Anotações do Curso ReactJS da B7Web - Módulo 02

(Por: Daniel Teixeira Quadros)

Sumário

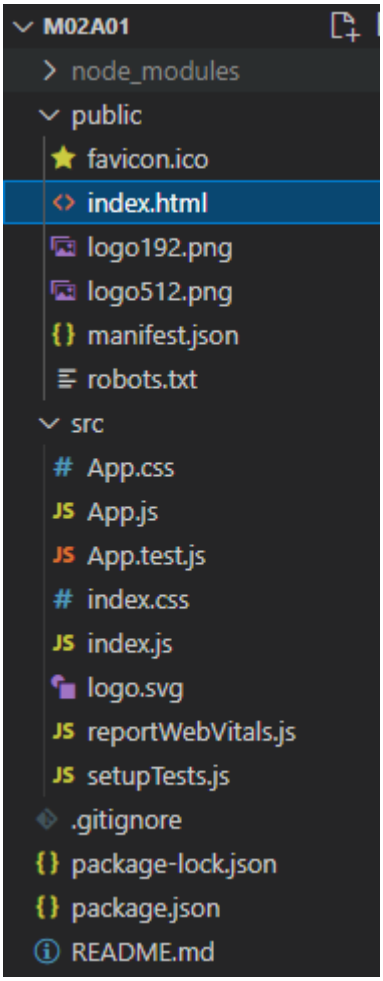
Aula 01 – Entendendo o React

Se, por algum motivo, for necessário parar o servidor do React com o **Ctrl+C** no terminal, para iniciá-lo novamente basta digitar o comando **npm start.**

```
PS D:\Informática\Programação\2021\Daniel\B7Web\ReactJS\modulo02\m02a01\m02a01> npm start
Compiled successfully!

You can now view m02a01 in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.3.3:3000

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

O React cria um servidor na máquina com o endereço **http://localhost:3000** que é utilizado para acessar o projeto a partir do navegador.

Ao abrirmos o projeto no navegador, o navegador procurará na pasta **public** pelo arquivo **index.html** e na pasta **src** pelo arquivo **index.js** :



Os arquivos index.html e index.js já vêm com um template pronto, inclusive com alguns itens desnecessários. Primeiramente, **vamos deixar o indexhtml e o index.js mais enxutos.**

É importante saber também que:
- A pasta **public** (pública) é a pasta que permite o acesso ao seu conteúdo externamente (pelo usuário).

- Já a pasta **src** (source → código fonte) não é acessível pelo usuário. É onde estará o nosso sistema. Primeiramente, no arquivo index.js vamos remover os seguintes itens:

Linha 3: `import './index.css';` Que importa o index.css → Arquivo CSS que configura a página de boas vindas do React(não precisamos dele).

Linha 5: `import reportWebVitals from './reportWebVitals';` Que importa (pelo npm) reportWebVitals.js que também não será utilizado

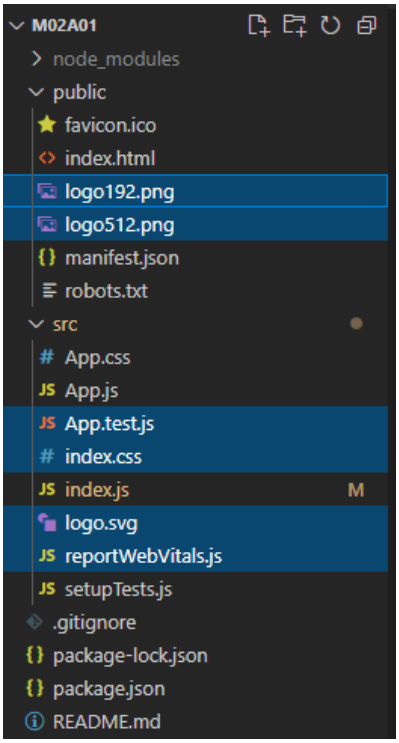
Removendo o reportWebVitals, podemos remover também as últimas linhas do index.js:

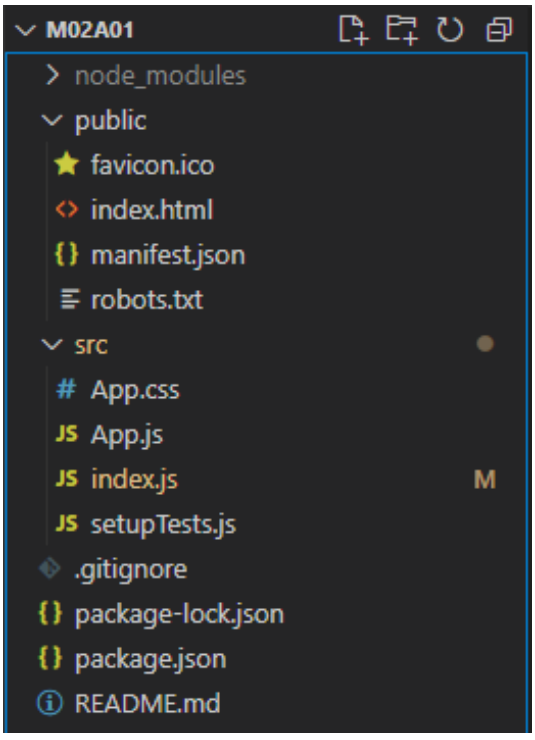
```
// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();
```

Nosso arquivo index.js iniciará assim:

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

Do nosso projeto, podemos excluir os arquivos:
Da pasta **public: logo192.png** e **logo512.png**
Da pasta **src: App.test.js**, **index.css**, **logo.svg** e **reportWebVitals.js**
Antes: Depois





Aula 02 – Classes e Funções

Para iniciar a aula apagamos todo o conteúdo do arquivo **App.js**

Obs.: Chamamos de **componente** todo o elemento que será exibido na tela. Inclusive, todo o sistema é um único e grande componente.

Vamos editar o App.js:

import React from 'react' → Importamos o React para o nosso sistema

Obs.: Depois de importarmos o react, criaremos os componentes. Após, para que o import App funcione do index.js, precisamos criar um export aqui.

Podemos criar componentes através de classes, funções ou variáveis.

Criação de componente utilizando classe:

```
class App extends React.Component {
  render(){
    return <h1>Testando 1,2,3...</h1>
  }
}
```

class App extends React.Component { → Criamos a classe App (Uma classe pode ter o nome do próprio arquivo, ou não).

extends React.Component → estende os componentes do React. Em outras palavras, estou declarando que a classe App é um componente.

Render(){ → função que renderiza (traduz, executa).

Criação de componente utilizando função:

```
function App(){
  return <h1>Testando 4,5,6...</h1>
}
```

Criação de componente utilizando variável:

Opção 1:

```
let App = () => {
  return <h1>Testando 7,8,9...</h1>
}
```

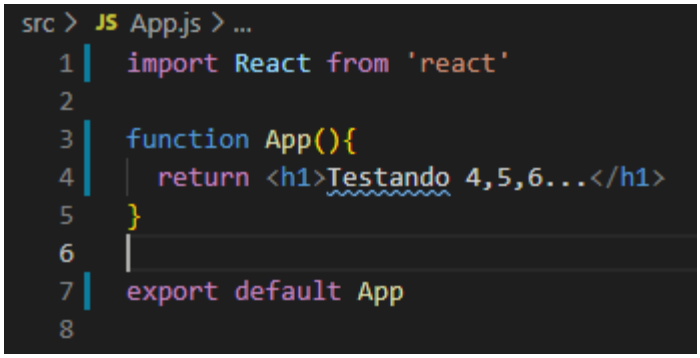
Opção 2:

```
let App = () => <h1>Testando 10,11,12...</h1>
```

Lembrando que função anônima com um retorno simples, pode ser declarada desta forma

Obs.: As formas mais utilizadas de criar um componente é utilizando uma função ou uma variável (Opção 1).

Terminamos a aula com o App.js assim:

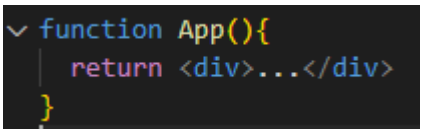


Aula 03 – Entendendo o JSX

É importante lembrar aqui que o JSX parece com HTML mas não é HTML.

1º Conceito: No JSX podemos retornar apenas 1 elemento, sempre. Exemplos

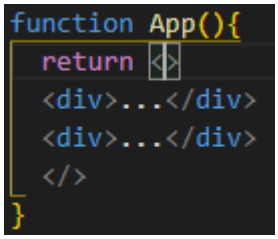
Apenas 1 elemento, funciona:



Dois elementos, não funciona:

Para retornarmos mais de um elemento, precisamos colocá-los, todos, dentro de um único elemento. Assim, funciona:

Podemos também, criar uma tag vazia, também funciona:



2º Conceito: Podemos colocar variáveis e valores dinâmicos dentro do JSX. Exemplos:

Exemplo 1: Podemos utilizar variáveis:

```
function App(){
  let nome = "Daniel"
  let idade = 40
  return <>
  <div>Meu nome é: {nome} e eu tenho {idade} anos</div>
  </>
}
```

Obs.: para imprimirmos estas variáveis juntamente com o texto, utilizamos o nome delas entre as chaves {}.

Exemplo 2: Podemos utilizar funções ou o que precisarmos em termos de javascript:

```
function formatarNome(usuario) {
  return usuario.nome+' '+usuario.sobrenome
}

function App(){
  let usuario = {
    nome: 'Daniel',
    sobrenome: 'Quadros'
  }
  return <>
  <div>Meu nome é: {formatarNome(usuario)}</div>
  </>
}
```

3º Conceito: Podemos utilizar atributos nas tags:

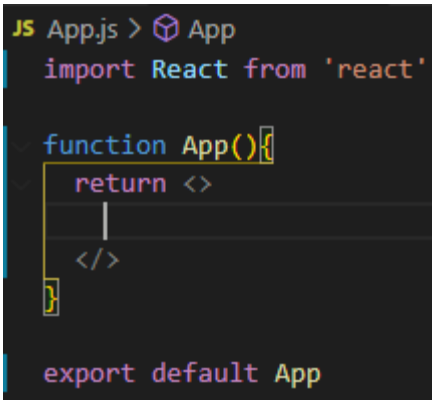
```
function App(){
  let image = 'https://www.google.com.br/google.jpg'

  return <>
    <img src={image}/>
  </>
}
```

Obs.: Como atributo de tag, não é utilizado a variável entre aspas.

Aula 04 – Componentes e Props

Iniciamos a aula 04 com o App.js assim:

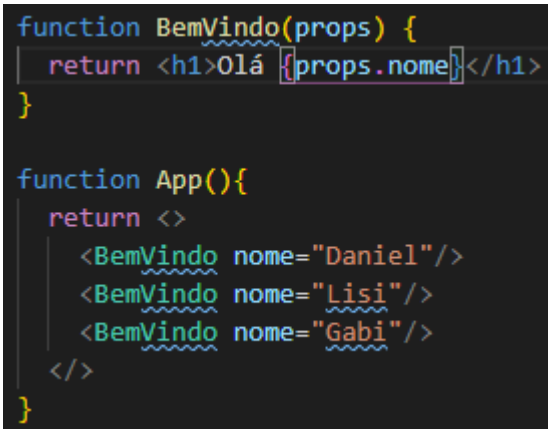
Como utilizamos componentes



Abaixo, criamos um componente utilizando uma function chamada BemVindo, e mais abaixo, no componente App, utilizamos o componente BemVindo como se fosse uma tag.

Props (properties → propriedades):

São propriedades que passamos juntamente com um componente, conforme o exemplo abaixo:



Adicionamos o parâmetro '**props**' no componente BemVindo e no return deste componente, entre chaves {},chamamos o parâmetro props com o nome da propriedade desejada **{props.nome}**

Obs.: Podemos passar quantas props quisermos:

```
function BemVindo(props) {
  return <h1>Olá {props.nome}, você tem {props.idade} anos!</h1>
}

function App(){
  return <>
    <BemVindo nome="Daniel" idade="40"/>
    <BemVindo nome="Lisi" idade="39"/>
    <BemVindo nome="Gabi" idade="7"/>
  </>
}
```

Vejamos a seguinte estrutura:

```
function Avatar(props) {
  return (
    <div>
      <img src={props.url} alt={props.name}/>
      <br/>
      <span>{props.name}</span>
    </div>
  )
}

function App(){
  let user = {
    url:"https://www.google.com.br/google.jpg",
    name:"Daniel Quadros"
  }
  return <>
    <Avatar url={user.url} name={user.name}/>
  </>
}
```

Neste exemplo acima,criamos o componente ‘Avatar’ e criamos uma estrutura no return. Aqui, passamos a utilizar estruturas como esta no return entre parênteses, apenas por questões de visualização.

No componente App, criamos um objeto ‘user’ com os itens ‘url’ e ‘name’ e no return, utilizamos o componente ‘Avatar’, com as propriedades ‘url e ‘name’ buscando essas informações do objeto ’user’.

Ainda, podemos passar para o componente ‘Avatar’, o objeto user por completo e só acessar os itens deste objeto diretamente dentro o componente ‘Avatar, veja o exemplo abaixo.

```
function Avatar(props) {
  return (
    <div>
      <img src={props.user.url} alt={props.user.name}/>
      <br/>
      <span>{props.user.name}</span>
    </div>
  )
}

function App(){
  let user = {
    url:"https://www.google.com.br/google.jpg",
    name:"Daniel Quadros"
  }
  return <>
    <Avatar user={user}/>
  </>
}
```

Aula 05 – Estilização com CSS e Stylesheet

Primeiramente veremos a estilização aplicando CSS diretamente em um componente, para isto, criaremos uma className chamada de ‘avatar’ em um componente e faremos alguns testes para entendermos o seu funcionamento:

```
function Avatar(props) {
  return (
    <div className="avatar">
      <img src={props.user.url} alt={props.user.name}/>
      <br/>
      <span>{props.user.name}</span>
    </div>
  )
}
```

Agora, no arquivo App.css, vamos apagar todo o conteúdo que vem neste arquivo, deixando ele vazio e criaremos uma estilização básica para a className avatar, apenas para teste:[

```
.avatar {
  border: 2px solid #000;
}
```

No App.js, para que seja reconhecido o App.css precisamos adicionar a seguinte linha:

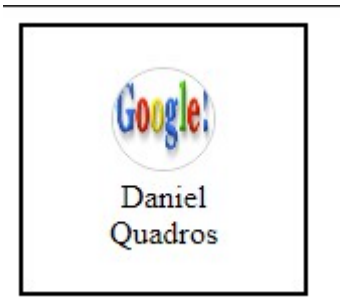
```
import './App.css'
```

Neste import, é importante perceber a sintaxe **‘./App.css’** esta é a sintaxe utilizada para a importação de um arquivo.

Realizamos outras estilizações, apenas para visualizar melhor este método:

```css
.avatar {
  border: 2px solid #000;
  text-align: center;
  padding: 20px;
  width: 100px;
}
.avatar img {
  width: 50px;
  height: 50px;
  border-radius: 50%;
  border: 1px solid #ccc;
}
```

E obtivemos o seguinte resultado no navegador:



Veremos agora outro método para estilizarmos a nossa aplicação que é a chamada **stylesheet**:

Primeiramente, removemos o import do App.css do arquivo App.js e o className avatar que havíamos criado anteriormente

prop style → Esta prop style se confunde com um style do CSS porque o resultado final é um style do CSS, mas ela não se comporta como um style do CSS (sintaxe é diferente).

```jsx
function Avatar(props) {
  return (
    <div style={{backgroundColor: '#f00', padding: 20, width: 150}}>
      <img src={props.user.url} alt={props.user.name}/>
      <br/>
      <span>{props.user.name}</span>
    </div>
  )
}
```

Utilizamos chaves **{ }** para colocarmos algo dentro do style e dentro das chaves colocamos um objeto. Desta forma, precisamos perceber que, nas propriedades do CSS que possuem um traço (-), por exemplo, não podemos utilizar este traço. Neste caso, devemos substituí-lo pela primeira letra da próxima palavra em maiúscula. Exemplo: backgroundColor.

Outra observação é que a sintaxe da configuração desejada, deve ser colocada entre aspas simples: backgroundColor: **'#f00'** ou se a configuração constar apenas de números, não colocamos entre aspas, mas é importante notar que não utilizamos a unidade de medida também. Ao declararmos um número, automaticamente, ele assume a unidade de medida como pixel (px): **padding: 20, width: 150.**

Obs. 1: Podemos utilizar os dois métodos simultaneamente, a estilização com o CSS e o stylesheet em conjunto.

Obs.: 2: Quando utilizamos stylesheet, precisamos aplicar a cada um dos elementos.

```
function Avatar(props) {
  return (
    <div style={{backgroundColor: '#f00', padding: 20, width: 150}}>
      <img style={{width:50, height:50}} src={props.user.url} alt={props.user.name}/>
      <br/>
      <span>{props.user.name}</span>
    </div>
  )
}
```

É importante conhecer essas duas formas, mas o interessante, por questões de organização visual, utilizarmos a estilização com CSS.

Aula 06 – Estilização com StyledComponents (1/3)

Para dar início a esta aula, mais uma vez, apagamos os componentes criados no App.js, deixando apenas o componente App com um return vazio:

```
import React from 'react'

function App(){
  return <>

  </>
}

export default App
```

Agora, iremos estilizar a nossa aplicação utilizando uma biblioteca externa chamada de **styledComponents** que é muito utilizada hoje em dia juntamente com o React.

Para isto, precisamos importar esta biblioteca utilizando o terminal:

- Crtl+C → Para parar o servidor e confirmamos com a tecla **Y**
- **npm install styled-components --save** executamos este comando no terminal que importará a biblioteca.

```
PS D:\Informática\Programação\2021\Daniel\B7Web\ReactJS\modulo02\m02a01\m02a01> npm install styled-components --save
npm WARN @babel/plugin-bugfix-v8-spread-parameters-in-optional-chaining@7.14.5 requires a peer of @babel/core@^7.13.0 but none is installed. You m
ust install peer dependencies yourself.
npm WARN tsutils@3.21.0 requires a peer of typescript@>=2.8.0 || >= 3.2.0-dev || >= 3.3.0-dev || >= 3.4.0-dev || >= 3.5.0-dev || >= 3.6.0-dev || >
= 3.6.0-beta || >= 3.7.0-dev || >= 3.7.0-beta but none is installed. You must install peer dependencies yourself.
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@2.3.2 (node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@2.3.2: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32"
,"arch":"x64"})
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.13 (node_modules\watchpack-chokidar2\node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.13: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32
","arch":"x64"})
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.13 (node_modules\webpack-dev-server\node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.13: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32
","arch":"x64"})

+ styled-components@5.3.0
added 12 packages from 10 contributors and audited 1942 packages in 40.459s

146 packages are looking for funding
  run `npm fund` for details

found 8 vulnerabilities (4 moderate, 4 high)
  run `npm audit fix` to fix them, or `npm audit` for details
PS D:\Informática\Programação\2021\Daniel\B7Web\ReactJS\modulo02\m02a01\m02a01>
```

Logo após a instalação, iniciamos mais uma vez o servidor: **npm start.**

Tudo que queremos utilizar no React, primeiramente precisamos importar, por isto, importaremos esta biblioteca também:

Normalmente chamamos este import de **styled** e é assim que vamos chamá-lo aqui. Sendo assim, adicionamos a seguinte linha no nosso App.js:

```
import styled from 'styled-components'
```

Para utilizarmos o styled é importante entender seu funcionamento.

Basicamente ele cria, em geral, um componente, e criamos este componente estilizando ele.

Normalmente utilizamos o styled para componentes que fazem parte da estrutura do nosso aplicativo.

Vejamos na prática para entendermos melhor:

```
import React from 'react'
import styled from 'styled-components'

const Title = styled.h1`

`
```

Aqui, criamos uma variável **'Title'**, e como é um componente, utilizamos a primeira letra em maiúsculo.
styled.h1 → Chamamos o styled e utilizamos a **tag html** que queremos criar.
`` → A estilização aqui vai entre crases.

Criado o componente, podemos utilizá-lo normalmente:

```
function App(){
  return <>
    <Title>Título de Teste</Title>
  </>
}
```

Agora podemos utilizar CSS normalmente no nosso componente criado:

```
const Title = styled.h1`
  color:#f00;
  font-size:18px;
`
```

Podemos ainda melhorar a estrutura do componente App, preparando para a criação de um componente que criaremos, o 'Site'

```
function App(){
  return (
    <Site>
      <Title>Título de Teste</Title>
    </Site>
  )
}
```

Criação do componente 'Site':

```
const Site = styled.div`
  width:400px;
  height:400px;
  background-color:#0f0;
`
```

Terminamos esta aula com o nosso App.js assim:

```
import React from 'react'
import styled from 'styled-components'

const Site = styled.div`
  width:400px;
  height:400px;
  background-color:#0f0;
`

const Title = styled.h1`
  color:#f00;
  font-size:18px;
`

const Botao = styled.button`
  font-size:19px;
  padding:10px 15px
`

function App(){
  return (
    <Site>
      <Title>Título de Teste</Title>
      <Botao>Clique aqui</Botao>
    </Site>
  )
}

export default App
```

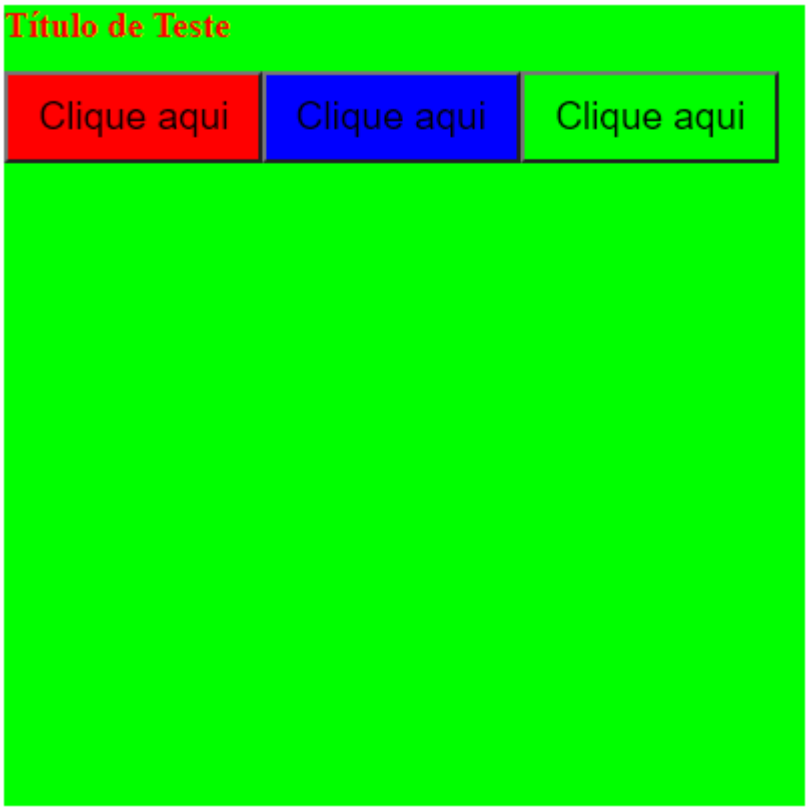
Aula 07 – Estilização com StyledComponents (2/3)

Iniciamos a aula adicionando mais dois componente ‘Botao’ no nosso componente ‘App’ para aprendermos como utilizar estilizações diferentes no mesmo componente ao utilizarmos ele em outro momento.

Para isto, criamos uma prop (color) para cada aplicação deste componente, neste caso o Botao, deixaremos o último Botao sem props, e na estilização (CSS) do componente, utilizamos esta prop.

```
const Botao = styled.button`
  font-size:19px;
  padding:10px 15px;
  background-color:${props => props.color || '#0f0'};
`

function App(){
  return (
    <Site>
      <Title>Título de Teste</Title>
      <Botao color="#f00">Clique aqui</Botao>
      <Botao color="#00f">Clique aqui</Botao>
      <Botao>Clique aqui</Botao>
    </Site>
  )
}
```

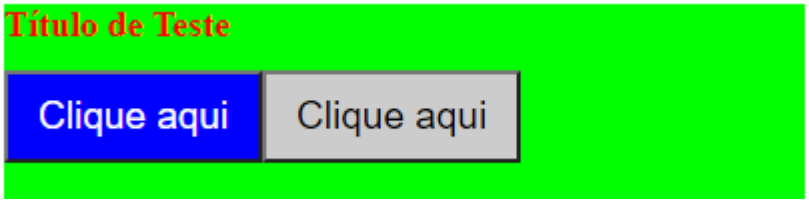
No CSS, o **${ }** permite que utilizemos Javascript no CSS e nesta aplicação, criaremos uma função anônima e chamamos o props.color. Utilizamos a opção **|| ‘#0f0’** para informar que em caso não seja encontrado um props chamado color, seja utilizado esta estilização. Temos o seguinte resultado no navegador:



Veremos agora, outra forma que é fazendo verificação:

```
const Botao = styled.button`
  font-size:19px;
  padding:10px 15px;
  background-color:${props => props.ativo == true ? '#00f' : '#ccc'};
  color:${props => props.ativo == true ? '#fff' : '#000'}
`

function App(){
  return (
    <Site>
      <Title>Título de Teste</Title>
      <Botao ativo={true}>Clique aqui</Botao>
      <Botao ativo={false}>Clique aqui</Botao>
    </Site>
  )
}
```

Utilizamos a qui a verificação de uma condição (true or false) e aplicamos a estilização de acordo com a condição recebida. Resultado:



Aula 08 – Estilização com StyledComponents (3/3)

Temos também como estender o uso do styled para outro componente. Vejamos:

Primeiramente, limpamos o nosso App.js deixando ele da seguinte maneira:

```
import React from 'react'
import styled from 'styled-components'

const Site = styled.div`
  width:400px;
  height:400px;
  background-color:#0f0;
`

const Botao = styled.button`
  font-size:19px;
  padding:10px 15px;
`

function App(){
  return (
    <Site>
      <Botao>Clique aqui</Botao>
      <Botao>Clique aqui</Botao>
    </Site>
  )
}

export default App
```

Adicionamos mais propriedades ao componente Botao:

```
const Botao = styled.button`
  font-size:19px;
  padding:10px 15px;
  border: 3px solid #f00;
  background-color:#fff;
  margin:5px;
  border-radius:5px;
`
```

Iremos criar agora outro componente que terá outras propriedades além das propriedades padrão que adicionamos ao componente Botao. Iremos aqui criar o que chamamos de **extensão:**

Utilizamos a seguinte sintaxe para utilizarmos esta extensão. Aqui no nossa aplicação, iremos fazer uma extensão do componente Botao e chamaremos este novo componente de BotaoPequeno:

```
const BotaoPequeno = styled(Botao)`
```

```
const Botao = styled.button`
  font-size:19px;
  padding:10px 15px;
  border: 3px dashed #f00;
  background-color:#fff;
  margin:5px;
  border-radius:5px;
`

const BotaoPequeno = styled(Botao)`
  padding:5px 10px;
  font-size:16px;
`

function App(){
  return (
    <Site>
      <Botao>Clique aqui</Botao>
      <BotaoPequeno>Clique aqui</BotaoPequeno>
    </Site>
  )
}
```

Nesta extensão, apenas adicionamos ou alteramos as propriedades que queremos que sejam diferentes do componente original, neste caso, o Botao.

Aula 09 – useState

Para iniciarmos a nossa aula, vamos limpar mais uma vez o App.js:

```
import React from 'react'
import styled from 'styled-components'

function App(){
  return (
    <>

    </>
  )
}

export default App
```

Até aqui, estávamos vendo formas de criar uma interface, a partir daqui, começaremos a ver algumas formas de modificar a nossa interface.

O useState nos permite realizar essas alterações na interface. Para visualizarmos este processo, criaremos um contador.

Para utilizarmos o useState precisamos importar juntamente com o React esta biblioteca, que é uma biblioteca interna do React.

```
import React, {useState} from 'react'
```

Obs.: Sempre que vermos a expressão ‘use’, geralmente se refere ao React hooks. (Mais a frente veremos mais conteúdos sobre o React Hooks).

Após importar o useState, criamos a seguinte estrutura:

```
import React, {useState} from 'react'
import styled from 'styled-components'

function App(){
  const [ contagem, setContagem ] = useState(0)
  const botaoAction = () => {
    setContagem(contagem + 1)
  }
  return (
    <>
      <div>{contagem} vazes</div>
      <button onClick={botaoAction}>Clique para aumentar</button>
    </>
  )
}

export default App
```

const [contagem, setContagem] = useState(0) → Nesta constante utilizamos uma estrutura semelhante a um array e neste array colocaremos duas informações:
contagem → O nome da variável.
setContagem (normalmente utilizamos a expressão 'set' e mais o nome da variável) → A função de modificar esta variável.
UseState(0) → Aqui, atribuímos o useState a esta variável e como parâmetro adicionamos o valor inicial para esta variável, neste caso, o número zero (0).

Desta forma, utilizando o setContagem, conseguimos modificar este valor inicial que está armazenado no variável contagem.

const botaoAction = () => { → Aqui, criamos a variável botaoAction com uma função anônima que se utilizará do setContagem para realizar as modificações de valores da variável contagem, que neste caso, somaremos mais 1 ao valor que estiver na variável
setContagem(contagem + 1) → Comando para somar + 1 ao valor da variável. Aqui, tentei utilizar o contagem ++, porém não funcionou.

Aula 10 – Campo de Input

Para iniciarmos a nossa aula, vamos limpar mais uma vez o App.js:

```
import React, {useState} from 'react'
import styled from 'styled-components'

function App(){

  return (
    <>

    </>
  )
}

export default App
```

Vamos criar um input utilizando a estrutura de componente, iniciamos com os códigos abaixo:

```
import React, {useState} from 'react'
import styled from 'styled-components'

const Input = styled.input`
  width:400px;
  height:30px;
  font-size:16px;
  padding:10px;
  border:1px solid #000;
`

function App(){

  return (
    <>
      <Input type="text"/>
    </>
  )
}

export default App
```

- Criamos o componente Input e utilizamos o styled.input nele
- Chamamos o Input no componente App

Agora, iremos criar a seguinte estrutura:

Criaremos uma state:

Iremos atrelar o valor do input à uma State, ou seja, o que tiver na State, terá no input e o que tiver no input terá na State

const [texto, setTexto] = useState('') → Criamos a nossa State, com os itens 'texto' e 'setTexto'. (Obs.: Sempre precisamos criar esses dois itens, mesmo que só utilizemos um deles.) Adicionamos o useState com uma string vazia.

<Input type="text" value={texto} onChange={handleInput}/> → Adicionamos o campo 'value' e atribuimos e ele o item 'texto'(O que tiver no state texto é o que terá no value do input)

onChange → É acionado quando adicionarmos alguma coisa no useState seja adicionado automaticamente aqui também.

={handleInput} → Função que o onChange chama e criaremos ela agora.

const handleInput = (e) => { → Criamos a variável handleInput e adicionamos um parâmetro que chamamos de 'e' (para se referir a um evento)

setTexto(e.target.value) → Adicionamos ao setTexto o valor que está sendo digitado.

Ficamos aqui com a seguinte estrutura:

```
function App(){
  const [texto, setTexto] = useState('')
  const handleInput = (e) => {
    setTexto(e.target.value)
  }

  return (
    <>
      <Input type="text" value={texto} onChange={handleInput}/>
    </>
  )
}
```

Adicionamos ainda no return do componente App a seguinte linha:

```
<p>{texto.length}</p>
```

Para mostrar na tela a quantidade de caracteres que estão sendo digitados.

Utilizamos esta mesma estrutura criamos um pequeno formulário para que o usuário digite seu e-mail, sua senha e ao clicar no botão, apareça um alerta mostrando o que foi digitado nos campos. Observe a estrutura:

```
function App(){
  const [email, setEmail] = useState('')
  const [password, setPassword] = useState('')

  const handleEmailInput = (e) => {
    setEmail(e.target.value)
  }
  const handlePasswordInput = (e) => {
    setPassword(e.target.value)
  }
  const handleButton = () => {
    alert(email+' - '+password)
  }

  return (
    <>
      <Input placeholder="Digite um E-mail" type="email" value={email} onChange={handleEmailInput}/>
      <Input placeholder="Digite sua senha" type="password" value={password} onChange={handlePasswordInput}/>
      <button onClick={handleButton}>Dizer</button>
    </>
  )
}
```

Ainda, se quisermos, podemos criar as funções anônimas que estão nas variáveis **handleEmailInput** e **handlePasswordInput** diretamente no **onChange** evitando assim de criar duas variáveis, conforme o exemplo abaixo:

```
import React, {useState} from 'react'
import styled from 'styled-components'

const Input = styled.input`
  width:400px;
  height:30px;
  font-size:16px;
  padding:10px;
  border:1px solid #000;
`

function App(){
  const [email, setEmail] = useState('')
  const [password, setPassword] = useState('')

  const handleButton = () => {
    alert(email+' - '+password)
  }

  return (
    <>
      <Input placeholder="Digite um E-mail" type="email" value={email} onChange={(e)=>setEmail(e.target.value)}/>
      <Input placeholder="Digite sua senha" type="password" value={password} onChange={(e)=>setPassword(e.target.value)}/>
      <button onClick={handleButton}>Dizer</button>
    </>
  )
}

export default App
```

Aula 11 – Condicional de Exibição

Para iniciarmos a aula, excluímos as linhas referentes ao password e ao button, iniciando com o App.js assim:

```
import React, {useState} from 'react'
import styled from 'styled-components'

const Input = styled.input`
  width:400px;
  height:30px;
  font-size:16px;
  padding:10px;
  border:1px solid #000;
`

function App(){
  const [email, setEmail] = useState('')

  return (
    <>
      <Input placeholder="Digite um E-mail" type="email" value={email} onChange={(e)=>setEmail(e.target.value)}/>
    </>
  )
}

export default App
```

Iremos aqui criar uma condição para que um campo apareça ou não, de acordo com a condição satisfeita ou não

Criamos um parágrafo que irá mostrar a quantidade de caracteres digitados no campo e-mail, porém, faremos com que se não houver nenhum caractere digitado (0 caracteres) o parágrafo não apareça na tela.

Para isto criaremos uma condicional:

```
    {email.length > 0 &&
      <p>{email.length} caracteres</p>
    }
```

Esta linha é similar a um **if** →
Se **email.length** > 0 (Se comprimento do campo e-mail for maior que 0)
&& → Aqui, os dois && não estão sendo usado como operador (**e**). Os dois && determinam o início da declaração do que irá aparecer se a condição anteriormente estabelecida for satisfeita.

Agora, ainda iremos criar uma outra condicional dentro desta. Para corrigir um erro de português.
Faremos com que o texto ‘caracteres’ só apareça no plural, caso a quantidade de caracteres digitados sejam maiores que 1, caso contrário, se for apenas 1 caractere digitado, deverá aparecer escrito ‘1 caractere’.

Para isto, fizemos a seguinte modificação no parágrafo:

```
<p>{email.length} caractere{email.length != 1 ? 's' : ''}
```

Removemos o s final da palavra caracteres e criamos uma condição que verifica a quantidade de caracteres e se a quantidade de caracteres for diferente de 1, será adicionado a letra ‘s’ no final da palavra caractere.

Veremos ainda uma terceira forma de condicional:

Criaremos aqui um botão que conterá a informação Fazer Login ou Sair, simulando um sistema de login de usuário.

Primeiramente, criamos uma variável no componente App:

```
const [isLogged, setIsLogged] = useState(false)
```

E abaixo, veremos duas maneiras de fazer este procedimento:

1ª Maneira:

```
{isLogged == true &&
  <button>Sair</button>
}
    {isLogged == false &&
  <button>Fazer Login</button>
}
```

2ª Maneira: (Mais simples)

```
{isLogged ? <button>Sair</button> : <button>Fazer Login</button>}
```

Aula 12 – Exercício: Calculadora de Gorjeta

```
import React, {useState} from 'react'
import styled from 'styled-components'

const Input = styled.input`
  width:400px;
  height:30px;
  font-size:16px;
  padding:10px;
  border:1px solid #000;
`
const Black = styled.p`
  font-weight: bolder;
  color: #000;
`


function App(){
  const [conta, setConta] = useState('0')
  const [gorjeta, setGorjeta] = useState('10')
```

```
  return (
    <>
      <h1>Calculadora de Gorjeta</h1>
      <p>Quanto deu a conta?</p>
      <Input type="number" value={conta} onChange={(e)=>setConta(parseFloat(e.target.value))}/>
      <p>Qual a porcentagem de gorjeta?</p>
      <Input type="number" value={gorjeta} onChange={(e)=>setGorjeta(parseFloat(e.target.value))}/>
      <hr/>
      {conta > 0  &&
        <>
          <p>Sub-total: R$ {conta.toFixed(2)}</p>
          <p>Gorjeta({gorjeta}%): R$ {((gorjeta/100) * conta).toFixed(2)}</p>
          <Black>Total: R$ {(conta + (gorjeta/100 * conta)).toFixed(2)}</Black>
        </>
      }
    </>
  )
}

export default App
```

Aula 13 – Ciclo de Vida (useEffect)

useEffect (effect hook) →

Para iniciarmos esta aula, vamos limpar novamente o nosso App.js.

No import do React, adicionamos o useEffect

```
import React, {useState, useEffect} from 'react'
```

E antes de utilizarmos o useEffect, vamos criar uma estrutura de contagem com um botão que ao clicarmos nele, é adicionado 1 a nossa contagem:

```
import React, {useState, useEffect} from 'react'
import styled from 'styled-components'

function App(){
  const [contagem, setContagem] = useState(0)
  const aumentarAction = () => {
    setContagem(contagem + 1)
  }
  return (
    <>
      <h1>Contagem: {contagem}</h1>
      <button onClick={aumentarAction}>Aumentar número</button>
    </>
  )
}

export default App
```

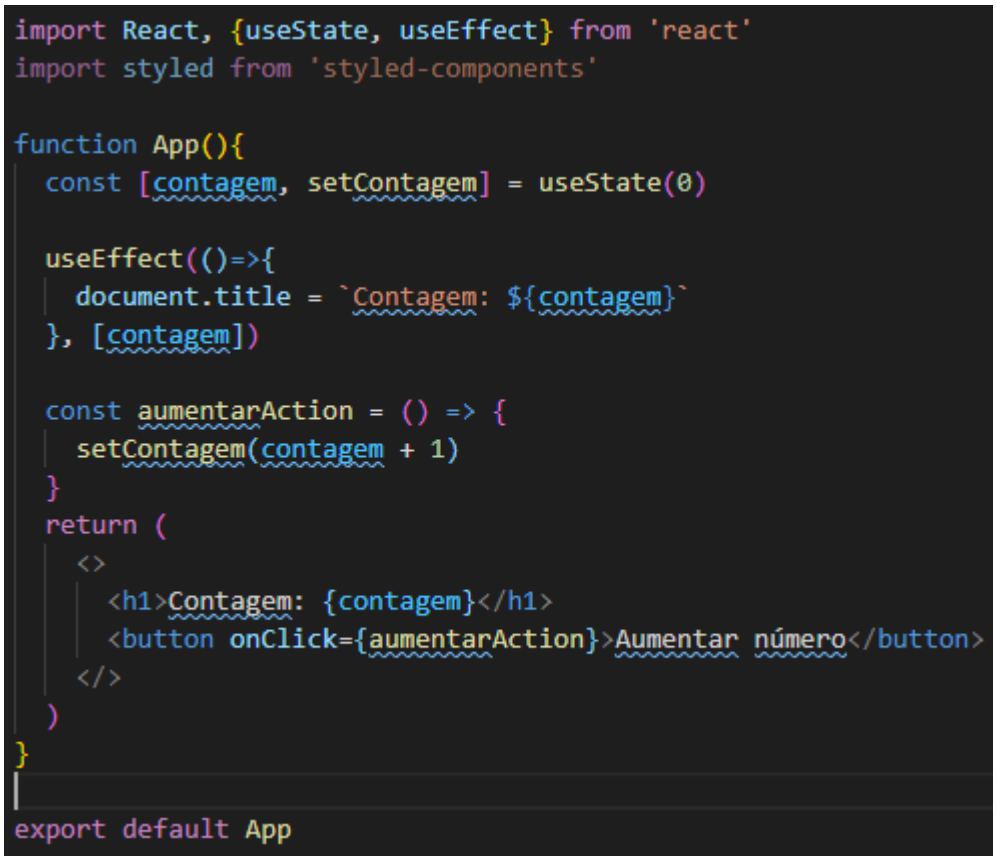
Agora iremos modificar a nossa estrutura, utilizando o useEffect de forma que consigamos alterar o título da página a cada vez que clicarmos no nosso botão Aumentar Número.

```
useEffect(()=>{
  document.title = `Contagem: ${contagem}`
}, [contagem])
```

useEffect é uma função que recebe dois parâmetros:

→ 1º Podemos colocar uma função anônima, por exêmplo. É a função que será executada quando alguma coisa acontecer.
→ 2º Recebe, geralmente, um array. É o observador. Ele fica observando as variáveis da lista deste array, e quando as variáveis desta lista for modificada ele rapidamente executa a função do primeiro parâmetro.

Terminamos a aula assim:



```
import React, {useState, useEffect} from 'react'
import styled from 'styled-components'

function App(){
  const [contagem, setContagem] = useState(0)

  useEffect(()=>{
    document.title = `Contagem: ${contagem}`
  }, [contagem])

  const aumentarAction = () => {
    setContagem(contagem + 1)
  }
  return (
    <>
      <h1>Contagem: {contagem}</h1>
      <button onClick={aumentarAction}>Aumentar número</button>
    </>
  )
}

export default App
```

Aula 14 – Separando em Componentes

Para iniciarmos esta aula, limpamos novamente o nosso App.js e começamos com a seguinte estrutura:

Criamos um título Lista de Tarefas

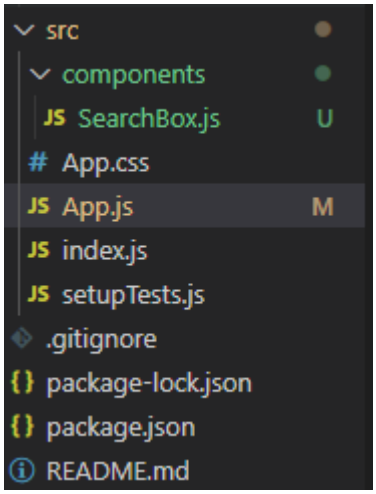
Abaixo do título Lista de Tarefas, criaremos um campo de busca. Porém, aqui, iremos criar este campo de busca para que ele possa ser reaproveitado em outras telas do nosso site e não apenas aqui. Para isto, criaremos ele em outro arquivo e sempre que quisermos utilizá-lo, bastará chamá-lo.

Aqui utilizamos a seguinte estrutura: Dentro da pasta src, criamos uma pasta chamada 'components' e dentro desta pasta, criamos o arquivo 'SearchBox.js'

Inicialmente ficamos com o App.js assim:

```jsx
import React, {useState, useEffect} from 'react'
import styled from 'styled-components'

function App(){

  return (
    <>
      <h1>Lista de Tarefas</h1>


    </>
  )
}

export default App
```

A estrutura criada para o arquivo SearchBox.js ficou assim:



E, criamos uma estrutura básica para o nosso SearchBox.js da seguinte maneira:

```jsx
import React, {useState} from 'react'
import styled from 'styled-components'

const InputText = styled.input`
    border: 2px solid #000;
    border-radios: 5px;
    padding: 15px;
    font-size: 17px;
    width: 300px;
`


function SearchBox(){

    return (
        <InputText/>
    )
}

export default SearchBox
```

Agora, voltamos para o App.js para utilizarmos o SearchBox.

Para isto, precisamos importar o SearchBox para cá:

```
import SearchBox from './components/SearchBox'
```

E podemos utilizá-lo normalmente:

```
return (
  <>
    <h1>Lista de Tarefas</h1>

    <SearchBox/>
  </>
)
```

Com esta conexão feita, podemos agora interagir entre os componentes. Vamos criar um placeholder para este input, e o texto deste placeholder será enviado do App.js para o componente SearchBox que irá tratar esta informação:

App.js:

```
<SearchBox frasePadrao="Faça sua busca..."/>
```

SearchBox.js adicionando o props:

```
function SearchBox(props){

    return (
        <InputText type="text" placeholder={props.frasePadrao}/>
    )
}
```

Ainda podemos criar uma condicional para este placeholder caso ele não receba nenhuma informação através da props:

```
<InputText
    type="text"
    placeholder={props.frasePadrao ?? "Digite alguma coisa"}/>
```

Terminamos a aula com o App.js assim:

```jsx
import React, {useState, useEffect} from 'react'
import styled from 'styled-components'
import SearchBox from './components/SearchBox'

function App(){

  return (
    <>
      <h1>Lista de Tarefas</h1>

      <SearchBox frasePadrao="Faça sua busca..."/>
      <SearchBox frasePadrao="Digite seu nome"/>
      <SearchBox/>
    </>
  )
}

export default App
```

Com o SearchBox.js assim:

```jsx
import React, {useState} from 'react'
import styled from 'styled-components'

const InputText = styled.input`
    border: 2px solid #000;
    border-radios: 5px;
    padding: 15px;
    font-size: 17px;
    width: 300px;
`

function SearchBox(props){

    return (
        <InputText
            type="text"
            placeholder={props.frasePadrao ?? "Digite alguma coisa"}/>
    )
}

export default SearchBox
```

E com o navegador assim:

Aula 15 – Trocando Dados Entre Componentes

Iniciamos esta aula deixando apenas um componente SearchBox na function App do App.js, assim:

```
function App(){

  return (
    <>
      <h1>Lista de Tarefas</h1>
      <SearchBox frasePadrao="Faça sua busca..."/>
    </>
  )
}
```

Agora, vamos para o SearchBox.js para Tronarmos este SearchBox um input funcional

`const [texto, setTexto] = useState('')` → Criamos aqui uma useState para o texto iniciando com uma String vazia.

```
<InputText
    type="text"
    value={texto}
    onChange={inputTextChange}
    placeholder={props.frasePadrao ?? "Digite alguma coisa"}/>
```

value={texto} → Associamos aqui a useState com o nosso input

onChange={inputTextChange} → Adicionamos este onChange para permitir a edição do 'value' e vinculamos com uma função chamada 'inputTextChange' que criaremos agora e que permitira a alteração deste valor

Acima do return com o InputText criamos a função **inputTextChange** que recebe o evento (e) como parâmetro e com o setTexto conseguimos trabalhar com a String:

```
function inputTextChange(e) {
    setTexto(e.target.value)
}
```

A partir daqui, o input está armazenando em 'texto' o que for digitado pelo usuário.

O componente SearchBox ficou assim:

```
function SearchBox(props){

    const [texto, setTexto] = useState('')

    function inputTextChange(e) {
        setTexto(e.target.value)
    }
    return (
        <InputText
            type="text"
            value={texto}
            onChange={inputTextChange}
            placeholder={props.frasePadrao ?? "Digite alguma coisa"}/>
    )
}
```

Após verificar que o processo está funcionando, reduzimos o componente eliminando a função 'inputTextChange' e criando uma função anônima diretamente no onChange do inputText, ficando com o componente assim:

```
function SearchBox(props){

    const [texto, setTexto] = useState('')

    return (
        <InputText
            type="text"
            value={texto}
            onChange={(e)=>setTexto(e.target.value)}
            placeholder={props.frasePadrao ?? "Digite alguma coisa"}/>
    )
}
```

Como próximo desafio, iremos exibir abaixo do input uma cópia do texto que o usuário estiver digitando. Para isto, voltamos para o App.js.

No SearchBox, criamos uma props chamada de onChangeText e passamos uma função que criaremos chamada 'handleSearchInput'. Ficamos com o componente App assim:

```
function App(){

  function handleSearchInput() {

  }
  return (
    <>
      <h1>Lista de Tarefas</h1>
      <SearchBox
        frasePadrao="Faça sua busca..."
        onChangeText={handleSearchInput}
      />
      <hr/>

      Texto procurado: ...

    </>
  )
}
```

Retornamos para o SearchBox. Adicionamos no **import** o **useEffect** porque agora iremos **monitorar 'texto'.**

Para isto, criamos o seguinte useEffect:

```
useEffect(()=>{
    props.onChangeText(texto)
}, [texto])
```

Que como primeiro parâmetro criamos uma função anônima que recebe a props 'onChangeText do SearchBox com o value texto e no segundo parâmetro indicamos a monitoração deste value.

Retornamos ao App.js.

Criamos uma useState para o texto que será exibido:

```
const [searchText, setSearchText] = useState('')
```

E vinculamos a edição deste searchText utilizando o setSearchText na functio handleSearchInput da seguinte forma:

```
function handleSearchInput(novoTexto) {
  setSearchText(novoTexto)
}
```

E para exibirmos na tela, apresentamos o searchText no return do componente App:

```
Texto procurado: {searchText}
```

Desta forma, conseguimos receber informações do usuário no SearchBox e Trazer de volta para a tela.

Agora Adicionaremos outro SearchBox no nosso App para fazermos o processo inverso.

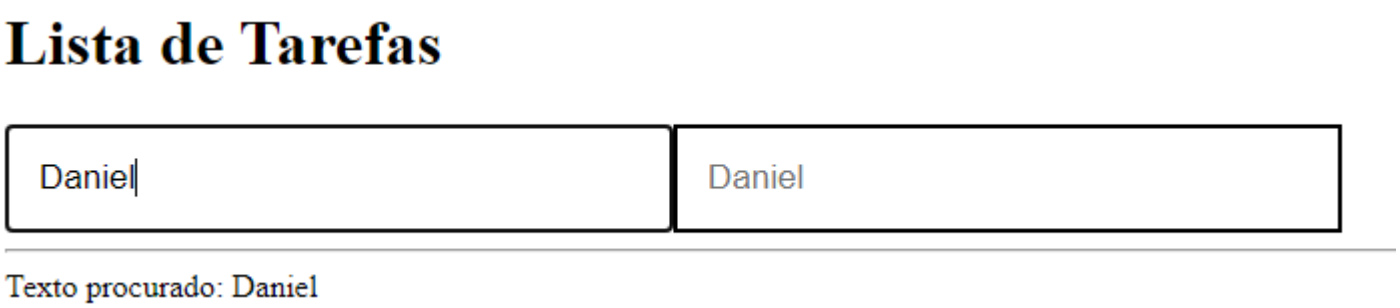
Neste segundo SearchBox iremos alterar o valor da props frasePadrao para {searchText} para receber o valor que estiver sendo digitado na tela e, excluiremos o onchangeText porque aqui, o usuário não irá editar nenhum valor neste input:

```
<SearchBox
  frasePadrao={searchText}
/>
```

No arquivo SearchBox.js, criaremos uma verificação de segurança no useEffect para os casos em que não exixtir a props onChangeText adicionando um if:

```
useEffect(()=>{
    if(props.onChangeText){
        props.onChangeText(texto)
    }
}, [texto])
```

Desta forma, o Segundo SearchBox recebe no placeholder o valor que for digitado no primeiro SearchBox:



Aula 16 – Exibindo Lista

Iniciamos a aula 16 excluindo o segundo campo de busca e o parágrafo 'Texto procurado: do App.js.

Vamos criar uma exibição de uma lista de tarefas. Para isto, vamos criar uma State para armazenar esta lista

```
const [list, setList] = useState([])
```

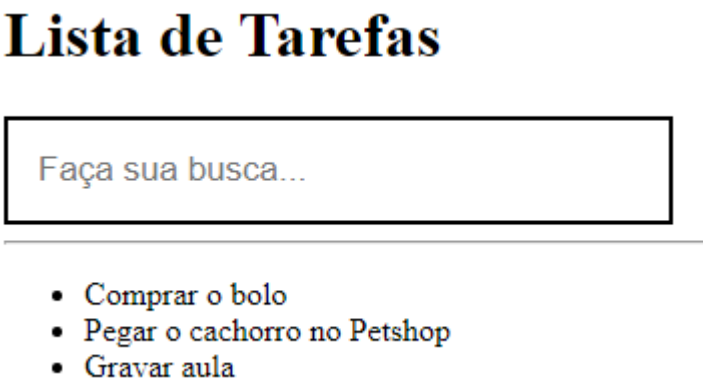
Após, criaremos uma useEffect com uma função anônima e que não irá observar ninguém (Irá ser executada uma única vez).
NO PRINT DO USEEFFECT ABAIXO, ENQUANTO ESTAVA NA AULA 17 PERCEBI A FALTA DO SEGUNDO PARÂMETRO. ESTE USEEFFECT DEVE TERMINAR ASSIM: `}, [])`E NÃO ASSIM: `})`
Nesta função anônima é que iremos colocar a nossa lista. Ela funcionará como se fosse a resposta à uma requisição a um banco de dados.

```
useEffect(()=>{

  setList([
    {title:'Comprar o bolo', done:false},
    {title:'Pegar o cachorro no Petshop', done:true},
    {title:'Gravar aula', done:false}
  ])

})
```

Agora, veremos como exibir o conteúdo desta lista na tela:
Abaixo do <hr> criamos a seguinte estrutura:

```
<ul>
  {list.map((item)=>{
    return(
      <li>{item.title}</li>
    )
  })}
</ul>
```

Criamos uma lista e dentro desta lista mapeamos (map) o array list ecom uma função anônima, retornamos todos os title dos objetos contidos neste array. Temos o seguinte resultado:



Podemos reduzir um pouquinho mais esta função anônima removendo o return:

```
<ul>
  {list.map((item)=>(
      <li>{item.title}</li>
    )
  )}
</ul>
```

Obs.: **MUITO IMPORTANTE** → Sempre que utilizarmos o **map** ele precisa ter uma **prop** chamada **key.** Essa prop key tem que receber uma String única. Cada item da minha lista tem que receber uma key única para que possamos identificá-los.

Se os objetos do meu banco de dados já possuírem uma chave única, podemos utilizá-los:

```
setList([
  {id:123, title:'Comprar o bolo', done:false},
  {id:124, title:'Pegar o cachorro no Petshop', done:true},
  {id:125, title:'Gravar aula', done:false}
])
```

Adicionamos o id no nosso setList para simularmos este id vindo de um banco de dados.

Precisamos lembrar que o nosso key para o map tem que ser uma string. Nesta estrutura, utilizamos o **toString**.

```
<ul>
  {list.map((item)=>(
      <li key={item.id.toString()}>{item.title}</li>
    )
  )}
</ul>
```

Porém, se não existir este id vindo do banco de dados, devemos criar um id para a key:

```
setList([
  {title:'Comprar o bolo', done:false},
  {title:'Pegar o cachorro no Petshop', done:true},
  {title:'Gravar aula', done:false}
])
```

Nesta estrutura, precisamos adicionar um segundo parâmetro na função anônima após o map chamado de index e adicionamos este index na key:

```
<ul>
  {list.map((item, index)=>(
    <li key={index}>{item.title}</li>
  ))}
</ul>
```

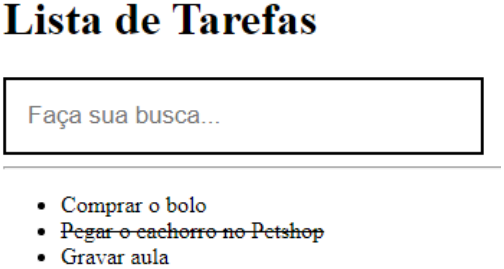
Obs.: É importante entender que esta estrutura de relação com o map e o key deve ser utilizado independentemente do componente que formos utilizar (sempre), mesmo que não fosse uma lista

Abaixo, adicionamos na nossa lista o done, porém transformamos o nosso boolean em uma String, apenas para visualizarmos na tela:

```
<ul>
  {list.map((item, index)=>(
    <li key={index}>{item.title} - {item.done.toString()}</li>
  ))}
</ul>
```

Abaixo, aprimoramos um pouco mais a estrutura de exibição da nossa lista criando uma condicional. Se o ítem da nossa lista estiver com o done true, iremos exibi-lo riscado, caso seja false, será exibido normalmente:

```
<ul>
  {list.map((item, index)=>(
    <li key={index}>
      {item.done &&
        <del>{item.title}</del>
      }
      {!item.done &&
        item.title
      }
      </li>
  ))}
</ul>
```



Terminamos a aula com a seguinte estrutura:

```
const [searchText, setSearchText] = useState('')
const [list, setList] = useState([])

useEffect(()=>{

  setList([
    {title:'Comprar o bolo', done:false},
    {title:'Pegar o cachorro no Petshop', done:true},
    {title:'Gravar aula', done:false}
  ])

})

function handleSearchInput(novoTexto) {
  setSearchText(novoTexto)
}
return (
  <>
    <h1>Lista de Tarefas</h1>
    <SearchBox
      frasePadrao="Faça sua busca..."
      onChangeText={handleSearchInput}
    />
    <hr/>
    <ul>
      {list.map((item, index)=>(
          <li key={index}>
            {item.done &&
              <del>{item.title}</del>
            }
            {!item.done &&
              item.title
            }
            </li>
        )
      )}
    </ul>
  </>
```

Aula 17 – Adicionando Novos Itens

Vamos alterar a função do nosso campo de busca para adicionar um item a lista. Para isto, iremos alterar o texto do placeholder deste campo para “Adicione um item” e iremos excluir o comando **onChangeText={handleSearchInput}** e a **function handleSearchInput** também

```
<h1>Lista de Tarefas</h1>
<SearchBox
  frasePadrao="Adicione um item"
/>
```

Agora, iremos trabalhar com o componente SearchBox.js para que ele entenda que quando algo for digitado no input e no momento em que for teclado o ENTER, ele receba o valor digitado na lista.

No SearchBox.js, no componente InputText adicionamos o comando onKeyUp e iremos adicionar uma função que criaremos logo acima chamada de handleKeyUp.

```
function handleKeyUp(e) {//Recebe o e(evento) do comando onKeyUp como parâmetro
    if(e.keyCode == 13) {//Confere se a tecla digitada foi o ENTER (13)
        if(props.onEnter){//Esta linha verifica a existência da props onEnter para evitar falhas no sistema
            props.onEnter(texto)//Envia o texto
            setTexto('')//Limpa o campo (Input)
        }
    }
}

return (
    <InputText
        type="text"
        value={texto}
        onChange={(e)=>setTexto(e.target.value)}
        onKeyUp={handleKeyUp}
        placeholder={props.frasePadrao ?? "Digite alguma coisa"}/>
)
```

O próximo passo, será adicionarmos a função onEnter no SearchBox lá no App.js e criaremos uma função chamada de addAction:

```
<SearchBox
  frasePadrao="Adicione um item"
  onEnter={addAction}
/>
```

Function addAction:

function addAction (newItem) { → recebe um novo item como parâmetro.

Para atualizarmos a nossa lista já existente, temos que dar dois passos e existem duas opções para fazermos este processo:

1ª opção:

criamos uma variável que chamaremos de newList que receberá uma cópia da minha lista já existente

```
let newList = list
```

Adicionamos o que foi digitado pelo usuário como título do novo item e a condição do done como false. Para isto utilizamos o push.
Depois de adicionado o novo item na cópia da lista, utilizamos o setList com o newList como parâmetro para atualizarmos a nossa lista original, agora contendo o novo item:

```
function addAction(newItem) {
  let newList = list
  newList.push({
    title:newItem,
    done:false
  })
  setList(newList)
}
```

Porém, esta estrutura que utilizamos acima, mesmo que funcione, não atualiza a nossa lista na tela, pois não é identificado que houve qualquer alteração. Para isto, a estrutura que realmente é a funcional é a que veremos abaixo:

→ **utilizaremos o spred do javascript**

```
function addAction(newItem) {
  let newList = [...list, {title:newItem, done:false}]
  setList(newList)
}
```

Na estrutura utilizando o spread, ele aproveita a lista existente e adiciona diretamente a ela o novo item e por último envia ao setList

2ª opção:

Utilizando o push →

```
function addAction(newItem) {
  let newList = [...list]
  newList.push({title:newItem, done:false})
  setList(newList)
}
```

Aula 18 – Marcando como feito

Podemos marcar o item como feito de basicamente duas formas, poderíamos criar um botão para isto ou podemos também, apenas clicar o item e ele será marcado ou desmarcado. Nesta aula veremos esta segunda opção:

Para isto, adicionaremos ao <li> o comando **onClick** com uma função anônima que criaremos com o nome de handleToggleDone enviando o index (para identificar o item específico que foi clicado) como parâmetro.

```
<ul>
  {list.map((item, index)=>(
      <li key={index} onClick={()=>handleToggleDone(index)}>
        {item.done &&
          <del>{item.title}</del>
        }
        {!item.done &&
          item.title
        }
        </li>
    )
  )}
</ul>
```

Estrutura da function handleToggleDone:

```
function handleToggleDone(index){// a function recebe o index como parâmetro
  let newList = [...list]//utilizamos o spread para adicionarmos a list em newList
  newList[index].done = !newList[index].done//inverte a condição do item done que for clicado
  setList(newList)//Atualiza a lista
}
```

Poderíamos também, se quiséssemos, utilizar um botão para marcar ou desmarcar. Embora, não utilizaremos esta estrutura, é interessante ver como seria:

```
<ul>
  {list.map((item, index)=>(
      <li key={index} >
        {item.done &&
          <del>{item.title}</del>
        }
        {!item.done &&
          item.title
        }
        <button onClick={()=>handleToggleDone(index)}>
          {item.done && 'Desfazer'}
          {!item.done && 'Fazer'}
        </button>
      </li>
    )
  )}
</ul>
```

Porém, vamos ficar com o o primeiro modelo clicando diretamente sobre o item e sem o botão:

```
<ul>
  {list.map((item, index)=>(
    <li key={index} onClick={()=>handleToggleDone(index)}>
      {item.done &&
        <del>{item.title}</del>
      }
      {!item.done &&
        item.title
      }
    </li>
  )
  )}
</ul>
```

Aula 19 – LocalStorage

Para iniciarmos esta aula, limpamos toda a nossa function App do App.js. Limpamos também os imports deixando apenas o import do React:

```
import React, {useState, useEffect} from 'react'

function App(){

  return (
    <>

    </>
  )
}

export default App
```

Abrimos o console no navegador da nossa página que está em branco e digitamos o seguinte comando:

```
> localStorage.setItem('name', 'Daniel')
< undefined
```

Como nós setamos este item no navegador, agora temos como pegar ele na nossa aplicação no VSCode. Para isto, utilizaremos uma useState:

```
const [name, setName] = useState(localStorage.getItem('name'))
```

A partir daqui, podemos utilizar esta informação na nossa aplicação:

```
return (
  <>
    Nome: {name}
  </>
```

Nome: Daniel

Desta forma conseguimos pegar um valor do localStorage.

E se quisermos setar um valor do localStorage, como fazemos?

Vamos criar um input

```
<input type="text" value={name} onChange={e=>setName(e.target.value)}/>
```

Neste exemplo acima, o nosso input começa com o localStorage e utilizamos o onChange para trocar o valor que estiver no value. Embora o campo esteja aceitando alterações no value, ele ainda não está trocando a informação que está no localStorage, pois se atualizarmos a página depois de editarmos o valor do input, o valor que aparecerá no input será aquele que adicionamos no localStorage originalmente.

Para alterarmos o valor do localStorage vamos utilizar um useEffect:

```
useEffect(()=>{
  localStorage.setItem('name', name)
},[name])
```

Neste useEffect criamos uma função anônima que altera o valor (setItem) do item 'name' do localStorage recebendo o valor que foi digitado pelo usuário no input e que recebemos através da nossa useState no item name. Além disto a nossa useEffect monitora o item name(`},[name])`), verificando e respondendo a qualquer alteração realizada neste input.

Esta é uma forma de armazenarmos um valor no localStorage. Terminamos a aula com o App.js assim:

```
import React, {useState, useEffect} from 'react'

function App(){

  const [name, setName] = useState(localStorage.getItem('name'))

  useEffect(()=>{
    localStorage.setItem('name', name)
  },[name])

  return (
    <>
      <input type="text" value={name} onChange={e=>setName(e.target.value)}/>
    </>
  )
}

export default App
```

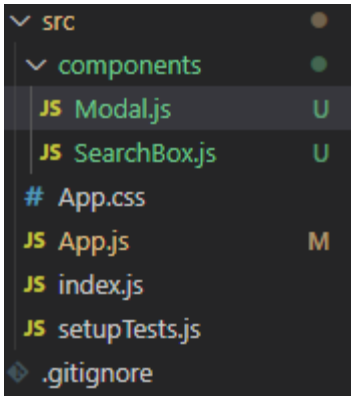
Aula 20 – Modal

Para iniciarmos esta aula, limpamos novamente a nossa function App:

```jsx
import React, {useState, useEffect} from 'react'

function App(){

  return (
    <>

    </>
  )
}

export default App
```

Para criarmos o nosso modal, poderíamos criá-lo diretamente no return, porém, criaremos ele em um componente separado, pois fica muito mais organizado e podemos reaproveitá-lo com mais facilidade se for necessário.

Para isto, dentro da nossa pasta components do nosso projeto, criaremos o arquivo **Modal.js**



Iniciamos o nosso Modal.js importando o React com o useState e o styled. Criamos a seguinte estrutura básica:

```jsx
import React, {useState} from 'react'
import styled from 'styled-components'

function Modal(){

    return (
        <>

        </>
    )
}

export default Modal
```

Voltamos para o nosso App.js e criamos o import do Modal.js:

```jsx
import React, {useState, useEffect} from 'react'
import Modal from './components/Modal'
```

E criamos a seguinte estrutura no return da function App:

```
return (
  <>
    <button>Abrir Modal</button>
    <Modal>
      <h1>Testando 1,2,3...</h1>
    </Modal>
  </>
)
```

Na function Modal do Modal.js, adicionamos uma props e acionamos ela com o comando props.children (filho):

```
function Modal(props){

    return (
        <>
            {props.children}
        </>
    )
}
```

Entendendo como colocamos o item dentro de um componente, agora, vamos fazer com que ele vire um Modal de verdade:

Para isto, vamos editar o Modal.js criando, primeiramente uma estrutura que irá gerar o efeito de escurecimento do fundo quando um modal é aberto:

```
import React, {useState} from 'react'
import styled from 'styled-components'

const ModalBackground = styled.div`
    position: fixed;
    left:0;
    top:0;
    right:0;
    bottom:0;
    z-index:90;
    background-color:rgba(0, 0, 0, 0.7);
`

function Modal(props){

    return (
        <ModalBackground>

        </ModalBackground>
    )
}

export default Modal
```

Agora, faremos o Modal efetivamente :

```jsx
function Modal(props){

    return (
        <ModalBackground>
            <ModalArea>
                {props.children}
            </ModalArea>
        </ModalBackground>
    )
}
```

E vamos criar o ModalArea;

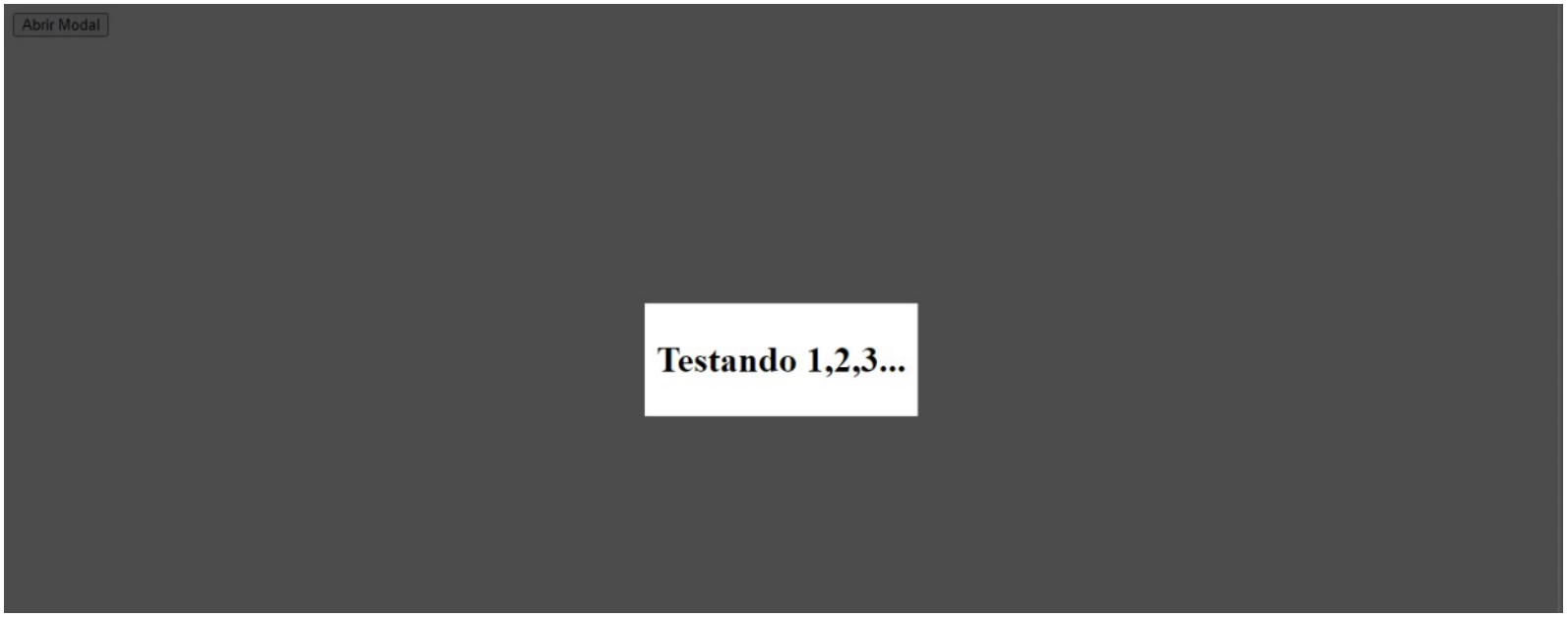
Para uma melhor estilização do ModalArea, vamos adicionar ao ModalBackground mais 3 propriedades do CSS: **display:flex; justify-content:center; e align-items:center:**

```jsx
const ModalBackground = styled.div`
    position:fixed;
    left:0;
    top:0;
    right:0;
    bottom:0;
    z-index:90;
    background-color:rgba(0, 0, 0, 0.7);
    display:flex;
    justify-content:center;
    align-items:center;
`
```

O nosso ModalArea, inicialmente será assim:

```jsx
const ModalArea = styled.div`
    background-color:#fff;
    padding:10px;
`
```

Temos o seguinte resultado no navegador:



Vamos configurar o modal, para que ele não seja exibido no momento em que a página for acessada. Um Modal, por padrão, só deve ser exibido quando for solicitado. Para isto, uma das opções que temos é a seguinte:

Criamos uma **props** que chamaremos de **visible** e atrelamos uma condição a ela. Se props.visible for true, o modal será exibido. Adicionamos esta estrutura dentro de um componente vazio (**<></>**).:

```
function Modal(props){

    return (
        <>
            {props.visible &&
                <ModalBackground>
                    <ModalArea>
                        {props.children}
                    </ModalArea>
                </ModalBackground>
            }
        </>
    )
}
```

Neste momento, no navegador, o nosso modal não está mais sendo exibido, pois ainda não definimos nada na nossa props visible.

Porém, se formos no nosso App.js e adicionarmos a props visible e configurarmos ela como true, por exemplo, o modal será exibido:

```
function App(){

  return (
    <>
      <button>Abrir Modal</button>
      <Modal visible={true}>
        <h1>Testando 1,2,3...</h1>
      </Modal>
    </>
  )
}

export default App
```

E se quisermos que ele suma, basta alterarmos o valor true para false.

Agora que entendemos a lógica, vamos colocar o valor de visible(true or false) em uma useState que chamaremos de modalVisible para que possamos modificá-lo e adicionaremos o onClick no button para que possamos acionar a modal quando clicarmos no button:

```
return (
  <>
    <button onClick={handleButtonClick}>Abrir Modal</button>
    <Modal visible={modalVisible}>
      <h1>Testando 1,2,3...</h1>
    </Modal>
  </>
)
```

No button, no comando onClick, acionamos uma variável que chamamos de handleButtonClick que contem uma função anônima que irá alterar o valor da props visible. Ficamos com a useState modalVisible e a variável handleButtonClick da seguinte maneira:

```
const [modalVisible, setModalVisible] = useState(false)

const handleButtonClick = ()=>{
  setModalVisible(true)
}
```

Acima, a useState modalVisible já inicia com o valor false e a variável handleButtonClick, utilizando o setModalVisible, altera este valor para true. Desta forma, a nossa aplicação, já responde ao comando do botão. Porém, ao clicarmos uma vez no botão, a nossa modal aparece e nunca mais retorna a condição de false, pois o nosso handleButtonClick apenas seta como true o modalViseble e não consegue retornar a condição de false.

Para isto, criaremos a seguinte estrutura:

Adicionamos mais uma props que chamaremos de setVisible e atrelamos ela a setModalVisible

```
<Modal visible={modalVisible} setVisible={setModalVisible}>
  <h1>Testando 1,2,3...</h1>
</Modal>
```

E voltamos para o Modal.js para utilizarmos esta nova prop:

Adicionamos o comando onClick no componente ModalBackground e adicionamos ao comando onClick uma variável que chamamos de handleBackGroundClick que contém uma função anônima:

```
{props.visible &&
    <ModalBackground onClick={handleBackgroundClick}>
        <ModalArea>
            {props.children}
        </ModalArea>
    </ModalBackground>
}
```

Criamos a variável handleBackgroundClick e adicionamos a função anônima a ela para que quando for clicado sobre a area do modal, a props setVisible envie o valor false para o setModalVisible fazendo com que o modal desapareça.

```
const handleBackgroundClick = ()=>{
    props.setVisible(false)
}
```

Terminamos a aula com o App.js assim:

```jsx
import React, {useState, useEffect} from 'react'
import Modal from './components/Modal'

function App(){

  const [modalVisible, setModalVisible] = useState(false)

  const handleButtonClick = ()=>{
    setModalVisible(true)
  }

  return (
    <>
      <button onClick={handleButtonClick}>Abrir Modal</button>
      <Modal visible={modalVisible} setVisible={setModalVisible}>
        <h1>Testando 1,2,3...</h1>
      </Modal>
    </>
  )
}

export default App
```

E o Modal.js assim:

```jsx
import React, {useState} from 'react'
import styled from 'styled-components'

const ModalBackground = styled.div`
    position:fixed;
    left:0;
    top:0;
    right:0;
    bottom:0;
    z-index:90;
    background-color:rgba(0, 0, 0, 0.7);
    display:flex;
    justify-content:center;
    align-items:center;
`
const ModalArea = styled.div`
    background-color:#fff;
    padding:10px;
`
```

```
function Modal(props){

    const handleBackgroundClick = ()=>{
        props.setVisible(false)
    }

    return (
        <>
            {props.visible &&
                <ModalBackground onClick={handleBackgroundClick}>
                    <ModalArea>
                        {props.children}
                    </ModalArea>
                </ModalBackground>
            }
        </>
    )
}

export default Modal
```

Aula 21 – Router: Básico 1

Antes de iniciar esta aula, ao invés de reaproveitar o projeto, resolvi criar um projeto do zero para exercitar. Para isto, segui os passos das aulas 06 do módulo 1, 01 do módulo 02 e 06 do módulo 02.

Após esta criação de um novo projeto, editamos o App,js assim:

,

```
import React from 'react'

function App(){

  return (
    <>

    </>
  )
}

export default App
```

Para utilizarmos o router, precisamos instalá-lo. Para isto, utilizamos o seguinte comando:

- Primeiramente, precisamos parar o nosso servidor no terminal com o comando **Ctrl+C.**
- **npm install react-router-dom**
- **npm start** → Para iniciar o nosso sistema novamente.

```
PS D:\Informática\Programação\2021\Daniel\B7Web\ReactJS\modulo02\m02a21\m02a21> npm install react-router-dom
npm WARN @babel/plugin-bugfix-v8-spread-parameters-in-optional-chaining@7.14.5 requires a peer of @babel/core@^7.13.0 but none is installed. You must instal
ll peer dependencies yourself.
npm WARN tsutils@3.21.0 requires a peer of typescript@>=2.8.0 || >= 3.2.0-dev || >= 3.3.0-dev || >= 3.4.0-dev || >= 3.5.0-dev || >= 3.6.0-dev || >= 3.6.0-b
eta || >= 3.7.0-dev || >= 3.7.0-beta but none is installed. You must install peer dependencies yourself.
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@2.3.2 (node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@2.3.2: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"
x64"})
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.13 (node_modules\watchpack-chokidar2\node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.13: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":
"x64"})
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.13 (node_modules\webpack-dev-server\node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.13: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":
"x64"})

+ react-router-dom@5.2.0
added 10 packages from 5 contributors and audited 1950 packages in 32.21s

146 packages are looking for funding
  run `npm fund` for details

found 3 moderate severity vulnerabilities
  run `npm audit fix` to fix them, or `npm audit` for details
PS D:\Informática\Programação\2021\Daniel\B7Web\ReactJS\modulo02\m02a21\m02a21>
```

Para iniciarmos a trabalhar com router, começaremos criando apenas doi caminhos, o ‘home’ e o ‘sobre’

O primeiro passo será importar alguns componentes do react-router-dom:

```
import {BrowserRouter, Switch, Route, Link} from 'react-router-dom'
```

E, na nossa function App, utilizaremos o BrowserRouter e criaremos a seguinte estrutura com os doi links:

```
return (
  <BrowserRouter>
    <header>
      <h1>Meu site legal</h1>
      <nav>
        <ul>
          <li><a href="/">Home</a></li>
          <li><a href="/sobre">Sobre</a></li>
        </ul>
      </nav>
    </header>
  </BrowserRouter>
)
```

Aqui, os nossos dois links estão funcionando, porém, quando clicamos num dos links, o navegador recarrega a página toda. O Router nos oferece o componente **Link** que faz com que o navegador recarregue apenas o conteúdo que for necessário, tornando a navegação mais leve, porque manterá o conteúdo que permanecer igual.

Para utilizarmos o Link:

```
return (
  <BrowserRouter>
    <header>
      <h1>Meu site legal</h1>
      <nav>
        <ul>
          <li>
            <Link to="/">Home</Link>
          </li>
          <li>
            <Link to="/sobre">Sobre</Link>
          </li>
        </ul>
      </nav>
    </header>
  </BrowserRouter>
)
```

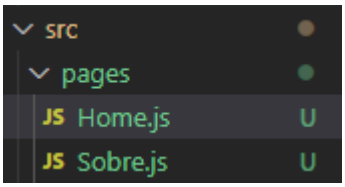
Aula 22 – Router: Básico 2

Para criarmos a parte da página que será alterada quando clicarmos em um link utilizaremos o componente **Switch** que receberá as rotas e funciona como o switch do javascript mesmo:

```
return (
  <BrowserRouter>
    <header>
      <h1>Meu site legal</h1>
      <nav>
        <ul>
          <li>
            <Link to="/">Home</Link>
          </li>
          <li>
            <Link to="/sobre">Sobre</Link>
          </li>
        </ul>
      </nav>
    </header>
    <hr/>
    <Switch>

    </Switch>
    <hr/>
    <footer>
      Todos os direitos reservados...
    </footer>
  </BrowserRouter>
)
```

Agora, dentro do Switch criaremos as rotas. Poderíamos criar diretamente no App.js cada página, porém, não é assim que se trabalha. Criamos dentro da pasta 'src' uma pasta que chamamos de **pages** e dentro dela criaremos as páginas.

Criamos a Home.js e Sobre.js. Com as seguintes estruturas básicas:



Home.js:

```
import React from 'react'

function Home(){

    return (
        <div>
            <h4>Página HOME</h4>
        </div>
    )
}

export default Home
```

Sobre.js:

```
import React from 'react'

function Sobre(){

    return (
        <div>
            <h4>Página Sobre</h4>
        </div>
    )
}

export default Sobre
```

Feito nossas duas páginas, precisamos agora criar as rotas para elas no nosso App.js.
O primeiro passo para isto é importar os arquivos para o App.js:

```
import React from 'react'
import {BrowserRouter, Switch, Route, Link} from 'react-router-dom'
import Home from './pages/Home'
import Sobre from './pages/Sobre'
```

Com os arquivos importados, criamos as rotas com a seguinte estrutura:

```
<Switch>
  <Route exact path="/">
    <Home/>
  </Route>
  <Route path="/sobre">
    <Sobre/>
  </Route>
</Switch>
```

Dentro do componente Route, adicionamos o **path** que é responsável por mostras o camonho

exact → É muito importante adicionarmos o exact no Home, pois ele faz com que a Home apareça apenas quando o endereço for exatamente */*. caso não adicionarmos, sempre que for encontrado */* ele exibira a Home, mesmo que tenha outros caracteres após a barra.

Obs.: podemos utilizar o exact em tadas as rotas, se quisermos.

```
<Switch>
  <Route exact path="/">
    <Home/>
  </Route>
  <Route exact path="/sobre">
    <Sobre/>
  </Route>
</Switch>
```

Aula 23 – Router: Parâmetros na URL

Vamos criar mais 3 links (Esportes, Notícias e Viagem) da seguinte forma:

```
<nav>
  <ul>
    <li>
      <Link to="/">Home</Link>
    </li>
    <li>
      <Link to="/sobre">Sobre</Link>
    </li>
    <li>
      <Link to="/categoria/esportes">Esportes</Link>
    </li>
    <li>
      <Link to="/categoria/noticias">Notícias</Link>
    </li>
    <li>
      <Link to="/categoria/viagem">Viagem</Link>
    </li>
  </ul>
</nav>
```

Criamos aqui estes 3 links e todos com o mesmo prefixo (categoria)

Podemos então criar uma rota para categoria da seguinte maneira:

```
<Switch>
  <Route exact path="/">
    <Home/>
  </Route>
  <Route path="/sobre">
    <Sobre/>
  </Route>
  <Route path="/categoria">
    <Categoria/>
  </Route>
</Switch>
```

E aqui mesmo no App.js já vamos criar o import para a página categoria que iremos criar:

```
import Categoria from './pages/Categoria'
```

Criamos o arquivo Categoria.js dentro da pasta pages e fizemos uma cópia da página sobre, alterando apenas as palavras sobre por categoria:

```
import React from 'react'

function Categoria(){

    return (
        <div>
            <h4>Página Categoria</h4>
        </div>
    )
}

export default Categoria
```

A partir daqui, quando clicamos em qualquer um dos três links novos (Esportes, Notícias ou Viagens), somos direcionados para a página Categorias

Para que possamos ter rotas personalizadas para as sub rotas de categoria adicionamos mais uma informação na rota:

```
<Route path="/categoria/:cat">
```

Esse **:cat** (nome definido por nós mesmos) funciona como uma variável (são os **:** que definem isto) que irá armazenar a informação adicional recebida para podermos definir em qual sub rota estamos

Agora que temos como pegar a informação exata da rota, vamos editar a página categoria (Categoria.js):

No Categoria.js iremos importar mais um componente (hookis):

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

Agora, utilizamos este componente para pegar as informações e podermos utilizar as sub-categorias:

```
let {cat} = useParams()
```

Obs.: Se quiséssemos, poderíamos ter diversas variáveis neste sentido. Por exemplo:
/categoria/:cat/:algumacoisa

Agora, temos o sub link armazenado na variável cat e podemos utilizá-lo. Veja , por exemplo:

```
import React from 'react'
import {useParams} from 'react-router-dom'

function Categoria(){

    let {cat} = useParams()

    return (
        <div>
            <h4>Página Categoria</h4>
            Exibindo itens da categoria {cat}
        </div>
    )
}

export default Categoria
```

Utilizamos na linha: Exibindo itens da categoria {cat} e ele respondeu como esperado no navegador:

Aula 24 – Router: Query

Como trabalhamos com Query String?

Primeiramente, apagamos o **:cat** da rota categoria no arquivo App.js:

```
<Route path="/categoria">
  <Categoria/>
</Route>
```

A partir daqui, o site não está mais pegando a sub categoria.

Como não estamos mais enviando nenhum parâmetro para o arquivo Categoria.js, vamos editar este arquivo:
- Apagamos o useParams do import.
- Apagamos a variával {cat}
- No import utilizamos agora o **useLocation** → Que nos permitirá utilizar Query String

```
import React from 'react'
import {useLocation} from 'react-router-dom'


function Categoria(){



    return (
        <div>
            <h4>Página Categoria</h4>
            Exibindo itens da categoria {cat}
        </div>
    )
}

export default Categoria
```

Iremos criar um Custome Hookie. Ele pode ser criado dentro da function Categoria, ou fora. Por questões de organização, criaremos fora da function:

```
function useQuery(){
    return new URLSearchParams(useLocation().search)
}
```

Chamamos esta function de useQuery que irá retornar o seguinte:

new URLSearchParams → Comando javascript que irá montar a classe useLocation. E aqui, utilizaremos o **useLocation().search**

Agora, podemos utilizar o useLocation através da nossa function useQuery.

Utilizaremos a useQuery na function Categoria

```
function Categoria(){
    let query = useQuery()
    let cat = query.get('tipo')

    return (
        <div>
            <h4>Página Categoria</h4>
            Exibindo itens da categoria {cat}
        </div>
    )
}
```

let query = useQuery() → Trazemos todos os recursos da useLocation através da function useQuery e armazenamos na variável **query**

let cat = query.get('tipo') → Criamos aqui a variável **cat** e utilizamos o recurso **get** para pegar a informação da rota que chamaremos de **tipo.**

Agora, precisamos acessar o App.js e editarmos os links das nossas sub categorias da seguinte forma:

```
<li>
  <Link to="/categoria?tipo=esportes">Esportes</Link>
</li>
<li>
  <Link to="/categoria?tipo=noticias">Notícias</Link>
</li>
<li>
  <Link to="/categoria?tipo=viagem">Viagem</Link>
</li>
```

Aula 25 – Router: Erro 404

Como criarmos uma rota para página não encontrada (Erro 404)

É importante entendermos um pouco melhor o Switch que estamos utilizando no App.js

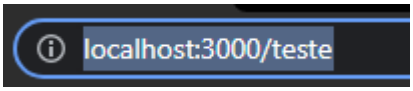
O Switch verifica rota por rota. Uma vez que ele encontra uma rota, ele para de procurar. Entendendo isto, significa que a última rota que colocarmos será a última a ser verificada. Se nesta última rota utilizarmos uma rota “genérica”, ou seja, uma rota que bate com qualquer URL, como um asterisco (*)por exemplo. Quando o Switch não encontrar nenhuma rota válida anteriormente, ele irá carregar esta última rota. Com este entendimento, podemos utilizar esta última rota como página não encontrada.

<Route path=”*”>
<h4>Página não encontrada</h4>
</Route>

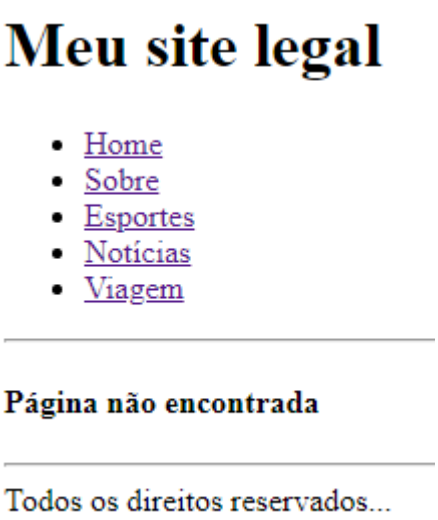
```
<Switch>
  <Route exact path="/">
    <Home/>
  </Route>
  <Route path="/sobre">
    <Sobre/>
  </Route>
  <Route path="/categoria">
    <Categoria/>
  </Route>
  <Route path="*">

  </Route>
</Switch>
```

Nesta rota, podemos criar uma página para ela. Neste exemplo, apenas adicionamos em um h4 a mensagem de página não encontrada.

Se digitarmos no navegador um link da página que não exista:



o navegador executa a nossa rota de página não encontrada:



Aula 26 – Router: Redirect

Redirecionamentos: Temos duas formas de fazer redirecionamentos. Uma delas é a partir de uma ação e a outra é através de uma condição.

Condição:

Para iniciarmos criaremos mais uma rota no App.js:

```
<Route path="/quem-somos">

</Route>
```

Vamos utilizar esta rota para que quando o usuário clique em ‘quem somos’ ele seja redirecionado para a rota ‘sobre’.

Para isto, adicionaremos mais um componente, o **Redirect**

```
import {BrowserRouter, Switch, Route, Link, Redirect} from 'react-router-dom'
```

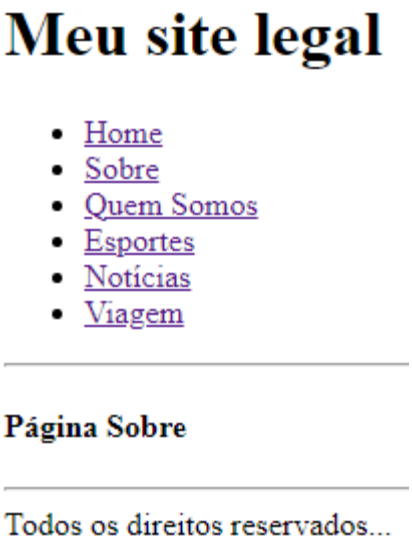
Então adicionamos o Redirect na rota ‘quem-somos’ apontando para a rota sobre:

```
<Route path="/quem-somos">
  <Redirect to="/sobre"/>
</Route>
```

e criamos o link para ‘quem-somos’:

```
<li>
  <Link to="/quem-somos">Quem Somos</Link>
</li>
```

A partir daqui, quando clicamos no link ‘Quem Somos’, somos redirecionados para a página ‘Sobre’:



Ação:

Primeiramente, vamos criar um botão na página Home. Quando o usuário clicar neste botão, o sistema executará uma ação que seja definida e será redirecionado para a página 'Sobre'.

Para isto, vamos criar um botão na Home.js:

```
<button onClick={handleButton}>Ir para a página SOBRE</button>
```

Neste botão adicionamos o onClick e vinculamos uma variável **handleButton** que criaremos:

Para criarmos esta variável, precisamos importar para cá um outro Hookies de 'react-router-dom':

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

Após importarmos este Hookies, armazenamos os seus recursos em uma variável que chamaremos de **history**:

```
let history = useHistory()
```

E então criamos a variável handleButton que está armazenando uma função anônima:

```
const handleButton = () => {
    history.replace('/sobre')
}
```

history.replace → Muda o histórico, neste caso, para **/sobre**.

Ficamos com o nosso Home.js assim:

```jsx
import React from 'react'
import {useHistory} from 'react-router-dom'

function Home(){
    let history = useHistory()
    const handleButton = () => {
        history.replace('/sobre')
    }
    return (
        <div>
            <h4>Página HOME</h4>

            <button onClick={handleButton}>Ir para a página SOBRE</button>
        </div>
    )
}

export default Home
```

A partir daqui, ao clicarmos no botão 'Ir para a página SOBRE', somos direcionados para a página SOBRE.

Vamos aqui criar uma ação para este botão para que ele direcione para a página sobre apenas 2 segundos após ser clicado no botão.

```jsx
const handleButton = () => {
    setTimeout(()=>{
        history.replace('/sobre')
    }, 2000)
}
```

Aula 27 – Router: Rotas Privadas (Controle de Acesso)

Rotas privadas → Rotas que precisam de determinado acesso para elas funcionarem.

Para iniciarmos, vamos remover do App.js as rotas quem-somos e categorias e dos links, deixaremos apenas o home ("/") e o sobre ("/sobre"). Tambem, excluiremos o import do Categoria:

```jsx
<Switch>
  <Route exact path="/">
    <Home/>
  </Route>
  <Route path="/sobre">
    <Sobre/>
  </Route>
  <Route path="*">
    <h4>Página não encontrada</h4>
  </Route>
</Switch>
```

```jsx
<nav>
  <ul>
    <li>
      <Link to="/">Home</Link>
    </li>
    <li>
      <Link to="/sobre">Sobre</Link>
    </li>
  </ul>
</nav>
```

Vamos criar o seguinte processo: A página ‘Home’ vamos manter para acesso público, porém para acessar a página ‘sobre’, o usuário deverá estar logado.

Agora criaremos uma variável que chamaremos de **isLogged** e começará com o valor **false** e utilizaremos para fazer este controle.

```
const isLogged = false
```

Temos duas formas de fazer este controle:

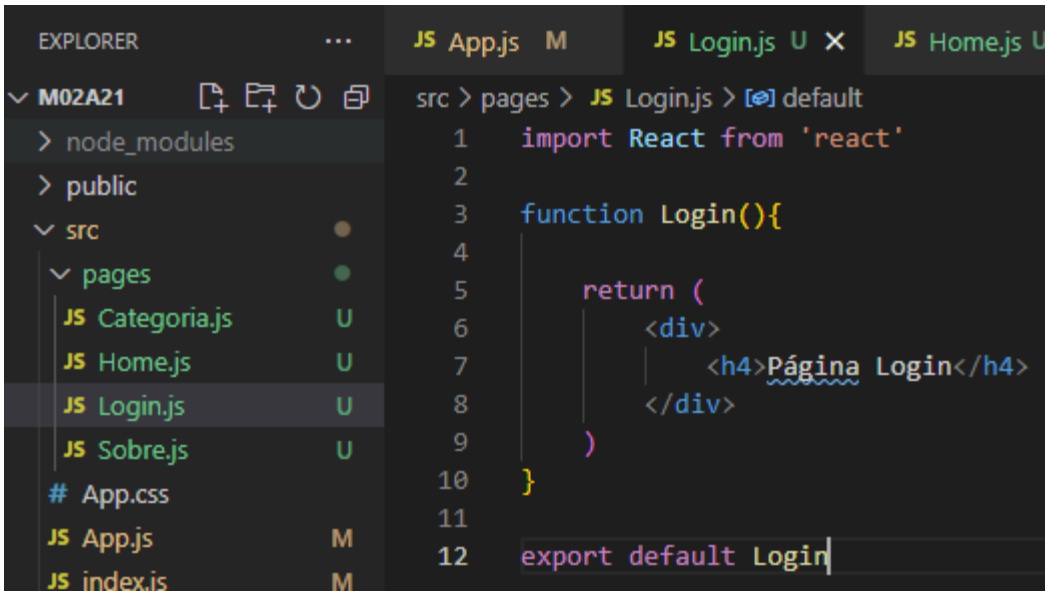
1ª Forma → Editamos a rota sobre:

```
<Route path="/sobre">
  {isLogged ? <Sobre/> : <Redirect to="/login"/>}
</Route>
```

Criamos aqui uma condicional para que o componente ‘Sobre’ só seja acessado se isLogged for true, caso contrário, será redirecionado para uma rota chamada login, que não criamos ainda.

Adicionamos a rota Login e criamos a página Login(Cópia da página Sobre):

```
<Route path="/login">
  <Login/>
</Route>
```



E no App.js, precisamos agora adicionar o import da página Login:

```
import Login from './pages/Login'
```

A partir daqui, o nosso sistema está funcionando, pois ao tentarmos acessar a página sobre, o sistema nos redireciona para a página de Login, porém se alterarmos o valor da nossa variável isLogged para true, o sistema permite que acessemos a página sobre normalmente.

2ª Forma → Criaremos um componente que fará este processo internamente.

Criaremos um componente que chamaremos de PrivateRoute que será responsável por fazer este controle de acesso:

```jsx
const PrivateRoute = ({children, ...rest}) => {
  return (
    <Route {...rest}>
      {isLogged ? children : <Redirect to="/login"/>}
    </Route>
  )
}
```

Este componente retorna o item que queremos acessar ou redireciona para a página de login:
A PrivateRoute recebe duas props:

children → O conteúdo que queremos acessar

...rest → Chamamos a segunda props de rest(resto). É responsável por receber as props que são enviadas lá na PrivateRoute

<Route {...rest}> → Mandamos o rest para a rota. Repetimos as props da PrivateRote aqui na Route.

{isLoged ? Children : <Redirect to=”/login”/>} → Utilizamos a mesma condição que criamos para o modelo anterior, ou seja: Está logado? Children, Não está logado? Redirect.

Mudamos a nossa Route e utilizamos a PrivateRoute que tem como conteúdo a Route Sobre (children)

```jsx
<PrivateRoute path="/sobre">
  <Sobre/>
</PrivateRoute>
```

A partir daqui, o nosso sistema está funcionando, pois ao tentarmos acessar a página sobre, o sistema nos redireciona para a página de Login, porém se alterarmos o valor da nossa variável isLogged para true, o sistema permite que acessemos a página sobre normalmente.