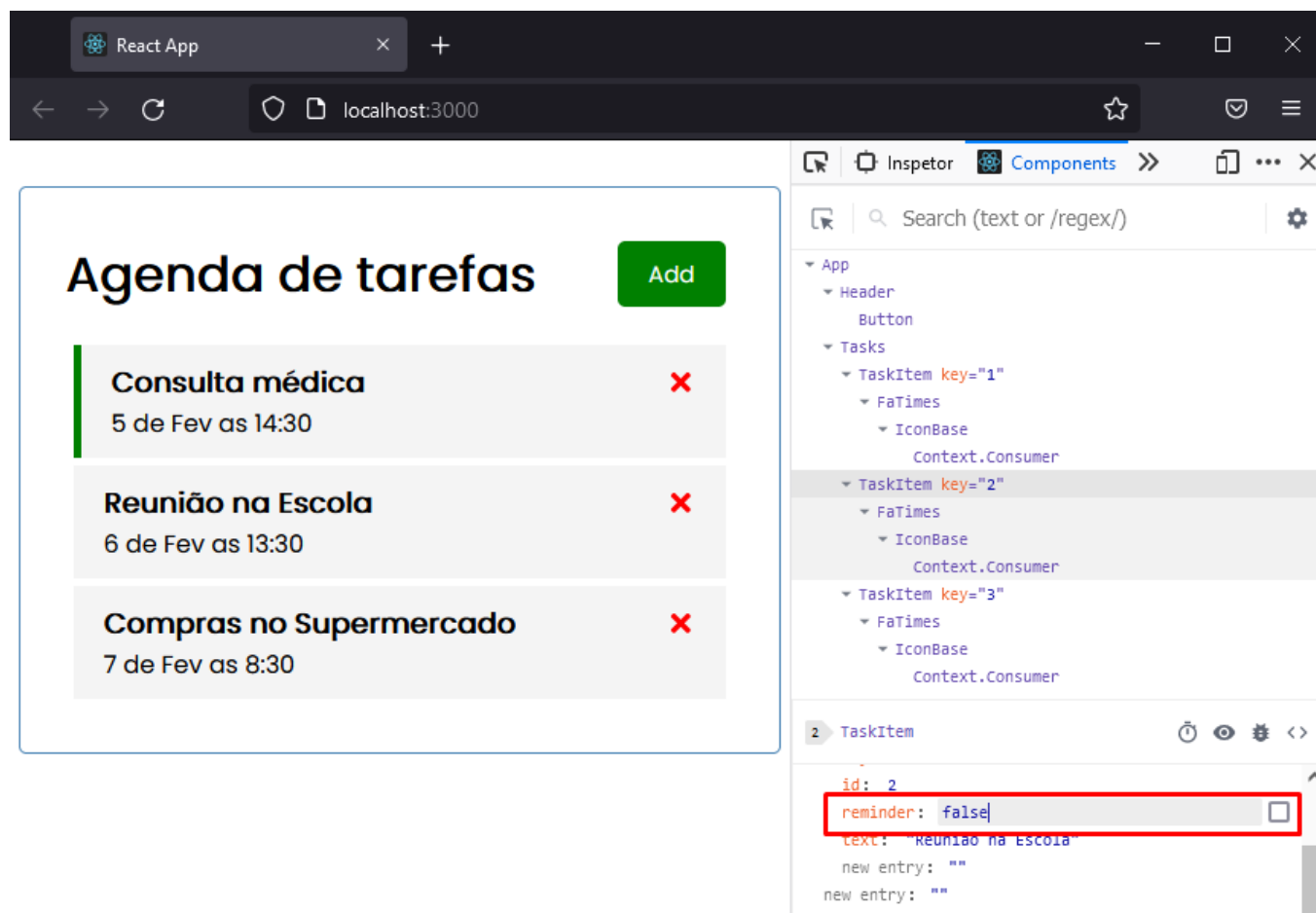
Compras no Supermercado

7 de Fev as 8:30

×

No navegador você pode ver como ficou nossa aplicação.

Observe na extensão **React Developer Tools** que clicamos no **TaskItem key="2"** e o campo **reminder** está com o valor **true**. Se você agora clicar **duas vezes na tarefa** o valor do campo **reminder** mudar, assim como a sua exibição no navegador web.




Importante!

Lembrando, essa ação só acontece no **Frontend**, se você carregar a página novamente tudo voltará para a exibição inicial, pois não temos um **Backend** para armazenar as alterações. Siga os seguintes passos para fazer a implementação.



Projeto Agenda de Compromissos - Parte 04

Os objetivos desta aula são:

- Implementar compromissos/tarefas utilizando a abordagem de componentes funcionais.

Bons estudos!



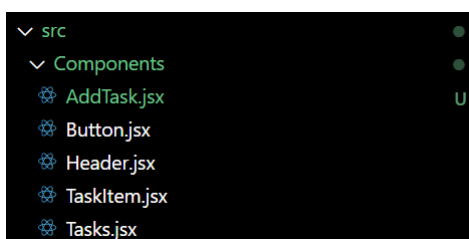
Importante!

Continue a implementação a partir do mesmo ponto que paramos no tema passado

Evento para adicionar uma tarefa na agenda

Vamos agora **criar** o **formulário** e o **evento** para **adicionar uma tarefa** e **exibi-la** na aplicação. Começaremos **criando** o arquivo **AddTask.jsx** no diretório **scr**. Nesse arquivo iremos implementar o componente responsável por **adicionar** as **tarefas** em nossa **agenda**. Siga os passos para atualizar o código.

No diretório **scr**, crie o arquivo **AddTask.jsx** e, então dentro dessa pasta teremos a seguinte estrutura.



No arquivo **AddTask.jsx**, devemos inserir o código do nosso componente como mostrado abaixo.

```
1. const AddTask = () => {
2.   return (
3.     <form className="add-form">
4.       <div className="form-control">
5.         <label>Tarefa</label>
6.         <input type="text" placeholder="Adicione uma tarefa" />
7.       </div>
8.       <div className="form-control">
9.         <label>Dia e Horário</label>
10.        <input type="text" placeholder="Adicione o dia e a hora" />
11.      </div>
12.      <div className="form-control form-control-check">
13.        <label>Reminder</label>
14.        <input type="checkbox" />
15.      </div>
16.      <input type="submit" value="Salvar a Tarefa" className="btn btn-block"/>
17.    </form>
18.  )
19. }
20.
21. export default AddTask
```

Note que o nosso componente possui a marcação **<form>**, isso significa que iremos criar o **formulário** com nesse código. O nosso formulário possuirá três campos:

- Uma marcação **<input>** para inserirmos a tarefa.
- Outra marcação **<input>** para definirmos a data e horário.
- Um **checkbox** para setarmos o reminder da tarefa ou não.

Agora, vamos **adicionar o novo componente** no arquivo **App.js** para ser renderizado no nosso Frontend. Portanto, temos que atualizar o código do arquivo **App.js** em dois lugares:

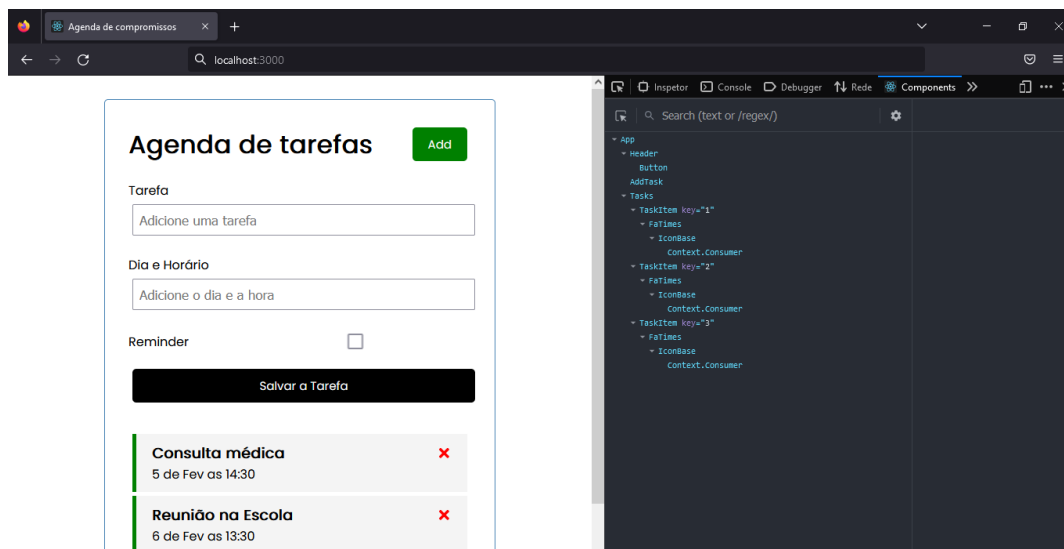
Primeiro, no início, temos que colocar a instrução para importar o novo componente como mostrado na linha 3 do trecho de código a seguir:

```
1. import Header from './Components/Header';
2. import Tasks from './Components/Tasks';
3. import { useState } from 'react';
```

Segundo, temos que atualizar o método **return()** do arquivo **App.js** inserido o novo componente na linha 44 logo abaixo do **<Header />** como mostrado no trecho de código abaixo:

```
41. return (
42.   <div className="container">
43.     <Header title="tarefas" />
44.     <AddTask />
45.     {tasks.length > 0 ? (
46.       <Tasks
47.         tasks={tasks}
48.         onDelete={deletaTarefa}
49.         onToggle={mudarReminder}
50.       />
51.     ) : (
52.       'Você não tem tarefas, pode tirar férias!'
53.     )}
54.   </div>
55. );
```

Se você, iniciar a aplicação **React**, verá que agora temos um formulário logo abaixo do título da nossa agenda.



Se abrirmos a extensão **React Dev Tools** poderemos visualizar o novo componente que agora faz parte da nossa página web.

Por enquanto, esse formulário não adiciona nenhuma tarefa. Então vamos implementar o código para fazer isso então.

Vamos **atualizar** o arquivo **AddTask.jsx** como mostrado abaixo.

```

1.  import { useState } from 'react';
2.
3.  const AddTask = ({ onAdd }) => {
4.      const [text, setText] = useState('');
5.      const [day, setDay] = useState('');
6.      const [reminder, setReminder] = useState(false);
7.
8.      const onSubmit = (e) => {
9.          e.preventDefault();
10.
11.          if(!text){
12.              alert("Por favor adicione uma tarefa!");
13.              return;
14.          }
15.
16.          onAdd({text, day, reminder});
17.
18.          setText('');
19.          setDay('');
20.          setReminder(false);
21.      }
22.
23.      return (
24.          <form className="add-form" onSubmit={onSubmit}>
25.              <div className="form-control">
26.                  <label>Tarefa</label>
27.                  <input type="text"
28.                      placeholder="Adicione uma tarefa"
29.                      value={text}
30.                      onChange={(e) => setText(e.target.value)}
31.                  />
32.              </div>
33.              <div className="form-control">
34.                  <label>Dia e Horário</label>
35.                  <input type="text"

```

```

36.             placeholder="Adicione o dia e a hora"
37.             value={day}
38.             onChange={(e) => setDay(e.target.value)}
39.         />
40.     </div>
41.     <div className="form-control form-control-check">
42.         <label>Reminder</label>
43.         <input type="checkbox"
44.             checked={reminder}
45.             value={reminder}
46.             onChange={(e) => setReminder(e.currentTarget.checked)}
47.         />
48.     </div>
49.     <input type="submit" value="Salvar a Tarefa" className="btn btn-block" />
50. </form>
51. )
52. }
53.
54. export default AddTask

```

No código mostrado acima inserimos a instrução na linha 1 para importar o **Hook useState** e criarmos três estados para o nosso componente (linhas 4, 5 e 6):

```

4. const [text, setText] = useState('');
5. const [day, setDay] = useState('');
6. const [reminder, setReminder] = useState(false);

```

Na linha 3, podemos ver a função **onAdd** sendo passada como **props** ao nosso **componente**. Essa função será implementada no componente App mais adiante.

Também implementamos a função **onSubmit()** nas linhas 8 até 21. Esta função será disparada quando clicarmos no botão **Salvar** a Tarefa do formulário. Na função, podemos ver a declaração **if** nas linhas 11 até 14 que testa se estado **text** está **vazio**, ou seja, se o campo tarefa do formulário está vazio. Caso verdadeiro, uma mensagem de alerta será exibida na nossa página web. Caso contrário, chamaremos a função **onAdd** que é passado como propriedade ao nosso componente. Ao invocarmos essa função na linha 16, passamos o conteúdo dos campos dos formulários através dos estados **text**, **day** e **reminder**. E nas linhas 18 a 20, **limpamos** os **campos do formulário**.

Na linha 24, inserimos o evento **onSubmit** na marcação **<form>** que invoca a função **onSubmit** implementada anteriormente.

Na marcação **<input>** que será usada para inserimos a tarefa, atualizamos nas linhas 13 e 14 o atributo **value** que recebe o estado **text** e o evento **onChange()** que utiliza a função **setText** para atualizar o estado **text** e consequentemente atualizar o campo do formulário.

Você pode observar que fazemos a mesma coisa com a marcação **<input>** do dia e horário, nas linhas 21 e 22 temos o atributo **value** que recebe o estado **day** e o evento **onChange()** que utiliza a função **setDay** para atualizar o estado **day** e consequentemente atualizar esse campo do formulário.

No checkbox reminder, inserimos nas linhas 28 e 29 o atributo **value** com o estado **reminder** e o evento **onChange()** para atualizar o valor do estado **reminder** e atualizar o checkbox do formulário.

Por fim, vamos ao arquivo **App.js** e inserir a função para adicionar a tarefa na nossa página web. Portanto, abaixo do estado **tasks** com acima da função implementada para deletar tarefas, vamos inserir o seguinte trecho de código.

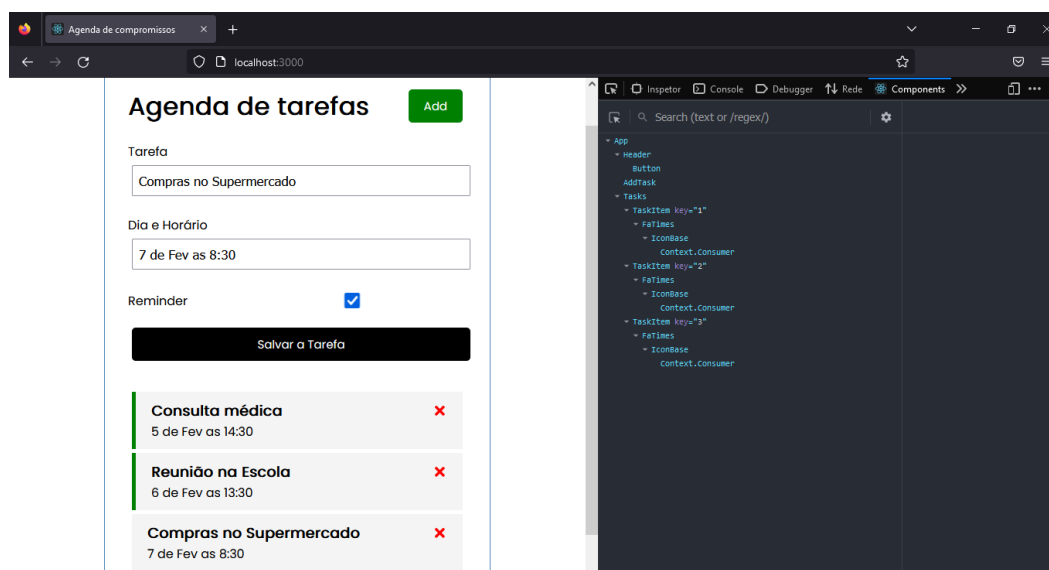
```
28. // Adicionar tarefa
29. const addTask = (task) => {
30.   const id = Math.floor(Math.random() * 100000) + 1;
31.   const newTask = { id, ...task};
32.   setTasks([...tasks, newTask]);
33. };
```

No código mostrado acima, temos a função **addTask**, que é passada como propriedade para o componente **AddTask** e invocada quando clicamos no botão **Salvar** a **Tarefa**. Essa função gera um **id aleatório** para a **nova tarefa** na linha 30, organiza a nova tarefa na variável **newTask** na linha 31 com a **id** e os **valores** do **text**, **day** e **reminder** passados pelo componente **AddTask**. E na linha 32, atualizamos o **estado** do componente **App** para exibir a **lista atualizada** de **tarefas**.

A outra parte do arquivo **App.js** que devemos atualizar é no método **return()**. Nessa parte devemos colocar a função **addTask** como propriedade do componente **AddTask** como mostrado na linha 52 do trecho de código abaixo:

```
49. return (
50.   <div className="container">
51.     <Header title="tarefas" />
52.     <AddTask onAdd={addTask} />
53.     {tasks.length > 0 ? (
54.       <Tasks tasks={tasks} onDelete={deletaTarefa} onToggle={mudarReminder}
55.     ) : (
56.       'Você não tem tarefas, pode tirar férias!'
57.     )}
58.   </div>
59. );
```

Agora podemos adicionar **quantas tarefas quisermos** na nossa **agenda**.



Usando o botão Add

Agora vamos usar o botão verde **Add** na nossa página para **exibir** ou **esconder** o nosso **formulário** para adicionar tarefas. Siga os passos para atualizarmos o nosso código.

Primeiro, vamos **atualizar** o componente **Header**, pois o botão verde Add está dentro desse componente.

```
1. import PropTypes from 'prop-types';
2. import Button from './Button';
3.
4. const Header = ({ title, onAdd }) => {
5.   const onClick = () => {
6.     onAdd();
7.   };
8.   return (
9.     <header className="header">
10.      <h1>Agenda de {title}</h1>
11.      <Button bgColor="green" text="Add" onClick={onClick} />
12.    </header>
13.  );
14. };
15.
16. Header.defaultProps = {
17.   title: 'compromissos',
18. };
19.
20. Header.propTypes = {
21.   title: PropTypes.string.isRequired,
22. };
23.
24. export default Header;
```

A mudanças que devemos fazer são:

- Inserir a função **onAdd** como propriedade desse componente na linha 4
- Inserir a chamada da função **onAdd()** na linha 6 da função **onClick()**.



Importante!

poderíamos apagar a função **onClick()** nas linhas 5 até 7 e colocar a função **onAdd()** na linha 11 que obteremos o mesmo resultado. Sendo assim, a linha 11 ficaria assim:

```
<Button bgColor="green" text="Add" onClick={onAdd} />
```

Agora devemos **atualizar o código** do arquivo **App.js**:

```

1. import Header from './Components/Header';
2. import Tasks from './Components/Tasks';
3. import AddTask from './Components/AddTask';
4. import { useState } from 'react';
5.
6. function App() {
7.   const [showAddTask, setShowAddTask] = useState(false);
8.   const [tasks, setTasks] = useState([
9.     {
10.      id: 1,
11.      text: 'Consulta médica',
12.      day: '5 de Fev as 14:30',
13.      reminder: true,
14.    },
15.    {
16.      id: 2,
17.      text: 'Reunião na Escola',
18.      day: '6 de Fev as 13:30',
19.      reminder: true,
20.    },
21.    {
22.      id: 3,
23.      text: 'Compras no Supermercado',
24.      day: '7 de Fev as 8:30',
25.      reminder: false,
26.    },
27.  ]);
28.
29.  // Adicionar tarefa
30.  const addTask = (task) => {
31.    const id = Math.floor(Math.random() * 100000) + 1;
32.    const newTask = { id, ...task };
33.    setTasks([...tasks, newTask]);
34.  };
35.
36.  // Deletar tarefa
37.  const deletaTarefa = (id) => {
38.    setTasks(tasks.filter((task) => task.id !== id));
39.  };
40.  // Alterar o reminder
41.  const mudarReminder = (id) => {
42.    setTasks(
43.      tasks.map((task) =>
44.        task.id === id ? { ...task, reminder: !task.reminder } : task
45.      )
46.    );
47.    console.log(id);
48.  };
49.
50.  return (
51.    <div className="container">
52.      <Header title="tarefas" onAdd={() => setShowAddTask(!showAddTask)} />
53.      {showAddTask && <AddTask onAdd={addTask} />}
54.      {tasks.length > 0 ? (
55.        <Tasks tasks={tasks} onDelete={deletaTarefa} onToggle={mudarReminder}
56.        />
57.      ) : (
58.        'Você não tem tarefas, pode tirar férias!'
59.      )}
60.    </div>
61.  );

```

```
58.     })
59.   </div>
60.   );
61. }
62.
63. export default App;
```

Devemos fazer apenas três modificações, a primeira na linha 7 adicionamos o **estado showAddTask** e o iniciamos com o valor false:

```
const [showAddTask, setShowAddTask] = useState(false);
```

Segunda na linha 52, devemos passar a função **onAdd** como propriedade do componente **<Header>**:

```
<Header title="tarefas" onAdd={() => setShowAddTask(!showAddTask)} />
```

E a terceira na linha 53, colocarmos a lógica **AND** para **exibir ou não** o componente **AddTask** de acordo com o valor lógico do estado **showAddTask**:

```
{showAddTask && <AddTask onAdd={addTask} />}
```

Agora você pode **exibir** ou **esconder** o formulário de **adicionar nova tarefa** clicando no botão verde **Add**.