

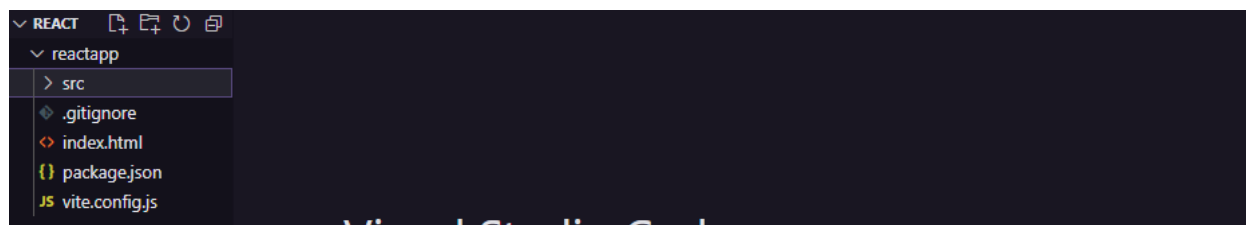
React JS

 [Stitches](#)

Fundamentos

Criar Projeto com Vite

```
npm create vite@latest nome --template react
```



Instalar as dependencias do projeto

```
npm install
```

Babel: [How to Setup Babel in Node.js](https://www.freecodecamp.org/learn/2021/react/2021-how-to-setup-babel-in-node-js) ([freecodecamp.org](https://www.freecodecamp.org/))

Eslint: [Setup ESLINT and PRETTIER in React app - DEV Community](#) 🏠👤

- `npm i -D eslint-import-resolver-typescript @typescript-eslint/eslint-plugin@`
- [Typescript and Eslint – Complete Intro to React v7](https://btholt.github.io/2019/04/27/typescript-and-eslint-complete-intro-to-react-v7/) (btholt.github.io)

Rodar o projeto

```
npm run dev
```

Estrutura do react

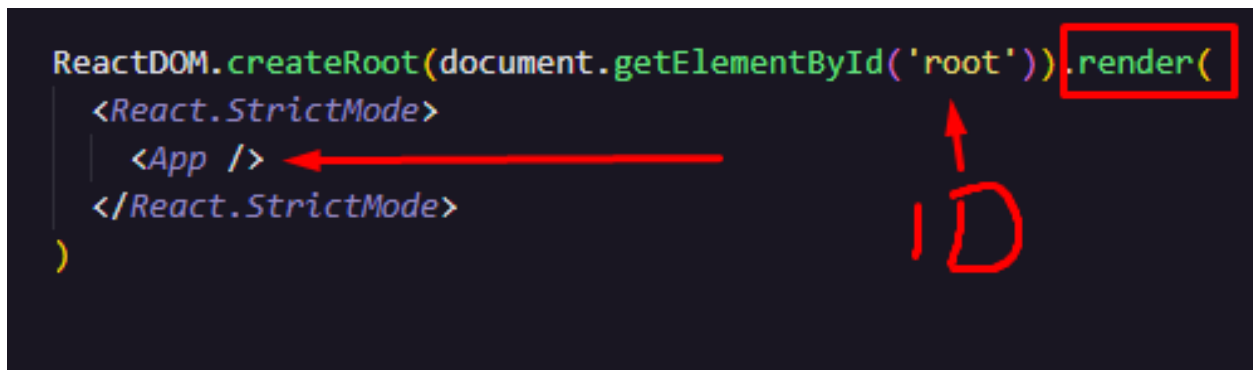
A página principal do nosso site possui um script tag com um arquivo chamado [main.jsx](#)

Esse main.jsx é um script que mistura JS e HTML. Dentro do main.jsx existe uma função para renderizar conteúdo.

Essa função faz parte da dependência ReactDOM, que é a que controla o documento html.

O ReactDOM manipula o index.html

Ele tem uma função que recebe a div root e injeta o conteúdo da nossa aplicação nela.

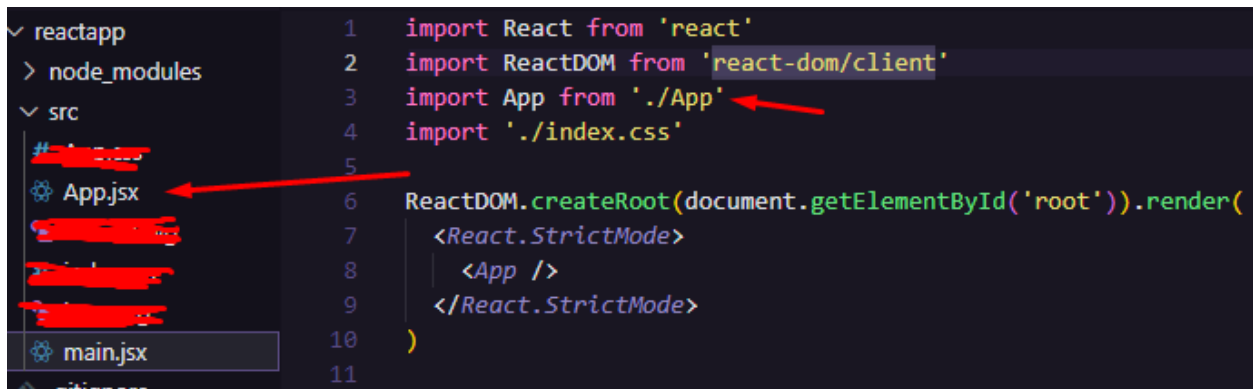


```
ReactDOM.createRoot(document.getElementById('root')).render(  
  <React.StrictMode>  
    <App />  
  </React.StrictMode>  
)
```

The image shows a code editor with the above code. A red box highlights the `render()` method. A red arrow points from the `<App />` tag to the `render()` method. A red circle with the letters 'ID' is drawn next to the `render()` method, with an arrow pointing to the `render()` method.

Veja que ele renderizou uma tag `<React.StrictMode>`. Que tem a tag `<App/>` dentro.

Essa tag APP é uma função de outro arquivo .jsx que é importado na main.



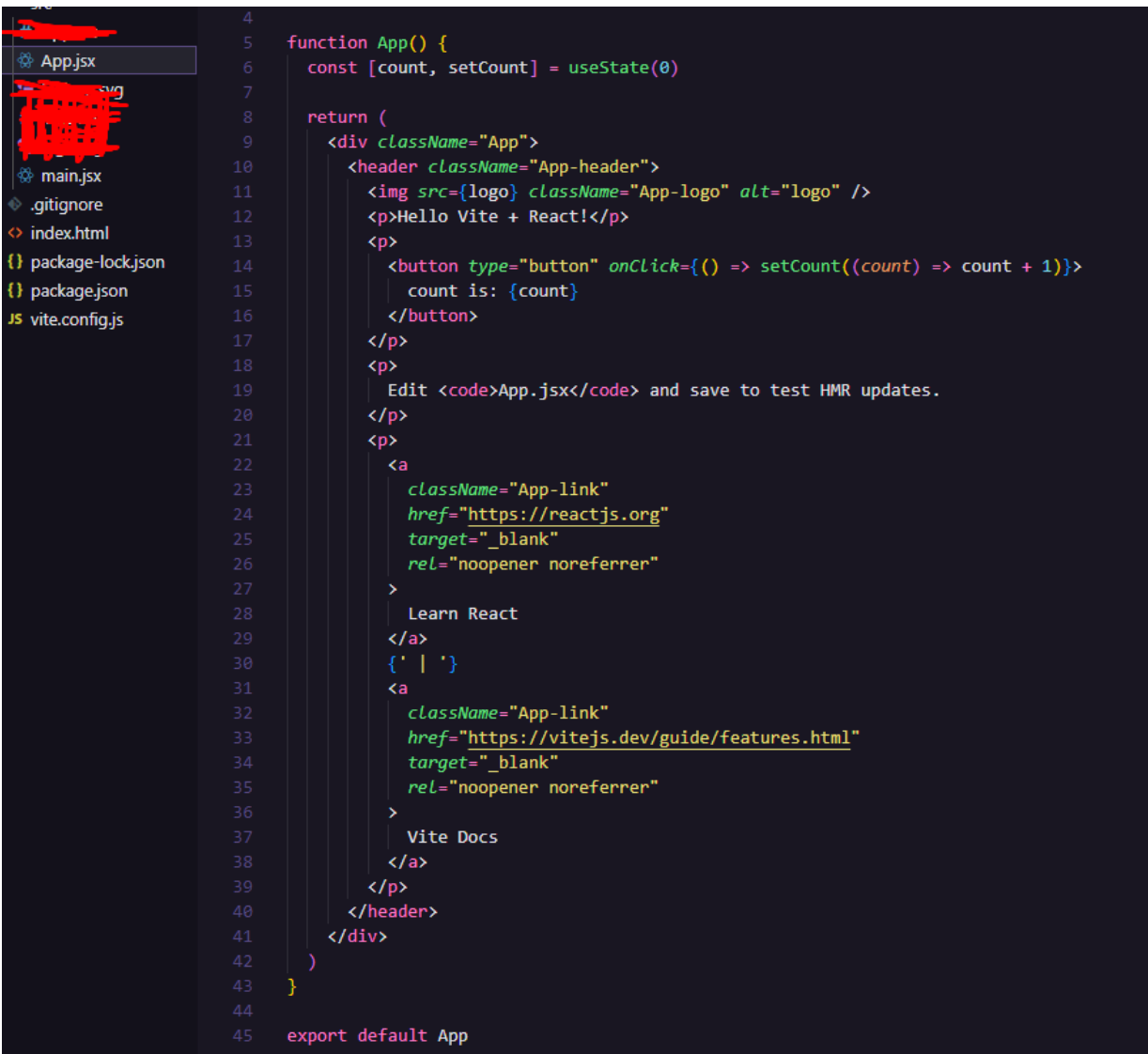
The image shows a code editor with a file explorer on the left and code in the main editor. The file explorer shows a project structure with `reactapp` folder containing `node_modules` and `src` folder. The `src` folder contains `App.jsx`, `main.jsx`, and `gitignore`. The `App.jsx` file is selected. The code in the main editor shows the following code:

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

A red arrow points from the `import App from './App'` line to the `App.jsx` file in the file explorer. Another red arrow points from the `<App />` tag in the code to the `App.jsx` file in the file explorer.

Esse arquivo App.jsx possui uma função chamada App que retorna um conteúdo em HTML.

Essa função é o chamado componente, e todo componente retorna um conteúdo HTML.



```
4
5 function App() {
6   const [count, setCount] = useState(0)
7
8   return (
9     <div className="App">
10      <header className="App-header">
11        <img src={logo} className="App-logo" alt="logo" />
12        <p>Hello Vite + React!</p>
13        <p>
14          <button type="button" onClick={() => setCount((count) => count + 1)}>
15            count is: {count}
16          </button>
17        </p>
18        <p>
19          Edit <code>App.jsx</code> and save to test HMR updates.
20        </p>
21        <p>
22          <a
23            className="App-link"
24            href="https://reactjs.org"
25            target="_blank"
26            rel="noopener noreferrer"
27          >
28            Learn React
29          </a>
30          { ' | ' }
31          <a
32            className="App-link"
33            href="https://vitejs.dev/guide/features.html"
34            target="_blank"
35            rel="noopener noreferrer"
36          >
37            Vite Docs
38          </a>
39        </p>
40      </header>
41    </div>
42  )
43 }
44
45 export default App
```

É possível trocar o nome desses components.

Criar uma pasta para os components é uma boa.

Regra do fragment

O retorno dessas funções só pode ser UMA tag, ou seja, **é necessário que todo o conteúdo esteja envelopado em uma tag só.**

Pode se usar tanto uma `<div>` pai quanto um fragment `<>`

Inserindo CSS.

Criar uma pasta para os styles.

No arquivo main.jsx `import './ styles/global.css'`

Nos arquivos do componente é da mesma maneira.

No react, `class=""` vira `className=""`

Lógica de exportação.

Nota-se que o `main.jsx` tem um `import`, se ele importa algo, então outra coisa está sendo exportada.

Cada arquivo `.jsx` além do main (o que não faz a renderização com o DOM) exporta uma função, e essa função retorna um código html.

Esse `.jsx` precisa explicitar que está exportando essa função. A função é um bloco do nosso UI

A melhor forma de exportar uma função é fazendo `export function Nome () {}`

Na hora do import é só dar `import { Nome } from diretorio do arquivo`

Para encaixar no código HTML, a função vira uma tag autocompleta: `<Nome />`

Pastas

Se eu tenho um componente, posso juntar o seu estilo e o `.jsx` em uma pasta só.

Nesse caso, é legal que o nome do `.jsx` seja `index.jsx`

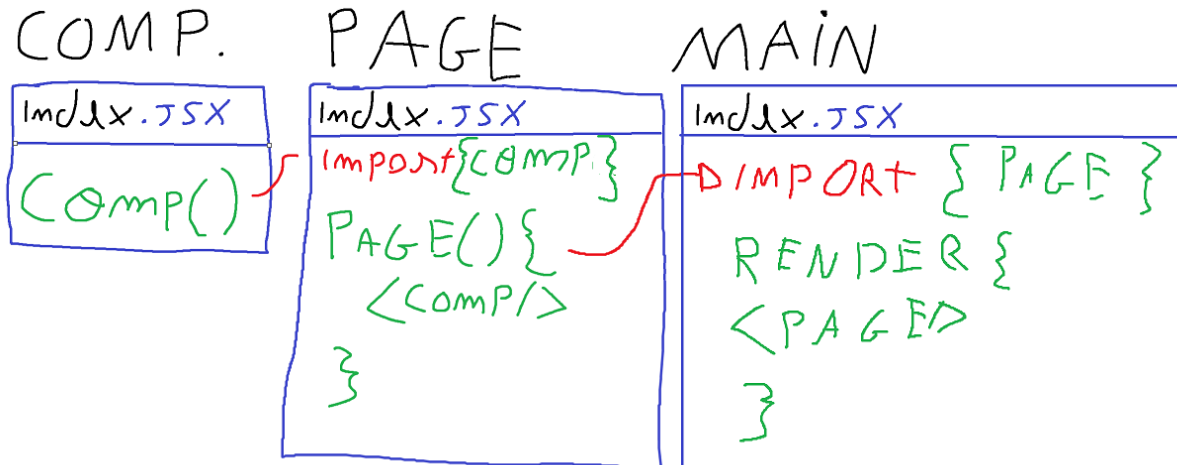
Dessa forma, ao exportar no `main.jsx`, não precisa declarar o nome do arquivo, apenas a pasta.

Exemplo.



Componentes

Componentes são pequenas partes da nossa UI que podem ser reutilizadas.



Componente > Página > Main

Vários componentes montam uma página

As diferentes páginas montam a aplicação main.

Componentes são criados da mesma forma que páginas são criadas, e eles são exportados pra dentro das páginas em vez da main.

Propriedades / Valores arbitrários → Valores dinamicos

Pode-se **preencher o conteúdo** do html **de um componente** se **passarmos propriedades** para ele na PAGE que ele está inserido.

Supondo que **Comp** está inserido em **Page**, ele é escrito no html de page como `<Comp/>`

Se adicionarmos atributos ao `<Comp/>` → `<Comp name="nome" data="10/04/2002" />`

Agora, lá no index.jsx do componente, em sua função, podemos passar variáveis que guardarão o valor dessas propriedades.

```

export function Comp(propriedades) {
  return (
    <>
      <div className="Comp">
        <span id="nome">{propriedades.name}</span>
        <span id="data">{propriedades.data}</span>
      </div>
    </>
  )
}

```

No lugar de propriedades, pode se passar também um conjunto `{}` com o nome das propriedades que nós inserimos lá na page → `Comp({nome, data})`

Pra passar uma variavel dentro do conteudo html, usa-se as chaves `{}`

UseStates → Guardando valores em tempo real.

useState é uma função do React a qual permite que nós mudemos o conteúdo da aplicação em tempo real baseado em nossa atividade.

Cria se um state da seguinte forma

```
const [ variavel , setVariavel ] = useState ( )
```

Variável age de forma referencial e **sempre guarda os valores enviados na função** `setVariavel()`

Dado um `<input>` queremos pegar o seu valor sempre que ele mudar

Input vai ter a propriedade `onChange={}` Que é uma propriedade em react que permite escrever um trecho de código javascript e executar alguma coisa.

Nesse caso, iremos pegar o valor atual do input e jogá-lo na variavel criada com o `useState()` por meio de sua função.

```
<input onChange ={evento => setVariavel (evento.target.value)}>
```

Principio da imutabilidade do useState.

O `useState` guarda uma variável, mas **se nós quisermos mudá-lo**, ele **não pode ser modificado, mas sim SUBSTITUÍDO**.

Ele é **imutável** pois **não tem como mudá-lo, somente trocá-lo** por um novo.

Dessa forma, se quisermos guardar o **valor de um useState antigo** dentro de um novo useState, **deve-se incluir esse prevState** na função `setVariavel` criada junto com ele.

```
setState(prevState => ...prevState, newState)
```

O novo state é igual ao espalhamento do state antigo mais o novo state, que é uma variável criada anteriormente por uma função.

Propriedade Key

Quando se cria componentes por meio de estruturas de repetição, é ideal que os itens dentro do iterável tenham uma propriedade chamada key, que é uma chave individual pra cada item. Isso melhora a performance do aplicativo.

Se meu useState for um array de usuarios, cada usuario tem que ter uma key.

UseEffect → Executar um comando sempre que um estado mudar ou um componente for criado.

useEffect é um hook que executa uma função quando seu componente é criado ou quando um ou mais states são alterados em tempo real.

Sintaxe: `useEffect (() => {arrow func} , [array com as variaveis states que queremos avaliar])`

useEffect é bem utilizados pra fazer consultas em apis ao iniciar um componente.

Cleanup Function useEffect

O useEffect pode retornar uma função, e essa função sempre é rodada antes do useEffect ser executado, ela pode ser utilizada para implementar mecanismos de espera quando um usuário estiver digitando ou fazendo ações repetitivas muito rápido, em vez de fazer vários requests pra cada uma dessas ações repetitivas, ele faz apenas uma com o estado final.

```
useEffect(() => {  
  timer = setTimeout(() => {  
    setOutput(name.toUpperCase());  
  }, 350);  
  
  return () => {  
    clearTimeout(timer);  
  };  
}, [name]);
```

UseEffect Assíncrono

o Use effect não aceita assincronia do tipo `async useEffect`. Para utilizar assincronismo no useEffect, devemos declarar a função callback do useEffect como uma `async function`.

```
useEffect(async function {  
  const img = await fetch(.....)  
  const bla bla = await fetch(.....)  
  
}, [nameState, passwordState])
```

FrontEnd Masters

Parcel, Prettier e Eslint

Pacotes úteis para bundling e organização respectivamente.

Hooks

useState

Não usar useStates dentro de condicionais ou loops.

Bom em inputs controlados.

useEffect

útil em chamadas de api

usar o `response.json()` para transformar a resposta da api para json

pode dar um **aviso** de eslint, recomendado é ignorar com `//eslint-disable-line react-hooks/exhaustive-deps` ao lado da linha.

Se não tiver nenhuma variável de dependencia no useEffect, então ele será executado sempre que qualquer coisa mudar. Se tiver

useRef

useRef é um hook que dá um valor a uma variável, mas esse valor segue o mesmo para todos os outros renders, sua posição de memória e valor é o mesmo. Isso é útil para se referir a elementos html dentro da pagina, muito usado para deletar modais junto com portais.

Custom Hooks

Custom hooks são funções criadas que fazem uma sequencia de useStates e useEffects dentro dela.

Criamos para poder reutilizá-las em várias partes do código.

Evita repetição de código.

Para criar um, é preciso criar outro arquivo .jsx ou .js e importar useEffect e useState.

Geralmente se usa um **objeto/dicionario localStorage**, que **evita uma nova chamada de api** se já tiver um dado carregado dentro dela.

Depois, escreve-se a `export function useCustomHook(variavel que muda ao chamar o hook)` e escrevemos os useStates, useEffects e requests.

useStates para armazenar variáveis

useEffect para chamar a api

Depois, o custom hook retorna itens, e diferente do useState e useEffect, o segundo não precisa ser uma função, pode ser uma variável.

Geralmente essa variável é de loading.

useDebugValue(valor)

Usado em customhooks, ele printa no console do react dev tools o valor dessa variavel, a fim de debugar o código.

PreventDefault em forms

Ao submeter um form, devemos usar o método preventDefault(); no evento, para que o formulário não faça o post do html e atualize a página.

```
onSubmit( (e) => { e.preventDefault(); } )
```

Criando vários componentes com arrays e Map.

Dada um array de dados, se você quiser criar vários componentes com cada um desses dados, usamos a função **map** do array.

Dentro do map, colocamos uma callback function que irá retornar o componente ou html que quisermos, com os dados do array inseridos.

```
export function App() {
  const lista = [1, 2, 3]

  return (
    <>
      { lista.map((item) => <h1>{item}</h1> }
        // 0
      </>
    )
  }
}
```

Essa função callback retorna um h1 com o nome de cada item.

Não se colocou o **return** pois está sendo usado um **retorno implícito**.

Se o **retorno ocupar mais de uma linha**, colocamos ele entre **parenteses ()**

NODE_ENV = DEVELOPMENT

O react quando tem essa propriedade definida como development, importa um package que auxilia o desenvolvedor mas deixa a aplicação mais lenta.

Na hora de dar deploy na aplicação, é necessário que esse valor seja mudado para **production**
O Parcel faz isso automaticamente, não sei o vite.

StrictMode

```
import { StrictMode } from 'react'
```

Strict mode é um componente que só funciona no node_env=development
Ajuda a prevenir erros no código e fazer com que o projeto seja futureproof

```
const App = () => {  
  return (  
    <StrictMode>  
      <div>  
        <h1>Adopt Me! </h1>  
        <SearchParams />  
      </div>  
    </StrictMode>  
  );  
};
```

React Router

Usado para fazer transições de rotas sem que haja uma atualização de página, renderizando componentes diferentes para cada rota.

npm install react-router-dom@6.2.1 ou 6.3.0

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

Esses 3 objetos são componentes, onde

BrowserRouter contém Routes que contém Router

BrowserRouter > Routes > Route

Route tem os atributos

path= Recebe a rota com o queryParams ou variavel do tipo :var ou ?var

ex: `path=details/:id`

esse valor de id estará disponível dentro do componente alvo.

element= Componente o qual aquele route vai renderizar quando aparecer.

```
<BrowserRouter>  
  <Routes>
```

```
<Route path="/" element={ <Componente 1 /> } />
<Route path="/:id" element={ <Componente 2 /> } />
</Routes>
</BrowserRouter>
```

useParams

useParams() é um hook do react-router-dom que pega os parâmetros da URL, com isso, podemos inserir esses parâmetros na implementação do componente, para ele exibir informações de um determinado item, baseado no parâmetro passado na url.

React router Link

Link é uma tag/componente do react router que serve como tag ancora (tag que quando clicamos somos levados para uma url).

A diferença da tag link é que ela faz a transição sem dar refresh na página.

```
import { Link } from 'react-router-dom'
```

Agora, todo botão que redirecionar a url deve ser feito com a tag link.

Atributos <Link />

to= rota

ex: `<Link to="/details/${id}" />`

Class components

Outra maneira de se escrever componentes.

EM VEZ DE FUNÇÕES, SÃO CRIADAS CLASSES.

E não se pode usar Hooks em Class Components

Criação

Primeiro, importa-se a classe Components da biblioteca “react”

```
import { Component } from 'react'
```

Após, cria-se uma classe com o nome do componente que herda a classe Component.

```
class MyComponent extends Component {  
  }  
}
```

Construtor

Essa classe deve ter um **construtor**, que chama os atributos da classe **super**

```
class MyComponent extends Component {  
  constructor(props) {  
    super(props)  
  }  
}
```

Deve ter também um **state**, que é um atributo dessa classe.

```
class MyComponent extends Component {  
  constructor(props) {  
    super(props)  
    this.state = { loading: true }  
  }  
}
```

Nesse caso, state é um objeto que possui loading como true;

Render

A classe deve ter um método render, que é o que retorna o html.

```

class MyComponent extends Component {
  constructor(props) {
    super(props)

    this.state = { loading: true }
  }

  render() {
    if (this.state.loading === true) {
      return <h1>Loading...</h1>
    }
    else {
      return <h1> Hello World </h1>
    }
  }
}

```

State

`this.state` é uma variável/objeto dentro da classe que contém todas as variáveis de estado do componente.

Dentro de `this.state`, temos os estados do nosso componente.

Estados com métodos Lifetime Cycles.

Para fazer uma chamada assim que o componente for montado, como o `useEffect`, a classe já conta com métodos que podem ser chamados em estados de vida.

`async componentDidMount ()` é uma função que executa assim que o componente for montado, então é usado para fazer requests da api.

Dentro dessa função, pode-se definir variáveis dentro do objeto estado com a função `setState()`

Lembrando da regra da imutabilidade, então, se desejamos reciclar o estado anterior, deve-se usar o **spread operator dentro do setState** na forma `setState({ ...this.state })`

Existem outros métodos de lifetime cycle, eles executam assim que ocorrem certas coisas dentro do componente.

Mount

`componentDidMount()` é um método que executa seus comandos assim que o componente aparecer na tela. Ele é executado depois do render.

Abaixo está a sequência de métodos que um class component inicialmente executa.

1. `constructor()`
2. `getDerivedStateFromProps()`
3. `render()`
4. `componentDidMount()`

`componentDidUpdate`

Declarando classes de forma simples com Babel

O babel é um transpilador que ajuda a transformar um js mais simples no js necessário para a aplicação rodar.

Criar um arquivo chamado `.babelrc` e cole o seguinte snippet

```
{
  "plugins": ["@babel/plugin-proposal-class-properties"]
}
```

Depois instale o `npm install --save-dev @babel/plugin-proposal-class-properties` pelo nodejs

Agora, a declaração da classe fica muito mais simples.

```
export default class MyComponent extends Component {  
  constructor(props) {  
    super(props)  
    this.state = { loading: true }  
  }  
}
```



```
export default class MyComponent extends Component {  
  state = {loading: true}  
}
```

Error Boundaries React

Um **error boundary** é um **class component** o qual envelope um componente que pode ter um erro na sua execução.

Ele é declarado assim como outros, mas usa o state e alguns métodos de lifetime cycle de forma diferente.

`state = { haserror: false }` → Estado do componente, iniciando sem erros.

getDerivedStateFromError

Função a qual é executada assim que se verificar um erro no componente filho o qual o error boundary está envelopando.

```
static getDerivedStateFromError() {  
  setState( {haserror: true} )  
}
```

componentDidCatch

Função executada para mostrar o erro na tela

```
componentDidCatch(error, info) {  
  console.log(error, info)  
}
```

componentDidUpdate

Famoso useEffect do class component, executa assim que um estado muda e depois renderiza novamente a aplicação

```
componentDidUpdate() {  
  setState(state = { x: y } )  
}
```

Código do error boundary do Front End Masters

```
// mostly code from reactjs.org/docs/error-boundaries.html  
import { Component } from "react";  
import { Link, Navigate } from "react-router-dom";  
  
class ErrorBoundary extends Component {  
  state = { hasError: false, redirect: false };  
  static getDerivedStateFromError() {  
    return { hasError: true };  
  }  
  componentDidCatch(error, info) {  
    console.error("ErrorBoundary caught an error", error, info);  
  }  
  componentDidUpdate() {  
    if (this.state.hasError) {  
      setTimeout(() => this.setState({ redirect: true }), 5000);  
    }  
  }  
  render() {  
    if (this.state.redirect) {  
      return <Navigate to="/" />;  
    } else if (this.state.hasError) {  
      return (  
        <h2>  
          There was an error with this listing. <Link to="/">Click here</Link>{" "  
            to back to the home page or wait five seconds.  
        </h2>  
      );  
    }  
    return this.props.children;  
  }  
}
```

```
export default ErrorBoundary;
```


Veja que no fim, **se o `haserror` for true**, ele **renderiza uma mensagem de erro** com um link para a página home. **Se não, ele retorna o componente filho do error boundary.**

Estrutura do componente completo.

Temos um wrapper que é um componente funcional, pois era necessário pegar os parâmetros da página pra fazer o `details` funcionar. Isso é o que se chamou no curso de **higher order components**.

```
const WrappedDetails = () => {
  const params = useParams();
  return (
    <ErrorBoundary>
      <Details params={params} />
    </ErrorBoundary>
  );
};

export default WrappedDetails;
```

O **`ErrorBoundary`** envelopa o conteúdo de **`Details`**, que é o nosso componente objetivo, então ele isola o erro que pode acontecer no componente **`Details`** e, caso ocorra, executa os comandos definidos na sua declaração. Caso não, o retorno é o `props.children`, que é o próprio `details`.

Context → Similar ao Redux

Context é um hook que permite que um estado a NÍVEL DE APLICATIVO seja compartilhado entre todos os componentes, ou seja, se existe uma informação acima que afeta todos os outros componentes, ela deve ser jogada no contexto.

O context resolve o problema de colocar props em todos os componentes para uma informação global.

Exemplos

- Qual o plano de assinatura do cliente
- Qual o tema que ele está usando

Criando um context

```
import { createContext } from "react"

export const Contexto = createContext(["valor inicial", () => {}])
```

```
/* const Contexto = createContext("") também funciona, mas o método ensinado
mesmo foi o de cima. O lance é que essa formatação se assemelha a um setState,
e dessa forma é possível utilizar outras ferramentas. */

export default Contexto
```

Importando contexto

```
import { Contexto } from "../contexto.jsx"
```

Contexto é um componente, a função `createContext` transforma ele em um contexto.

No nosso arquivo **App.jsx**, **podemos envelopar o nosso aplicativo com o componente Contexto**, e assim, todos os componentes dentro de app podem consultar o contexto usando o hook **useContext**.

A questão é que, ainda é necessário delimitar um valor para o Contexto, é regra de utilização.

Então cria-se um **useState** que atribui um valor a uma variável **para que ela seja usada no value** do componente `<Contexto />`.

```
import { Contexto } from "../contexto.jsx"
import { useState } from "react"

export function App() {

  const [tema, setTema] = useState("dark")
  //Ou
  const tema = useState("dark") //Recomendado!
  //Ou
  const [tema] = useState("dark")

  return (
    <Contexto.Provider value={tema} > //Value recebe tema definido no useState
      <ComponenteFilho />
      <ComponenteFilho />
    </Contexto.Provider />
  )
}

export default App
```

Consultar e mudar o contexto dentro dos componentes.

```
import { Contexto } from "../contexto.jsx"
import { useContext } from "react"
```

```
export function Componente() {
  const [tema, setTema] = useContext(Contexto) // Leitura e escrita
  const [tema] = useContext(Contexto) // Leitura
  //Está sendo usada a desestruturação de objetos para pegar os dados

  return (
    <h1>{tema}</h1>
    <button onClick={() => setTema("novotema")}>
  )
}
```

Passo a passo criar contexto

1. Criar um arquivo **.jsx** para guardar o **Contexto**, importar `createContext` e declará-lo
2. Ir no nível de **App**, importar o componente **Contexto**, **criar uma variavel com useState** dando o valor inicial do contexto à ela.
3. Colocar o `<Contexto.Provider value={variavel do useState}>` em volta dos componentes que desejar.
4. **Dentro dos componentes que desejar**, **importar o componente Contexto e o `useContext()`** para consultar o valor do contexto.
5. **declarar uma variavel e uma função** para **consultar e alterar** o contexto com o `useContext(Contexto)`
 - a. Exemplo: `[tema, setTema] = useContext (Contexto)`

▼ ContextAPI no Typescript

Criar um context no Typescript é bem mais chatinho do que no Js.

createContext e **useContext**

createContext é uma função do react que cria um contexto. Esse contexto é como se fosse uma struct que recebe duas coisas, uma variável de informação e uma arrow function.

```
interface TokenContextType {
  token: string;
  setTokenValue: (value: string) => void;
}
```

O exemplo a seguir é implementando um context pra accessToken

Essa variável de informação pode receber outros tipos, incluindo tipos mais elaborados.

Essa estrutura é necessária pois precisamos de uma variável pra consultar e uma função que seta o valor dessa variável.

É necessário criar essa interface pois o typescript é muito chato e quer que você esclareça que tipo a função createContext vai retornar para a sua variável context.

Nesse sentido a primeira coisa a se fazer é:

```
import { createContext, useState, ReactNode } from "react";

interface TokenContextType {
  token: string;
  setTokenValue: (value: string) => void;
}

#####
### ALGUMA COISA ###
#####

const TokenContext = createContext<TokenContextType>({0 QUE DEVO COLOCAR AQUI??})
```

Agora surge a pergunta, o que colocar ali? Bom, é necessário colocar um valor padrão para o context, e ele tem que ser uma variável exatamente do tipo que vc definiu. O valor padrão geralmente não tem nada.

Assim, o código inteiro, até agora, fica assim:

```
import { createContext, useState, ReactNode } from "react";

interface TokenContextType {
  token: string;
  setTokenValue: (value: string) => void;
}

const defaultValue = {
  token: "",
  setTokenValue: () => {null}
}

const TokenContext = createContext<TokenContextType>(defaultValue)
```

Criando um provider

O provider é um componente que serve para entregar o contexto para seus componentes filhos, ele recebe um props chamado `{ children }` do tipo `ReactNode`.

Dentro desse provider setamos uma variável e uma função com useState, a variável guarda o valor do context e a função é chamada pela arrow function do contexto (meio confuso, mas lendo o código dá pra entender) a fim de mudar a variável.

Depois retorna-se a estrutura seguinte com value sendo a desestruturação do contexto. Coincidentemente, o useState que usamos casa com os itens passados no value do provider, o ideal é que os nomes da variável/função do provider sejam iguais ao do tipo definido TokenContextType, pra usar do par `nome:nome` no dicionário e deixá-lo enxuto.

```
export const AuthProvider = ({ children }: ContextProps) => {
  const [token, setToken] = useState(defaultValue.token);

  const setTokenValue = (value: string) => {
    setToken(value);
  };

  return <AuthContext.Provider value={{ token, setTokenValue }}>{children}</AuthContext.Provider>;
};
```

Código Completo

```
import { createContext, useState, ReactNode } from "react";

interface ContextProps {
  children: ReactNode;
}

interface TokenContextType {
  token: string;
  setTokenValue: (value: string) => void;
}

const defaultValue = {
  token: "",
  setTokenValue: () => {null}
};

const AuthContext = createContext<TokenContextType>(defaultValue);

export const AuthProvider = ({ children }: ContextProps) => {
  const [token, setToken] = useState(defaultValue.token);

  const setTokenValue = (value: string) => {
    setToken(value);
  };

  return <AuthContext.Provider value={{ token, setTokenValue }}>{children}</AuthContext.Provider>;
};

export default AuthContext;
```

Exemplo com estruturas de dados “complexa”

```

import { createContext, useState, ReactNode } from "react";

interface ContextProps {
  children: ReactNode;
}

interface dataStructure {
  nome: string;
  idade: number;
}

interface TokenContextType {
  value: dataStructure;
  setValue: (value: dataStructure) => void;
}

const defaultValue = {
  value: {
    nome: "",
    idade: 0,
  },
  setValue: () => {
    (null);
  },
};

const Context = createContext<TokenContextType>(defaultValue);

export const Provider = ({ children }: ContextProps) => {
  const [value, setToken] = useState(defaultValue.value);

  const setValue = ({ nome, idade }: dataStructure) => {
    setToken({ nome, idade });
  };

  return <AuthContext.Provider value={{ value, setTokenValue }}>{children}</AuthContext.Provider>;
};

export default AuthContext;

```

React Portals

Um react portal é um componente criado usando o `createPortal` que pode ser chamado dentro de outros componentes do react mas ele é renderizado no topo da árvore do arquivo html, fazendo com que fique acima das outras divs.

Nesse sentido, portais são úteis para renderizar conteúdos que precisam ficar lá em cima da árvore mas serem chamados no fundo dela. É um portal do fundo da árvore para o topo.

Modais são um excelente uso para portais.

Inserir implementação depois de praticar.

React Intermediário

Todos os hooks

useState

Já escrito

useEffect

Função que roda outra função callback sempre que um estado na sua lista de dependencias muda.

A lista de dependencias é o segundo parametro do useEffect:

- Nada na lista: Roda sempre que um estado mudar
- Lista vazia: Só roda uma vez.
- Estados especificos na lista: Roda somente quando estes estados mudarem.

A função callback de dentro do useEffect pode ter um retorno, esse retorno pode ser uma função que é executada assim que o componente é desmontado ou esse useEffect é rodado de novo.

Essa função é chamada cleanup function, e é útil para limpar coisas que o useEffect criou, como setInterval, setTimeouts etc.

useContext

Já escrito

Usado para informações de app level, para evitar prop drill.

useRef

useRef é um tipo de useState que faz uma variável conter o valor mais atual possível em todos os renders, além disso, se o objeto não mudar, ele continua sendo o mesmo em questão de espaço de memória.

```
const ref = useRef("valor")
```

Esse valor pode ser consultado e alterado com `ref.current`

ref tag html

Pode se atribuir um elemento html a uma variavel ref, isso faz com que esse elemento possa ser guardado nessa variável e ser manipulado com funções useEffect.

```
const refh1 = useRef()

useEffect(() => {
  refh1.focus()
}, [])

return <h1 ref={refh1}>Olá Mundo</h1>
```

useReducer

É um tipo de setState mais organizado e utilizado quando se existem muitos estados a serem guardados. Funciona parecido com o Redux, por isso o nome.

Uma vez que um componente possui muitos estados, é possível organizá-los em um objeto que possui diferentes estados.

O useReducer cria um objeto o qual cada atributo pode ser um estado diferente.

```
const [state, dispatch] = useReducer (reducer, {nome : "gabriel" })
```

State é o nosso objeto de estados, e nesse caso, `state.nome` é `"gabriel"`

dispatch é a nossa função parecida com o setState, essa variável recebe a função **reducer**, que nós montamos.

A função reducer é onde os estados são alterados.

Ela é uma função que muda o objeto state **dependendo de uma ação**. Essa ação pode ter um type e um payload.

action é um objeto, possui type que define qual estado será alterado, e payload, caso o estado precise ser preenchido com algum tipo de informação digitada pelo usuário.

É utilizado um switch case para definir que operação fazer dependendo da ação escrita.

```
const reducer = (state, action) => {
  switch (action.type) {
    case "mudarNome":
      return {...state, nome: "novoNome"}
    default:
      return null
  }
}
```

Ao clicar em um botão para mudar o estado, por exemplo, usamos, por exemplo.

```
onClick={ () => { dispatch( { type: "novoNome" }) } }
```


Isso chama a função `dispatch`, com o parametro `state` já subentendido e o `action` é desconstruído dentro da chamada.

Se quiséssemos passar dados pro `dispatch`, colocamos `type` e `payload` dentro do `dispatch`, sendo `payload` as informações/strings/numeros/dados/etc.

useMemo

`useMemo` é um hook o qual serve 100% para performance e otimização.

Dada uma variável que precisa chamar uma função muito cara computacionalmente, é interessante usar o `useMemo` para que ela só seja executada quando necessário.

Por exemplo, a função de fibonacci é muito cara para valores altos, seria péssimo executar ela toda vez que renderizar o componente (executar a função do componente).

Se o valor da sequência não alterar de um render pra outro, não é necessário fazer essa chamada novamente, é interessante que ele seja memorizado.

Podemos fazer com que o valor já computado seja guardado numa variável e seja executado depois somente quando uma dependencia de estados informada mude (geralmente a variável que está sendo passada como parâmetro da função).

```
const valorDaSequencia = useMemo (() => fibonacci(numero), [numero])
```

Parecido com o `useEffect`, o `useMemo` só será executado quando `numero` mudar, pois está no seu array de dependencia.

useCallback

A escrever

Validação e atualização de AccessToken com axios.

useLayoutEffect

É um `useEffect` utilizado para puxar dados precisos como as dimensões da tela, para criar animações ou coisas que movem na tela.

```
const LayoutEffectComponent = () => {
  const [width, setWidth] = useState(0);
  const [height, setHeight] = useState(0);
  const el = useRef();

  useEffect(() => {
    setWidth(el.current.clientWidth);
    setHeight(el.current.clientHeight);
  });
}
```

Experiencia

O vite não atualiza sozinho quando executa o projeto no wsl

Usa-se variáveis de ambiente no vite com `import.meta.env.VARIAVEL`, e essa variável só está disponível pro vite se ela tiver um prefixo `VITE_`

Esse import meta precisa de uma tipagem no typescript.

O nome da tipagem já é pre-definido, `ImportMeta` e ela tem que ser escrita no

`/src/vite-env.d.ts`

Exemplo:

```
/// <reference types="vite/client" />

interface ImportMetaEnv {
  readonly VITE_API_URL: string;
}

interface ImportMeta {
  readonly env: ImportMetaEnv;
}
```

Children

Um dos atributos dos props de um componente é o children, children é um atributo de props que representa todos o conteúdo filho de um componente.

Exemplo:

```
<Interface >
  <h1>Olá!</h1>
  <p>Como estão?</p>
</Interface />
```

Dentro do nosso componente `Interface` podemos nos referir ao `h1` e `p` por meio do `props.children`

```
export const Interface = (props) => {  
  return (  
    {props.children}  
  )  
}
```

Esse snippet faz com que interface renderize todos os *filhos* que estão dentro dele. É útil para a criação de modais, por exemplo.

Passar cookies para o back automaticamente pelo axios

É necessário adicionar um parâmetro nas opções do `axios.create`

```
export default axios.create({  
  baseURL: import.meta.env.VITE_API_URL,  
  withCredentials: true,  
});
```

Erro do axios withCredentials e o cors ao passar cookies do navegador para o back.

No cors options é necessário ativar uma opção no json.

```
credentials: true
```

```
const corsOptions: any = {  
  origin: (origin: any, callback: any) => {  
    if (whitelist.indexOf(origin) !== -1 || !origin) {  
      callback(null, true);  
    } else {  
      callback(new Error("Not allowed by CORS"));  
    }  
  },  
  optionsSuccessStatus: 200,  
  credentials: true,  
};  
  
export default corsOptions;
```

Navegador não registrava cookies httponly

Para resolver, coloca-se mais algumas opções na hora de enviar o cookie httponly para o front

```
res.cookie("jwt", refreshToken, {  
  httpOnly: true,  
  maxAge: 60 * 60 * 1000,  
  sameSite: "none",  
  secure: true,  
});
```