**Introdução**

Seja bem vindo(a) ao primeiro curso de React da Alura! Meu nome é Paulo Scalercio, e eu serei o seu guia durante o desenvolvimento de uma aplicação para a Casa do Código. Se você tem interesse em entender mais sobre o React, veio ao lugar certo! Não se esqueça de fazer os exercícios e, se tiver qualquer dúvida, comentar no fórum do curso.

Bons estudos!

**Preparando o ambiente**

Para começarmos a trabalhar com o React, precisaremos montar o nosso ambiente. Durante o curso, usaremos a versão 10.15.3 LTS do Node, que pode ser baixada em <https://nodejs.org/en/>. Em seguida, basta seguir os passos da instalação.

Como editor de texto, usaremos O VSCode (Visual Studio Code). Também usaremos o create-react-app, uma ferramenta do Facebook que gerará automaticamente um projeto inicial do React. No Prompt de Comando, acessaremos a pasta do projeto, onde armazenaremos todo o código, e executaremos npm install create-react-app para instalar essa ferramenta.

Terminada a instalação, utilizaremos o NPX, o package runner do NPM, para criarmos nosso projeto. Executaremos npx create-react-app curso-react-alura e aguardaremos a instalação dos pacotes. Ao fim, teremos a sugestão de executar cd curso-react-alura (para entrarmos na pasta do projeto) e então npm start para iniciarmos o servidor de desenvolvimento. Feito isso, será aberta uma página no navegador com um projeto React padrão carregado.

Após ter feito tudo isso, estaremos preparados para iniciar o curso. Vamos lá!

# Instalando NodeJS e VSCode

Para prosseguir com o curso, precisamos instalar as devidas dependências.

**VSCode**

É recomendado que você utilize o mesmo editor de texto do curso, você pode baixar e instalar o **VsCode** [nesse link](https://code.visualstudio.com/)

**NodeJS no Windows**

Para começar, utilizaremos a versão **10.15.3** do NodeJS, é de extrema importância que seja utilizada essa versão durante o curso. Você precisará fazer o download do instalador [nesse link](https://nodejs.org/en/), feito isso, utilize o instalador para prosseguir.

É importante não alterar as configurações recomendadas pelo instalador, dessa forma ele instalará o npm que precisaremos durante o curso.

Você pode verificar se a instalação foi um sucesso através do comando:

node -v

A saída deve ser a versão do node instalada.

**NodeJS no MacOS**

Para começar, utilizaremos a versão **10.15.3** do NodeJS, é de extrema importância que seja utilizada essa versão durante o curso. Você precisará fazer o download do instalador [nesse link](https://nodejs.org/en/), feito isso, utilize o instalador para prosseguir.

É importante não alterar as configurações recomendadas pelo instalador, dessa forma ele instalará o npm que precisaremos durante o curso

Execute o NodeJS.pkg para iniciar a instalação.

Você pode verificar se a instalação foi um sucesso através do comando:

node -v

A saída deve ser a versão do node instalada.

**Ubuntu**

Para começar, utilizaremos a versão **10.15.3** do NodeJS, é de extrema importância que seja utilizada essa versão durante o curso.

Para instalar o node no Ubuntu, execute a linha abaixo:

sudo apt-get install nodejs@10.15.3

Você pode verificar se a instalação foi um sucesso através do comando:

node -v

A saída deve ser a versão do node instalada.

2) Também precisaremos do NPM, para instalá-lo, rode o comando abaixo no terminal

sudo apt-get install npm

Você pode verificar se a instalação foi um sucesso através do comando:

npm --version

A saída deve ser a versão do npm instalada.

# Criando um projeto do zero

Nesta atividade criaremos um projeto do zero utilizando o create-react-app. Como essa ferramenta é constantemente atualizada, pode ser que o projeto padrão criado seja diferente do utilizado em aula. Caso isso aconteça, você pode **baixar o projeto utilizado pelo instrutor na próxima atividade**, junto com as instruções para rodar o projeto.

# Criando um projeto do zero.

1) Crie um diretório para seu projeto.

2) Abra esse diretório no terminal do seu sistema operacional e rode o comando abaixo:

npm install create-react-app

3) Após a instalação, crie o projeto executando a linha:

npx create-react-app react-alura

4) Feito isso, entre no diretório criado pela ferramenta e então execute:

npm start

# Importando o projeto da aula

# Importando o projeto

Para utilizar o mesmo projeto do instrutor, você deverá baixar o projeto [nesse link](https://caelum-online-public.s3.amazonaws.com/1196-react-parte1/01/react-0.zip) e seguir os passos abaixo:

## Inicializando o projeto baixado

Com o node, npm instalados, siga os passos abaixo:

1) Faça o [download](https://caelum-online-public.s3.amazonaws.com/1196-react-parte1/01/react-0.zip) do projeto e extraia onde desejar.

2) Abra o diretório do projeto no terminal e execute o comando:

npm install

Esse comando fará com que o npm instale todas as dependências necessárias.

3) Após a conclusão, execute:

npm start

Nesse ponto, seu projeto será inicializado e um navegador será aberto automaticamente.

# Entendendo o projeto

No último vídeo nós preparamos o ambiente do curso e utilizamos o create-react-app para criarmos a aplicação, chamada curso-react-alura, que utilizaremos para começar o desenvolvimento do nosso projeto. Ao final, executamos npm start dentro do diretório do projeto para iniciarmos o servidor de desenvolvimento da aplicação, e abrimos o endereço [http://localhost:3000](http://localhost:3000/) no navegador, nos direcionando para um projeto padrão do React que é criado automaticamente pelo pacote create-react-app.

O projeto nos mostra a seguinte mensagem:

Edit src/App.js and save to reload.

Ou seja, deveremos editar o arquivo App.js dentro do diretório "src" e salvá-lo para recarregar a página da projeto. Além disso, temos um link "Learn React" que direciona para a documentação dessa biblioteca.

No diretório "curso-react-alura", temos uma pasta "node\_modules", que contém os módulos do node que a aplicação precisará para funcionar. Temos também uma pasta "public" contendo um arquivo favicon.ico, o ícone que é mostrado na aba do navegador; um manifest.json contendo algumas configurações; e um arquivo index.html.

Nesse HTML, temos um <title>, tag que define o que aparece na aba do navegador - nesse caso, "React App". No corpo desse arquivo, temos uma <div id="root"> que não possui nenhum conteúdo, o que é um pouco curioso - e guardaremos essa informação para o futuro.

Abriremos a pasta "src" do projeto no Visual Studio Code. Nela, temos um arquivo App.js, cuja edição nos foi indicada na página inicial da aplicação, um arquivo de estilo App.css, além de um index.css e um index.js. Vamos analisar o código deste último:

ReactDOM.render(<App />, document.getElementById('root'));

// If you want your app to work offline and load faster, you can change

// unregister() to register() below. Note this comes with some pitfalls.

// Learn more about service workers: https://bit.ly/CRA-PWA

serviceWorker.unregister();

Encontramos nesse arquivo um ReactDOM que está chamando o métood render(), e um App, importado do arquivo App.js, que está sendo inserido em um elemento que possiu uma ID root - que é justamente o nosso index.html. Ou seja, todo o conteúdo de App.js, que se parece com um HTML (mas na verdade é outra coisa), está sendo renderizado por meio do método render() dentro de um elemento com o ID root.

Para confirmarmos essa afirmação, inspecionaremos a página [http://localhost:3000](http://localhost:3000/). Encontraremos uma <div id="root">, dentro da qual temos uma <div class="App"> e todo o header de um projeto padrão do React. Aparentemente, se editarmos o arquivo App.js, isso será refletido na tela. Pensando nisso, removeremos todo o conteúdo a partir do header, mantendo somente a <div className="App">. Em seguida, como todo desenvolvedor que está aprendendo uma nova tecnologia, escreveremos um "Hello World!", por enquanto na forma de título (<h1>).

function App() {

return (

<div className="App">

<h1>Hello World!</h1>

</div>

);

}

export default App;

Em [http://localhost:3000](http://localhost:3000/), a mensagem que escrevemos no código será exibida com sucesso. Porém, no Prompt, receberemos a mensagem "Compiled with warnings.":

Compiled with warnings.

.src/App.js

Line 2: 'logo' is defined but never used no-unused-vars

Search for the keywords to learn more about each warning.

To ignore, add //enlist-disable-next-line to the line before.

Isso significa que, na linha 2 de App.js, temos uma variável logo que está sendo definida mas não utilizada. Essa variável representa o arquivo logo.svg, que é a imagem rotativa antes mostrada na página inicial da aplicação. Sendo assim, vamos removê-la. Feito isso, passaremos a receber a mensagem "Compiled successfully!".

No próximo vídeo entenderemos um pouco mais sobre o arquivo App.js.

# Entendendo o projeto

No último vídeo nós preparamos o ambiente do curso e utilizamos o create-react-app para criarmos a aplicação, chamada curso-react-alura, que utilizaremos para começar o desenvolvimento do nosso projeto. Ao final, executamos npm start dentro do diretório do projeto para iniciarmos o servidor de desenvolvimento da aplicação, e abrimos o endereço [http://localhost:3000](http://localhost:3000/) no navegador, nos direcionando para um projeto padrão do React que é criado automaticamente pelo pacote create-react-app.

O projeto nos mostra a seguinte mensagem:

Edit src/App.js and save to reload.

Ou seja, deveremos editar o arquivo App.js dentro do diretório "src" e salvá-lo para recarregar a página da projeto. Além disso, temos um link "Learn React" que direciona para a documentação dessa biblioteca.

No diretório "curso-react-alura", temos uma pasta "node\_modules", que contém os módulos do node que a aplicação precisará para funcionar. Temos também uma pasta "public" contendo um arquivo favicon.ico, o ícone que é mostrado na aba do navegador; um manifest.json contendo algumas configurações; e um arquivo index.html.

Nesse HTML, temos um <title>, tag que define o que aparece na aba do navegador - nesse caso, "React App". No corpo desse arquivo, temos uma <div id="root"> que não possui nenhum conteúdo, o que é um pouco curioso - e guardaremos essa informação para o futuro.

Abriremos a pasta "src" do projeto no Visual Studio Code. Nela, temos um arquivo App.js, cuja edição nos foi indicada na página inicial da aplicação, um arquivo de estilo App.css, além de um index.css e um index.js. Vamos analisar o código deste último:

ReactDOM.render(<App />, document.getElementById('root'));

// If you want your app to work offline and load faster, you can change

// unregister() to register() below. Note this comes with some pitfalls.

// Learn more about service workers: https://bit.ly/CRA-PWA

serviceWorker.unregister();

Encontramos nesse arquivo um ReactDOM que está chamando o métood render(), e um App, importado do arquivo App.js, que está sendo inserido em um elemento que possiu uma ID root - que é justamente o nosso index.html. Ou seja, todo o conteúdo de App.js, que se parece com um HTML (mas na verdade é outra coisa), está sendo renderizado por meio do método render() dentro de um elemento com o ID root.

Para confirmarmos essa afirmação, inspecionaremos a página [http://localhost:3000](http://localhost:3000/). Encontraremos uma <div id="root">, dentro da qual temos uma <div class="App"> e todo o header de um projeto padrão do React. Aparentemente, se editarmos o arquivo App.js, isso será refletido na tela. Pensando nisso, removeremos todo o conteúdo a partir do header, mantendo somente a <div className="App">. Em seguida, como todo desenvolvedor que está aprendendo uma nova tecnologia, escreveremos um "Hello World!", por enquanto na forma de título (<h1>).

function App() {

return (

<div className="App">

<h1>Hello World!</h1>

</div>

);

}

export default App;

Em [http://localhost:3000](http://localhost:3000/), a mensagem que escrevemos no código será exibida com sucesso. Porém, no Prompt, receberemos a mensagem "Compiled with warnings.":

Compiled with warnings.

.src/App.js

Line 2: 'logo' is defined but never used no-unused-vars

Search for the keywords to learn more about each warning.

To ignore, add //enlist-disable-next-line to the line before.

Isso significa que, na linha 2 de App.js, temos uma variável logo que está sendo definida mas não utilizada. Essa variável representa o arquivo logo.svg, que é a imagem rotativa antes mostrada na página inicial da aplicação. Sendo assim, vamos removê-la. Feito isso, passaremos a receber a mensagem "Compiled successfully!".

No próximo vídeo entenderemos um pouco mais sobre o arquivo App.js.

# Mãos na Massa

Chegou a hora de você pôr em prática o que foi visto na aula. Para isso, execute os passos listados abaixo.

1 - Abra a pasta src com o editor de texto de sua escolha

2 - Abra o arquivo App.js

3 - Substitua o JSX desse arquivo por :

<div className=”App”>

<h1> Hello World </h1>

</div>

4 - Remova a linha de importação da Logo do React:

import logo from ‘./logo.svg’;

5 - O código final do App.js deve ser:

import React from ‘react’;

import ‘./App.css’;

function App(){

return(

<div className=”App”>

<h1> Hello World </h1>

</div>

);

}

export default App;

# O que aprendemos?

Nesta aula, aprendemos:

* Ambiente necessário para criação do projeto
* Criação de um projeto inicial utilizando create-react-app
* Inicialização do projeto inicial

# Primeiro componente

No último vídeo fizemos um "Hello World!", e agora começaremos a realmente trabalhar no nosso projeto. Na introdução, vimos que o objetivo é criarmos um sistema para a Casa do Código. De início, precisaremos de uma tabela de autores contendo seus livros, um botão de remover, entre outras features. Para isso, escrevemos uma <table> no arquivo App.js.

Uma <table> é composta por um <thead> e um <tbody>. Dentro de <thead>, adicionaremos uma linha com <tr> ("table row") e várias <th> contendo os textos "Autores", "Livros" e "Preços". Em <tbody>, também teremos várias <tr>, cada uma delas com várias <td> respectivas a essas categorias. Em <thead>, adicionaremos uma <th> de "Remover", e para cada <tr> do corpo adicionaremos um <button>, também de "Remover".

function App() {

return (

<div className="App">

<table>

<thead>

<tr>

<th>Autores</th>

<th>Livros</th>

<th>Preços</th>

<th>Remover</th>

</tr>

</thead>

<tbody>

<tr>

<td>Paulo</td>

<td>React</td>

<td>1000</td>

<td><button>Remover</button></td>

</tr>

<tr>

<td>Nico</td>

<td>React</td>

<td>1000</td>

<td><button>Remover</button></td>

</tr>

<tr>

<td>Daniel</td>

<td>React</td>

<td>1000</td>

<td><button>Remover</button></td>

</tr>

</tbody>

</table>

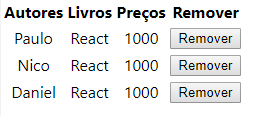
</div>

);

}

export default App;

Após salvarmos, o código compilará corretamente e teremos uma tabela sendo exibida no Front End, ainda que o botão "Remover" não tenha funcionalidade.



A ideia do React é justamente criarmos o Front End por meio de componentes reutilizáveis para nossa aplicação. Sendo assim, queremos transformar essa tabela em um componente que poderá ser utilizado em diversas partes do nosso código. Começaremos esse processo criando uma classeTabela.js na qual importaremos a biblioteca React, e que herdará de Component. Todo componente obrigatoriamente precisa de um método render(), que irá renderizar algo na tela. Esse método retornará justamente o template de tabela que criamos anteriormente. Por fim, exportaremos a Tabela.

import React, {Component} from 'react';

class Tabela extends Component {

render() {

return(

<table>

<thead>

<tr>

<th>Autores</th>

<th>Livros</th>

<th>Preços</th>

<th>Remover</th>

</tr>

</thead>

<tbody>

<tr>

<td>Paulo</td>

<td>React</td>

<td>1000</td>

<td><button>Remover</button></td>

</tr>

<tr>

<td>Nico</td>

<td>React</td>

<td>1000</td>

<td><button>Remover</button></td>

</tr>

<tr>

<td>Daniel</td>

<td>React</td>

<td>1000</td>

<td><button>Remover</button></td>

</tr>

</tbody>

</table>

);

}

}

export default Tabela;

No arquivo App.js, importaremos a classe Tabela e, na <div>, a utilizaremos por meio da tag <Tabela />.

import React from 'react';

import './App.css';

import Tabela from './Tabela';

function App() {

return (

<div className="App">

<Tabela />

</div>

);

}

export default App;

Salvando nossas alterações, a tabela que criamos continuará sendo exibida em [http://localhost:3000](http://localhost:3000/). Já começamos a componentizar nossa aplicação, mas ainda podemos subcomponentizá-la ainda mais. Repare que a nossa <table> é composta de um <thead> e um <tbody>. A ideia, então, é continuarmos com um <thead> fixo, mas tornarmos o conteúdo do <tbody> dinâmico. Sendo assim, vamos transformá-lo em um componente.

Primeiro, fora da classe, criaremos uma constante TableHead que receberá uma arrow function cujo retorno será o template de <thead>. Em seguida, adicionaremos o componente <TableHead /> na Tabela.

const TableHead = () => {

return (

<thead>

<tr>

<th>Autores</th>

<th>Livros</th>

<th>Preços</th>

<th>Remover</th>

</tr>

</thead>

);

}

class Tabela extends Component {

render() {

return (

<table>

<TableHead />

<tbody>

<tr>

<td>Paulo</td>

<td>React</td>

<td>1000</td>

<td><button>Remover</button></td>

</tr>

<tr>

<td>Nico</td>

<td>React</td>

<td>1000</td>

<td><button>Remover</button></td>

</tr>

<tr>

<td>Daniel</td>

<td>React</td>

<td>1000</td>

<td><button>Remover</button></td>

</tr>

</tbody>

</table>

);

}

}

Como estamos trabalhando no mesmo documento, não precisaremos exportar ou importar nada, e a nossa tabela continuará sendo exibida corretamente. Repetiremos esse processo para <tbody>.

const TableBody = () => {

return(

<tbody>

<tr>

<td>Paulo</td>

<td>React</td>

<td>1000</td>

<td><button>Remover</button></td>

</tr>

<tr>

<td>Nico</td>

<td>React</td>

<td>1000</td>

<td><button>Remover</button></td>

</tr>

<tr>

<td>Daniel</td>

<td>React</td>

<td>1000</td>

<td><button>Remover</button></td>

</tr>

</tbody>

);

}

class Tabela extends Component {

render() {

return (

<table>

<TableHead />

<TableBody />

</table>

);

}

}

Da mesma forma, nossa tabela continuará sendo exibida corretamente. Agora, nossa classe Tabela é um componente composto por dois subcomponentes, <TableHead /> e <TableBody />. No próximo vídeo continuaremos trabalhando nesse projeto.

# Passando dados entre componentes

No último vídeo nós componentizamos um componente <Tabela />, criando dois subcomponentes, <TableHead /> e <TableBody />. A ideia agora é tornarmos os dados de <TableBody /> um pouco mais dinâmicos, de modo que a nossa tabela saiba renderizar esses dados recebidos, por exemplo, a partir de um array.

Para começarmos, criaremos um array de autores em App.js (preparado previamente pelo instrutor).

function App() {

const autores = [

{

nome: 'Paulo',

livro: 'React',

preco: '1000'

},

{

nome: 'Daniel',

livro: 'Java',

preco: '99'

},

{

nome: 'Marcos',

livro: 'Design',

preco: '150'

},

{

nome: 'Bruno',

livro: 'DevOps',

preco: '100'

}

];

return (

<div className="App">

<Tabela />

</div>

);

}

export default App;

Temos então um array de JSON com os nomes dos autores, seus respectivos livros e preços diferentes. De alguma forma, precisaremos passar esse array de autores para o nosso componente <Tabela />, que deverá renderizá-los corretamente - se temos quatro elementos no array, deveremos ter quatro linhas.

Em React, para passarmos uma informação de um componente para outro, precisamos utilizar os chamados Props, que têm uma notação semelhante às propriedades do HTML. No retorno de App.js, escolheremos o nome desse prop, nesse caso autores, e passaremos a ele uma informação - mais especificamente, o array de autores.

return (

<div className="App">

<Tabela autores = { autores }/>

</div>

);

Feito isso, precisaremos receber tal elemento no componente Tabela. No corpo de render(), criaremos um elemento com o mesmo nome (const { autores }), e, por meio de this.props, tal elemento receberá automaticamente os dados armazenados naquele objeto.

class Tabela extends Component {

render() {

const {autores} = this.props;

return (

<table>

<TableHead />

<TableBody />

</table>

);

}

}

Para verificarmos se tal comportamento realmente está acontecendo, faremos um console.log() do elemento autores e, após a compilação, inspecionaremos a página da aplicação. No console, encontraremos um array:

0: {nome: "Paulo", livro: "React", preco: "1000"}

1: {nome: "Daniel", livro: "Java", preco: "99"}

2: {nome: "Marcos", livro: "Design", preco: "150"}

3: {nome: "Bruno", livro: "DevOps", preco: "100"}

Ou seja, realmente estamos conseguindo passar as informações. Sendo assim, removeremos o console.log() e voltaremos a trabalhar. O componente Tabela é composto por dois subcomponentes. Dentre eles, é o <TableBody /> que precisará receber o array autores para trabalhar. Sendo assim, da mesma forma que fizemos no App.js, criaremos um Prop autores para o qual passaremos o elemento autores.

class Tabela extends Component {

render() {

const { autores } = this.props;

return (

<table>

<TableHead />

<TableBody autores = { autores }/>

</table>

);

}

}

O TableBody é um *function component*, e pode receber parâmetros. No caso, passaremos um objeto props, de forma que ela possa receber outras propriedades além de autores. Como queremos criar uma lógica de renderização dos dados de um array, apagaremos os autores que criamos anteriormente nesse componente.

const TableBody = props => {

}

Começaremos criando uma constante linhas, que serão uma sequência de *table rows*, ou seja, <tr>. A partir de props, pegaremos o nosso array autores e faremos um map() desses elementos, recebendo também uma *arrow function*. Nela, retornaremos uma <tr> contendo três <td> simples e um botão "Remover".

const TableBody = props => {

const linhas = props.autores.map(() => {

return (

<tr>

<td></td>

<td></td>

<td></td>

<td><button>Remover</button></td>

</tr>

)

});

}

No método map(), a função receberá linha, o elemento individual do momento da iteração, e um index, que é a posição desse elemento.

const TableBody = props => {

const linhas = props.autores.map((linha, index) => {

return (

<tr>

<td></td>

<td></td>

<td></td>

<td><button>Remover</button></td>

</tr>

)

});

}

Com isso, dentro do return, usaremos linha.nome, linha.livro e linha.preco para acessar cada uma das propriedades dos itens no nosso array.

const TableBody = props => {

const linhas = props.autores.map((linha, index) => {

return (

<tr>

<td>{linha.nome}</td>

<td>{linha.livro}</td>

<td>{linha.preco}</td>

<td><button>Remover</button></td>

</tr>

)

});

}

Agora, precisaremos compor esse componente e organizar todas essas linhas em um <TableBody />. Para isso, retornaremos um <tbody> dentro do qual teremos as nossas linhas.

const TableBody = props => {

const linhas = props.autores.map((linha, index) => {

return (

<tr>

<td>{linha.nome}</td>

<td>{linha.livro}</td>

<td>{linha.preco}</td>

<td><button>Remover</button></td>

</tr>

)

});

return(

<tbody>

{linhas}

</tbody>

);

}

Salvando as alterações, nossa nova tabela será exibida em [http://localhost:3000](http://localhost:3000/).



Dessa forma, temos uma tabela dinâmica que recebe os valores de um array. Inclusive, se quisermos, podemos adicionar outro autores em autores.

const autores = [

{

nome: 'Paulo',

livro: 'React',

preco: '1000'

},

{

nome: 'Daniel',

livro: 'Java',

preco: '99'

},

{

nome: 'Marcos',

livro: 'Design',

preco: '150'

},

{

nome: 'Bruno',

livro: 'DevOps',

preco: '100'

},

{

nome: 'Nico',

livro: 'Java',

preco: '9999'

}

];



# Mãos na Massa

Chegou a hora de você pôr em prática o que foi visto na aula. Para isso, execute os passos listados abaixo.

1 - Crie um novo arquivo chamado Tabela.js

2 - Faça a importação do módulo do React, além do Component.

import React, {Component} from ‘react’;

3 - Crie um function component para o cabeçalho da tabela e outra para o corpo.

import React, {Component} from ‘react’;

const TableHead = () =>{

return (

<thead>

<tr>

<th>Autor</th>

<th>Titulo</th>

<th>Preço</th>

<th>Remover</th>

</tr>

</thead>

);

}

const TableBody = props =>{

const linhas = props.autores.map((linha, index) => {

return (

<tr key={index}>

<td>{linha.nome}</td>

<td>{linha.titulo}</td>

<td>{linha.preco}</td>

<td><button>Remover</button></td>

</tr>

);

});

return (

<tbody>

{linhas}

</tbody>

)

}

4 - Agora, no mesmo arquivo, logo abaixo desses dois componentes, crie um Class Component Tabela, passando autores como props para o TableBody, lembre-se de exportá-lo.

class Tabela extends Component {

render(){

const { autores } = this.props;

return(

<table>

<TableHead />

<TableBody autores={autores} />

</table>

);

}

}

export default Tabela;

5 - Agora abra o arquivo App.js e crie um array de autores, veja abaixo:

import React, { Component } from 'react';

import './App.css';

class App extends Component {

autores = [

{

nome: 'Paulo',

livro: 'React',

preco: '1000'

},

{

nome: 'Daniel',

livro: 'Java',

preco: '99'

},

{

nome: 'Marcos',

livro: 'Design',

preco: '150'

},

{

nome: 'Bruno',

livro: 'DevOps',

preco: '100'

}

];

//RESTO DO CÓDIGO

}

export default App;

6 - Agora importe o componente Tabela que foi criado, e utilize-o no App.js, passando esse Array criado como props.

import React, { Component } from 'react';

import './App.css';

import Tabela from './Tabela';

class App extends Component {

autores = [

{

nome: 'Paulo',

livro: 'React',

preco: '1000'

},

{

nome: 'Daniel',

livro: 'Java',

preco: '99'

},

{

nome: 'Marcos',

livro: 'Design',

preco: '150'

},

{

nome: 'Bruno',

livro: 'DevOps',

preco: '100'

}

];

render() {

return (

<Tabela autores={autores}/>

);

}

}

export default App;

7 - Feito isso, verifique seu navegador e veja se está tudo certo!

# O que aprendemos?

Nesta aula, aprendemos:

* Criar um Class componente no React
* Criar sub-componentes internos como Function Component
* Comunicação entre componentes utilizando props.

# Trabalhando com o state

Na última aula, conseguimos fazer com que nosso código gerasse de maneira dinâmica, e com base em um array de autores, as linhas da nossa tabela. Se verificarmos a saída no Prompt, veremos que a compilação ocorreu com sucesso. Entretanto, se inspecionarmos a página e acessarmos o console, veremos um aviso de que não definimos uma propriedade key para os elementos filhos de uma lista.

No caso, nosso <tbody> possui elementos filhos, que são as linhas (<tr>), e temos que declarar uma chave para cada um deles. No React, a propriedade key serve para identificar qual elemento está sendo alterado (adicionado ou removido) no componente. Para resolvermos nosso problema, adicionaremos como chave o index, que é justamente a posição do elemento no nosso array de autores.

const linhas = props.autores.map((linha, index) => {

return (

<tr key={index}>

<td>{linha.nome}</td>

<td>{linha.livro}</td>

<td>{linha.preco}</td>

<td><button>Remover</button></td>

</tr>

)

});

Feito isso, deixaremos de receber um erro no console. De volta ao arquivo App.js, repare que nosso array de autores parece meio "jogado" no nosso código. As informações armazenadas nele estão refletindo na nossa tabela - sendo assim, quando adicionamos um novo elemento ao array, queremos que ele seja adicionado também à tabela; e, da mesma forma, se removemos um elemento do array, queremos que o componente sofra uma alteração de estado, perdendo uma das linhas da tabela.

Sendo assim, esse array nada mais é do que o controle de estado da aplicação, e tudo que diz respeito ao controle de estado de um componente tem um lugar específico para ser armazenado no React: o "state" ("estado" em inglês). Pensando nisso, alteraremos nosso código para começarmos a utilizar essa ferramenta.

É possível utilizar o state dentro de um *function component*, mas, por fim didáticos, transformaremos nosso elemento App.js em um *class component* importando a classe Component e fazendo com que ela herde dessa classe. Em seguida, criaremos um state que receberá um objeto ({}), dentro do qual armazenaremos o nosso array. Esse array se tornará uma chave de um objeto JavaScript (recebendo : ao invés de =), e teremos que adicionar uma vírgula ao final de sua declaração, já que outros elementos poderão ser adicionados no futuro.

import React, {Component} from 'react';

import './App.css';

import Tabela from './Tabela';

class App extends Component {

state = {

autores : [

{

nome: 'Paulo',

livro: 'React',

preco: '1000'

},

{

nome: 'Daniel',

livro: 'Java',

preco: '99'

},

{

nome: 'Marcos',

livro: 'Design',

preco: '150'

},

{

nome: 'Bruno',

livro: 'DevOps',

preco: '100'

},

{

nome: 'Nico',

livro: 'Java',

preco: '9999'

}

],

};

return (

<div className="App">

<Tabela autores = { autores }/>

</div>

);

}

export default App;

O VSCode acusará um erro no nosso return, já que um componente criado por meio de uma classe precisa, obrigatoriamente, da declaração de um método render(). Passaremos nosso retorno para dentro desse método.

render() {

return (

<div className="App">

<Tabela autores = { autores }/>

</div>

);

}

Nosso código ainda não funcionará, já que agora estamos armazenando os autores dentro de um state, o que nos impede de passar o array diretamente para a tabela. Para obtermos esses dados, acessaremos a chave autores por meio do state.

render() {

return (

<div className="App">

<Tabela autores = { this.state.autores }/>

</div>

);

}

Feito isso, nosso código compilará com sucesso e continuará funcionando como deveria. A seguir, implementaremos a remoção de um elemento.

No decorrer do projeto, utilizamos state e props para compor a lógica da nossa aplicação, marque abaixo a alternativa que descreve a diferença entre esses dois objetos.

**State:** Serve para guardar valores/estado que podem vir a mudar com a interação do usuário com o componente, tendo efeito na renderização do mesmo, o state pode ser passado como props!

**Props:** Valores para a configuração de um componente, essas props são passadas pelo elemento acima (que utiliza o componente que irá receber) e são imutáveis, utilizado para a comunicação de componentes.

Alternativa correta! Esse é justamente o uso desses dois objetos, Props para configuração de componentes e State armazenando estado/valores!

# Método de remoção

Chegou o momento de implementarmos a funcionalidade de remoção de um autor. A ideia é que, a partir de um clique em um dos botões "Remover" nas nossas linhas, removermos toda a linha em questão.

Quando começamos a trabalhar com o state, que servirá para armazenarmos o estado da nossa aplicação, toda alteração de estado deverá alterá-lo. Toda vez que o state é alterado, ele automaticamente redesenha o nosso componente. Sendo assim, se utilizarmos um método para removermos um dos autores do state, nosso componente será redesenhado na tela automaticamente.

Nossa estratégia será justamente alterarmos o state, removendo um elemento do array autores, por meio de ume método chamado removeAutor(), que por enquanto receberá uma arrow function vazia.

removeAutor = () =>{

}

Queremos remover uma linha a partir de um botão que está sendo exibido na tabela. Sendo assim, precisaremos passar o método removeAutor() para nossa Tabela.js, o que pode ser feito por meio da adição de um prop removeAutor = { this.removeAutor } no método render().

render() {

return (

<div className="App">

<Tabela autores = { this.state.autores } removeAutor = { this.removeAutor } />

</div>

);

}

Como não é possível alterar um state diretamente, teremos que utilizar um método específico do React para efetuar essa alteração, chamado setState(). Esse método recebe um JSON que servirá para fazer o merge dos dados que já estão no state e aqueles que estão sendo passados. No caso, nosso objetivo é alterar a chave autores, da qual removeremos o elemento que sofreu o clique na tabela. Antes de efetuarmos a mudança, precisaremos do array autores atual, que conseguiremos criando uma constante autores recebendo this.state.

A partir de autores, chamaremos o método filter(), dentro do qual escreveremos um método que recebe o autor (a unidade dentro de autores) e a posAtual dessa iteração.

removeAutor = () =>{

const { autores } = this.state;

this.setState(

{

autores : autores.filter((autor, posAtual) => {

}),

}

);

}

Faremos um teste nesse método pedindo um console.log() de autor e posAtual.

removeAutor = () =>{

const { autores } = this.state;

this.setState(

{

autores : autores.filter((autor, posAtual) => {

console.log(autor, posAtual);

return posAtual;

}),

}

);

}

Na nossa tabela, receberemos o método removeAutor como prop e o passaremos, também como prop, para o TableBody.

class Tabela extends Component {

render() {

const { autores, removeAutor } = this.props;

return (

<table>

<TableHead />

<TableBody autores = { autores } removeAutor = {removeAutor} />

</table>

);

}

}

No botão "Remover", adicionaremos uma propriedade onClick recebendo uma arrow function que chamará pros.removeAutor().

const TableBody = props => {

const linhas = props.autores.map((linha, index) => {

return (

<tr>

<td>{linha.nome}</td>

<td>{linha.livro}</td>

<td>{linha.preco}</td>

<td><button onClick = {() => {props.removeAutor() }}>Remover</button></td>

</tr>

)

});

return(

<tbody>

{linhas}

</tbody>

);

}

Perceba que a única forma de reconhecermos qual elemento está sofrendo o clique é justamente pelo index, ou seja, a posição do autor no array, Sendo assim, passaremos esse index para o método removeAutor().

const linhas = props.autores.map((linha, index) => {

return (

<tr>

<td>{linha.nome}</td>

<td>{linha.livro}</td>

<td>{linha.preco}</td>

<td><button onClick = {() => {props.removeAutor(index) }}>Remover</button></td>

</tr>

)

});

Na definição do método, receberemos esse index e o exibiremos no console.log().

removeAutor = index =>{

const { autores } = this.state;

this.setState(

{

autores : autores.filter((autor, posAtual) => {

console.log(index, posAtual);

return;

}),

}

);

}

Feito isso, se clicarmos em algum dos botões "Remover" listados na tela, todos os items serão removidos. No log, receberemos a posição do item que foi clicado e a posição do momento de iteração do filter(). A ideia é excluirmos um autor quando a posição da iteração atual for igual ao index sendo recebido. Sendo assim, removeremos o console.log() e retornaremos posAtual quando ela for diferente do index.

removeAutor = index =>{

const { autores } = this.state;

this.setState(

{

autores : autores.filter((autor, posAtual) => {

return posAtual !== index;

}),

}

);

}

Agora, se clicarmos em algum dos botões "Remover", quando o index e a posAtual do filtro forem diferentes, o elemento será mantido no array. Entretanto, quanto eles coincidirem, o elemento será removido.

# Para saber mais: React Hooks

# React Hooks

A recente atualização do React 16.8 trouxe a possibilidade de utilizarmos state em function components.

A forma de fazer isso é utilizando o **useState**, lembrando que temos que importá-lo conforme abaixo:

import React, {useState} from 'react';

O useState nos retorna o state atual e uma função que deve ser utilizada para atualizar esse estado, e recebe o estado inicial do componente.

Veja um exemplo abaixo:

Vamos criar um componente que irá contar a quantidade de clicks realizados pelo usuário, primeiro vamos importar o que utilizaremos:

import React, { useState } from 'react';

Feito isso, vamos criar nosso componente!

function ContaClicks() {

return(

<div>

<p>Você clicou x vezes </p>

<button>clicar</button>

</div>

);

}

Nosso componente está criado, agora vamos utilizar o **useState** para usar state em um function component!

function ContaClicks() {

const [contador, setContador] = useState(0);

return(

<div>

<p>Você clicou x vezes </p>

<button>clicar</button>

</div>

);

}

Repare que recebemos um contador (estado atual), e um método para atualizá-lo, e o **useState** recebe 0 que é nosso estado inicial, feito isso, vamos fazer tudo acontecer!

function ContaClicks() {

const [contador, setContador] = useState(0);

return(

<div>

<p>Você clicou {contador} vezes </p>

<button onClick={() => setContador(contador + 1)}>clicar</button>

</div>

);

}

Feito isso nosso componente está completo! Nosso código completo é:

import React, { useState } from 'react';

function ContaClicks() {

const [contador, setContador] = useState(0);

return(

<div>

<p>Você clicou {contador} vezes </p>

<button onClick={() => setContador(contador + 1)}>clicar</button>

</div>

);

}

# Mãos na Massa

Chegou a hora de você pôr em prática o que foi visto na aula. Para isso, execute os passos listados abaixo.

1 - Modifique o Array autores para que ele seja armazenado no state

state = {

autores: [

{

nome: 'Paulo',

livro: 'React',

preco: '1000'

},

{

nome: 'Daniel',

livro: 'Java',

preco: '99'

},

{

nome: 'Marcos',

livro: 'Design',

preco: '150'

},

{

nome: 'Bruno',

livro: 'DevOps',

preco: '100'

}

]

}

2 - Feito isso, crie um método chamado removeAutor que recebe um index (posição) do autor em App.js

removeAutor = index => {

const { autores } = this.state;

this.setState({

autores : autores.filter((autor, posAtual) => {

return posAtual !== index;

}),

});

}

3 - Dentro do seu JSX, passe esse método como props para que ele seja utilizado pelo componente Tabela e corrija a passagem de autores, agora dentro de state:

<Tabela autores= {this.state.autores} removeAutor= {this.removeAutor} />

4 - Altere o seu componente Tabela para que ele receba o método removeAutor e passe também esse método para o TableBody:

class Tabela extends Component{

render(){

const { autores, removeAutor } = this.props;

return(

<table>

<TableHead />

<TableBody autores={autores} removeAutor = { removeAutor }/>

</table>

);

}

}

5 - Após isso, modifique o TableBody para que a remoção seja feita no click do botão:

<td><button onClick={ () => props.removeAutor(index)} >Remover</button></td>

6 - Após todas as alterações, veja se tudo funciona, abaixo você encontra como nosso código está no momento:

App.js

import React, { Component } from 'react';

import './App.css';

import Tabela from './Tabela';

class App extends Component {

state = {

autores : [

{

nome: 'Paulo',

livro: 'React',

preco: '1000'

},

{

nome: 'Daniel',

livro: 'Java',

preco: '99'

},

{

nome: 'Marcos',

livro: 'Design',

preco: '150'

},

{

nome: 'Bruno',

livro: 'DevOps',

preco: '100'

}

],

}

removeAutor = index => {

const { autores } = this.state;

this.setState({

autores : autores.filter((autor, posAtual) => {

return posAtual !== index;

}),

})

}

render() {

return (

<Tabela autores={this.state.autores} removeAutor = { this.removeAutor }/>

);

}

}

export default App;

Tabela.js

import React, { Component } from 'react';

const TableHead = () => {

return(

<thead>

<tr>

<th>Autor</th>

<th>Livro</th>

<th>Preco</th>

<th>Remover</th>

</tr>

</thead>

);

}

const TableBody = props =>{

const linhas = props.autores.map((linha, index)=>{

return(

<tr key={index}>

<td>{linha.nome}</td>

<td>{linha.livro}</td>

<td>{linha.preco}</td>

<td><button onClick={ () => props.removeAutor(index)} >Remover</button></td>

</tr>

);

});

return(

<tbody>

{linhas}

</tbody>

);

}

class Tabela extends Component{

render(){

const { autores, removeAutor } = this.props;

return(

<table>

<TableHead />

<TableBody autores={autores} removeAutor = { removeAutor }/>

</table>

);

}

}

export default Tabela;

# O que aprendemos?

Nesta aula, aprendemos:

* Trabalhar com state
* Método setState()
* Comportamento ao se alterar o state
* Método de remoção com filter

# Novo component para o formulário

No último vídeo nós conseguimos implementar a remoção de um autor, e agora trabalharemos com a inserção. Para isso, utilizaremos um formulário. Nosso instrutor preparou previamente um template de formulário para utilizarmos nessa aula:

<form>

<label for="nome">Nome</label>

<input

id="nome"

type="text"

name="nome"

/>

<label for="livro">Livro</label>

<input

id="livro"

type="text"

name="livro"

/>

<label for="preco">Preço</label>

<input

id="preco"

type="text"

name="preco"

/>

<button type="button">Salvar

</button>

</form>

No projeto, criaremos uma classe Formulario.js na qual importaremos o React e o Component. Essa classe obrigatoriamente terá um método render() que retornará o nosso template.

import React, { Component } from 'react'

class Formulario extends Component {

render() {

return (

<form>

<label for="nome">Nome</label>

<input

id="nome"

type="text"

name="nome"

/>

<label for="livro">Livro</label>

<input

id="livro"

type="text"

name="livro"

/>

<label for="preco">Preço</label>

<input

id="preco"

type="text"

name="preco"

/>

<button type="button">Salvar

</button>

</form>

)

}

}

export default Formulario;

Criaremos também um constructor(), afinal, como temos um formulário no qual dados serão inseridos, iremos alterar o estado de um elemento. Nesse construtor, receberemos os props e os passaremos para o Component com o super(). Criaremos então um stateInicial, um objeto com o estado inicial do nosso formulário, no qual todos os campos estarão vazios. Por fim, setaremos o state como esse stateInicial.

constructor(props){

super(props);

this.stateInicial = {

nome:'',

livro:'',

preco:'',

}

this.state = this.stateInicial;

}

Repare que no formulário temos a palavra for, que é uma palavra reservada do JavaScript, e que não poderemos utilizar no JSX. Sendo assim, vamos alterá-la para o equivalente htmlFor. Além disso, no render(), precisaremos pegar os valores de state para os adicionarmos no formulário. Para isso, criaremos as chaves nome, livro e preco, que pegaremos a partir de this.state.

Com isso, no formulário, conseguiremos informar os valores (value) dos inputs com suas respectivas chaves.

render() {

const {nome, livro, preco} = this.state;

return (

<form>

<label htmlFor="nome">Nome</label>

<input

id="nome"

type="text"

name="nome"

value={nome}

/>

<label htmlFor="livro">Livro</label>

<input

id="livro"

type="text"

name="livro"

value={livro}

/>

<label htmlFor="preco">Preço</label>

<input

id="preco"

type="text"

name="preco"

value={preco}

/>

<button type="button">Salvar

</button>

</form>

)

}

Feito isso, importaremos o formulário no nosso App.js com import Form from './Formulario'. No método render(), adicionaremos a tag <Form /> e removeremos a <div> que foi criada quando iniciamos o nosso projeto.

import React, {Component} from 'react';

import './App.css';

import Tabela from './Tabela';

import Form from './Formulario';

class App extends Component {

state = {

autores: [

{

nome: 'Paulo',

livro: 'React',

preco: '1000'

},

{

nome: 'Daniel',

livro: 'Java',

preco: '99'

},

{

nome: 'Marcos',

livro: 'Design',

preco: '150'

},

{

nome: 'Bruno',

livro: 'DevOps',

preco: '100'

},

{

nome: 'Nico',

livro: 'Java',

preco: '9999'

}

],

};

removeAutor = index =>{

const { autores } = this.state;

this.setState(

{

autores : autores.filter((autor, posAtual) => {

return posAtual !== index;

}),

}

);

}

render() {

return (

<Tabela autores = { this.state.autores } removeAutor = { this.removeAutor } />

<Form />

);

}

}

export default App;

Fazendo isso, teremos um erro de compilação, já que agora temos duas tags raiz - <Tabela /> e <Form /> - o que é inválido em uma linguagem de marcação. Porém, se começarmos a adicionar divs no código, vamos poluir nosso DOM com muitos elementos. Para solucionarmos isso, utilizaremos uma ferramenta do React chamada Fragment. Criaremos então um <Fragment> que conseguirá retornar tags irmãs sem necessariamente ter um wrapper que as renderize no HTML.

render() {

return (

<Fragment>

<Tabela autores = { this.state.autores } removeAutor = { this.removeAutor } />

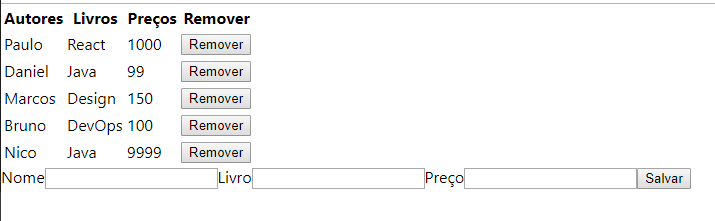
<Form />

</Fragment>

);

}

Feito isso, nossa tabela e o formulário serão renderizados com sucesso:



Entretanto, não conseguiremos preencher os campos do formulário. Além disso, receberemos um erro no console da página. No <input> do formulário, estamos setando o value como sendo igual ao nome que estamos recebendo do state. Já esse state está sendo setado pelo stateInicial, que é vazio. Sendo assim, não conseguiremos inserir novas informações nesses campos. Assim, quando o usuário for disparar um evento de mudança de estado, deveremos pegar a entrada do usuário e atualizar o estado do formulário de modo que ele consiga escrever nesse campo. Trabalharemos nisso no próximo vídeo.

# Resolvendo problema do formulário

Nós criamos um formulário, mas ainda não conseguimos digitar qualquer coisa nos campos desse componente. O motivo é que estamos o tempo todo setando o valor desses campos com o valor recebido de stateInicial, que é sempre vazio. Começaremos a resolver esse problema criando um método escutadorDeInput(). Esse método receberá um event, um parâmetro opcional do JavaScript, a partir do qual trabalharemos com uma arrow function.

A partir de event.target, pegaremos o name e o value do campo.

escutadorDeInput = event => {

const { name, value } = event.target;

}

A cada evento disparado de mudança do input, queremos atualizar o estado do componente para que essa mudança seja refletida nele e exibida na tela. Faremos isso por meio do setState(), utilizando uma notação do ECMAScript 6 para, a partir do name, colocar o value.

escutadorDeInput = event => {

const { name, value } = event.target;

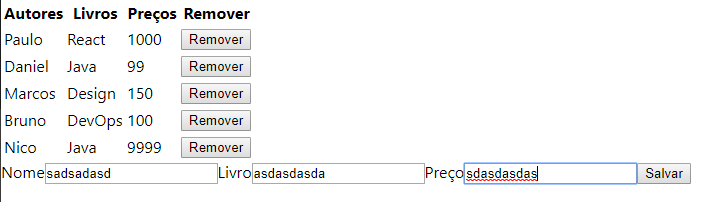
this.setState({

[name] : value

});

}

Agora precisamos acionar esse método escutadorDeInput() de acordo com o evento de mudança do formulário. No render(), adicionaremos um atributo onChange que chama this.escutadorDeInput em todos os nossos campos. Feito isso, nosso código compilará com sucesso, sem erros no console, e conseguiremos escrever nos campos do formulário.



No próximo vídeo aprenderemos a salvar um novo autor no nosso state para que ele seja refletido no componente da tabela.

# Criando um autor na tabela

Já resolvemos o problema do nosso formulário e conseguimos entrar com valores nos campos, mas ainda precisamos fazer com que o botão de "Salvar" funcione, adicionando um novo autor à tabela. Quando alteramos o state de um componente, ele se redesenha. Sendo assim, quando adicionarmos um novo autor na chave autores, nosso componente se redesenhará com esse autor redesenhado.

Em App.js, criaremos um método escutadorDeSubmit() que receberá a unidade autor e setará o estado do nosso componente utilizando o *spread operator*, adicionando a unidade autor que foi recebida.

escutadorDeSubmit = autor => {

this.setState({ autores:[...this.state.autores, autor]})

}

Por fim, passaremos esse método como props do nosso formulário.

render() {

return (

<Fragment>

<Tabela autores = { this.state.autores } removeAutor = { this.removeAutor } />

<Form escutadorDeSubmit={this.escutadorDeSubmit}/>

</Fragment>

);

}

Voltando a Formulario.js, criaremos um método submitFormulario() que irá, a partir do props, chamar o escutadorDeSubmit(), passando como parâmetro this.state, enviando as informações do novo autor para o App.js. Em seguida, zeraremos os campos desse formulário setando-o comthis.stateInicial.

submitFormulario = () => {

this.props.escutadorDeSubmit(this.state);

this.setState(this.stateInicial);

}

Esse método deverá ser disparado no clique do botão "Salvar". Faremos essa definição com um atributo onClick que recebe this.submitFormulario.

<button type="button" onClick={this.submitFormulario}>Salvar

</button>

Feito isso, conseguiremos inserir um novo autor por meio do nosso formulário. No próximo capítulo trabalharemos no visual da nossa aplicação.

# Mãos na Massa

Chegou a hora de você pôr em prática o que foi visto na aula. Para isso, execute os passos listados abaixo.

1 - Crie um novo arquivo chamado Formulario.js.

2 - O componente formulário será um Class Component, crie conforme abaixo:

import React, { Component } from ‘react’;

class Formulario extends Component {

constructor(props){

super(props);

}

}

export default Formulario;

3 - Crie um estado inicial no construtor do Formulario que será o estado atual da aplicação.

import React, { Component } from ‘react’;

class Formulario extends Component {

constructor(props){

super(props);

this.stateInicial = {

nome: ‘’,

livro: ‘’,

preco: ‘’

}

this.state = this.stateInicial;

}

}

export default Formulario;

4 - Crie seu método render retornando o devido JSX conforme abaixo:

render() {

return (

<form>

<label htmlFor="nome">Nome</label>

<input

id="nome"

type="text"

name="nome"

/>

<label htmlFor="livro">Livro</label>

<input

id="livro"

type="text"

name="livro"

/>

<label htmlFor="preco">Preço</label>

<input

id="preco"

type="text"

name="preco"

/>

<button type="button">Salvar

</button>

</form>

);

}

5 - Crie os métodos escutadorDeInput e submitFormulario e edite seu render para que os dados inputados pelo usuário sejam refletidos na tela.

Repare na constante criada no render.

render() {

const { nome, livro, preco } = this.state;

return (

<form>

<label htmlFor="nome">Nome</label>

<input

id="nome"

type="text"

name="nome"

value={nome}

onChange={this.escutadorDeInput} />

<label htmlFor="livro">Livro</label>

<input

id="livro"

type="text"

name="livro"

value={livro}

onChange={this.escutadorDeInput} />

<label htmlFor="preco">Preço</label>

<input

id="preco"

type="text"

name="preco"

value={preco}

onChange={this.escutadorDeInput} />

<button onClick={this.submitFormulario} type="button">Salvar

</button>

</form>

);

}

Métodos:

submitFormulario = () => {

this.props.escutadorDeSubmit(this.state);

this.setState(this.stateInicial);

}

escutadorDeInput = event => {

const { name, value } = event.target;

this.setState({

[name]: value

});

}

6 - O arquivo final deve ser:

import React, { Component } from 'react';

class Formulario extends Component {

constructor(props) {

super(props);

this.stateInicial = {

nome: '',

livro: '',

preco: ''

}

this.state = this.stateInicial;

}

submitFormulario = () => {

this.props.escutadorDeSubmit(this.state);

this.setState(this.stateInicial);

}

escutadorDeInput = event => {

const { name, value } = event.target;

this.setState({

[name]: value

});

}

render() {

const { nome, livro, preco } = this.state;

return (

<form>

<label htmlFor="nome">Nome</label>

<input

id="nome"

type="text"

name="nome"

value={nome}

onChange={this.escutadorDeInput} />

<label htmlFor="livro">Livro</label>

<input

id="livro"

type="text"

name="livro"

value={livro}

onChange={this.escutadorDeInput} />

<label htmlFor="preco">Preço</label>

<input

id="preco"

type="text"

name="preco"

value={preco}

onChange={this.escutadorDeInput} />

<button onClick={this.submitFormulario} type="button">Salvar

</button>

</form>

);

}

}

export default Formulario

7 - Agora abra o arquivo App.js, importe o formulário e utilize-o no JSX

import Formulario from ‘./Formulario’;

render() {

return (

<div>

<Tabela autores={this.state.autores} removeAutor = { this.removeAutor }/>

<Formulario />

</div>

);

}

8 - Agora será necessário criar o método escutadorDeSubmit que deverá ser passado como props para o Formulário, esse método alterará o state autores.

escutadorDeSubmit = autor => {

this.setState({ autores: [...this.state.autores, autor] });

}

render() {

return (

<div>

<Tabela autores={this.state.autores} removeAutor = { this.removeAutor }/>

<Formulario escutadorDeSubmit={this.escutadorDeSubmit}/>

</div>

);

}

9 - Feito isso, teste tudo no navegador!

# O que aprendemos?

Nesta aula, aprendemos:

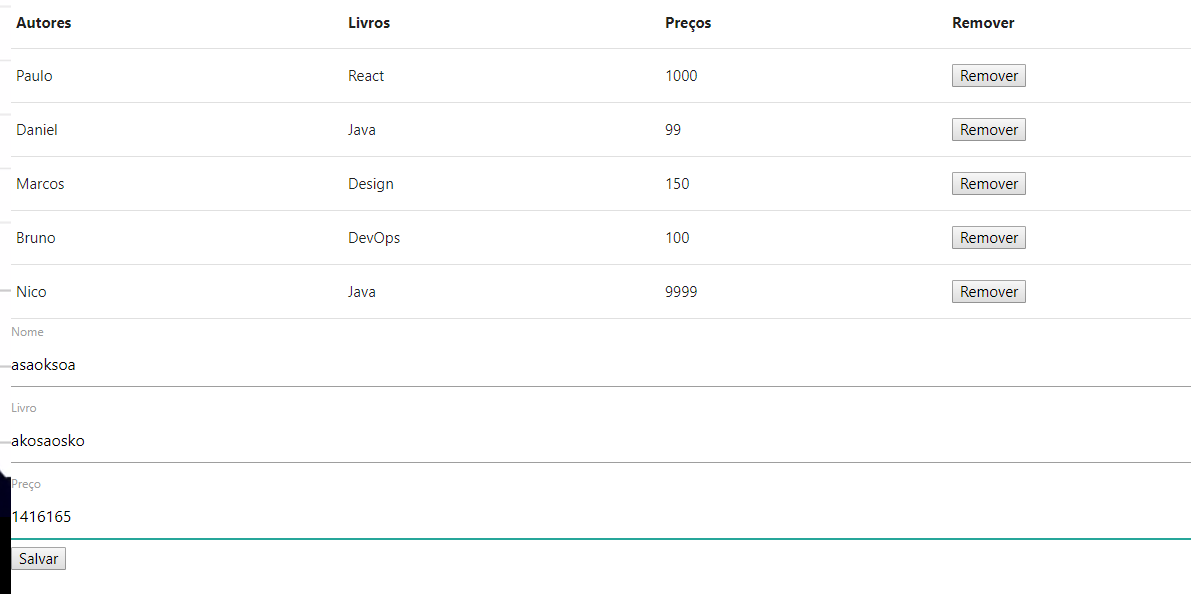
* Trabalhar com formulários
* Utilizando o evento onChange
* Submissão de formulário
* Criar um elemento dinamicamente

# Utilizando Materialize

Na última aula implementamos a inserção de elementos em uma tabela, começaremos a estilizar o nosso projeto, deixando-o com uma aparência mais profissional. Para isso, utilizaremos uma biblioteca de estilos chamada [Materialize](https://materializecss.com/), que instalaremos por meio do npm.

No Prompt, finalizaremos a execução da nossa aplicação e executaremos npm install materialize-css@1.0.0, instalando a versão mais atual na publicação desse curso. Na documentação, é possível ter acesso às diversas funcionalidades e exemplos de como utilizá-las.

Voltando ao projeto, ao invés de importarmos o arquivo App.css do diretório do projeto, importaremos materialize-css/dist/css/materialize.min.css. Subindo novamente a aplicação e acessando [http://localhost:3000](http://localhost:3000/), nossa aplicação terá ganhado uma nova aparência.



No próximo vídeo começaremos a utilizar classes para trabalhar com esse estilo.

**Novo component e estilo**

Na última aula aprendemos a fazer a instalação do Materialize via npm, e conseguimos acesso a essa folha de estilo no nosso projeto. Feitas as importações, nossa aplicação já ganhou uma aparência nova. Nosso objetivo agora é utilizarmos as classes do Materialize para estilar o nosso Front End, a começar pelo cabeçalho.

De início, usaremos o modelo *Right Aligned Links*, que alinha os links à direita.

<nav>

<div class="nav-wrapper">

<a href="#" class="brand-logo">Logo</a>

<ul id="nav-mobile" class="right hide-on-med-and-down">

<li><a href="sass.html">Sass</a></li>

<li><a href="badges.html">Components</a></li>

<li><a href="collapsible.html">JavaScript</a></li>

</ul>

</div>

</nav>

Na pasta "src", criaremos um componente de cabeçalho Header.js no qual importaremos o React. Como este será um componente de função, criaremos uma constante Header recebendo uma *arrow function* que retornará o template anterior, no qual faremos algumas modificações: removeremos a id, substituiremos todos os links por cerquilha (#) e removerremos o hide-on-med-and-down, que esconde os links em resoluções mais baixas. Por fim, exportaremos esse cabeçalho com export default Header.

import React from 'react';

const Header = () => {

return (

<nav>

<div class="nav-wrapper">

<a href="#" class="brand-logo">Logo</a>

<ul class="right">

<li><a href="#">Sass</a></li>

<li><a href="#">Components</a></li>

<li><a href="#">JavaScript</a></li>

</ul>

</div>

</nav>

);

}

export default Header;

Em App.js, recuperemos esse cabeçalho importando-o.

import React, {Component, Fragment} from 'react';

import 'materialize-css/dist/css/materialize.min.css';

import Header from './Header';

//...///

render() {

return (

<Fragment>

<Header />

<Tabela autores = { this.state.autores } removeAutor = { this.removeAutor } />

<Form escutadorDeSubmit={this.escutadorDeSubmit}/>

</Fragment>

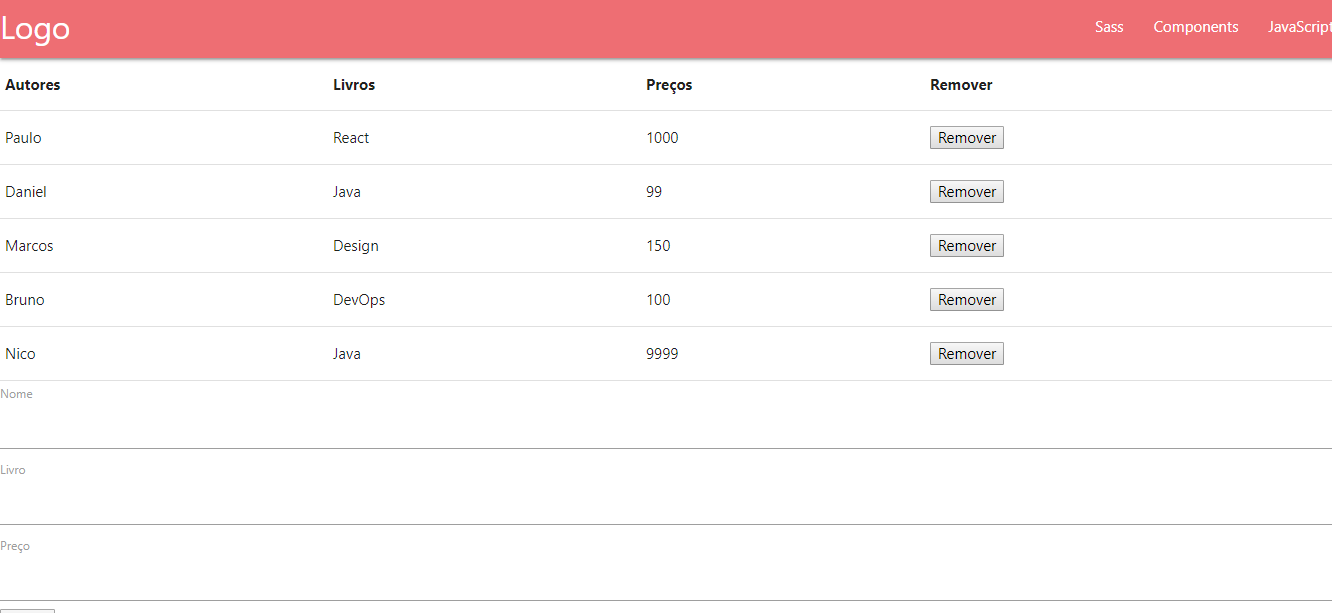
);

}

}

export default App;

Feito isso, nossa aplicação ganhará um cabeçalho com "Sass", "Components" e "JavaScript" no canto direito e "Logo" no canto esquerdo.



Em Header.js, trocaremos essa última palavra por "Casa do Código"; "Sass" será substituído por "Autores"; "Component" por "Livros"; e "Javascript" por "Sobre".

import React from 'react';

const Header = () => {

return (

<nav>

<div class="nav-wrapper">

<a href="#" class="brand-logo">Casa do Código</a>

<ul class="right">

<li><a href="#">Autores</a></li>

<li><a href="#">Livros</a></li>

<li><a href="#">Sobre</a></li>

</ul>

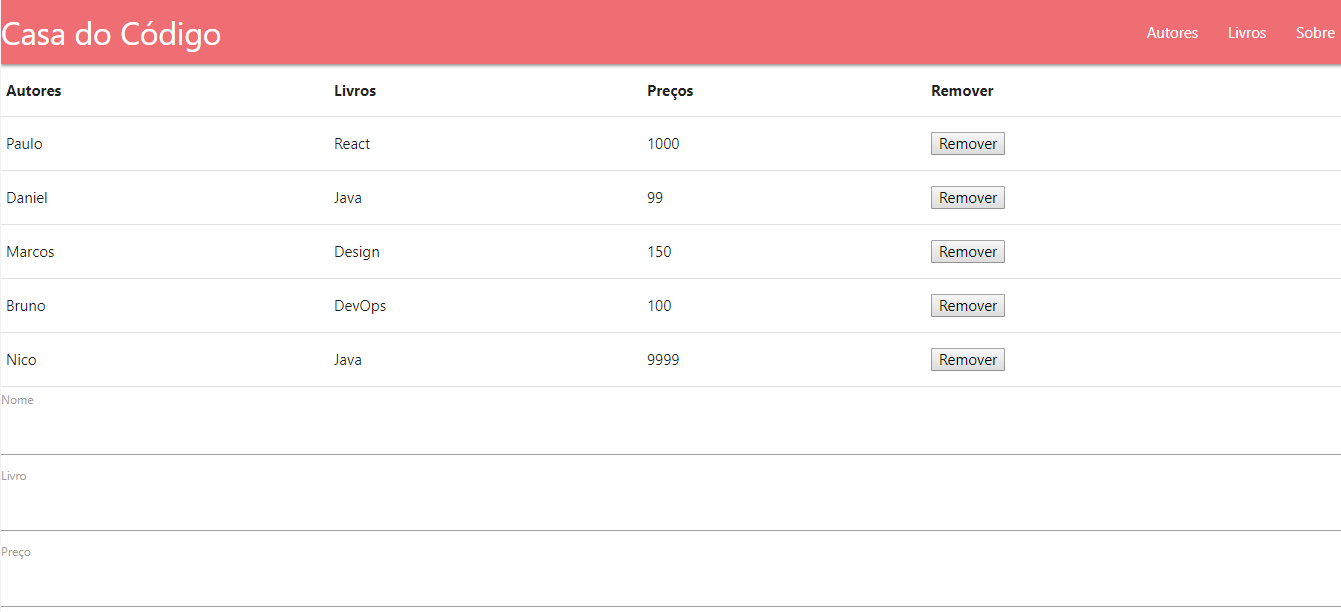
</div>

</nav>

);

}

export default Header;



Também trocaremos a cor do cabeçalho por um tom de azul identificado comoindigo lighten-2.

const Header = () => {

return (

<nav>

<div class="nav-wrapper indigo lighten-2">

<a href="#" class="brand-logo">Casa do Código</a>

<ul class="right">

<li><a href="#">Autores</a></li>

<li><a href="#">Livros</a></li>

<li><a href="#">Sobre</a></li>

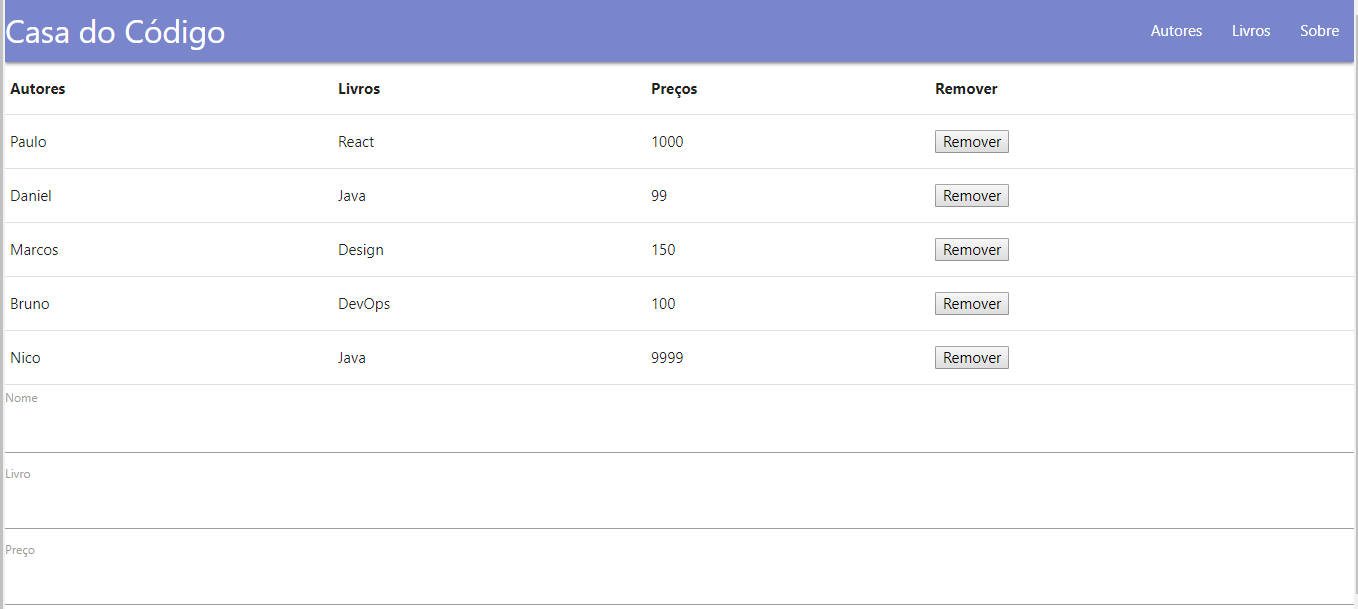
</ul>

</div>

</nav>

);

}



Partindo para a estilização da tabela, adicionaremos centralização e uma função que escure a linha quando passamos o mouse sobre ela. Faremos isso com className="centered highlight", utilizando classNamejá que class é uma palavra reservada do JavaScript. Também alteraremos o botão "Remover" com className="waves-effect waves-light indigo lighten-2 btn", fazendo com que ele ganhe uma estilização, seja realçado ao passar do mouse e tenha a mesma cor do cabeçalho.

const TableBody = props => {

const linhas = props.autores.map((linha, index) => {

return (

<tr key={index}>

<td>{linha.nome}</td>

<td>{linha.livro}</td>

<td>{linha.preco}</td>

<td><button onClick = {() => {props.removeAutor(index) }} className="waves-effect waves-light indigo lighten-2 btn">Remover</button></td>

</tr>

)

});

return(

<tbody>

{linhas}

</tbody>

);

}

class Tabela extends Component {

render() {

const { autores, removeAutor } = this.props;

return (

<table class="centered highlight">

<TableHead />

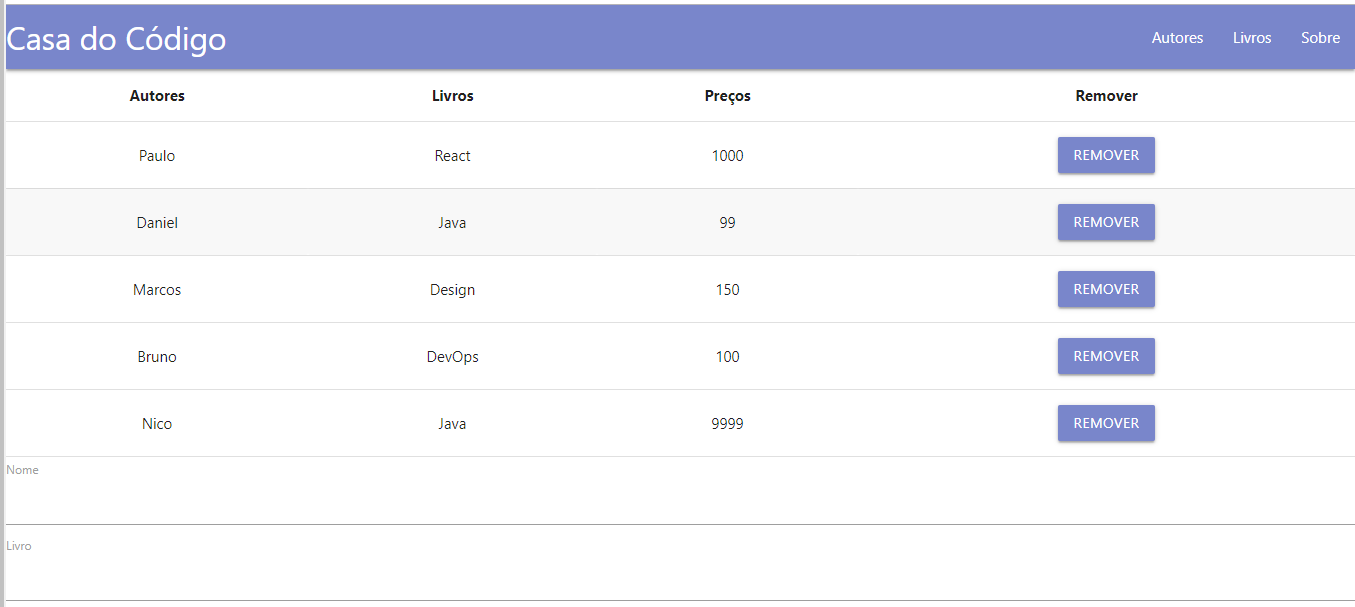
<TableBody autores = { autores } removeAutor = {removeAutor} />

</table>

);

}

}



Terminada a estilização da tabela, trabalharemos também com o formulário. Podemos utilizar um className="row" para posicionarmos esses elementos na mesma linha e ocupando exatamente o mesmo espaço. Para isso, em Formulario.js, englobaremos todos os campos, exceto o botão "Salvar", em uma <div> com esse clasName. Em seguida, cada um dos campos receberá individualmente uma <div className="input-field col s4">, definindo que são campos de entrada e que eles devem ocupar 4 posições (em um grid de 12). Cada <label> também receberá uma className="input-field".

render() {

const { nome, livro, preco } = this.state;

return (

<form>

<div className="row">

<div className="input-field col s4">

<label className="input-field" htmlFor="nome">Nome</label>

<input

id="nome"

type="text"

name="nome"

value={nome}

onChange={this.escutadorDeInput}

/>

</div>

<div className="input-field col s4">

<label className="input-field" htmlFor="livro">Livro</label>

<input

id="livro"

type="text"

name="livro"

value={livro}

onChange={this.escutadorDeInput}

/>

</div>

<div className="input-field col s4">

<label className="input-field" htmlFor="preco">Preço</label>

<input

id="preco"

type="text"

name="preco"

value={preco}

onChange={this.escutadorDeInput}

/>

</div>

</div>

<button onClick={this.submitFormulario} type="button">Salvar

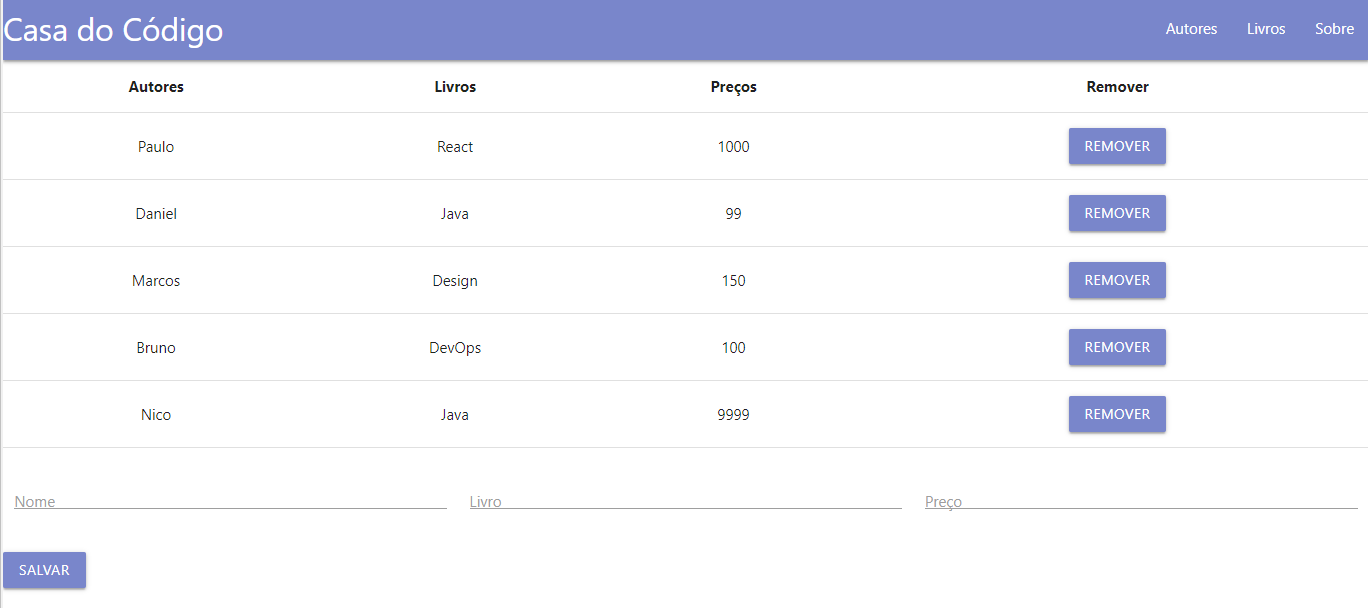
</button>

</form>

)

}

Com className="validade", podemos incluir uma validação que mostrará ao usuário quando algum input for inválido. Por fim, adicionaremos className="waves-effect waves-light indigo lighten-2 btn" ao botão "Salvar".



Assim, terminamos a estilização da nossa página, mas ainda podemos fazer alguns ajustes finos. Em App.js, colocaremos nossa <Tabela /> e nosso <Formulario /> dentro de uma <div className="container">, centralizando os elementos da aplicação. Porém, o botão "Salvar" ficará no extremo inferior da página. Para não precisarmos criar outro elemento, importaremos o App.css novamente e, nesse arquivo, criaremos uma classe .mb-10 na qual aplicaremos um margin-bottom com 1 em.

.mb-10{

margin-bottom: 1em;

}

Em seguida, adicionaremos essa classe ao nosso container:

<div className="container mb-10">

Assim, ganharemos algum espaço entre o botão "Salvar" e a parte inferior da página. Nossa aplicação continuará funcionando normalmente, incluindo as operações de adicionar ou remover items, mas o terminal indicará um problema em Header.js, informando que o nosso atributo href não é válido. A documentação do JavaScript diz que strings do JS, valores vazios ou cerquilhas não são considerados valores válidos para o href, e precisaremos alterá-los para que o terminal deixe de nos mostrar avisos. Sendo assim, paremos um endereço a cada um desses campos.

const Header = () => {

return (

<nav>

<div class="nav-wrapper indigo lighten-2">

<a href="/" class="brand-logo">Casa do Código</a>

<ul class="right">

<li><a href="/autores">Autores</a></li>

<li><a href="/livros">Livros</a></li>

<li><a href="/sobre">Sobre</a></li>

</ul>

</div>

</nav>

);

}

export default Header;

Dessa forma, nosso código passará a compilar corretamente.

# class x className

# class e className funcionaram?

No último vídeo, vimos que no componente **Header.js**, mantivemos a utilização do **class** para a atribuição de classes CSS, enquanto nos outros, realizamos as mudanças para **className**.

O problema inicial da utilização de **class** para definição de estilos de componentes era devido ao fato de que **class** no Javascript é uma palavra reservada.

Vamos ao problema:

//Válido em ES5

//Inválido em ES3

var props = { class: 'foo' };

Essa situação gerava problemas de suporte para o IE8, que tornava o uso de **class** um problema.

Porém, a maioria de nós não escreve mais código usando ES3, ou utiliza o Babel para transpilar seu código ou está 'mirando' no IE9+, o React em si não suporta o IE8.

Então, o único problema com **class** agora seria:

// não é válido de jeito nenhum

const class = props.class;

const { class } = props;

Porém, com **destructuring**, podemos fazer:

// Válido

const {class: cls} = foo;

const cls = props.class;

O que acontece é que normalmente, passamos mais tempo escrevendo classes em componentes do que lendo essas classes. Um exemplo é que teríamos que escrever <div className...> o tempo todo. Com o suporte ao **class**, conseguimos salvar algum tempo de digitação, e como sabemos, tempo é dinheiro.

## Mas por que então className?

Os componentes React utilizavam as propriedades do DOM via API do Javascript, e para isso, era necessário acessar utilizando a propriedade **className** e não **class**

ex:

node.className

Durante o curso, optamos por seguir com a utilização do **className** por ser a forma recomendada no [site](https://reactjs.org/docs/faq-styling.html#how-do-i-add-css-classes-to-components) da biblioteca.

Caso queira saber mais, você pode verificar [esse link](https://github.com/facebook/react/issues/13525#issuecomment-417818906).

# Resumo do curso

No último vídeo nós terminamos a estilização do nosso projeto, e agora faremos uma recapitulação de tudo que aprendemos até o momento. Começamos fazendo a instalação do create-react-app, uma ferramenta do Facebook que gera um projeto automaticamente,. Após analisarmos o que foi criado, passamos a trabalhar na pasta "src".

Aprendemos a diferença de um *function component*, um componente mais simples, e de um *class component*, que, além de ser mais robusto, obrigatoriamente possui um método render() que é chamado toda vez que o estado desse componente é alterado. Ainda aprendemos a criar e utilizar componentes em diversos pontos do nosso código, exportando-os e importando-os quando necessário - lembrando que não é necessário passar uma extensão nessa importação, um açúcar sintático disponibilizado pelo React.

Partindo de uma tabela com grandes componentes, conseguimos dividi-los em subcomponentes - mais especificamente, um TableBody e um TableHead -, e aprendemos a passar informações entre os componentes por meio dos props. Também aprendemos a guardar informações de estado da nossa aplicação para alterar a tabela, a criar e a enviar em um formulário por meio de um input do usuário.

# Para saber mais: Styled Components

# Styled Components

Quando começamos a desenvolver aplicações utilizando **React**, começamos também a perceber uma gama de módulos e ferramentas criadas pela comunidade para facilitar nossa vida.

Um desses módulos ataca exatamente a forma com que estilizamos nossos componentes no **React**

## O que é o Styled Component

É um framework de estilização via JavaScript (CSS In JS) que utiliza template string (template literals) e CSS provendo uma plataforma onde é possível estilizar seus componentes usando CSS. Ou seja, é possível estilizar seus componentes usando CSS dentro de seu código!

## Instalando o módulo Styled Components

Para fazer a instalação, abra o terminal no diretório do seu projeto e execute o comando:

npm install styled-components

Note: É recomendado que você instale um plugin para styled-components no seu editor de texto.

## Criando seu primeiro componente estilizado

1 - Antes de tudo, é necessário importar o módulo no seu arquivo.

import styled from 'styled-components';

2 - Feito isso, crie um componente Button

const Button = styled.button``;

É dentro desses (**`**) acentos graves que entrará o código CSS

3 - Estilize seu button da forma que desejar.

OBS: Podemos criar qualquer componente utilizando styled.TAG

4 - Continue a estilização do componente.

const Button = styled.button`

background: transparent;

border-radius: 3px;

border: 2px solid palevioletred;

color: palevioletred;

margin: 0.5em 1em;

padding: 0.25em 1em;

`;

5 - Podemos também adaptar o nosso componente criando funções dentro das template literals, veja:

const Button = styled.button`

background: transparent;

border-radius: 3px;

border: 2px solid palevioletred;

color: palevioletred;

margin: 0.5em 1em;

padding: 0.25em 1em;

${props => props.primary && css`

background: palevioletred;

color: white;

`}

`;

Acima estamos dizendo que queremos mudar a cor de background caso a exista a propriedade primary

render(

<div>

<Button>Normal Button</Button>

<Button primary>Primary Button</Button>

</div>

);

6 - Podemos criar mais um componente para centralizar os botões, veja o código completo abaixo:

const Button = styled.button`

background: transparent;

border-radius: 3px;

border: 2px solid palevioletred;

color: palevioletred;

margin: 0.5em 1em;

padding: 0.25em 1em;

${props => props.primary && css`

background: palevioletred;

color: white;

`}

`;

const Container = styled.div`

text-align: center;

`

//MÉTODO RENDER

render(

<Container>

<Button>Normal Button</Button>

<Button primary>Primary Button</Button>

</Container>

# Mãos na Massa

Chegou a hora de você pôr em prática o que foi visto na aula. Para isso, execute os passos listados abaixo.

1 - Faça a instalação do materialize executando a linha abaixo no seu terminal

npm install materialize-css@1.0.0

2 - Importe a folha de estilos do materialize no arquivo App.js.

import 'materialize-css/dist/css/materialize.min.css'

3 - Crie um novo arquivo chamado Header.js

4 - Crie um Function component que retorne o Header da aplicação com todos os estilos.

import React from 'react';

const Header = () =>{

return (

<nav>

<div className="nav-wrapper indigo lighten-2">

<a href="/" className="brand-logo ml-3">Casa do Código</a>

<ul id="nav-mobile" className="right">

<li><a href="/">Autores</a></li>

<li><a href="/">Livros</a></li>

<li><a href="/">Sobre</a></li>

</ul>

</div>

</nav>

);

}

export default Header;

5 - Importe seu Header no App.js e utilize-o no seu template

import Header from ‘./Header’;

<Header />

6 - No arquivo App.css, crie uma nova classe para o espaçamento inferior do botão de submit.

.mb-10{

margin-bottom:1em;

}

8- Importe o arquivo App.css no projeto.

import './App.css';

Utilize essa classe criada no App.js

8 - Estilize os outros componentes da aplicação:

App.js

<Fragment>

<Header />

<div className="container mb-10">

<h1>Casa do código</h1>

<Tabela autores={this.state.autores} removeAutor={this.removeAutor} />

<Formulario escutadorDeSubmit={this.escutadorDeSubmit} />

</div>

</Fragment>

Tabela.js

//TABELA

<table className="centered highlight">

<TableHead />

<TableBody autores={autores} removeAutor = { removeAutor }/>

</table>

// TABLEBODY

<tr key={index}>

<td>{linha.nome}</td>

<td>{linha.livro}</td>

<td>{linha.preco}</td>

<td><button onClick={ () => props.removeAutor(index)} className="waves-effect waves-light indigo lighten-2 btn">Remover</button></td>

</tr>

//TABLEHEAD

<thead>

<tr>

<th>Autor</th>

<th>Livro</th>

<th>Preco</th>

<th>Remover</th>

</tr>

</thead>

Formulario.js

<form>

<div className="row">

<div className="input-field col s4">

<label className="input-field" htmlFor="nome">Nome</label>

<input

className="validate"

id="nome"

type="text"

name="nome"

value={nome}

onChange={this.escutadorDeInput} />

</div>

<div className="input-field col s4">

<label className="input-field" htmlFor="livro">Livro</label>

<input

className="validate"

id="livro"

type="text"

name="livro"

value={livro}

onChange={this.escutadorDeInput} />

</div>

<div className="input-field col s4">

<label className="input-field col s4" htmlFor="preco">Preço</label>

<input

className="validate"

id="preco"

type="text"

name="preco"

value={preco}

onChange={this.escutadorDeInput} />

</div>

</div>

<button onClick={this.submitFormulario} className="btn waves-effect waves-light indigo lighten-2" type="button">Salvar

</button>

</form>

9 - Após estilizado, verifique tudo no navegador!

# O que aprendemos?

Nesta aula, aprendemos:

* Instalação do módulo do materialize-css.
* Importação de um arquivo de estilo.
* Utilizar o materialize.
* Estilizar o projeto utilizando className para atribuição de classes.

**Conclusão**

Chegamos ao fim da primeira parte do nosso curso de React. Esperamos que você tenha gostado e tirado bastante proveito das informações passadas nesse curso. Se tiver qualquer dúvida, não deixe de procurar nossos instrutores, moderadores e alunos lá no fórum.

Bons estudos e até a próxima!

**Introdução**

Seja bem vindo(a) à parte 2 do nosso curso de React, na qual, utilizando o React, continuaremos trabalhando na aplicação da Casa do Código, criando uma listagem de autores e livros (com notificações na navegação entre telas), uma página de "Sobre", persistência dos dados inseridos ou removidos, validação dos campos do formulário, entre outras funcionalidades.

Bons estudos!

# Instalação do módulo Validator

Atualmente, se clicarmos no botão "Salvar" da nossa aplicação sem passarmos nenhuma informação, criaremos uma linha (autor, livro e preço) sem nenhum valor. Seria interessante fazermos uma validação desses campos para só então criarmos um item na tabela. Implementaremos essa lógica de validação utilizando um pacote do npm chamado validator.js, cuja última versão é a 10.11.0.

No Prompt, executaremos npm install validator@10.11.0 para instalarmos esse pacote no projeto. Na [página do Validator](https://www.npmjs.com/package/validator) é possível encontrar várias informações, como os métodos auxiliares que nos ajudarão a validar os dados. Ao longo do curso, usaremos o isEmpty() para verificarmos se realmente houve uma entrada no campo, e isInt() para verificarmos se o valor enviado para o input de preço é um inteiro.

# Começando nossa validação

Concluída a instalação do Validator, começaremos a implementar a validação no nosso projeto. A ideia é validarmos algo que acontece no nosso Formulario.js, mais especificamente em submitFormulario(), permitindo o envio de informações apenas se os campos estiverem preenchidos. Em "src", criaremos uma classe FormValidator.js para isolar o código de validação de modo que possamos utilizar essa funcionalidade em outros pontos no futuro, caso isso seja necessário. Nosso objetivo é importar o validador e travar o submit caso ele esteja em branco.

Nessa classe, teremos um método valida() que recebe um state e, por enquanto, executa um console.log() com a mensagem "validado", além de retornar um false. Por fim, exportaremos o FormValidator.

import validador from 'validator';

class FormValidator {

valida(state) {

console.log('validado');

return false;

}

}

export default FormValidator;

Em Formulario.js, importaremos o FormValidator e, no construtor, criaremos uma instância dessa classe.

constructor(props) {

super(props);

this.validador = new FormValidator();

this.stateInicial = {

nome: '',

livro: '',

preco: '',

}

this.state = this.stateInicial;

}

No método submitFormulario(), usaremos o operador if para verificarmos o retorno da chamada do método valida(). Quando verdadeiro, o botão "Salvar" do formulário enviará as informações, adicionando um elemento na tela. Do contrário (else), escreveremos uma mensagem "Submit Bloqueado" na tela com um console.log().

Subindo novamente nossa aplicação, teremos um aviso dizendo que, apesar da compilação ter ocorrido com sucesso, existe um validador que não é utilizado. Por enquanto não nos preocuparemos com isso. Se acessarmos <http://localhost:3000/> e tentarmos salvar sem preenchermos nenhum dos campos, nenhum item será adicinado à lista, e receberemos a mensagem "Submit Bloqueado" no console. Já se preenchermos os campos...a mesma coisa irá ocorrer. Solucionaremos esse problema no próximo vídeo.

# Validando um campo

No último vídeo nós conseguimos travar o formulário retornando false na classe FormValidator, e agora passaremos a utilizar o pacote Validator do Node para validarmos os nossos campos, começando pelo nome.

Em Formulario.js, a instanciação de FormValidator receberá um objeto JSON com a regra de validação do campo, composta pelo nome do campo e o método que queremos utilizar do Validator, por exemplo isEmpty().

this.validador = new FormValidator({

campo:'nome',

metodo:'isEmpty'

});

Em submitFormulario(), removeremos o else por enquanto. No FormValidator.js, criaremos um constructor() que receberá a validacao.

class FormValidator {

constructor(validacao){

this.validacao = validacao;

}

valida(state) {

console.log('validado');

return false;

}

}

export default FormValidator;

No método valida(), recuperaremos o valor do campo (campoValor) a partir de state[this.validacao.campo.toString()] - ou seja, pegaremos o valor recebido no campo e parseando os dados para string, pois a biblioteca Validator trabalha somente com strings. Em seguida, recuperaremos o método sendo passado pelo validador (metodoValidacao) a partir de validador[this.validacao.metodo].

class FormValidator {

constructor(validacao){

this.validacao = validacao;

}

valida(state) {

const campoValor = state[this.validacao.campo.toString()];

const metodoValidacao = validador[this.validacao.metodo];

}

}

export default FormValidator;

Repare que estamos passando o método isEmpty como uma string. Isso é feito para, ao invés de importarmos um módulo em vários pontos do código, o encapsularmos em um único arquivo e, de algum modo, informar à classe qual método deverá ser utilizado.

Continuando, usaremos o operador if para verificarmos se o retorno de metodoValidacao(campoValor, [], state) é true. Em caso positivo, nosso formulário é inválido, e escreveremos a mensagem "FORM INVÁLIDO" com console.log(), retornando false. Do contrário, escreveremos "FORM VÁLIDO" e retornaremos true.

Nota: o parâmetro [] consiste em um array vazio de configurações opcionais dos métodos.

class FormValidator {

constructor(validacao){

this.validacao = validacao;

}

valida(state) {

const campoValor = state[this.validacao.campo.toString()];

const metodoValidacao = validador[this.validacao.metodo];

if(metodoValidacao(campoValor, [], state) === true){

console.log("FORM INVÁLIDO");

return false;

}else{

console.log("FORM VÁLIDO");

return true;

}

}

}

export default FormValidator;

O trecho de código === true está presente apenas por questões didáticas, e pode ser omitido sem qualquer interferência no funcionamento da aplicação.

Feito isso, se clicarmos no botão "Salvar" da nossa aplicação sem preenchermos nenhum dos campos, receberemos a mensagem "FORM INVÁLIDO" no console e nenhum item será adicionado. Se preenchermos os campos, os dados serão adicinados corretamente e a mensagem "FORM VÁLIDO" será exibida no console.

No próximo vídeo, faremos nosso código funcionar para todos os campos.

# Mãos na Massa

Chegou a hora de você pôr em prática o que foi visto na aula. Para isso, execute os passos listados abaixo.

1 - No arquivo **Formulario.js**, altere o **constructor** inicializando os seguintes atributos:

constructor(props){

super(props);

this.validador = new FormValidator({

campo: 'nome',

metodo: 'isEmpty'

});

}

2 - Altere o método **submitFormulario**:

submitFormulario = () => {

if (this.validador.valida(this.state)){

//Lógica para inserção na tabela

}else{

console.log('submit bloqueado');

}

}

3 - Crie o arquivo **FormValidator.js**.

import validador from 'validator';

class FormValidator {

constructor(validacao) {

this.validacao = validacao;

}

valida(state) {

const campoValor = state[this.validacao.campo.toString()];

const metodoValidacao = validador[this.validacao.metodo];

if (metodoValidacao(campoValor, [], state)) {

console.log('FORM INVALIDO');

return false;

} else {

console.log('FORM VALIDO');

return true;

}

}

}

export default FormValidator;

4 - Importe o novo **FormValidator** no seu **Formulario.js**.

import FormValidator from './FormValidator';

5 - Teste no seu navegador!

# O que aprendemos?

Nesta aula, aprendemos:

* Como utilizar o módulo validator.
* Estratégia para concentrar a lógica e importação de módulos para validação.
* Criar uma classe sem necessariamente criar um componente.
* Lógica para validação de campos.

# Definindo regras genéricas

No último vídeo conseguimos validar o campo "Nome", e nosso objetivo agora é generalizarmos o código, validando todos os campos. Para começarmos esse processo, ao invés de passarmos somente um JSON para o campo nome, passaremos um array de JSONs, um para cada campo.

Além disso, passaremos no JSON uma informação validoQuando que nos permitirá saber quando o retorno de uma verificação é válida ou não. No caso, quando utilizarmos o método isEmpty(), o retorno true será inválido, e o false válido. Entretanto, quando o método utilizado for isInt(), no caso do campo "Preço", o retorno true será válido.

Também passaremos uma mensagem personalizada para cada um dos campos. No caso do preço, incluiremos um argumento args, um array de JSOn com min: 0 e max: 99999.

this.validador = new FormValidator([

{

campo:'nome',

metodo:'isEmpty',

validoQuando: false,

mensagem: 'Entre com um nome'

},

{

campo:'livro',

metodo:'isEmpty',

validoQuando: false,

mensagem: 'Entre com um livro'

},

{

campo:'preco',

metodo:'isInt',

args: [{min:0, max:99999}],

validoQuando: true,

mensagem: 'Entre com um valor numérico'

}

]);

Assim, nosso construtor receberá um array de objetos contendo as nossas regras. Em FormValidator, precisaremos iterar por esses JSONs para fazermos a validação. Antes disso, criaremos um método valido() no qual pegaremos todas as informações da validação e devolveremos um objeto mais estruturado ao formulário.

Esse método terá uma constante validacao recebendo um objeto JSON a ser modificado dentro dele. Alteraremos o nome do objeto no nosso construtor para validacoes, já que consiste em um array.

constructor(validacoes){

this.validacoes= validacoes;

}

No método valido() faremos um mapeamento de validacao[regra.campo] para {isInvalid: false, message: ''}. Ao fim, retornaremos um objeto com {isValid: ture, ...validacao}, utilizando o spread operator.

valid(){

const validacao = {

}

this.validacoes.map(regra => (

validacao[regra.campo] = {isInvalid: false, message: ''}

));

return {isValid: true, ...validacao};

}

Ou seja, estamos criando um objeto JSON informando que o formulário é válido e passando todas as regras que foram aplicadas no método valida().

Nesse método, geraremos um objeto com let validacao = this.valido(). Em seguida, para cada regra de validação no array validacoes, executaremos uma lógica de validação bastante parecida com a que já temos, portanto copiaremos o código.

valida(state) {

let validacao = this.valido();

this.validacoes.forEach(regra => {

const campoValor = state[this.validacao.campo.toString()];

const metodoValidacao = validador[this.validacao.metodo];

if(metodoValidacao(campoValor, [], state) === true){

console.log("FORM INVÁLIDO");

return false;

}else{

console.log("FORM VÁLIDO");

return true;

}

});

}

Estamos recuperando o campoValor e o metodoValidacao, e precisaremos também do args que deverá ser utilizado no campo preço. Assim, criaremos uma constante args que receberá regra.args ou um array vazio.

Em metodoValidacao, adicionaremos um typeof regra.metodo === 'string' ?. Dessa forma, se regra.metodo for uma string, utilizaremos o validador para recuperar o método. Do contrário, aplicaremos regra.metodo diretamente. Assim, nosso código estará pronto para lidar com importação do validador.

No operador if, ao invés de passarmos um array vazio, passaremos o spread operator ...args. Se regra.validaQuando for diferente do retorno de metodoValidacao(), identificaremos que há algo de errado e criaremos um objeto com isInvalid: true e message: regra.mensagem (a mensagem que escrevemos nas regras do formulário). Abaixo, modificaremos validacao com isValid = false, removeremos o else e retornaremos o objeto modificado.

valida(state) {

let validacao = this.valido();

this.validacoes.forEach(regra => {

const campoValor = state[this.validacao.campo.toString()];

const args = regra.args || [];

const metodoValidacao = typeof regra.metodo === 'string' ? validador[this.validacao.metodo] : regra.metodo

if(metodoValidacao(campoValor, ...args, state) !== regra.validoQuando){

validacao[regra.campo] = {

isInvalid: true,

message: regra.mensagem

}

validacao.isValid = false;

}

});

return validacao;

}

Agora precisamos utilizar o objeto gerado no nosso formulário. Em this.stateInicial, criaremos uma nova chave validacao que começará como válida (this.validador.valido()).

this.stateInicial = {

nome: '',

livro: '',

preco: '',

validacao: this.validador.valido(),

}

Em submitFormulario(), criaremos uma constante validacao recebendo a chamada de this.validador.valida() com o parâmetro this.state. No operador if, faremos validacao.isValid. Em caso positivo, como já descrito no código, mudaremos o state da aplicação, adicionaremos um elemento novo e isso se refletirá na tela. Do contrário (else), precisaremos recuperar os campos que tiveram erro e exibir suas respectivas mensagens na tela.

Para isso, a partir de validacao, recuperaremos {nome, livro, preco} e os armazenaremos em um array campos. Em seguida, criaremos uma constante camposInvalidos, filtraremos os elementos do array (elem) e aplicaremos uma regra - no caso, elem.isInvalid. Por fim, para cada elemento em camposInvalidos, aplicaremos um console.log.

submitFormulario = () => {

const validacao = this.validador.valida(this.state);

if(validacao.isValid) {

this.props.escutadorDeSubmit(this.state);

this.setState(this.stateInicial);

}else{

const {nome, livro, preco} = validacao;

const campos = [nome, livro, preco];

const camposInvalidos = campos.filter(elem => {

return elem.isInvalid;

});

camposInvalidos.forEach(console.log);

}

}

Feito isso, se tentarmos salvar um novo autor na nossa aplicação, receberemos um erro indicando que a propriedade campo não pode ser lida. Isso acontece porque, em Formulario.js, esquecemos de trocar this.validacao por regra.

this.validacoes.forEach(regra => {

const campoValor = state[regra.campo.toString()];

const args = regra.args || [];

const metodoValidacao = typeof regra.metodo === 'string' ? validador[regra.metodo] : regra.metodo

if(metodoValidacao(campoValor, ...args, state) !== regra.validoQuando){

validacao[regra.campo] = {

isInvalid: true,

message: regra.mensagem

}

validacao.isValid = false;

}

});

Corrigido o erro, se clicarmos em "Salvar" sem passarmos nenhuma informação, receberemos no console as mensagens de erro para cada um dos campos, e nenhum item será adicionado. Se digitarmos somente o nome, as mensagens referentes ao livro e ao valor numérico serão exibidas, e novamente nada será adicionado. Se preenchermos todos os campos com um texto, inclusive "preço", receberemos a mensagem pedindo a entrada de um valor numérico. Já se preenchermos todos os campos corretamente, um novo item será adicionado à lista e nenhuma mensagem será exibida no console.

Assim, concluímos a nossa validação. No próximo vídeo aprenderemos a exibir uma mensagem de alto nível ao usuário.

# Entendendo a validação à fundo

Nosso código de validação é um pouco complexo. Se algo não ficou claro para você, recomendo que leia a explicação abaixo, caso contrário, fique à vontade para pular para a próxima atividade!

Para começar, vamos partir do componente **Formulario**.

# Formulário.js

Vimos no capítulo anterior que validar um campo específico é relativamente mais fácil do que um formulário inteiro.

Nosso objetivo é fazer com que nosso código seja genérico o suficiente a ponto de conseguir validar todos os campos do meu formulário atual e também de outros formulários que podem vir a existir na minha aplicação. Em outras palavras, estamos desenvolvendo uma solução para validação e não uma validação para aquele formulário em específico.

Para validarmos de forma genérica, precisamos de algumas informações básicas, por isso adicionamos algumas outras chaves em cada regra nossa que é criada no **constructor**

this.validador = new FormValidator([

{

campo: 'nome',

metodo: 'isEmpty',

validoQuando: false,

mensagem: 'Entre com um nome'

},

{

campo: 'livro',

metodo: 'isEmpty',

validoQuando: false,

mensagem: 'Entre com um livro'

},

{

campo: 'preco',

metodo: 'isInt',

args: [{min: 0, max: 99999}],

validoQuando: true,

mensagem: 'Entre com um valor numérico'

}

]);

Perceba como a condição de comparação muda de acordo com o campo, enquanto a definição de válido para os campos **nome** e **livro** é diferente da definição de válido para o campo **preco**.

Enquanto **livro** e **nome** são válidos quando o retorno é **false**, **preco** é válido para retorno **true**. Por isso criamos a chave **validoQuando**.

Precisamos também de uma mensagem específica para cada campo, para que consigamos emitir uma mensagem específica de cada campo.

No **submitFormulario**, condicionamos a submissão à validade dos campos e capturamos os campos inválidos e emitimos a mensagem específica.

# FormValidator.js

Agora no **FormValidator**, onde está nossa lógica de validação.

Nessa classe, temos dois métodos: O **valido** e o **valida**.

Enquanto o **valido** cria um grande objeto dizendo que o formulário é válido, o método **valida** verifica cada campo de acordo com as regras definidas e **modifica** esse objeto criado pelo **valido** de maneira a tornar esse formulário inválido caso seja necessário.

Dando uma olhada no método válido:

valido() {

//criação do objeto

const validacao = {};

//populando o objeto de acordo com a quantidade de campos

//criando a chave isInvalid e atribuindo false para cada

//\*\*TODOS OS CAMPOS COMEÇAM VÁLIDOS!\*\*

this.validacoes.map(regra => (

validacao[regra.campo] = { isInvalid: false, message: '' }

));

//retorna um grande objeto com uma chave externa isValid

//junto com todos os outros campos.

return { isValid: true, ...validacao };

}

}

Para ficar mais claro, a estrutura do objeto retornado é:

{

isValid: true,

nome: { isInvalid: false, message: '' },

livro: { isInvalid: false, message: '' },

preco: { isInvalid: false, message: ''}

}

Agora com esse objeto em mãos, podemos começar a verificar os valores de entrada do usuário e modificá-lo caso necessário.

Com isso, vamos agora para o método **valida**

O objetivo do método **valida** é verificar a entrada do usuário utilizando o método que foi passado e comparando com o valor de **validoQuando**

Vamos ver esse método:

valida(state) {

//itera pelo array de regras de validação e constrói

//um objeto validacao e retorna-o

//começa assumindo que está tudo válido, recebe o

//objeto do método valido.

let validacao = this.valido();

this.validacoes.forEach(regra => {

//Se o campo não tiver sido marcado

//anteriormente como invalido por uma regra.

if (!validacao[regra.campo].isInvalid) {

//Determina o valor do campo, o método a ser invocado

//e os argumentos opcionais pela definição da regra

const campoValor = state[regra.campo.toString()];

const args = regra.args || [];

//if ternário para estar preparado caso

//alguém passe o método direto sem ser string

const metodoValidacao = typeof regra.metodo === 'string' ?

validador[regra.metodo] : regra.metodo;

//invoca o método específico da regra

if (metodoValidacao(campoValor, ...args, state) !== regra.validoQuando) {

//modifica o objeto no campo específico

validacao[regra.campo] = {

isInvalid: true,

message: regra.mensagem

};

validacao.isValid = false;

}

}

});

return validacao;

}

Repare que quando recuperamos o método a ser utilizado, estamos nos blindando caso alguém decida passar o método diretamente e não apenas o nome em formato de string.

Chamamos o método específicado e comparamos com o valor de **validoQuando**, se for diferente, modificamos nosso objeto, alterando a chave **isInvalid** para **true** e passando a mensagem que recebemos nas regras.

Feito isso, alteramos a chave **isValid** para **false**, agora que nosso formulário se tornou inválido e então retornamos todo o objeto.

Apesar de parecer um pouco completo, se pensarmos passo à passo as coisas ficam mais fáceis.

Se algo ainda não ficou claro para você, não hesite em me procurar no fórum!

# Errata: Nome da chave

# Errata

No vídeo **"Exibindo mensagens de alto nível"**, no minuto **5:21**, o nome da chave no componente **Formulario** deve ser em inglês, conforme criado na classe **FormValidator**.

O código deve ser:

PopUp.exibeMensagem('error', campo.message);

Pedimos desculpas pelo inconveniente.

# Exibindo mensagens de alto nível

No último vídeo terminamos de implementar a validação dos nossos campos. Porém, só temos mensagens de erro de baixo nível, exibidas no console, e seria interessante as enviarmos também para o usuário do sistema, e podemos utilizar o Materialize para isso.

Essa biblioteca possui os chamados "Toasts", que nada mais são do que notificações em pop-ups. O Toast é utilizado por meio de um JavaScript do Materialize que deverá ser importado. Em "src", criaremos um arquivo PopUp.js e, seguindo o padrão indicado na documentação, importaremos m do pacote materialize-css.

Criaremos uma constante PopUp que será um JSON contendo uma chave exibeMensagem na qual teremos uma *arrow function*. Essa função receberá um dado e verificará se precisaremos exibir uma mensagem de erro ou uma mensagem de sucesso. Sendo assim, receberemos na função um status e a msg que queremos exibir.

Se o status da mensagem for igual a success, chamaremos M.toast() para criarmos um Toast com três especificações: html, que receberá a msg que criamos; classes, que nos permite configurar uma cor, nesse caso green; e displayLenght, que define a duração da mensagem, e que setaremos como 2000 milissegundos.

Em seguida, criaremos outro Toast para quando o status da mensagem for igual a error, e a única diferença será que teremos a cor red ao invés de green. Por fim, exportaremos o PopUp.

import M from 'materialize-css';

const PopUp = {

exibeMensagem: (status, msg) => {

if(status === 'success'){

M.toast({html: msg, classes: 'green', displayLenght: 2000});

}

if(status === 'error'){

M.toast({html: msg, classes: 'red', displayLength: 2000});

}

}

}

export default PopUp;

Criados os toasts, temos que utilizá-los nos nossos métodos. Em Formulario.js, importaremos o arquivo PopUp do seu diretório. Ao final do método submitFormulario(), no qual fazemos um loop (forEach()) para cada mensagem de erro, receberemos um campo e, para cada campo, chamaremos Pop.exibeMensagem(), passando como parâmetros 'error e campo.message.

camposInvalidos.forEach(campo => {

PopUp.exibeMensagem('error', campo.message)

});

Feito isso, se tentarmos adicionar um item à lista passando somente o seu "Nome", por exemplo, as mensagems "Entre com um livro" e "Entre com um valor numérico" serão exibidas em vermelho na tela. Agora precisamos criar também uma mensagem de sucesso.

Em App.js, quando enviamos um formulário, chamamos o escutadorDeSubmit() que está sendo recebido por props. Após importarmos o PopUp do seu diretório, acessaremos esse método e chamaremos PopUp.exibeMensagem() com o parâmetro 'success' e a mensagem "Autor adicionado com sucesso".

escutadorDeSubmit = autor => {

this.setState({ autores:[...this.state.autores, autor]});

PopUp.exibeMensagem('success', "Autor adicionado com sucesso");

}

Agora, se preenchermos todos os campos corretamente e clicarmos em "Salvar", a mensagem definida será exibida em uma pop-up verde. Também seria interessante incluirmos uma mensagem confirmando a remoção de um autor. Em removeAutor, depois de setarmos o state da aplicação,chamaremos PopUp.exibeMensagem() com o parâmetro error (para termos uma pop-up vermelha) e a mensagem "Autor removido com sucesso".

removeAutor = index =>{

const { autores } = this.state;

this.setState(

{

autores : autores.filter((autor, posAtual) => {

return posAtual !== index;

}),

}

);

PopUp.exibeMensagem('error', "Autor removido com sucesso");

}

Feito isso, nossa mensagem de remoção passará a ser exibida sempre que clicarmos em "Remover".

# Mãos na Massa

Chegou a hora de você pôr em prática o que foi visto na aula. Para isso, execute os passos listados abaixo.

1 - Crie um arquivo PopUp.js e FormValidator.js e importe no seu componente Formulário

import FormValidator from './FormValidator';

import PopUp from './PopUp';

2 - Corrija a criação do **FormValidator** no construtor do componente Formulário.

//DENTRO DO CONSTRUTOR.

this.validador = new FormValidator([

{

campo: 'nome',

metodo: 'isEmpty',

validoQuando: false,

mensagem: 'Entre com um nome'

},

{

campo: 'livro',

metodo: 'isEmpty',

validoQuando: false,

mensagem: 'Entre com um livro'

},

{

campo: 'preco',

metodo: 'isInt',

args: [{min: 0, max: 99999}],

validoQuando: true,

mensagem: 'Entre com um valor numérico'

}

]);

this.stateInicial = {

nome: '',

livro: '',

preco: '',

validacao: this.validador.valido()

}

3 - Modifique o método **submitFormulario** conforme abaixo:

submitFormulario = () => {

const validacao = this.validador.valida(this.state);

console.log(validacao)

if(validacao.isValid){

this.props.escutadorDeSubmit(this.state);

this.setState(this.stateInicial);

}else {

const { nome, livro, preco } = validacao;

const campos = [nome, livro, preco];

const camposInvalidos = campos.filter(elem => {

return elem.isInvalid

});

camposInvalidos.forEach(campo => {

PopUp.exibeMensagem('error', campo.message);

});

}

}

4- Abra seu arquivo **FormValidator** crie sua classe e adicione os métodos **valida** e **valido**:

import validador from 'validator';

class FormValidator {

constructor(validacoes) {

// validacoes é um array específico do formulário com regras de validação

this.validacoes = validacoes;

}

valida(state) {

//itera pelo array de regras de validação e constrói

//um objeto validacao e retorna-o

//começa assumindo que está tudo válido

let validacao = this.valido();

this.validacoes.forEach(regra => {

//Se o campo não tiver sido marcado anteriormente como invalido por uma regra.

if (!validacao[regra.campo].isInvalid) {

//Determina o valor do campo, o método a ser invocado

//e os argumentos opcionais pela definição da regra

const campoValor = state[regra.campo.toString()];

const args = regra.args || [];

const metodoValidacao = typeof regra.metodo === 'string' ?

validador[regra.metodo] : regra.metodo;

if (metodoValidacao(campoValor, ...args, state) !== regra.validoQuando) {

validacao[regra.campo] = {

isInvalid: true,

message: regra.mensagem

};

validacao.isValid = false;

}

}

});

return validacao;

}

//cria um objeto validaçao para um form válido

valido() {

const validacao = {};

this.validacoes.map(regra => (

validacao[regra.campo] = { isInvalid: false, message: '' }

));

return { isValid: true, ...validacao };

}

}

export default FormValidator;

5 - Abra seu arquivo PopUp.js e adicione o código abaixo:

import M from 'materialize-css';

const PopUp = {

exibeMensagem: (status, msg) => {

if (status === "success")

M.toast({ html: msg, classes: "green", displayLength: 2000 })

if (status === "error")

M.toast({ html: msg, classes: "red", displayLength: 2000 })

}

}

export default PopUp;

6 - Teste no navegador e veja se tudo funciona!

# O que aprendemos?

Nesta aula, aprendemos:

* Fazer com que a lógica funcione para qualquer campo.
* Blindar o código para saber trabalhar caso o método. seja o próprio método ou string.
* Utilizar o Toast para emitir mensagens de alto nível.
* Criar um componente de PopUp para notificar usuários.

# Instalando o módulo de rotas

No capítulo anterior nós concluímos a validação e ainda exibimos uma mensagem de alto nível para o usuário. Agora trabalharemos em outro ponto do nosso código. No momento temos um grande componente App renderizando as informações na tela. Na parte superior direita, temos três links: "Autores", "Livros" e "Sobre". Entretanto, nenhum desses links apresenta alguma funcionalidade até o momento.

A ideia é que, de acordo com a rota exibida na URL - por exemplo, <http://localhost:3000/autores>, <http://localhost:3000/livros> ou <http://localhost:3000/sobre> -, mudarmos o componente que é renderizado na tela. Assim, ao invés de App.js, exibiríamos o Livro.js, e assim por diante. Faremos isso utilizando um módulo do React chamado react-router-dom na versão 5.0.0.

No Prompt de Comando, executaremos npm install react-router-dom@5.0.0. Após a instalação, poderemos utilizar esse módulo na nossa aplicação importando-o:

import router from 'react-router-dom';

No próximo vídeo começaremos a linkar as URLs com um componente específico a elas.

# Trabalhando com rotas

No último vídeo fizemos a instalação do pacote de rotas do React, e agora é hora de utilizá-lo. Primeiro, faremos um ajuste de estilo em App.js, adicionando um título <h1> "Casa do Código".

render() {

return (

<Fragment>

<Header />

<div className="container mb-10">

<h1>Casa do Código</h1>

<Tabela autores = { this.state.autores } removeAutor = { this.removeAutor } />

<Form escutadorDeSubmit={this.escutadorDeSubmit}/>

</div>

</Fragment>

);

}

A ideia é, a partir de uma rota, direcionarmos o usuário a um componente específico. Por exemplo, <http://localhost:3000/livros> levará a uma listagem de livros, e assim por diante. Antes de começarmos, é importante lembrar que o App.js está sendo renderizado em index.js, que pega o root na pasta "public" e joga o componente dentro dele. Sendo assim, é esse arquivo que iremos alterar.

Para fazermos o roteamento, precisaremos importar o BrowserRouter, o Switch e o Route, todos provindos de react-router-dom. No método render(), passaremos a utilizar o roteamento. Incluiremos uma tag <BrowserRouter>, indicando que queremos utilizar as rotas, e uma tag <Switch>, que conterá as rotas propriamente ditas da aplicação. Para cada uma delas, teremos uma tag <Route> contendo um path, que é o caminho.

Começaremos pelo caminho /, que deverá renderizar um component={App}. Seguindo esse molde, definiremos as outras rotas da nossa aplicação. É uma boa prática incluirmos também um <Route> sem a definição de um caminho, mas renderizando componente NotFound. Dessa forma, se uma rota não descrita for acessada pelo usuário, nós o manteremos no nosso domínio, renderizando uma página vazia, mas com o cabeçalho estilizado.

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

ReactDOM.render(

<BrowserRouter>

<Switch>

<Route path='/' component={App}/>

<Route path='/sobre' component={Sobre}/>

<Route path='/livros' component={Livros}/>

<Route path='/autores' component={Autores}/>

<Route component={NotFound}/>

</Switch>

</BrowserRouter>,

document.getElementById('root'));

Antes de prosseguirmos, imagine uma situação em que temos as rotas /sobre e /sobre/informacao. Como a comparação das rotas é parcial, podemos ter problemas no acesso a elas. Para definirmos que uma rota só pode ser acessada pelo caminho exato passado no path, podemos adicionar um parâmetro exact=[true], evitando esse tipo de problema. Pensando nisso, adicionaremos esse parâmetro na rota /.

Agora criaremos os componentesSobre e NotFound. Como desafio, você pode criar sozinho(a) os componentes Livros e Autores, e [a solução do nosso instrutor pode ser encontrada aqui](https://cursos.alura.com.br/course/react-validacao-rotas-api/task/56911). Com os conhecimentos aprendidos até aqui, temos certeza de que você não terá problemas nesse desafio.

Começaremos criando um arquivo Sobre.js no qual importaremos o React, o Fragment e o Header (para usarmos o cabeçalho). Nesse arquivo, teremos um function component Sobre() que retornará um <Fragment> contendo o cabeçalho e um título <h1> "Sobre". Por fim, exportaremos esse componente.

import React, { Fragment } from 'react';

import Header from './Header';

const Sobre = () =>{

return (

<Fragment>

<Header />

<h1>Sobre</h1>

</Fragment>

);

}

export default Sobre;

O processo será praticamente idêntico no componente NotFound.js, mudando apenas a nomeclatura e o texto exibido na tela.

import React, { Fragment } from 'react';

import Header from './Header';

const NotFound = () =>{

return (

<Fragment>

<Header />

<h1>Página Não Encontrada</h1>

</Fragment>

);

}

export default NotFound;

Por enquanto, para não recebermos um erro na aplicação, criaremos os componentes Autores e Livros com a mesma construção do Sobre, deixando para você a implementação da lógica desses componentes.

Feito isso, subiremos novamente a aplicação e testaremos as rotas criadas. Como esperado, elas exibirão corretamente os textos que preparamos. Inclusive, se testarmos uma rota que não existe na aplicação, uma página com o texto "Página Não Encontrada" será renderizada na tela.

Ainda precisamos ajustar um ponto: repare que sempre que clicamos em algum link, a página é recarregada. Entretanto, nosso objetivo é construirmos uma SPA (Single Page Application), e tal comportamento fere esse conceito.

O react-router-dom nos fornece uma tag específica, chamada Link, para trabalharmos com rotas sem carregarmos uma página. Em Header.js, vamos importá-la.

No retorno, substituiremos todas as tags <a> pela tag <Link>, passando o parâmetro to para cada uma das rotas da aplicação.

import React from 'react';

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

const Header = () => {

return (

<nav>

<div className="nav-wrapper indigo lighten-2">

<Link to="/" className="brand-logo">Casa do Código</Link>

<ul className="right">

<li><Link to='/autores'>Autores</Link></li>

<li><Link to='/livros'>Livros</Link></li>

<li><Link to='/sobre'>Sobre</Link></li>

</ul>

</div>

</nav>

);

}

export default Header;

Assim, conseguiremos navegar por todas as nossas páginas sem que o navegador recarregue a página. Podemos melhorar ainda mais a nossa visualização indicando no menu qual é a página ativa, e é isso que faremos no próximo vídeo. Recomendamos implementar os componentes de autores e de livros antes disso!

# Melhorando nosso código

Já terminamos de construir as rotas da nossa aplicação, e agora queremos melhorar um pouco o nosso código. No momento estamos importando a tag Link no arquivo Header.js. Agora, imagine um cenário no qual estamos utilizando essa tag em vários componentes e, por alguma razão, o react-router-dom deixou de dar suporte a ela. Nesse caso, teremos que alterar a nossa importação e todos os pontos que utilizam a tag no nosso código, o que é bastante problemático.

Seria mais interessante termos uma tag própria que, por baixo dos panos, utilizasse o Link do react-router-dom. Para isso, utilizaremos o chamado "High Order Component", um componente que retorna outro por baixo dos panos. Além disso, gostaríamos de estilizar os links para que eles fossem realçados de acordo com a página que está sendo exibida, o que não é possível com a tag Link.

Começaremos a solucionar esses problemas criando um arquivo LinkWrapper.js, no qual importaremos o React e, do react-router-dom, a NavLink, que se comporta de maneira semelhante ao Link, mas com possibilidades de estilização.

Criaremos então uma constante LinkWrapper recebendo props (para saber para qual rota o usuário deverá ser enviado) e retornará um <NavLink>. O <NavLink> possui uma propriedade activeStyle que nos permite passar um estilo, por exemplo fontWeight: "bold" (para que o link da página ativa fique em negrito). Para passarmos as rotas, usaremos o spread operator ...props.

import React from 'react';

import {NavLink} from 'react-router-dom';

const LinkWrapper = props => {

return (

<NavLink activeStyle={{fontWeight: "bold"}}{...props}/>

);

}

export default LinkWrapper;

Dessa forma, estaremos recebendo todos os atributos e aplicando-os ao <NavLink>. Isso facilitará o nosso trabalho e o de futuros colegas de desenvolvimento nesse projeto. Além disso, será possível sobrescrever o activeStyle, caso isso seja necessário por algum motivo, pois poderemos passá-lo como props. Isso também se dá pela ordem em que esses atributos estão dispostos na tag - como o ...pros está por último, ele sempre será aplicado por último.

Feito isso, deixaremos de importar Link no arquivo Header.js e passaremos a importar o LinkWrapper. Além disso, usaremos a tag <LinkWrapper> nos pontos em que a antiga era aplicada.

import React from 'react';

import LinkWrapper from './LinkWrapper';

const Header = () => {

return (

<nav>

<div className="nav-wrapper indigo lighten-2">

<LinkWrapper to="/" className="brand-logo">Casa do Código</LinkWrapper>

<ul className="right">

<li><LinkWrapper to='/autores'>Autores</LinkWrapper></li>

<li><LinkWrapper to='/livros'>Livros</LinkWrapper></li>

<li><LinkWrapper to='/sobre'>Sobre</LinkWrapper></li>

</ul>

</div>

</nav>

);

}

export default Header;

Assim, nossos links passarão a ser negritados quando estivermos em suas respectivas páginas. Se quisermos definir outro comportamento, basta passarmos uma propriedade activeStyle (vazia, por exemplo) no link desejado

É importante trabalharmos com High Order Components para deixarmos nosso código "livre" de funcionalidades de outros módulos, envelopando-os e criando um único ponto de alteração caso eles sejam alterados em algum ponto do futuro.

# Solução para a página de Autores e de Livros

Nesse capítulo nós deixamos o desafio de implementar os componentes Autores e Livros. Existem várias maneiras de construir essas duas páginas, e, se você quiser, pode compartilhar a sua solução conosco no fórum do curso! Aqui mostraremos a solução proposta pelo nosso instrutor.

A ideia é que as páginas Autores.js e Livros.js têm basicamente a mesma estrutura, e, por debaixo dos panos, termos um mesmo componente capaz de renderizar as duas tabelas de acordo com as informações recebidas. Vamos analisar o código de Autores.js:

import React, { Component, Fragment } from 'react';

import Header from './Header';

import DataTable from './DataTable';

class Autores extends Component {

constructor(props) {

super(props);

this.state = {

autores: [

{

nome: 'Paulo',

livro: 'React',

preco: '1000'

},

{

nome: 'Daniel',

livro: 'Java',

preco: '99'

},

{

nome: 'Marcos',

livro: 'Design',

preco: '150'

},

{

nome: 'Bruno',

livro: 'DevOps',

preco: '100'

},

{

nome: 'Nico',

livro: 'Java',

preco: '9999'

}

],

titulo: 'Autores'

};

}

render() {

return (

<Fragment>

<Header />

<div className='container'>

<h1>Página de Autores</h1>

<DataTable dados={this.state.autores} titulo={this.state.titulo} colunas={['nome']}/>

</div>

</Fragment>

);

}

}

export default Autores;

Repare que criamos um componente DataTable para o qual estamos passando os autores, o titulo utilizado na tabela e quais colunas queremos renderizar. Além disso, no state, temos uma chave titulo. Vamos analisar a implementação desse DataTable.js:

import React from 'react';

const DataTable = props => {

let linhas = props.dados.map((item) =>

<tr key={item.id}>

{props.colunas.map((coluna) =>

<td key={`${item.id}${item[coluna]}`}>{item[coluna]}</td>)}

</tr>);

return (

<table className='centered highlight'>

<thead>

<tr>

<th>{props.titulo}</th>

</tr>

</thead>

<tbody>

{linhas}

</tbody>

</table>

);

}

export default DataTable;

Ele recebe as propriedades enviadas pelo componente - no caso, Autores.js - e iteramos pelos autores, criando uma <tr> para cada um deles, colocando como key o item.id. Em seguida, iteramos pelas colunas que queremos criar, criando um <td> para cada um desses elementos.

Repare que nesse key estamos concatenando dois valores, item.id e o valor guardado na coluna. Isso é uma boa prática, já que no React não podemos ter dois elementos diferentes com a mesma key. Por fim, retornamos a tabela montada com o valor do título, recebido por props, e jogando as linhas em um <tbody>.

O componente Livros é praticamente idêntico ao Autores:

import React, { Component, Fragment } from 'react';

import Header from './Header';

import DataTable from './DataTable';

class Livros extends Component {

constructor(props) {

super(props);

this.state = {

autores: [

{

nome: 'Paulo',

livro: 'React',

preco: '1000'

},

{

nome: 'Daniel',

livro: 'Java',

preco: '99'

},

{

nome: 'Marcos',

livro: 'Design',

preco: '150'

},

{

nome: 'Bruno',

livro: 'DevOps',

preco: '100'

},

{

nome: 'Nico',

livro: 'Java',

preco: '9999'

}

],

titulo: 'Livros'

};

}

render() {

return (

<Fragment>

<Header />

<div className='container'>

<h1>Página de Livros</h1>

<DataTable dados={this.state.autores} titulo={this.state.titulo} colunas={['livro']}/>

</div>

</Fragment>

);

}

}

export default Livros;

Antes nós estávamos criando componentes mais simples (function components), mas os transformamos em class components justamente para conseguirmos guardar o state de cada um desses elementos. Sempre devemos começar criando nossos componentes como funções, e só transformá-los em classes quando surge a necessidade.

Essa foi a solução do nosso instrutor, e se você fez de maneira diferente ou não conseguiu realizar o exercício, converse conosco no fórum! Bons estudos e até o próximo capítulo!

# Faça o que eu fiz na aula

Chegou a hora de você pôr em prática o que foi visto na aula. Para isso, execute os passos listados abaixo.

1 - Instale o módulo validator no seu projeto:

No terminal:

npm install validator

2 - Crie um componente Livros e escreva o seguinte código:

import React, { Component, Fragment } from 'react';

import Header from './Header';

import DataTable from './DataTable'

class Livros extends Component {

constructor(props) {

super(props);

this.state = {

autores: [

{

nome: 'Paulo',

livro: 'React',

preco: '1000'

},

{

nome: 'Daniel',

livro: 'Java',

preco: '99'

},

{

nome: 'Marcos',

livro: 'Design',

preco: '150'

},

{

nome: 'Bruno',

livro: 'DevOps',

preco: '100'

}

],

titulo: 'Livros'

}

}

render() {

return (

<Fragment>

<Header />

<div className="container">

<h1>Livros</h1>

<DataTable dados={this.state.autores} titulo={this.state.titulo} colunas={['livro']} />

</div>

</Fragment>

);

}

}

export default Livros;

3 - Crie um componente Autores e escreva o seguinte código:

import React, { Component, Fragment } from 'react';

import Header from './Header';

import DataTable from './DataTable';

class Autores extends Component {

constructor(props) {

super(props);

this.state = {

autores: [

{

nome: 'Paulo',

livro: 'React',

preco: '1000'

},

{

nome: 'Daniel',

livro: 'Java',

preco: '99'

},

{

nome: 'Marcos',

livro: 'Design',

preco: '150'

},

{

nome: 'Bruno',

livro: 'DevOps',

preco: '100'

}

],

titulo: 'Autores'

}

}

render() {

return (

<Fragment>

<Header />

<div className="container">

<h1>Autores</h1>

<DataTable dados={this.state.autores} titulo={this.state.titulo} colunas={['nome']} />

</div>

</Fragment>

);

}

}

export default Autores;

4 - Crie um componente para a página Sobre e coloque o seguinte código:

import React, { Fragment } from 'react';

import Header from './Header';

const Sobre = () => {

return(

<Fragment>

<Header />

<h1> Sobre </h1>

</Fragment>

);

}

export default Sobre;

5 - Crie um *High Order Component* **LinkWrapper** para isolar a utilização do módulo validator

import React from 'react';

import { NavLink } from 'react-router-dom';

const LinkWrapper = (props) => {

return (

<NavLink activeStyle={{ fontWeight: "bold" }} {...props}/>

);

}

export default LinkWrapper;

6 - Utilize o **LinkWrapper** no componente **Header**

import React from 'react';

import LinkWrapper from './LinkWrapper';

const Header = () =>{

return (

<nav>

<div className="nav-wrapper indigo lighten-2">

<LinkWrapper to="/" className="brand-logo ml-3" activeStyle={{}}>Casa do Código</LinkWrapper>

<ul id="nav-mobile" className="right">

<li><LinkWrapper to="/autores">Autores</LinkWrapper></li>

<li><LinkWrapper to="/livros">Livros</LinkWrapper></li>

<li><LinkWrapper to="/sobre">Sobre</LinkWrapper></li>

</ul>

</div>

</nav>

);

}

export default Header;

7 - Modifique o arquivo **index.js** para determinar as rotas e seus respectivos componentes.

import React from 'react';

import ReactDOM from 'react-dom';

import './index.css';

import App from './App';

import \* as serviceWorker from './serviceWorker';

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

import Autores from './Autores';

import Livros from './Livros';

import Sobre from './Sobre';

import NotFound from './NotFound'

ReactDOM.render(

<BrowserRouter>

<Switch>

<Route path="/" exact={true} component={App} />

<Route path="/sobre" component={Sobre} />

<Route path="/livros" component={Livros} />

<Route path="/autores" component={Autores} />

<Route component={NotFound} />

</Switch>

</BrowserRouter>, document.getElementById('root'));

serviceWorker.unregister();

8 - Teste no navegador se tudo continua funcionando normalmente!

# O que aprendemos?

Nesta aula, aprendemos:

* Instalação do módulo de rotas react-router-dom v4.
* Trabalhar com rotas no React.
* High Order Component.
* DataTable genérico.

# Acessando uma API externa

No último capítulo nós trabalhamos com as rotas da nossa aplicação, e agora é o momento de armazenarmos os dados de tabela em algum lugar e a recebê-los desse mesmo lugar, trabalhando com uma API externa. [A API que utilizaremos nesse processo, bem simples e feita em Node, pode ser baixada aqui.](https://caelum-online-public.s3.amazonaws.com/1274-react-parte2/04/curso-react-api.zip)

Após extrairmos o conteúdo do ZIP, acessaremos o diretório pelo Prompt e executaremos npm install. Terminada a instalação, executaremos npm start para subirmos a aplicação, que poderá ser acessada em [http://localhost:8000](http://localhost:8000/). Na página <http://localhost:8000/api/autor>, encontraremos como retorno uma mensagem de sucesso e um objeto data contendo quatro elementos com seus respectivos IDs, nomes, livros e preços. Se acessarmos <http://localhost:8000/api/autor/1>, por exemplo, teremos como retorno somente o primeiro elemento, e assim por diante. Nosso objetivo agora é acessarmos o endpoint dessa API e utilizar esses dados no nosso projeto.

No nosso curso, utilizaremos o fetch(), uma funcionalidade do ECMAScript 6, para realizarmos requisições. Existem outras formas de fazer requisições assíncronas, como o AJAX, mas escolhemos essa por não depender de recursos externos. No render() de App.js, usaremos o método fetch() passando o endereço da nossa aplicação. Em seguida, chamaremos .then() para executarmos res => res.json(), parseando os dados em formato JSON, e depois novamente o .then() para executarmos um console.log e exibirmos tais dados no console.

render() {

fetch('http://localhost:8000/api/autor')

.then(res => res.json())

.then(console.log);

return (

<Fragment>

<Header />

<div className="container mb-10">

<h1>Casa do Código</h1>

<Tabela autores = { this.state.autores } removeAutor = { this.removeAutor } />

<Form escutadorDeSubmit={this.escutadorDeSubmit}/>

</div>

</Fragment>

);

}

Se acessarmos a nossa aplicação, teremos no console um Object com a mensagem "success" e os nossos elementos. No segundo .then(), podemos executar um res => console.log(res.data) para recebermos nossos dados em um array.

fetch('http://localhost:8000/api/autor')

.then(res => res.json())

.then(res => console.log(res.data));

Assim, teremos como retorno um array de autores.

# Rodando a API

# Rodando a API do curso

Para começar, faça o download da **API** [aqui](https://caelum-online-public.s3.amazonaws.com/1274-react-parte2/04/curso-react-api.zip)

1 - Após realizar o download, extraia o diretório no local de sua preferência.

2 - Utilizando o terminal, abra o diretório da **API** e rode o comando:

npm install

Aguarde enquanto o **npm** instala todas as depêndencias necessárias.

3 - Rode o comando:

npm start

Para inicializar o servidor, no seu terminal deve aparecer uma mensagem de confirmação. ("Servidor escutando na porta....")

# Criando nosso serviço API

No último vídeo nós conseguimos fazer uma requisição do tipo GET para nossa API, recebendo como retorno um array de quatro elementos. Agora precisamos realmente escrever o código que realizará essas requisições para nós.

O método fetch() consegue realizar pedidos para uma API, e é com ele que iremos trabalhar. Porém, ao invés de incluirmos esse método em diversos pontos da nossa aplicação, "sujando-a" com uma URL que algum dia poderá mudar, criaremos um arquivo específico que irá encapsular todos os códigos referentes a requisições. Assim, se existir alguma mudança na nossa aplicação, saberemos exatamente onde trabalhar.

Em "src", criaremos um arquivo ApiService.js. Nele, criaremos uma constante ApiService recebendo um objeto.

const ApiService = {

}

Começaremos então a implementar o nosso primeiro endpoint. Abaixo, você pode encontrar todos os endpoints da nossa aplicação, incluindo requisições do tipo GET, POST, PATCH e DELETE.

Lista de endpoints da API:

GET

api/autor -> Retorno: Array de autores (nome, livro, preço, id)

api/autor/livro -> Retorno: Array de livros (livro, id)

api/autor/nome -> Retorno: Array de nomes (nome, id)

api/autor/:id -> Retorno: Autor com respectivo id (nome, livro, preço, id)

POST

api/autor -> Criação de autor (Retorna o autor inteiro do banco)

PATCH

api/autor/:id -> Atualiza um autor no banco (Retorna o autor inteiro)

DELETE

api/autor/:id -> Remove o autor do banco

Criaremos um método ListaAutores que servirá para pegarmos os autores da nossa API. Com uma arrow function, retornaremos um fetch() fazendo uma requisição para <http://localhost:8000/api/autor>. Então, usaremos o .then() para executarmos um res => res.json(), parseando nossos dados para um JSON.

const ApiService = {

ListaAutores: () =>{

return fetch('http://localhost:8000/api/autor')

.then(res => res.json());

}

}

export default ApiService;

EmApp.js, ao invés de fazermos o fetch(), importaremos o ApiService e a usaremos no método render() chamando ApiSerivce.ListaAutores(). Como receberemos um JSON, criaremos uma constante autores e, por fim, rodaremos um console.log() de autores.

render() {

const autores = ApiService.ListaAutores();

consolg.log(autores);

return (

<Fragment>

<Header />

<div className="container mb-10">

<h1>Casa do Código</h1>

<Tabela autores = { this.state.autores } removeAutor = { this.removeAutor } />

<Form escutadorDeSubmit={this.escutadorDeSubmit}/>

</div>

</Fragment>

);

}

No console da nossa aplicação, receberemos uma Promise. Isso acontece pois, por algum motivo, a nossa requisição ainda não foi completada. Como o método ListaAutores() já faz o parseamento em JSON das nossas informações, faremos then(res => console.log(res.data)) para exibirmos os dados no console.

render() {

ApiService.ListaAutores();

then(res => console.log(res.data));

return (

<Fragment>

<Header />

<div className="container mb-10">

<h1>Casa do Código</h1>

<Tabela autores = { this.state.autores } removeAutor = { this.removeAutor } />

<Form escutadorDeSubmit={this.escutadorDeSubmit}/>

</div>

</Fragment>

);

}

Dessa vez teremos como retorno os dados armazenados na nossa API. Continuaremos criando nossos endpoints, dessa vez com o método CriaAutor(). Esse método receberá um autor e nos retornará um fetch() direcionando para <http://localhost:8000/api/autor>.

A única diferença entre os métodosListaAutores() e CriaAutor() é o verbo utilizado para fazer essa requisição. No caso, este último receberá um dado em json contendo o método, o tipo de conteúdo (application/json) e um corpo representando um autor. Por fim, devolveremos uma resposta em JSON com then(res => res.json()).

const ApiService = {

ListaAutores: () =>{

return fetch('http://localhost:8000/api/autor')

.then(res => res.json());

},

CriaAutor: autor =>{

return fetch('http://localhost:8000/api/autor', {method: 'POST', headers: {'content-type': 'application/json'}, body: autor})

.then(res => res.json());

},

}

export default ApiService;

Criado esse método, partiremos para o ListaNomes(). A ideia é a mesma de ListaAutores(), mudando apenas o endpoint para <http://localhost:8000/api/autor/nome>.

const ApiService = {

ListaAutores: () =>{

return fetch('http://localhost:8000/api/autor')

.then(res => res.json());

},

CriaAutor: autor =>{

return fetch('http://localhost:8000/api/autor', {method: 'POST', headers: {'content-type': 'application/json'}, body: autor})

.then(res => res.json());

},

ListaNomes: () =>{

return fetch('http://localhost:8000/api/autor/nome')

.then(res => res.json());

}

}

export default ApiService;

Em App.js, trocaremos o método ListaAutores() para ListaNomes().

ApiService.ListaNomes().then(res => console.log(res.data));

Acessando nossa aplicação, teremos no console um array contendo os nomes e ids dos elementos. Vamos continuar implementando os métodos ListaLivros() e RemoveAutor(). O primeiro é idêntico aos métodos ListaAutores() e ListaNomes(), só que acessando o endpoint <http://localhost:8000/api/autor/livro>. Já o método RemoveAutor() recebe um id e direciona, por meio de uma template string, para http://localhost:8000/api/autor${id}, concatenando a URL com a ID recebida no método. Além disso, passaremos o método DELETE e o tipo do conteúdo (application/json).

const ApiService = {

ListaAutores: () =>{

return fetch('http://localhost:8000/api/autor')

.then(res => res.json());

},

CriaAutor: autor =>{

return fetch('http://localhost:8000/api/autor', {method: 'POST', headers: {'content-type': 'application/json'}, body: autor})

.then(res => res.json());

},

ListaNomes: () =>{

return fetch('http://localhost:8000/api/autor/nome')

.then(res => res.json());

},

ListaLivros: () =>{

return fetch('http://localhost:8000/api/autor/livro')

.then(res => res.json());

},

RemoveAutor: id =>{

return fetch(`http://localhost:8000/api/autor/${id}`, {method: 'DELETE', Headers: {'content-type': 'application/json'}})

.then(res => res.json());

}

}

export default ApiService;

Com isso, terminamos a construção da ApiService, atendo a todos os endpoints da nossa aplicação. Antes de prosseguirmos, deletaremos o código que estávamos utilizando para realizar testes em App.js.

ApiService.ListaLivros()

.then(res => console.log(res.data));

No próximo vídeo usaremos a ApiService que criamos para transformarmos as requisições em dados que podem ser exibidos na tela.

# Utilizando o serviço

No último vídeo implementamos um ApiService, um serviço que encapsula os métodos que fazem as nossas requisições, e agora vamos utilizá-lo. Em App.js, faremos a importação do ApiService e criaremos um constructor para essa classe, recebendo props como argumento. Passaremos esse argumento com o super() e moveremos o state criado anteriormente para dentro do construtor. Removeremos os autores codificados no array, mantendo apenas a chave vazia, já que receberemos esses autores de uma API.

constructor(props) {

super(props);

this.state = {

autores: [],

};

}

Para recuperarmos os autores, precisaremos conhecer um método do ciclo de vida do React chamado componentDidMount(). Esse método é executado logo após o render(), ou seja, ele é chamado depois que o componente é construído, faz uma requisição para a API e altera o estado do componente com os novos dados, chamando novamente o render() e redesenhando a tela.

Dentro desse método, faremos a nossa requisição, começando por ListaAutores(), que nos retornará uma Promise. Sendo assim, usaremos o then() para setarmos o estado com res => {this.setState()}, passando a chave autores recebendo dois spread operators, ...this.state.autores (para mantermos os autores que já estão nesse array) e ...res.data (para inserirmos os novos).

componentDidMount(){

ApiService.ListaAutores()

.then(res => {

this.setState({autores: [...this.state.autores, ...res.data]})

});

}

Salvando o código, nossa página passará a listar os autores recebidos da API. Agora vamos implementar a remoçãop de um autor. No método removeAutor(), estamos utilizando no filtro um index recebido a partir do map() de TableBody na nossa Tabela.js. Isso nos trará um problema, já que, na API, o primeiro ID (que é utilizado para encontrar o item no banco de dados) é 1, e a primeira posição de um array é 0. Sendo assim, não conseguiremos fazer a nossa lógica funcionar com o filtro criado atualmente.

Para corrigirmos isso, ao invés de utilizar o index, utilizaremos linha.id, passando-o para a chamada do método removeAutor().

const TableBody = props => {

const linhas = props.autores.map((linha) => {

return (

<tr key={linha.id}>

<td>{linha.nome}</td>

<td>{linha.livro}</td>

<td>{linha.preco}</td>

<td><button onClick = {() => {props.removeAutor(linha.id) }} className="waves-effect waves-light indigo lighten-2 btn">Remover</button></td>

</tr>

)

});

return(

<tbody>

{linhas}

</tbody>

);

}

Voltando ao App.js, receberemos como parâmetro de removeAutor() o id e o compararemos com o autor recebido no filtro. Por fim, após enviarmos a mensagem, faremos a requisição propriamente dita com ApiService.RemoveAutor() passando o id como argumento.

removeAutor = id =>{

const { autores } = this.state;

this.setState(

{

autores : autores.filter((autor) => {

return autor.id !== id;

}),

}

);

PopUp.exibeMensagem('error', "Autor removido com sucesso");

ApiService.RemoveAutor(id);

}

Assim, conseguiremos remover com sucesso os autores da lista. Ainda precisamos de uma lógica de criação de autor. Para isso, primeiro precisamos saber onde exatamente esse autor é criado, mudando o estado da aplicação, que é no ewscutadorDeSubmit(). Nesse método, chamaremos ApiService.CriaAutor(), passando como argumento um autor em string, o que conseguiremos com JSON.stringify(). Como nosso retorno será uma Promise, usaremos .then(res => res.data) para parsear os dados.

Em seguida, usaremos outro .then(), dessa vez colocando o autor dentro do nosso estado, uma lógica que já havíamos implementado antes nesse método.

escutadorDeSubmit = autor => {

ApiService.CriaAutor(JSON.stringify(autor))

.then(res => res.data)

.then(autor => {

this.setState({ autores:[...this.state.autores, autor]});

PopUp.exibeMensagem('success', "Autor adicionado com sucesso");

});

}

Com isso, conseguiremos adicionar novos autores na plataforma sem problemas. Ainda precisamos atualizar a nossa página de autores e de livros, já que ambas estão *hard coded* em nossa aplicação. Em Autores.js, removeremos o conteúdo do array autores e trocaremos a variável para nomes, mantendo o array vazio. No método render, alteraremos this.state.autores para this.state.nomes.

class Autores extends Component {

constructor(props) {

super(props);

this.state = {

nomes: [

],

titulo: 'Autores'

};

}

render() {

return (

<Fragment>

<Header />

<div className='container'>

<h1>Página de Autores</h1>

<DataTable dados={this.state.nomes} titulo={this.state.titulo} colunas={['nome']}/>

</div>

</Fragment>

);

}

}

export default Autores;

Da mesma forma que na tabela principal da nossa aplicação, utilizaremos o componentDidMount() para fazermos uma requisição. Dentro dele, chamaremos ApiService.ListaNomes(), retornando uma Promise. A partir da resposta res, chamaremos this.setState(), passando um array com nomes: [...this.state.nomes, ...res.data].

componentDidMount(){

ApiService.ListaNomes()

.then(res => {

this.setState({nomes: [...this.state.nomes, ...res.data]})

});

}

Feito isso, nossa página de autores passará a listar os nomes cadastrados na API. Repetiremos esse processo para os livros, alterando as variáveis necessárias e utilizando o método ApiService.ListaLivros().

class Livros extends Component {

constructor(props) {

super(props);

this.state = {

livros: [],

titulo: 'Livros'

};

}

componentDidMount(){

ApiService.ListaLivros()

.then(res => {

this.setState({livros: [...this.state.livros, ...res.data]})

});

}

render() {

return (

<Fragment>

<Header />

<div className='container'>

<h1>Página de Livros</h1>

<DataTable dados={this.state.livros} titulo={this.state.titulo} colunas={['livro']}/>

</div>

</Fragment>

);

}

}

export default Livros;

Com isso, a página de livros passará a listar os nomes recebidos, portanto conseguimos implementar a criação, remoção e exibição dos itens em nossa API.

# Para saber mais: Axios

# Axios

Uma outra forma de se realizar requisições assíncronas é utilizando o módulo Axios, que trabalha de forma diferente do fetch, que utilizamos no curso, veja abaixo:

Antes de tudo, podemos fazer a instalação do módulo direto no terminal, com a linha abaixo:

npm install axios

Agora podemos importar o módulo no projeto:

import axios from ‘axios’;

Feito isso, podemos utilizar o Axios para realizar as requisições!

# Get

axios.get(‘http://localhost:800/api/autor’)

.then(res => {

const autores = res.autores;

})

Repare que o Axios já nos devolve um objeto JSON, sem necessidade de realizarmos um parse!

# Post

Semelhante ao fetch, o método post recebe um JSON de dados

axios.post(‘http://localhost:800/api/autor’, { ... })

.then(res => {

const autores = res.autores;

})

# Delete

axios.delete(`http://localhost:8000/api/autor/${id}`)

.then(res => {

console.log(res);

})

}

Tanto o fetch quanto o Axios funcionam nos navegadores mais atuais, apesar de termos um problema de compatibilidade com o fetch em navegadores mais antigos.

Durante o curso, optamos pela utilização do **fetch** por já ser nativo do JavaScript.

Porém, cabe a você, desenvolvedor, escolher o que funciona melhor no seu caso!

# Mãos na Massa

Chegou a hora de você pôr em prática o que foi visto na aula. Para isso, execute os passos listados abaixo.

1 - Crie seu arquivo **ApiService** e coloque o código:

const ApiService = {

ListaAutores : () =>{

return fetch('http://localhost:8000/api/autor')

.then(res => res.json());

},

CriaAutor : autor => {

return fetch('http://localhost:8000/api/autor', {method: 'POST', headers: {'content-type': 'application/json'}, body: autor})

.then(res => res.json());;

},

ListaNomes: () =>{

return fetch('http://localhost:8000/api/autor/nome')

.then(res => res.json());;

},

ListaLivros: () => {

return fetch('http://localhost:8000/api/autor')

.then(res => res.json());;

},

RemoveAutor: id => {

return fetch(`http://localhost:8000/api/autor/${id}`, {method: 'DELETE', headers: { 'content-type' : 'application/json'},})

.then(res => res.json());;

}

}

export default ApiService;

2 - Importe a **ApiService** e modifique o componente **Autores** e **Livros** para realizar a requisição da **API**

# Autores

import React, { Component, Fragment } from 'react';

import Header from './Header';

import DataTable from './DataTable';

import ApiService from './ApiService';

import PopUp from './PopUp';

class Autores extends Component {

constructor(props) {

super(props);

this.state = {

nomes: [],

titulo: 'Autores'

};

}

componentDidMount(){

ApiService.ListaNomes()

.then(res => {

this.setState({nomes: [...this.state.nomes, ...res.data]});

PopUp.exibeMensagem('success', 'Autores Listados com sucesso');

});

}

render() {

return (

<Fragment>

<Header />

<div className='container'>

<h1>Página de Autores</h1>

<DataTable dados={this.state.nomes} titulo={this.state.titulo} colunas={['nome']} />

</div>

</Fragment>

);

}

}

export default Autores;

# Livros

import React, { Component, Fragment } from 'react';

import Header from './Header';

import DataTable from './DataTable';

import ApiService from './ApiService';

import PopUp from './PopUp';

class Livros extends Component {

constructor(props) {

super(props);

this.state = {

livros: [],

titulo: 'Livros'

};

}

componentDidMount(){

ApiService.ListaLivros()

.then(res => {

this.setState({livros : [...this.state.livros, ...res.data]});

PopUp.exibeMensagem('success', 'Livros listados com sucesso');

});

}

render() {

return (

<Fragment>

<Header />

<div className='container'>

<h1>Página de Livros</h1>

<DataTable dados={this.state.livros} titulo={this.state.titulo} colunas={['livro']} />

</div>

</Fragment>

);

}

}

export default Livros;

4 - Feito isso, importe **ApiService** e altere seu componente **App** para realizar as requisições.

import React, { Component, Fragment } from 'react';

import 'materialize-css/dist/css/materialize.min.css';

import './App.css';

import Header from './Header';

import Tabela from './Tabela';

import Form from './Formulario';

import PopUp from './PopUp';

import ApiService from './ApiService';

class App extends Component {

constructor(props) {

super(props);

this.state = {

autores: [],

};

}

removeAutor = id => {

const { autores } = this.state;

this.setState(

{

autores: autores.filter(autor => {

return autor.id !== id;

}),

}

);

PopUp.exibeMensagem("error", "Autor removido com sucesso");

ApiService.RemoveAutor(id);

}

escutadorDeSubmit = autor => {

ApiService.CriaAutor(JSON.stringify(autor))

.then(res => res.data)

.then(autor => {

this.setState({ autores:[...this.state.autores, autor]});

PopUp.exibeMensagem("success", "Autor adicionado com sucesso");

})

}

componentDidMount() {

ApiService.ListaAutores()

.then(res => {

this.setState({ autores: [...this.state.autores, ...res.data] })

})

}

render() {

return (

<Fragment>

<Header />

<div className="container mb-10">

<h1>Casa do Código</h1>

<Tabela autores={this.state.autores} removeAutor={this.removeAutor} />

<Form escutadorDeSubmit={this.escutadorDeSubmit} />

</div>

</Fragment>

);

}

}

export default App;

5 - Teste tudo no navegador!

# O que aprendemos?

Nesta aula, aprendemos:

* Acesso a uma **\*API** externa.
* Utilizar o fetch para requisições no React.
* Criar uma **ApiService**.
* Utilizar uma **ApiService**.

# Tratamento de erros

Já conseguimos implementar as funcionalidades que modificam os dados da nossa API, mas ainda temos alguns pontos a melhorar. Por exemplo, no método removeAutor(), estamos executando uma lógica para remover um item do state, exibindo a mensagem de que o autor foi removido com sucesso e só então fazendo uma requisição para a API. Porém, nada nos garante que essa requisição será bem sucedida. Sendo assim, precisamos garantir que essa mensagem só seja mostrada e o state só seja atualizado se essa requisição for concluída.

Primeiro, criaremos em ApiService.js um novo elemento chamado TrataErros. Ele receberá a resposta (res) e checará se o retorno de res.ok é false. Nesse caso, lançaremos um Error() passando res.responseText. Já se a requisição for um sucesso, retornaremos res.json().

TrataErros: res =>{

if(!res.ok){

throw Error(res.responseText);

}

return res.json();

}

A ideia é que o TrataErros() seja sempre o primeiro método a ser chamado dentre os nossos serviços. Como já estamos parseando a resposta para JSON ao final dele, removeremos esse processo dos outros métodos.

const ApiService = {

ListaAutores: () =>{

return fetch('http://localhost:8000/api/autor')

},

CriaAutor: autor =>{

return fetch('http://localhost:8000/api/autor', {method: 'POST', headers: {'content-type': 'application/json'}, body: autor})

},

ListaNomes: () =>{

return fetch('http://localhost:8000/api/autor/nome')

},

ListaLivros: () =>{

return fetch('http://localhost:8000/api/autor/livro')

},

RemoveAutor: id =>{

return fetch(`http://localhost:8000/api/autor/${id}`, {method: 'DELETE', headers: {'content-type': 'application/json'},})

},

TrataErros: res =>{

if(!res.ok){

throw Error(res.responseText);

}

return res.json();

}

}

export default ApiService;

Em App.js, reescreveremos o método removeAutor(). Começaremos criando um novo array chamado autoresAtualizado, que será o resultado do filtro. Sendo assim, ele receberá a chamada de autores.filter(), método para o qual passaremos um autor e no qual retornaremos o resultado da verificação autor.id !== id. Obtendo assim o array atualizado, não precisaremos mais do filtro, portanto removeremos o this.setState() por enquanto, ficando somente com a mensagem e a requisição para a API, que moveremos para cima.

removeAutor = id =>{

const { autores } = this.state;

const autoresAtualizado = autores.filter(autor =>{

return autor.id !== id

});

ApiService.RemoveAutor(id);

PopUp.exibeMensagem('error', "Autor removido com sucesso");

}

Abaixo dela, chamaremos o .then() e, recebendo a resposta res, executaremos ApiService.TrataErros() passando essa resposta como parâmetro. Em seguida, faremos outro .then, novamente recebendo res, e iniciaremos uma comparação. No caso, quando um elemento é deletado com sucesso da nossa API, temos como retorno uma mensagem "deleted". Sendo assim, verificaremos se res.message é igual a deleted. Em caso positivo, exibiremos a mensagem construída anteriormente e chamaremos this.setState() passando para autores o array ...autoresAtualizado. Do contrário, ou seja, se o método TrataErros() lançar uma exceção, utilizaremos o método catch() para recuperar esse erro (err) e, com uma arrow function, executaremos PopUp.exibeMensagem(), exibindo na tela o texto "Erro na comunicação com a API ao tentar remover o autor".

removeAutor = id =>{

const { autores } = this.state;

const autoresAtualizado = autores.filter(autor =>{

return autor.id !== id

});

ApiService.RemoveAutor(id)

.then(res => ApiService.TrataErros(res))

.then(res => {

if(res.message === 'deleted') {

this.setState({autores: [...autoresAtualizado]});

PopUp.exibeMensagem('error', "Autor removido com sucesso");

}

})

.catch(err => PopUp.exibeMensagem('error', "Erro na comunicação com a API ao tentar remover o autor"))

}

Ainda não conseguiremos testar, pois, ao removermos o res.json() da nossa ApiService, a aplicação toda parou de funcionar. Continuaremos trabalhando, agora corrigindo o escutadorDeSubmit(). Nele, depois da chamada de ApiService.CriaAutor(), incluiremos um .then() chamando ApiService.TrataErros(), passando a resposta res como argumento. Em seguida, resumiremos as duas últimas chamadas de .then() em uma única, passando res e executando this.setState().

escutadorDeSubmit = autor => {

ApiService.CriaAutor(JSON.stringify(autor))

.then(res => ApiService.TrataErros(res))

.then(res =>

this.setState({ autores:[...this.state.autores, autor]});

PopUp.exibeMensagem('success', "Autor adicionado com sucesso");

});

}

Quando adicionamos um elemento na API, ela nos retorna uma mensagem "success". Sendo assim, no segundo .then(), checaremos de res.message é igual a success. Em caso positivo, setaremos o estado da aplicação com um novo autor (res.data) e mostraremos a mensagem de sucesso an tela. Do contrário, usaremos o catch() para capturarmos a exceção err e enviaremos uma mensagem "Erro na comunicação com a API ao tentar criar o autor".

escutadorDeSubmit = autor => {

ApiService.CriaAutor(JSON.stringify(autor))

.then(res => ApiService.TrataErros(res))

.then(res => {

if(res.message === 'success'){

this.setState({ autores:[...this.state.autores, autor]});

PopUp.exibeMensagem('success', "Autor adicionado com sucesso");

}

})

.catch(err =>PopUp.exibeMensagem('error', "Erro na comunicação com a API ao tentar criar o autor"));

}

Para fazermos a listagem, precisaremos mexer no componentDidMount(), que é onde chamamos o método ListaAutores(). Aqui, também usaremos o .then(), pegaremos a resposta res e chamaremos ApiService.TrataErros() passando essa resposta como argumento. Quando listamos os autores, também recebemos uma mensagem de sucesso do banco de dados. Sendo assim, usaremos o operador if para verificar se res.message é igual a 'success'. Em caso positivo, setaremos o estado normalmente. Do contrário, recuperaremos o erro lançado e enviaremos uma mensagem ao usuário.

componentDidMount(){

ApiService.ListaAutores()

.then(res => ApiService.TrataErros(res))

.then(res => {

if(res.message === 'success'){

this.setState({autores: [...this.state.autores, ...res.data]})

}

})

.catch(err =>PopUp.exibeMensagem('error', "Erro na comunicação com a API ao tentar listar os autores"));

}

Terminadas essas alterações, nossa aplicação voltará a funcionar normalmente. Para testarmos, podemos trocar a URL da remoção de um autor em ApiService.js. Dessa forma, ao tentarmos remover um item, receberemos a mensagem "Erro na comunicação com a API ao tentar remover o autor" na tela, e nenhum autor será removido.

Ainda precisamos corrigir a listagem de autores e de livros, mas isso ficará como desafio para você! Mas não se preocupe, pois uma possível solução será apresentada no próximo vídeo.

# Solução para a página Livros e Autores

Nesse vídeo iremos resolver o desafio deixado anteriormente, que basicamente consistia em implementar o mesmo padrão de tratamento de erros para as páginas de autores e de livros. Em Autores.js, importaremos o PopUp, que usaremos para exibir a mensagem de erro. No método componentDidMount(), após a chamada de ListaNomes(), passaremos um .then() para executarmos ApiServices.TrataErros() recebendo uma res.

Depois, checaremos se res.message é igual a 'success', que também é a mensagem devolvida por esse endpoint. Em caso positivo, faremos o this.setState() preparado anteriormente e usaremos o PopUp.exibeMensagem() para enviar uma mensagem de sucesso ao usuário. Do contrário, usaremos o catch() para capturar o erro lançado e o PopUp.exibeMensagem() para mandarmos a mensagem "Falha na comunicação com a API ao listar os autores".

componentDidMount(){

ApiService.ListaNomes()

.then(res => ApiService.TrataErros(res))

.then(res => {

if(res.message === 'success'){

PopUp.exibeMensagem('error', 'Autores listados com sucesso')

this.setState({nomes: [...this.state.nomes, ...res.data]})

}

})

.catch(err => PopUp.exibeMensagem('error', 'Falha na comunicação com a API ao listar os autores'));

}

Feito isso, conseguiremos listar os autores corretamente na aplicação. Repetiremos esse processo para os livros:

componentDidMount(){

ApiService.ListaLivros()

.then(res => ApiService.TrataErros(res))

.then(res => {

if(res.message === 'success'){

PopUp.exibeMensagem('error', 'Livros listados com sucesso')

this.setState({livros: [...this.state.livros, ...res.data]})

}

})

.catch(err => PopUp.exibeMensagem('error', 'Falha na comunicação com a API ao listar os livros'));

}

Assim, a listagem de livros também passará a funcionar, e toda a nossa aplicação está funcionando como deveria. Perceba que o tratamento de erros foi incluído na própria ApiService. Dependendo do escopo do tratamento, vale a pena criar um serviço somente para isso. Porém, como estamos trabalhando em um escopo reduzido, optamos por esse caminho.

# Mãos na Massa

Chegou a hora de você pôr em prática o que foi visto na aula. Para isso, execute os passos listados abaixo.

1 - Modifique sua **ApiService** criando o método **TrataErros** e utilize-o encandeando-o nos outros métodos.

const ApiService = {

ListaAutores : () =>{

return fetch('http://localhost:8000/api/autor')

.then(res => ApiService.TrataErros(res))

.then(res => res.json());

},

CriaAutor : autor => {

return fetch('http://localhost:8000/api/autor', {method: 'POST', headers: {'content-type': 'application/json'}, body: autor})

.then(res => ApiService.TrataErros(res))

.then(res => res.json());;

},

ListaNomes: () =>{

return fetch('http://localhost:8000/api/autor/nome')

.then(res => ApiService.TrataErros(res))

.then(res => res.json());;

},

ListaLivros: () => {

return fetch('http://localhost:8000/api/autor')

.then(res => ApiService.TrataErros(res))

.then(res => res.json());;

},

RemoveAutor: id => {

return fetch(`http://localhost:8000/api/autor/${id}`, {method: 'DELETE', headers: { 'content-type' : 'application/json'},})

.then(res => ApiService.TrataErros(res))

.then(res => res.json());;

},

TrataErros : res =>{

if(!res.ok){

throw Error(res.responseText);

}

return res;

}

}

export default ApiService;

2 - Feito isso, capture os erros nos pontos de utilização da **ApiService**

# Home.js

import React, { Component, Fragment } from 'react';

import 'materialize-css/dist/css/materialize.min.css';

import './Home.css';

import Header from '../../Components/Header/Header';

import Tabela from '../../Components/Tabela/Tabela';

import Form from '../../Components/Formulario/Formulario';

import PopUp from '../../utils/PopUp';

import ApiService from '../../utils/ApiService';

class Home extends Component {

constructor(props){

super(props);

this.state = {

autores: [],

};

}

removeAutor = id => {

const { autores } = this.state;

const autoresAtualizado = autores.filter(autor => {

return autor.id !== id;

});

ApiService.RemoveAutor(id)

.then(res =>{

if(res.message === 'deleted'){

this.setState({autores : [...autoresAtualizado]})

PopUp.exibeMensagem("error", "Autor removido com sucesso");

}

})

.catch(err => PopUp.exibeMensagem("error", "Erro na comunicação com a API ao tentar remover o autor"));

}

escutadorDeSubmit = autor => {

ApiService.CriaAutor(JSON.stringify(autor))

.then(res =>{

if(res.message === 'success'){

this.setState({ autores:[...this.state.autores, res.data] });

PopUp.exibeMensagem("success", "Autor adicionado com sucesso");

}

})

.catch(err => PopUp.exibeMensagem("error", "Erro na comunicação com a API ao tentar criar o autor"));

}

componentDidMount(){

ApiService.ListaAutores()

.then(res => {

if(res.message === 'success'){

this.setState({autores : [...this.state.autores, ...res.data]})

}

})

.catch(err => PopUp.exibeMensagem("error", "Erro na comunicação com a API ao tentar listar os autores"));

}

render() {

return (

<Fragment>

<Header />

<div className="container mb-10">

<h1>Casa do Código</h1>

<Tabela autores={this.state.autores} removeAutor={this.removeAutor} />

<Form escutadorDeSubmit={this.escutadorDeSubmit}/>

</div>

</Fragment>

);

}

}

export default Home;

# Livros.js

import React, { Component, Fragment } from 'react';

import Header from '../../Components/Header/Header';

import DataTable from '../../Components/DataTable/DataTable';

import ApiService from '../../utils/ApiService';

import PopUp from '../../utils/PopUp';

class Livros extends Component {

constructor(props) {

super(props);

this.state = {

livros: [],

titulo: 'Livros'

};

}

componentDidMount(){

ApiService.ListaLivros()

.then(res => {

if(res.message === 'success'){

PopUp.exibeMensagem('success', 'Livros listados com sucesso');

this.setState({livros : [...this.state.livros, ...res.data]});

}

})

.catch(err => PopUp.exibeMensagem('error', 'Falha na comunicação com a API ao listar os livros'));

}

render() {

return (

<Fragment>

<Header />

<div className='container'>

<h1>Página de Livros</h1>

<DataTable dados={this.state.livros} titulo={this.state.titulo} colunas={['livro']} />

</div>

</Fragment>

);

}

}

export default Livros;

# Autores.js

import React, { Component, Fragment } from 'react';

import Header from '../../Components/Header/Header';

import DataTable from '../../Components/DataTable/DataTable';

import ApiService from '../../utils/ApiService';

import PopUp from '../../utils/PopUp';

class Autores extends Component {

constructor(props) {

super(props);

this.state = {

nomes: [],

titulo: 'Autores'

};

}

componentDidMount(){

ApiService.ListaNomes()

.then(res => {

if(res.message === 'success'){

PopUp.exibeMensagem('success', 'Autores Listados com sucesso');

this.setState({nomes: [...this.state.nomes, ...res.data]});

}

})

.catch(err => PopUp.exibeMensagem('error', 'Falha na comunicação com a API ao listar os autores'));

}

render() {

return (

<Fragment>

<Header />

<div className='container'>

<h1>Página de Autores</h1>

<DataTable dados={this.state.nomes} titulo={this.state.titulo} colunas={['nome']} />

</div>

</Fragment>

);

}

}

export default Autores;

3 - Teste tudo no seu navegador.

# O que aprendemos?

Nesta aula, aprendemos:

* Tratamento de erros.
* Exibir erros utilizando o PopUp.js.
* Solução para a página de Autores e Livros.

# Resumo do curso

No último vídeo terminamos de organizar o nosso projeto, e agora faremos um apanhado de tudo que fizemos ao longo desse treinamento. Começamos trabalhando com um FormValidator.js, que primeiramente travava o submit de informações. Depois, fizemos com que ele validasse se o campo "Nome" estava vazia ou não, utilizando o pacote validator do npm. No capítulo seguinte, generalizamos esse código, fazendo com que ele trabalhasse da mesma forma independentemente do campo ao qual fosse aplicado. Para isso, criamos um array de regras de validação que seria utilizado nesses campos, concentrando toda a lógica de validação em um único arquivo.

Preparada a validação, começamos a trabalhar com rotas em nosso index.js. A ideia era exibirmos um componente a depender da rota acessada pelo usuário, como /autores ou /livros. Para isso, utilizamos o react-router-dom, instalado como npm, e as tags <BrowserRouter>, <Switch> e <Route>. Assim, definimos as rotas / (a página inicial), /sobre, /livros e /autores, além de um componente NotFound que é devolvido quando a rota acessada pelo usuário não correspondente com nenhum conteúdo da aplicação, mantendo-o em nosso domínio.

Também alteramos o Header.js, utilizando uma tag <NavLink> do próprio react-router-dom linkar os elementos do cabeçalho. Porém, percebemos que seria necessário importar essa tag em vários pontos do código, poluindo-o com uma funcionalidade provinda de um pacote externo e cuja estabilidade não podemos garantir.

Pensando nisso, e seguindo boas práticas de programação, criaremos um *High Order Component* chamado LinkWrapper que envolve o <NavLink>, concentrando a sua utilização em um único arquivo e permitindo que acessássemos suas funcionalidades em outros pontos do código. Isso também nos trouxe a possibilidade de utilizar o activeStyle para deixarmos um link em negrito quando sua respectiva rota estivesse ativa.

Lançamos o desafio de fazer a tabela de Autores e de Livros, e imaginamos que a maioria tenha criado tabelas separados para cada um desses elementos. Já o nosso instrutor criou uma DataTable.js, uma tabela genérica que consegue trabalhar a partir dos dados que recebe.

Para enviarmos e recebermos dados de uma API, criamos uma ApiService.js contendo métodos para cada um dos endpoints da API disponibilizada para o treinamento. Aprendemos a fazer requisições corretamente, utilizando as Promises para não notificarmos o usuário caso a requisição não esteja completa. Também utilizamos a ApiService.js para fazermos o tratamento dos erros, capturando-os na chamada e exibindo uma mensagem ao usuário a partir do PopUp.js.

Ainda temos muito o que aprender no universo do React, e trataremos de alguns temas, como o Redux, no próximo treinamento.

# Faça o que eu fiz na aula

Chegou a hora de você pôr em prática o que foi visto na aula. Para isso, execute os passos listados abaixo.

1 - Crie suas pastas **Components**, **Pages** e **utils**

2 - Modifique o nome do seu **App.js** para **Home.js** e toda a declaração da sua classe.

3 - Na pasta **Components**, crie uma pasta para cada componente, coloque os respectivos arquivos e corrija as importações necessárias.

A estrutura é a seguinte

Componentes/

|

-- DataTable/

|

--DataTable.js

|

-- Formulario/

|

--Formulario.js

|

-- Header/

|

--Header.js

|

-- Tabela/

|

--Tabela.js

4 - Faça o mesmo processo anterior, mas agora para Pages. Lembre-se de corrigir as importações.

A estrutura é a seguinte

Pages/

|

-- Autores/

|

--Autores.js

|

-- Home/

|

--Home.js

|

-- Livros/

|

--Livros.js

|

-- NotFound/

|

--NotFound.js

|

-- Sobre/

|

--Sobre.js

5 - Faça o mesmo processo anterior para a pasta **utils**, lembre-se de corrigir as importações

utils/

|

-- ApiService.js

|

-- FormValidator.js

|

-- LinkWrapper.js

|

-- PopUp.js

6 - Veja se tudo continua funcionando!

# O que aprendemos?

Nesta aula, aprendemos:

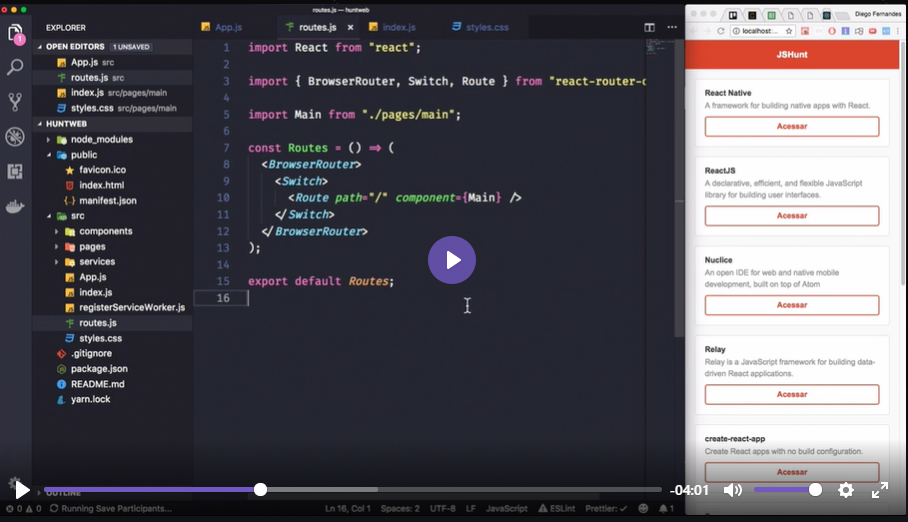
* Organizar um projeto React.
* Separar os componentes por pastas
* Separar as utilidades
* Separar os Page Components

**Conclusão**

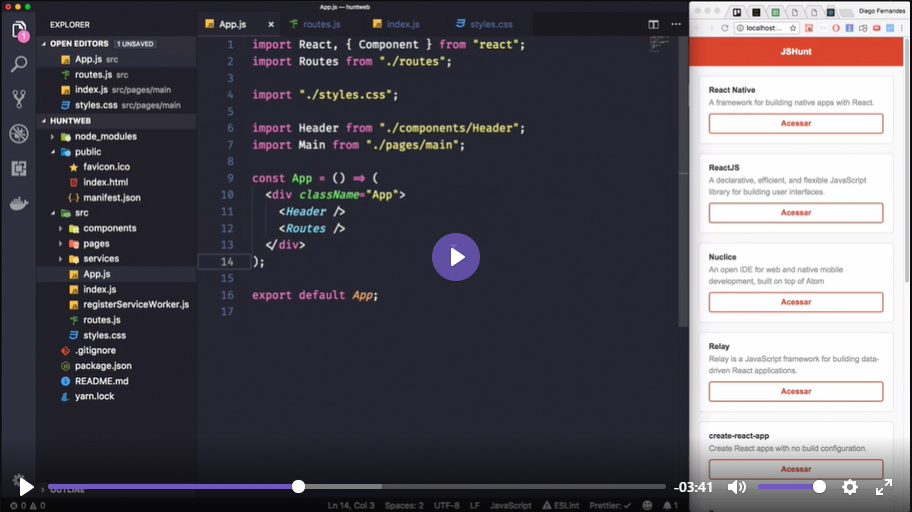
Chegamos ao fim da segunda parte do nosso curso de React. Esperamos que você tenha gostado e que consiga tirar proveito de todas as informações aprendidas ao longo desse treinamento, e que consiga, a partir de hoje, utilizar as ferramentas aprendidas no seu dia-a-dia. Se tiver dúvidas, procure nossos instrutores, monitores e outros alunos no fórum da Alura.

Bons estudos e até a próxima!

**Rotas de maneira Elegante, Rocketseat**

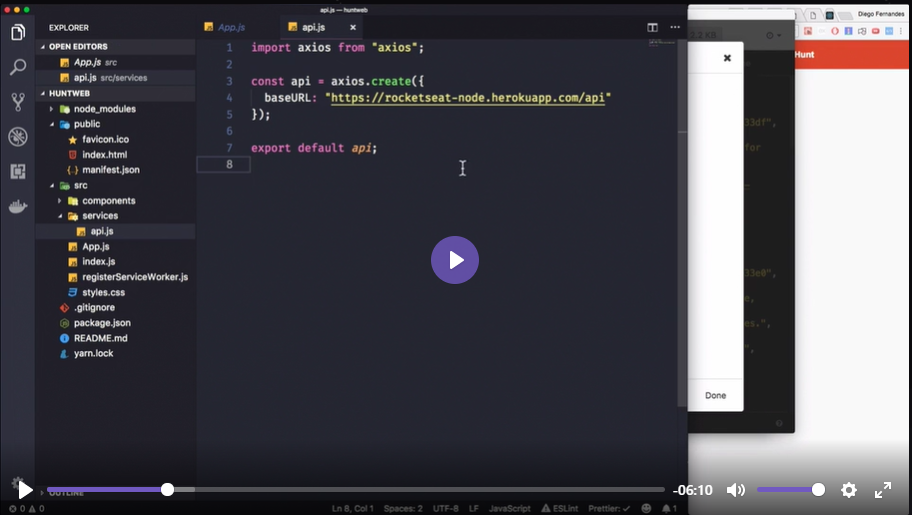


Criar o arquivo routes.js e nele colocar todas as rotas

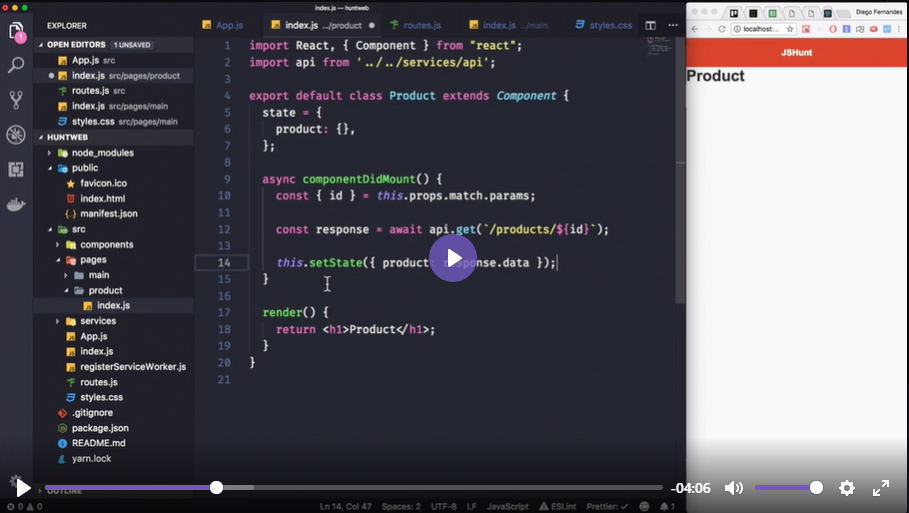


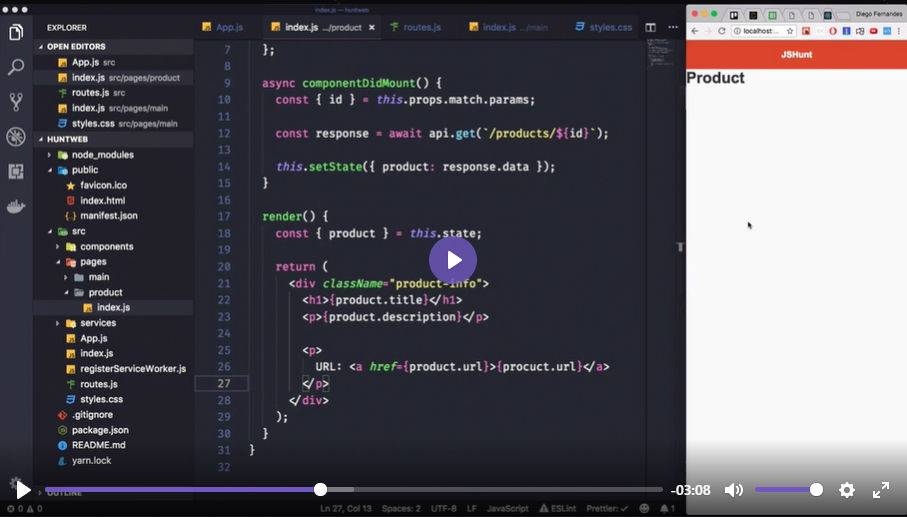
No arquivo onde tudo começa, o principal, importa o Routes e coloca ele dentro do “html”

**Requisições de maneira Elegante, Rocketseat**



Usando axios para fazer as requisições





Usando async e await com o axios e pegando o id da url