



eFront - React

O que é e como iniciar com React

React é uma biblioteca JavaScript para a criação de interfaces de usuário que utilizam o padrão SPA (Single Page Application).

Está bem luri, mas o que é uma SPA?

SPA é um padrão em que seu carregamento dos recursos (como CSS, JavaScript e HTML das páginas) ocorre apenas uma única vez: na primeira vez em que o usuário acessa a aplicação. Isso faz com que o carregamento das novas páginas seja bem mais rápido.

React foi criado pelo Meta (antiga empresa Facebook) em meados de 2011. Com o intuito de resolver um problema que os engenheiros do Facebook identificaram, que é a renderização da aplicação. Com React, na medida em que os dados mudam apenas os pedaços (que contém os dados) de uma determinada parte é renderizada. Diferente quando trabalhamos em uma aplicação feita com HTML, CSS e JavaScript onde a renderização é feita na página toda.

E como isso é possível?

Com o React, as interfaces do usuário são compostas por pedaços de código isolados, chamados de "componentes". Isso torna o código mais fácil de dar manutenção e reutiliza o mesmo código para outras funcionalidades.

Create React App

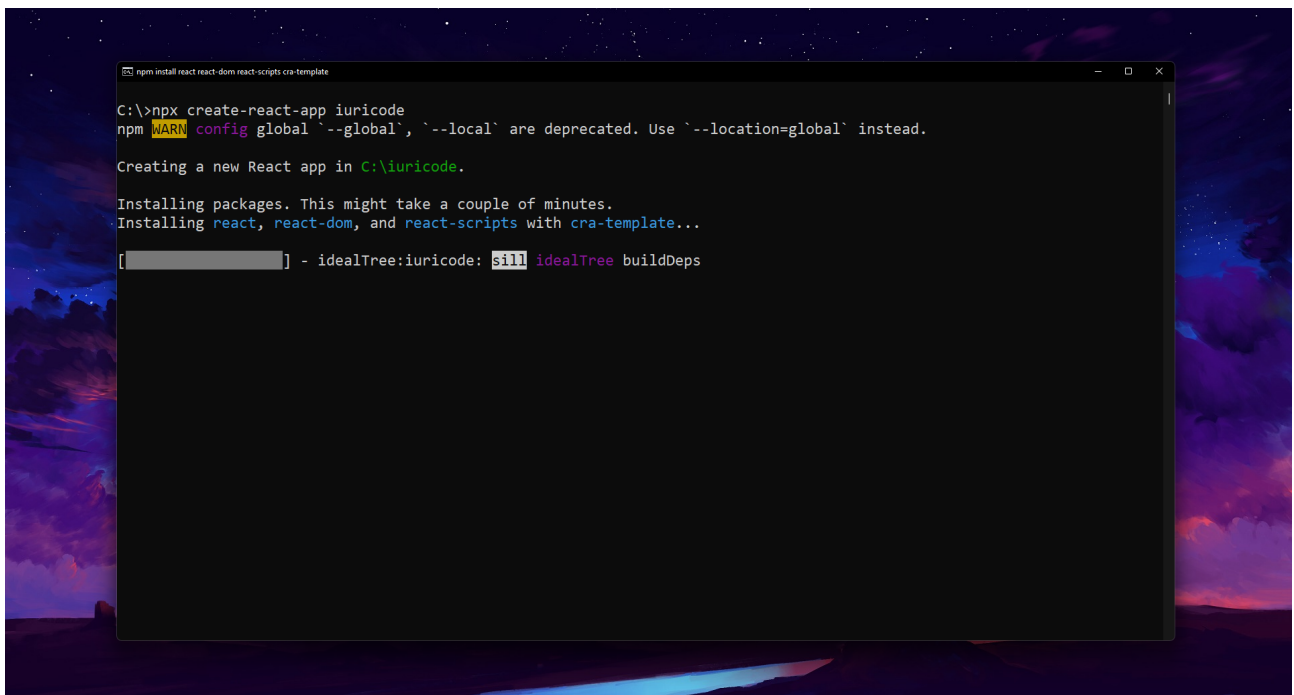
O CRA (create-react-app) é um conjunto de ferramentas e funcionalidades pré-configuradas para que você possa iniciar um projeto React.

Além de configurar seu ambiente de desenvolvimento para utilizar as funcionalidades mais recentes do JavaScript, ele fornece uma experiência de desenvolvimento agradável, e

otimiza a sua aplicação para produção. Será necessário ter o [Node](#) na sua máquina para criar um projeto em React.

Após instalar o Node já podemos criar um projeto, para isso damos o seguinte comando em nosso terminal:

```
npx create-react-app iuricode
```

A screenshot of a terminal window with a dark background and a colorful, abstract landscape wallpaper. The terminal shows the command 'npx create-react-app iuricode' being executed. The output includes a warning about deprecated flags, confirmation of the new React app creation in 'C:\iuricode', and the installation of 'react', 'react-dom', and 'react-scripts' using 'cra-template'. The progress bar at the bottom shows 'idealTree:iuricode: sill idealTree buildDeps'.

- **npx**: npx é um executor responsável por executar as bibliotecas que podem ser baixadas do site npm.
- **create-react-app**: responsável por criar as funcionalidades.
- **iuricode**: nome do projeto.

Após o npx instalar os pacotes iremos dar o seguinte comando para entrar na aplicação criada.

```
cd iuricode
```

```
Prompt de Comando

Created git commit.

Success! Created iuricode at C:\iuricode
Inside that directory, you can run several commands:

  npm start
    Starts the development server.

  npm run build
    Bundles the app into static files for production.

  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 iuricode
  npm start

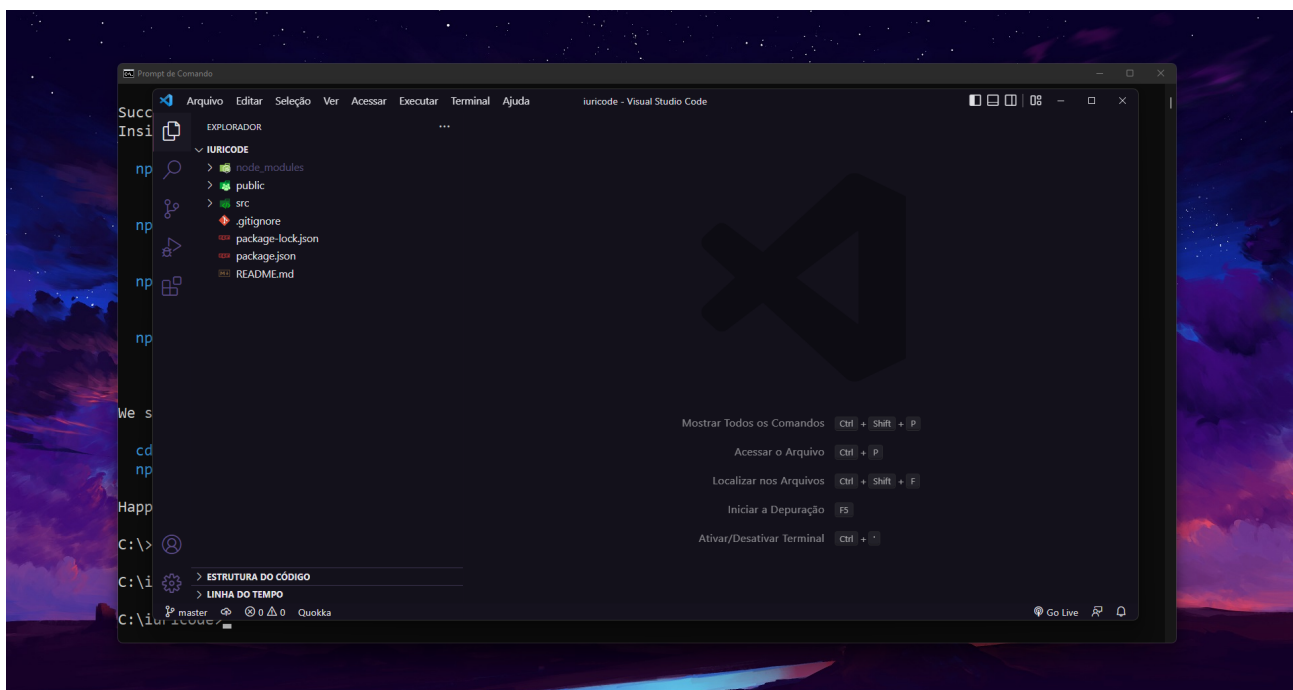
Happy hacking!

C:\>cd iuricode
C:\iuricode>
```

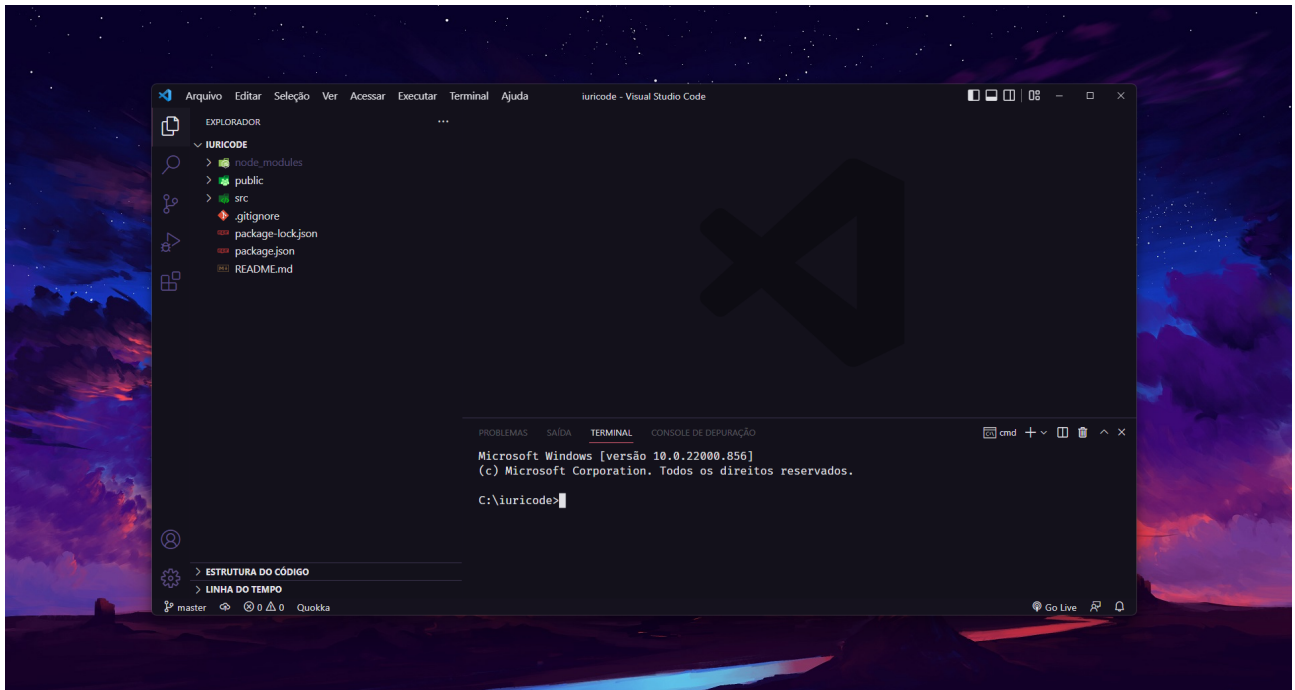
- **cd**: permite a mudança do diretório atual, nesse caso acessamos a pasta iuricode

Depois de estar no diretório do projeto criado podemos dar o seguinte comando para abrir o projeto dentro do nosso [Visual Studio Code](#):

code .

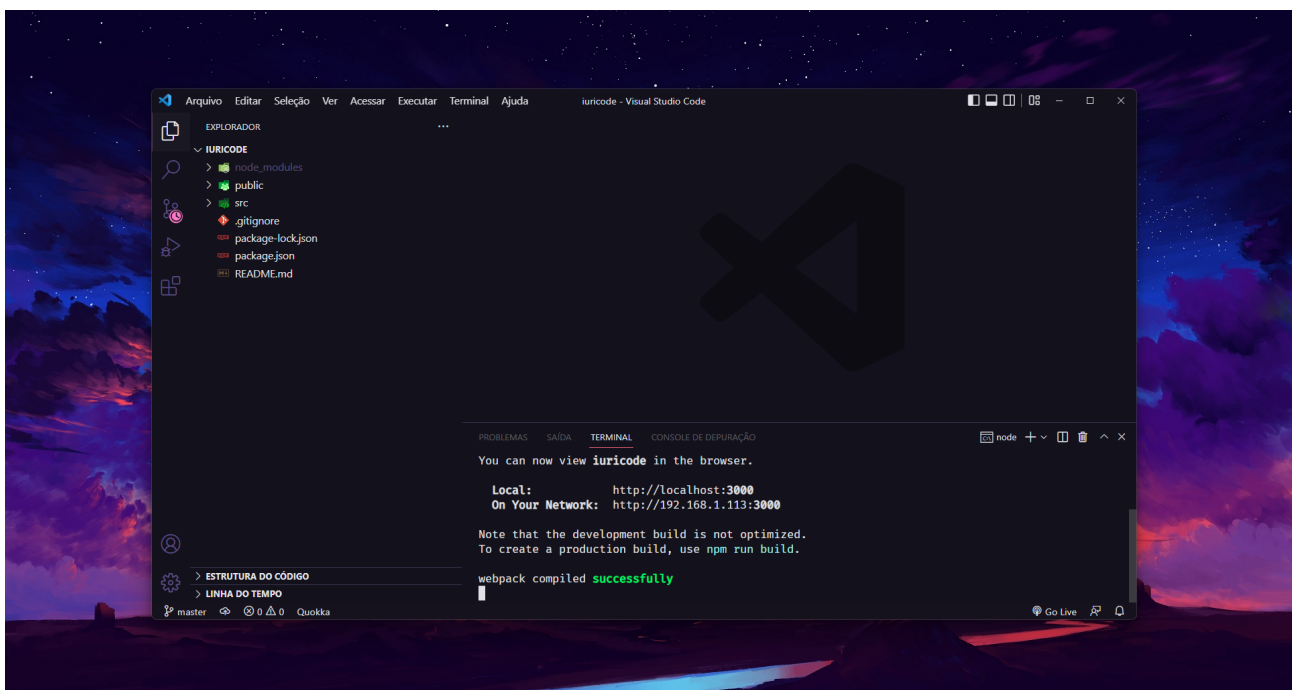


Após abrir o Visual Studio Code pressione as teclas Ctrl + ` (aspa simples), dessa forma irá abrir um terminal integrado no Visual Studio Code.



Em nosso terminal integrado iremos dar o seguinte comando para rodar nossa aplicação:

npm start



- **npm start:** é usado para executar a aplicação.

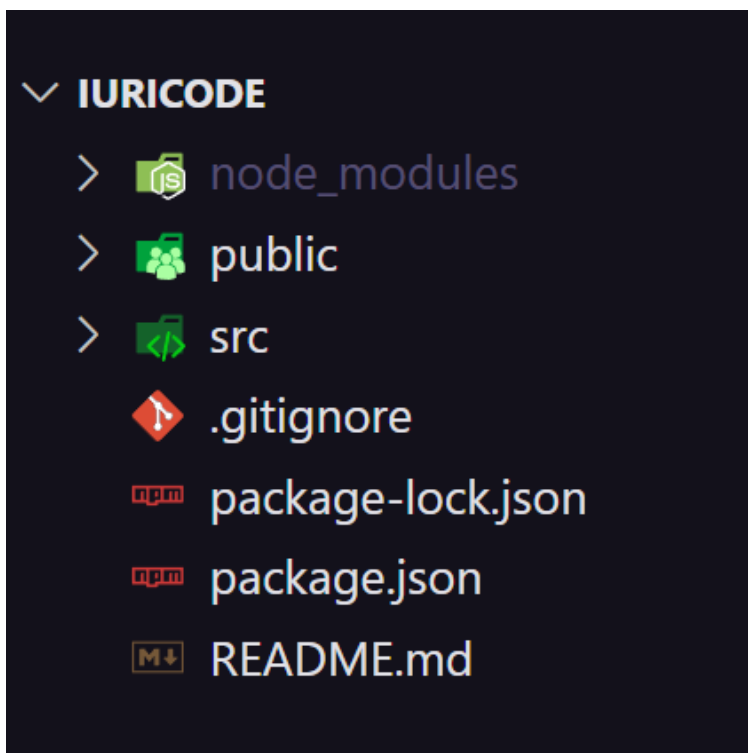
Após realizar o comando irá abrir (automaticamente) uma aplicação com o padrão do CRA na porta 3000 (três mil). Caso não seja iniciado automaticamente a aplicação no navegador você pode acessá-la em:

<http://localhost:3000/>

Uma coisa super importante é que às vezes será preciso para o comando npm start e rodar novamente. Em muitos dos casos será preciso fazer essa tarefa quando mexemos na configuração, mudamos os formatos dos arquivos, instalamos um novo pacote e entre outros casos.

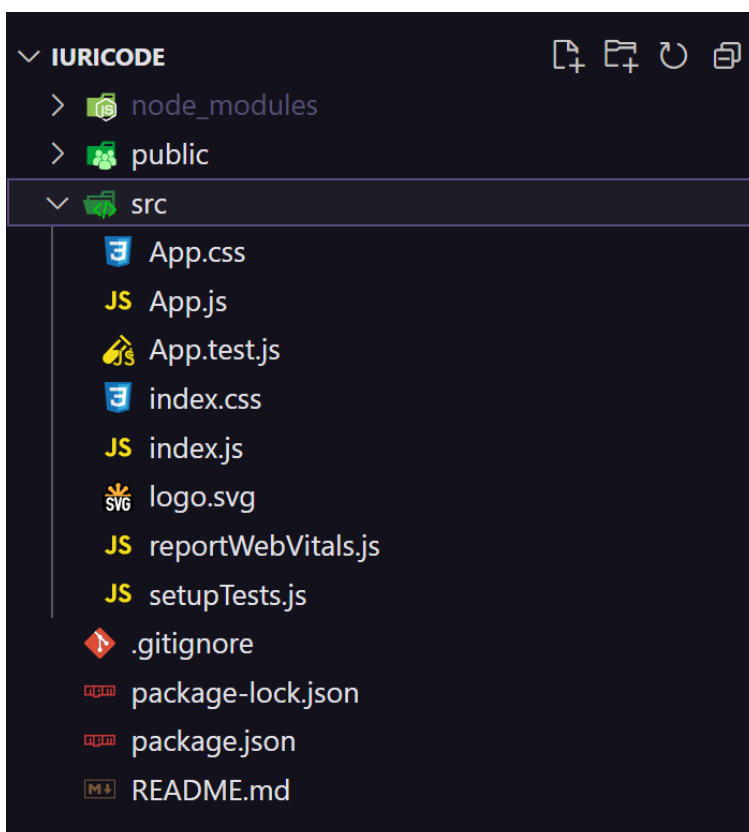
Estrutura da pasta

Está bem, mas o que exatamente o CRA criou para nós, Iuri? Como dito anteriormente, o CRA criou um conjunto de ferramentas e funcionalidades pré-configuradas para a gente iniciar nossa aplicação React. Na imagem a seguir iremos ver quais foram os conjuntos criados:



- **node_modules**: É o diretório criado pelo npm para rastrear cada pacote que você instala. Dependemos dele para que nossos pacotes sejam executados.
- **public**: que contém os recursos públicos da aplicação, como imagens. Além disso, temos um arquivo HTML que contém o id "root" . É neste id que nossa aplicação React será renderizada e vai ser exibida.
- **src**: Contém os arquivos do componente React que é o responsável por renderizar a aplicação toda.
- **.gitignore**: Aqui ficaram os arquivos ignorados na hora de subir no Git.
- **package-lock.json**: Ele descreve as características das dependências usadas na aplicação, como versões, sub-dependências, links de verificação de integridade, dentre outras coisas.
- **package.json**: É um repositório central de configurações para ferramentas. Também é onde o npm armazena os nomes e versões de todos os pacotes instalados.
- **README.md**: É um arquivo que contém informações necessárias para entender o objetivo do projeto.

Dentro da pasta **src** temos alguns arquivos que não iremos utilizar na aplicação, mas antes deletá-los irei explicar sobre os mesmos pois é sempre bom saber para que servem.



- **App.js**: Esse arquivo contém todo o conteúdo que aparece em tela quando damos o comando npm start.
- **index.js**: Esse arquivo é feito a conexão do código React com o arquivo index.html que contém o id "root".
- **App.test.js**: É um arquivo de teste, ele é executado quando você roda o comando npm test.
- **App.css & index.css**: São arquivos de estilos CSS.
- **logo.svg**: Esse arquivo SVG é a logo do React.
- **reportWebVitals.js**: Esse arquivo permite medir o desempenho e a capacidade de resposta da sua aplicação.
- **setupTests.js**: Tudo o que ele possui são alguns métodos expect personalizados.

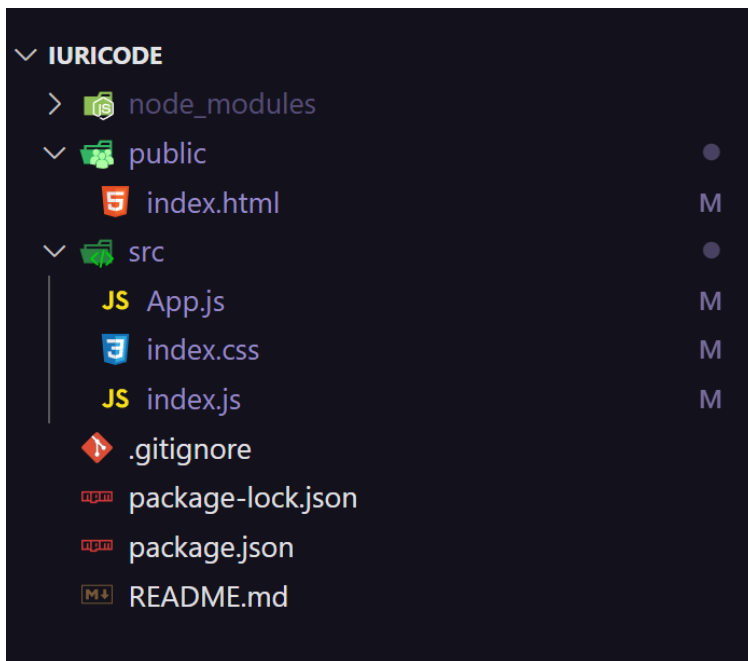
Geralmente deletamos os seguintes arquivos dentro da pasta src:

- App.css
- logo.svg
- reportWebVitals.js
- setupTests.js
- App.test.js

Além desses arquivos que não serão utilizado por nós, iremos limpar também os arquivos dentro da pasta public e as importações das imagens no index.html:

- favicon.ico
- logo192.png
- logo512.png
- manifest.json
- robots.txt

O resultado final da nossa aplicação será essa:



JSX

No topo do arquivo `index.js`, é importando o React, ReactDOM, `index.css`, App e `serviceWorker`. Ao importar o React, você está, na verdade, extraindo código para converter o JSX em JavaScript. JSX são os elementos similares ao HTML.

Mas o que é esse JSX, luri?

Essa é uma extensão de sintaxe especial e válida para o React, seu nome vem do JavaScript XML. Normalmente, mantemos o código em HTML, CSS e JavaScript em arquivos separados. Porém no React, isso funciona de um modo um pouco diferente.

Nos projetos em React, não criamos arquivos em HTML separados, pois o JSX nos permite escrever uma combinação de HTML e JavaScript em um mesmo arquivo. Você pode, no entanto, separar seu CSS em outro arquivo.

Como dito anteriormente, o JSX possui uma sintaxe muito semelhante ao HTML. O código abaixo demonstra claramente esta característica. Apesar de muito parecido, o código a seguir não é HTML e sim um trecho de código JSX.

```
import './App.css';
```



```
function App() {  
  return <h1>Sou um título</h1>;  
}  
  
export default App;
```

Para melhor o processo de desenvolvimento podemos mudar (ou criar) os arquivos com o formato .js para .jsx

Alguns editores de código, como Visual Studio Code, podem vir mais preparados.

Por exemplo: se você utilizar .jsx, pode ser que tenha vantagens no autocomplete das tags e também no highlight do código.

Dessa forma sempre é bom mudar de .js para .jsx.

Uma coisa muito importante sobre o JSX é que sempre que utilizamos mais de uma tag HTML no retorno da função, precisamos englobadas em uma tag pai. No exemplo abaixo é uma demonstração incorreta.

```
import './App.css';  
  
function App() {  
  return (  
    <h1>Sou um título</h1>  
    <p>Sou uma descrição</p>  
  );  
}  
  
export default App;
```

Agora iremos corrigir a função acima englobando as tags H1 e P dentro de uma Div.

```
import './App.css';
```

```
function App() {
  return (
    <div>
      <h1>Sou um título</h1>
      <p>Sou uma descrição</p>
    </div>
  );
}

export default App;
```

Porém nem sempre queremos que o retorno tenha uma tag pai englobando tudo. Para isso temos o Fragment do React, que nada menos é uma tag vazia/sem significado.

```
import "./App.css";

function App() {
  return (
    <>
      <h1>Sou um título</h1>
      <p>Sou uma descrição</p>
    </>
  );
}

export default App;
```

Você pode fazer dessa forma também (observe que importamos o React):

```
import React from "react";
import "./App.css";

function App() {
  return (
    <React.Fragment>
      <h1>Sou um título</h1>
    </React.Fragment>
  );
}
```

```
<p>Sou uma descrição</p>
</React.Fragment>
);
}

export default App;
```

Componentes

No exemplo anterior foi mostrado a utilização do JSX, o exemplo nada mais é do que um componente no React.

Ok, mas o que faz um componente?

Componentes permitem você dividir a UI em partes independentes, reutilizáveis, ou seja, trata cada parte da aplicação como um bloco isolado, livre de outras dependências externas. Componentes são como as funções JavaScript. Eles aceitam entradas e retornam elementos React que descrevem o que deve aparecer na tela.

E como isso é importante?

Vou criar um simples exemplo. Imagine que temos um card que representa a apresentação de uma notícia. Sua estrutura poderá se parecer como essa:

```
<article>
  

  <div>
    <h2>Título de um post</h2>
    <p>Olá, eu sou apenas uma pequena descrição sobre esse card.</p>
  </div>

  <a href="www.iuricode.com">Acessar post</a>
</article>
```

Até aqui está tudo bem. Mas imagine que queremos 6 (seis) cards iguais a esse. Normalmente com HTML iremos duplicar esse código em 6 (seis) vezes, correto?

Agora imagine que por algum motivo seja preciso adicionar uma tag span para sinalizar a data da postagem. Teríamos que mudar em todos! Pior ainda, imagine que existe este card em várias páginas em nossa aplicação. Concorde que iria demorar muito para dar manutenção neste código, por causa de uma simples alteração?

E como o React resolve isso?

Com os componentes!

Para criar um componente é bem simples, basta você criar um novo arquivo, exemplo, Card.jsx e criar a estrutura de um componente.

```
function Card() {  
  return (  
    <article>  
        
  
      <div>  
        <h2>Título de um post</h2>  
        <p>Olá, eu sou apenas uma pequena descrição sobre esse card.</p>  
      </div>  
  
      <a href="www.iuricode.com">Acessar post</a>  
    </article>  
  );  
}  
  
export default Card;
```

Após fazendo bastante chamar nosso arquivo criado no App.js, dessa forma:

```
import Card from './Card';  
  
function App() {  
  return (  
    <div>  
      <Card />  
      <Card />  
      <Card />  
    </div>  
  );  
}
```

```
</div>  
);  
}  
  
export default App;
```

Perceba que para chamar o componente Card foi preciso fazer o importe do mesmo. O que vale ser comentado aqui é que um componente sempre irá começar com a letra maiúscula, exemplo, Card, Footer, Menu... caso a primeira letra seja minúscula o React irá interpretar como se fosse uma tag HTML. Outra coisa importante é o './Card' na importação do componente, essa parte nada mais é o caminho onde se encontra o arquivo chamado. Nesse caso o arquivo se encontra na mesma raiz do arquivo que está chamando './' mas com o nome de Card, você pode importar dizendo o formato do arquivo, como './Card.tsx'.

Lembra da pasta public? No exemplo do Card tivemos o seguinte trecho de código:

```

```

Bem, tudo que tiver dentro da pasta public pode ser acessado apenas com o nome e formato do arquivo, diferente do HTML tradicional que teria que escrever o caminho das pasta até acessar o arquivo desejado.

Ok Iuri, mas se caso eu queira que cada componente do arquivo Card tenha um título diferente?

A resposta para isso é props!

Props

Outro conceito importante dos componentes é a forma como se comunicam. O React tem um objeto especial, chamado de prop (que significa propriedade), que usamos para transportar dados de um componente para o outro. Isso é, as props são valores personalizados e tornam os componentes mais dinâmicos.

Vamos utilizar o componente Card já mostrado para ver como passar dados com as props.

Primeiro, precisamos definir uma prop no componente Card e atribuir um valor a ela (isso no arquivo App.jsx).

```
import Card from "./Card";

function App() {
  return (
    <div>
      <Card title="React" />
      <Card title="Front-end" />
      <Card title="iuricode" />
    </div>
  );
}

export default App;
```

Como o componente Card é o elemento filho aqui, precisamos definir as props do seu pai (App.jsx), para podermos passar os valores e obter o resultado simplesmente acessando a prop "title".

```
function Card(props) {
  return (
    <article>
      

      <div>
        <h2>{props.title}</h2>
        <p>Olá, eu sou apenas uma pequena descrição sobre esse card.</p>
      </div>

      <a href="www.iuricode.com">Acessar post</a>
    </article>
  );
}

export default Card;
```

Dessa forma conseguimos utilizar o código do componente Card e apresentar dados de forma dinâmica.

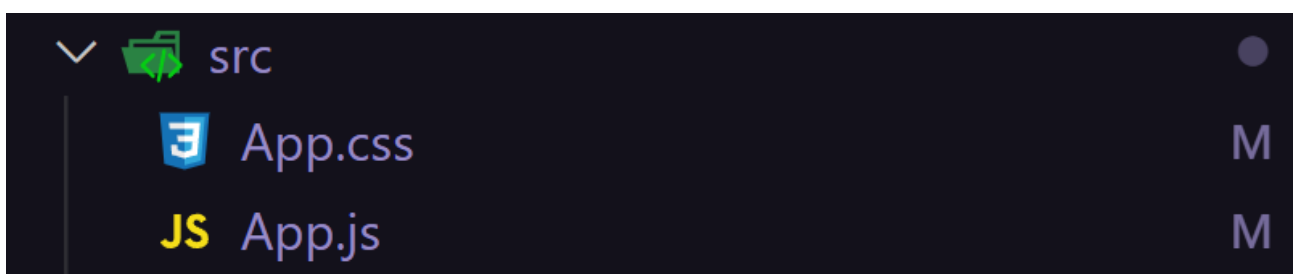
CSS Modules

Quando criamos nossa aplicação com CRA, foram criados dois arquivos CSS (index.css e App.css) que são responsáveis por adicionar estilos na aplicação toda. Porém existe um problema com os classNames ao adicionar estilos dessa forma que foi feito pelo CRA. O problema com os classNames ocorre da seguinte maneira. Imaginem um nome de className bastante utilizado por todos, no caso irei explicar utilizando o nome "title". Todos os lugares em que utilizamos o "title" precisamos criar um nome composto para que não haja globalidade entre os estilos, então caso formos utilizar dentro de um card, criaremos o "card-title", caso seja uma modal, será "modal-title", e cada vez ficará mais difícil para pensar num bom nome de className para cada componente que precise de um "title".

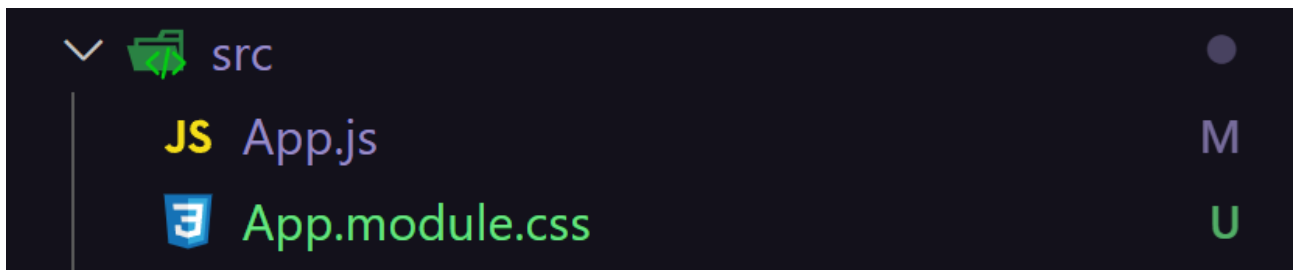
Porém temos a opção de criar estilos únicos para cada componente, utilizando os css-modules. Os css-modules são arquivos CSS em que os classNames são definidos localmente, isso significa que os estilos ali criados, só serão declarados dentro daquele escopo, e não globalmente, evitando conflitos entre estilos.

O primeiro passo é alterar o nome do seu arquivo de estilos para o seguinte formato: [name].module.css

Antes:



Depois:



Depois de fazer isso, deverá alterar seu import dos estilos e a utilização dos classNames para a seguinte forma.

Não utilizando CSS Modules:

```
import './App.css';

function App() {
  return <h1 className="title">Hello World</h1>;
}

export default App;
```

Utilizando CSS Modules:

```
import styles from './App.module.css';

function App() {
  return <h1 className={styles.title}>Hello World</h1>;
}

export default App;
```

Você deve estar se perguntando como que isso funciona por baixo dos panos. Bom, o css-modules cria um className único para cada local em que é utilizado.

Resultado final não utilizando CSS Modules:



Hello World

Resultado final utilizando CSS Modules:



Hello World

Eventos

Manipular eventos com elementos React é muito semelhante ao tratamento de eventos em elementos DOM. Existem algumas diferenças sintáticas para as quais devemos atentar:

- Os eventos React são nomeados usando camelCase, em vez de minúsculas;
- Com o JSX, você passa uma função como manipulador de eventos, em vez de uma string;

Por exemplo, o evento HTML definido abaixo:

```
<button onclick="ativarDesconto()">
  Ativar Desconto
</button>
```

Será definido da seguinte forma no React:

```
<button onClick="{ativarDesconto}">
  Ativar Desconto
</button>
```

Vamos criar um novo componente bem simples para mostrar como escutar eventos no React.

No App.jsx defina o seu código a seguir:

```
function App() {  
  const exibirAviso = () => {  
    alert("Oi!");  
  };  
  
  return <button onClick={exibirAviso}>Exibir aviso</button>;  
}  
  
export default App;
```

No exemplo acima, o atributo onClick é nosso manipulador de eventos e é adicionado ao elemento de destino para especificar a função a ser executada quando esse elemento for clicado.

O atributo onClick é definido para a função exibirAviso que exibe uma mensagem.

Assim, sempre que o botão for clicado, a função exibirAviso será chamada que, por sua vez, mostra a caixa de aviso.

Essa é uma forma bem simples de tratar eventos no React.

Hooks

Hooks foram adicionados ao React na versão 16.8.

Hooks permitem que componentes de função tenham acesso ao estado e outros recursos do React. Por causa disso, os componentes de classe geralmente não são mais necessários.

Embora Hooks geralmente substituam componentes de classe, não há planos para remover classes do React.

Mas o que são Hooks?

Basicamente, Hooks nos permitem "ligar" em recursos do React, como métodos de estado e ciclo de vida.

Aqui está um exemplo de um Hook. Não se preocupe se não fizer sentido. Já entrarei em mais detalhes.

useState

```
import React, { useState } from "react";
function App() {
  // Declaramos a nova variável de estado, que chamaremos de "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Você clicou {count} vezes</p>
      <button onClick={() => setCount(count + 1)}>Click aqui</button>
    </div>
  );
}
export default App;
```

Para usar o Hook useState (ou qualquer Hook), primeiro precisamos importá-lo em nosso componente.

```
import React, { useState } from "react";
```

Observe que estamos desestruturando o `useState` a partir do `React` pois é uma exportação nomeada.

Mas o que o `useState` está fazendo exatamente?

Aqui estamos usando o Hook `useState` para acompanhar o estado da aplicação.

`useState` aceita um estado inicial e retorna dois valores:

- O estado atual.
- Uma função que atualiza o estado.

O estado geralmente se refere aos dados ou propriedades da aplicação que precisam ser rastreados.

Depois de importar o `useState` do `React`. Inicialize o estado na parte superior do componente da função.

```
import React, { useState } from "react";

function App() {
  const [count, setCount] = useState(0);
}

export default App;
```

Observe que, novamente, estamos desestruturando os valores retornados de `useState`.

- O primeiro valor, `count`, é o nosso estado atual.
- O segundo valor, `setCount`, é a função que é usada para atualizar nosso estado.

Por fim, definimos o estado inicial como um número 0: `useState(0)`

Agora podemos incluir nosso estado em qualquer lugar em nosso componente.

E como irei atualizar o estado atual?

Para atualizar nosso estado, usamos nossa função de atualização de estado.

Nunca devemos atualizar o estado diretamente. Ex: `color = "red"` não é permitido.

Sempre iremos utilizar o segundo valor para adicionar o novo valor, e o primeiro valor para apresentar o valor atual. Exemplo:

Correto:

```
import React, { useState } from "react";

function App() {
  const [color, setColor] = useState("vermelho");

  return (
    <div>
      <h1>Minha cor favorita é {color}!</h1>
      <button type="button" onClick={() => setColor("azul")}>
        Mudar para azul
      </button>
    </div>
  );
}

export default App;
```

Errado (erro está no button):

```
import React, { useState } from "react";

function App() {
  const [color, setColor] = useState("vermelho");

  return (
    <div>
      <h1>Minha cor favorita é {color}!</h1>
      <button type="button" onClick={() => color("azul")}>
        Mudar para azul
      </button>
    </div>
  );
}

export default App;
```

```

    </button>
  </div>
);
}

export default App;

```

Obs: faça um teste para atender melhor a utilização do useState.

O Hook useState pode ser usado para rastrear strings, números, booleanos, arrays, objetos e qualquer combinação destes.

Poderíamos criar vários Hooks de estado para rastrear valores individuais.

```

import React, { useState } from "react";

function App() {
  const [brand, setBrand] = useState("Ford");
  const [model, setModel] = useState("Mustang");
  const [year, setYear] = useState("1964");
  const [color, setColor] = useState("red");
}

export default App;

```

Ou podemos usar apenas um estado e incluir um objeto!

```

import React, { useState } from "react";

function App() {
  const [car, setCar] = useState({
    brand: "Ford",
    model: "Mustang",
    year: "1964",
    color: "red",
  });
}

export default App;

```

useEffect

O `useEffect` é um Hook do React. Ele permite que você execute efeitos colaterais no seu código. Mas o que seriam esses efeitos colaterais? Buscar dados de uma API, mudar a DOM e cronômetros, por exemplo, são algumas opções de efeitos colaterais no seu código.

Na sua declaração, ele vai receber dois parâmetros, que funcionam assim:

```
useEffect(<Função>, <Dependência>)
```

Por fim, no trecho de código abaixo você vai ver o `useEffect` usado para mudar o título de uma página usando como base o estado da variável `'count'`. Note que, como dependência, o `useEffect` vai utilizar o `'count'` justamente porque o `'document.title'` precisa ser executado novamente toda vez que o `'count'` mudar de estado. Portanto, se não houvesse dependência, a função seria executada somente uma vez ao renderizar o componente.

```
import React, { useState, useEffect } from "react";

function App() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `Você clicou ${count} vezes`;
  });

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

export default App;
```

Claro que esses não são os únicos Hooks disponíveis no React. Abaixo deixa a lista dos principais Hook:

- useState
- useEffect
- useContext
- useRef
- useReducer
- useCallback
- useMemo

API

Agora que entendemos como funciona os Hooks useState e useEffect já podemos trazer um exemplo prático da sua utilização. Em nosso exemplo com os Hooks, teremos um componente que será responsável por listar os nossos repositórios do GitHub.

Iremos construir uma aplicação que lista os repositórios da nossa conta do GitHub, então precisarei acessar a API deles. Faremos uma requisição HTTP utilizando o fetch do JavaScript para buscar a lista de repositórios desta API. Vale ressaltar que o fetch gera efeitos colaterais em nosso código, pois ele é uma operação de input e output.

O Hook useEffect nos auxilia a lidar com os efeitos colaterais.

```
import React, { useEffect } from "react";

function App() {
  useEffect(() => {
    async function carregaRepositorios() {
      const resposta = await fetch(
        "https://api.github.com/users/iuricode/repos"
      );

      const repositorios = await resposta.json();
      return repositorios;
    }
  });
}
```



```

    carregaRepositorios();
  }, []);

  return <h1>Retorno da função</h1>;
}

export default App;

```

No exemplo acima, os efeitos colaterais são a chamada API. Este Hook recebe dois parâmetros: o primeiro é uma função que será executada quando o componente for inicializado e atualizado. Em nosso exemplo, este primeiro parâmetro é uma arrow function assíncrona, que faz uma requisição à API e guarda na const response, em formato de json, e, depois, na const repositorios. Já o segundo parâmetro indica em qual situação os efeitos colaterais irão modificar. No nosso caso, como queremos carregar a lista somente uma vez, passamos um array vazio [], pois ele garante a execução única.

Com os dados da API em mãos, agora só armazená-los em uma lista e depois exibi-los em tela. Para que possamos lidar com as mudanças de estado da nossa aplicação, iremos usar o hook useState. Para isso, usamos o hook desta forma:

```

import React, { useState, useEffect } from "react";

function App() {
  const [repositorio, setRepositorio] = useState([]);

  useEffect(() => {
    async function carregaRepositorios() {
      const resposta = await fetch(
        "https://api.github.com/users/iuricode/repos"
      );

      const repositorios = await resposta.json();
      return repositorios;
    }
  });
}

```

```

    carregaRepositorios();
  }, []);

  return <h1>Retorno da função</h1>;
}

export default App;

```

Como a nossa função `setRepositorio` é a responsável por alterar o estado de repositório, precisamos colocá-la dentro do escopo da função `useEffect`, porque ela é a responsável por pegar os dados que vão modificar o estado da nossa aplicação. Desta forma, nossa função `useEffect` fica assim:

```

import React, { useState, useEffect } from "react";

function App() {
  const [repositorio, setRepositorio] = useState([]);

  useEffect(() => {
    async function carregaRepositorios() {
      const response = await fetch(
        "https://api.github.com/users/julio-cesar96/repos"
      );

      const repositorios = await response.json();
      setRepositorio(repositorios);
    }
    carregaRepositorios();
  }, []);

  return <h1>Retorno da função</h1>;
}

export default App;

```

A diferença dentro dessa função é que retiramos o return repositorios e adicionamos a função setRepositorios(), passando como parâmetro a constante repositorios, pois, como foi explicado, é essa função responsável por atualizar os estado da nossa aplicação.

Agora que já temos os dados dos repositórios vindo da API do GitHub e estamos atualizando o estado da nossa aplicação com eles, o último passo é exibir ele de forma dinâmica dentro da tag ul desta forma.

```
import React, { useState, useEffect } from "react";

function App() {
  const [repositorio, setRepositorio] = useState([]);

  useEffect(() => {
    async function carregaRepositorios() {
      const response = await fetch(
        "https://api.github.com/users/iuricode/repos"
      );

      const repositorios = await response.json();
      setRepositorio(repositorios);
    }

    carregaRepositorios();
  }, []);

  return (
    <ul>
      {repositorio.map((repositorio) => (
        <li key={repositorio.id}>{repositorio.name}</li>
      ))}
    </ul>
  );
}
```

```
export default App;
```

Para que fosse possível gerar dinamicamente, utilizamos o método `map()` para poder percorrer o nosso array repositório, que possui a lista de repositórios, e imprimir um a um na tela.

Dessa forma você acabou de aprender a utilizar os conceitos básicos do React. Este ebook teve objetivo de ensinar aqueles que estão começando no mundo React, e também para lembrar aqueles que já têm conhecimento sobre essa tecnologia tão manda pelos devs.