

Módulo 01

01 hello world `npx create-react-app "nome do projeto"` e para iniciar o react `npm run start`

02 Criação do projeto com vite `npm create vite@latest` e para iniciar o vite `npm run dev`

Módulo 02

01 criando componentes

Criando componentes

- Na maioria dos projetos os componentes ficam em uma pasta chamada **components**, que devemos criar;
- Geralmente são nomeados com a **camel case**: `FirstComponent.js`;
- No arquivo **criamos uma função**, que contém o código deste componente (a lógica e o template);
- E também precisamos **exportar esta função**, para reutilizá-lo;
- Vamos ver na prática!

```
modulo02 > 01criandoComponentes > src > components > FirstComponents.js > FirstComponents
1  const FirstComponents = () => {
2    return(
3      <div>
4        <h1>Meu primeiro Componente</h1>
5      </div>
6    )
7  }
8
9  export default FirstComponents
```

02 importação de componente

Importando componente

- A importação é a maneira que temos de **reutilizar o componente**;
- Utilizamos a sintaxe: **import X from './components/X'** onde X é o nome do componente;
- Para colocar o componente importado em outro componente, precisamos colocá-lo em forma de tag: **<FirstComponent />**
- E então finalizamos o ciclo de importação;
- Vamos importar o `FirstComponent` em `App`;

```
modulo02 > 01criandoComponentes > src > App.jsx > ...
1  import FirstComponents from './components/FirstComponents'
2
3  function App() {
4
5    return (
6      <>
7        <h1>Fundamento React</h1>
8        <FirstComponents />
9      </>
10    )
11  }
```

03 conhecendo o jsx

- **JSX** é o HTML do React;
- Onde vamos declarar as tags de HTML que serão exibidas no navegador;
- Ficam no **return** do componente;
- Temos algumas diferenças do HTML, por exemplo: class será **className**;
- Isso se dá pelas **instruções semelhantes de JS e HTML**, pois o JSX é JavaScript, então algumas terão nomes diferentes;
- O JSX pode ter apenas **um elemento pai**;

```
ulo02 > 01criandoComponentes > src > components > FirstComponents.js > default
1  const FirstComponents = () => {
2    return(
3      <div>
4        <h1>Meu primeiro Componente</h1>
5        <p className="teste">Meu texto</p>
6      </div>
7    )
8  }
```

04 Comentários no componentes

Comentários no componente

- Podemos inserir comentários de **duas maneiras** no componente;
- Na parte da função, onde é executada a lógica, a sintaxe é: **// Algum comentário**;
- E também no JSX: **{/* Algum comentário */}**
- As chaves nos permitem **executar sentenças em JavaScript**, veremos isso mais adiante;
- Vamos testar os comentários!

```
modulo02 > 01criandoComponentes > src > components > FirstComponents.js > FirstComponents
1  // arquivo de estilo
2
3  const FirstComponents = () => {
4    // essa função faz isso
5    return(
6      <div>
7        {/* Comentário no componente */}
8        <h1>Meu primeiro Componente</h1>
9      </div>
10    )
11  }
```

05 template expressions

- **Template Expressions** é o recurso que nos permite executar JS no JSX e também **interpolam variáveis**;
- Isso será muito útil ao longo dos seus projetos em React;
- A sintaxe é: **{ algumCódigoEmJS }**
- **Tudo que está entre chaves é processado em JavaScript** e nos retorna um resultado;
- Vamos ver na prática!

```
modulo02 > 01criandoComponentes > src > components > TemplateExpressions.jsx > TemplateExpressions
1  const TemplateExpressions = () => {
2    const name = 'Romario'
3    const data = {
4      age: 29,
5      job: 'Programmer'
6    }
7
8    return (
9      <div>
10        <h1>Olá {name}, tudo bem?</h1>
11        <p>você atua como: {data.job}</p>
12      </div>
13    )
14  }
15
16  export default TemplateExpressions
```

06 Hierarquia de componentes

- Os componentes que criamos **podem ser reutilizados em vários componentes**;
- E ainda componentes **podem formar uma hierarquia**, sendo importados uns dentro dos outros, como fizemos em App;
- Vamos ver na prática estes conceitos!

```
modulo02 > 01criandoComponentes > src > components > MyComponent.jsx > ...
```

```
1  const MyComponent = () => {
2    return (
3      <div>
4        <h1>Estou sendo reaproveitado em vários lugares</h1>
5      </div>
6    )
7  }
8
9  export default MyComponent
```

```
const FirstComponents = () => {
  // essa função faz isso
  return(
    <div>
      { /* Comentário no componente */ }
      <h1>Meu primeiro Componente</h1>
      <p className="teste">Meu texto</p>
      <MyComponent />
    </div>
  )
}
```

07 eventos

- Os eventos para o front-end são **essenciais**;
- Em várias situações vamos precisar do click, como ao **enviar formulários**;
- No React os eventos já estão 'prontos', podemos utilizar **onClick** para ativar uma função ao clicar em um elemento;
- Esta função é criada na própria função do componente;
- As funções geralmente tem o padrão **handleAlgumaCoisa**;

modulo02 > 01criandoComponentes > src > components > Events.jsx > Events > handleMyEvent

```
1  const Events = () => {
2
3    const handleMyEvent = (e) => { // o e relacionado ao evento
4      console.log(e)
5    }
6
7    return (
8      <div>
9        <div>
10         <button onClick={handleMyEvent}>clique aqui</button>
11        </div>
12      </div>
13    )
14  }
15
16  export default Events
```

08 Funções no evento

- Quando as funções são simples, podemos **realizar a lógica no próprio evento**;
- Isso **torna nossa código mais 'complicado'**, por atrelar lógica com HTML;
- Mas em **algumas situações** é aplicável;
- Vamos ver na prática!

modulo02 > 01criandoComponentes > src > components > Events.jsx > default

```
1  const Events = () => {
2    const active = true;
3
4    const handleMyEvent = (e) => { // o e relacionado ao evento
5      console.log(e)
6    }
7
8    return (
9      <div>
10        <div>
11          <button onClick={handleMyEvent}>clique aqui</button>
12        </div>
13        <div>
14          <button onClick={() => console.log('Clicou!')}>Clique aqui também</button>
15          <button onClick={() => {
16            if (active) {
17              console.log('isso não deveria está aqui')
18            }
19          }}>aqui também</button>
20        </div>
21      </div>
22    )
23  }
24
25  export default Events
```

09 Funções de renderização

- Podemos criar **funções que retornam JSX**;
- Isso serve para criar situações que **dependam de outras condições**;
- Ou seja, o JSX a ser renderizado pode variar por alguma variável, por exemplo;
- Vamos ver na prática!

```
7
8   const renderSomething = (x) => {
9     if (x) {
10      return <h1>Renderizando isso</h1>
11    } else {
12      return <h1>Ou isso</h1>
13    }
14  }

29
30    {renderSomething(true)}
31  </div>
32  )
33 }
```

Modulo 03

01 Imagens públicas no react

```
modulo03 > src > App.jsx > App
1  function App() {
2
3    return (
4      <>
5        <h1>Avançando em React</h1>
6        <div>
7          {/* Imagem em public acessa pela / */}
8          
9        </div>
10     </>
11   )
12 }
13
14 export default App
```

02 imagens em src

- A pasta public pode ser utilizada para colocar imagens, como fizemos na aula passada;
- Mas um padrão bem utilizada para as imagens dos projetos **é colocar em uma pasta chamada assets**, em src;
- Ou seja, você vai encontrar projetos com as **duas abordagens**;
- Em assets **precisamos importar as imagens**, e o **src é dinâmico** com o nome de importação;


```

modulo03 > src > App.jsx > App
1  import Naruto from './assets/naruto02.jpg'
2
3  function App() {
4
5    return (
6      <>
7        <h1>Avançando em React</h1>
8        <div>
9          { /* Imagem em public access pela */ }
10         
11       </div>
12       <div>
13         { /* Imagem com src */ }
14         <img src={Naruto} alt="naruto" />
15       </div>
16     </>
17   )
18 }
19
20 export default App

```

03 O que são hooks

- Recursos do React que tem **diversas funções**;
- Como: **guardar e alterar o estado de algum dado** na nossa aplicação;
- Todos os hooks começam com **use**, por exemplo: **useState**;
- Podemos criar os nossos hooks, isso é chamado de **custom hook**;
- Os hooks precisam ser **importados**;
- Geralmente são úteis em todas as aplicações, **utilizaremos diversos ao longo do curso**;

07 introdução as props

- **Props** é outro recurso fundamental do React;
- Nos permite **passar valores de um componente pai para um componente filho**;
- Isso será muito útil quando os dados forem carregados via banco de dados, por exemplo;
- As props vem em um objeto no **argumento da função do componente**;
- Vamos ver na prática!

```

const ShowUserName = (props) => {
  return (
    <div>
      <h2>O nome do usuário é {props.nameUser}</h2>
    </div>
  )
}

export default ShowUserName

<ShowUserName nameUser="Romário" />
</>

```

08 desestrutura props

- É super comum passar **mais de uma prop em um componente**;
- Para facilitar isso o React nos permite **desestruturar as propriedades que chegam**, com o recurso de destructuring;
- Se temos duas props: name e age;
- Podemos fazer assim `function MyComponent({name, age})`
- Agora **não precisamos mais utilizar** `props.algumaCoisa`;
- Vamos ver na prática!

```
export function CarDetails({ brand, km, color }) {  
  return(  
    <div>  
      <h2>Detalhes do carros</h2>  
  
      <ul>  
        <li>Marca: {brand}</li>  
        <li>Km: {km}</li>  
        <li>Cor: {color}</li>  
      </ul>  
    </div>  
  )  
}
```

```
<CarDetails brand='bmw' km={1000} color='red' />
```

09 Reaproveitamento de componentes

- Com **props** a **reutilização de componentes** começa a fazer muito sentido;
- Se temos os dados de 1000 carros por exemplo, podemos **reaproveitar o nosso CarDetails 1000 vezes**;
- Isso torna nosso código mais padronizado, facilitando a manutenção;
- Vamos ver na prática!

```
<ShowUserName nameUser="Romário" />  
<CarDetails brand='bmw' km={1000} color='red' newCar={false} />  
<CarDetails brand='fiat' km={0} color='blue' newCar={true} />  
<CarDetails brand='Uno' km={105600} color='white' newCar={false} />
```

10 reutilização com loop

- Os arrays de dados podem ter **muitos itens** também;
- Então o correto é utilizar uma **estrutura de loop (map)** para a sua exibição;
- E com isso conseguimos conciliar os **três conceitos**: renderização de listas, reaproveitamento de componentes e props;
- Vamos ver na prática!

```
function App() {
  const cars = [
    { id: 1, brand: 'bmw', km: 1000, color: 'red', newCar: false },
    { id: 2, brand: 'Uno', km: 0, color: 'white', newCar: true },
    { id: 3, brand: 'Gol Bola', km: 100230, color: 'black', newCar: false },
    { id: 4, brand: 'Ferra', km: 10000, color: 'red', newCar: false },
  ]

  {cars.map((car) => (
    <CarDetails brand={car.brand} km={car.km} color={car.color} newCar={car.newCar} />
  ))}
}
```

11 react fragments

- Os **React fragments** são interessantes para quando precisamos ter mais de um elemento pai em um componente;
- Criamos uma tag vazia: `<> ... </>`
- **E ela serve como elemento pai**, não alterando a estrutura do HTML com uma div, por exemplo;
- Vamos ver na prática!

```
export function Fragment() {
  return (
    <>
      <h2>Título 01</h2>
      <h2>Título 02</h2>
    </>
  )
}
```

12 children props

- **Children prop** é um recurso utilizado para quando um componente precisa ter JSX dentro dele;
- Porém **este JSX vem do componente pai**;
- Então o componente age como um **container**, abraçando estes elementos;
- E children é considerada uma **prop do componente**;
- Vamos ver na prática!

```
export function Container({children, myValue}) {
  return(
    <div>
      <h2>Esse é o título do container</h2>
      {children}
      <p>meu valor é {myValue}</p>
    </div>
  )
}
```

```
<Container myValue='teste'>
  <p>Este é o conteúdo</p>
</Container>
```


13 funções com props

- As **funções podem ser passadas para as props** normalmente;
- Basta criar a função no componente pai e **enviar como prop** para o componente;
- No componente filho ela pode ser ativada por um evento, por exemplo;
- Vamos ver na prática!

```
function showMessage() {  
  console.log('Evento do componente pai')  
}
```

```
<ExecuteFunction onMyFunction={showMessage} />  
</>  
)
```

modulo03 > src > components > ExecuteFunction.jsx > ExecuteFunction

```
1 export function ExecuteFunction({onMyFunction}) {  
2   return(  
3     <div>  
4       <p>daldla awlalwl</p>  
5       <button onClick={onMyFunction}>clique aqui</button>  
6     </div>  
7   )  
8 }
```

14 elevação de state

- Elevação de state ou **State lift** é quando um valor é elevado do componente filho para o componente pai;
- Geralmente temos **um componente que usa o state e outro que o altera**;
- Então precisamos passar a alteração para o componente pai, e este passa para o componente que usa o state;
- Vamos ver na prática!

```
const [message, setMessage] = useState('')
```

```
function handleMessage(msg) {  
  setMessage(msg)  
}
```

```
<Mensagem msg={message} />  
<ChangeMessage onChangeMessage={handleMessage} />  
</>
```

modulo03 > src > components > Message.jsx > Mensagem

```
1 export function Mensagem({msg}) {  
2   return(  
3     <div>  
4       <p>A mensagem é: {msg}</p>  
5     </div>  
6   )  
7 }
```

```

modulo03 > src > components > ChangeMessage.jsx > ChangeMessage
1  export function ChangeMessage({onChangeMessage}) {
2    const messages = ['oi', 'olá', 'roudy']
3    return (
4      <div>
5        <button onClick={() => onChangeMessage(messages[0])}>1</button>
6        <button onClick={() => onChangeMessage(messages[1])}>2</button>
7        <button onClick={() => onChangeMessage(messages[2])}>3</button>
8      </div>
9    )
10 }

```

Módulo 04

01 css global o index.css é o css global da aplicação

```

modulo04 > src > index.css > body
1  body {
2    margin: 0;
3    padding: 0;
4    font-family: 'Courier New', Courier, monospace;
5    text-align: center;
6  }
7
8  h1 {
9    color: red;
10 }

import App from './App.jsx'
import './index.css'

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

```

02 css de componentes usar o class name para a estilização não vazar

- O **CSS de componente** é utilizado para um componente em específico;
- Geralmente é criado um arquivo com o mesmo nome do componente e este é **importado no componente**;
- Note que este método **não é scoped**, ou seja, o CSS vaza para outros componentes se houver uma regra em colisão;
- O React já cria um exemplo desta técnica com o App.css/js;
- Vamos ver na prática!

```

modulo04 > src > components > MyComponents.jsx > MyComponents
1  import './MyComponent.css'
2
3  export function MyComponents() {
4    return (
5      <div>
6        <h2>Meu componente</h2>
7
8        <p className="pMyComponent">Este é o paragrafo do componente</p>
9      </div>
10   )
11 }

modulo04 > src > components > MyComponent.css > .pMyComponent
1  .pMyComponent {
2    color: green;
3    background-color: black;
4  }

```

03 css inline

- O inline style do React é **igual o do CSS**;
- Por meio do **atributo style** conseguimos aplicar regras diretamente em um elemento;
- **Devemos optar por outras maneiras de CSS**, o inline pode dificultar a manutenção ou deixar o código imprevisível em algumas situações;
- Vamos ver na prática!

```
<p style={{ color: "red", backgroundColor: "green", fontWeight: "bold" }}>paragrafo inline</p>  
</div>
```

04 css inline dinâmico

- O **CSS dinâmico inline** aplica estilo baseado em uma condicional;
- Vamos inserir no atributo um **if ternário**;
- Dependendo da condição podemos mudar que regras de estilo um elemento recebe;
- Vamos ver na prática!

```
function App() {  
  const n = 10  
  
  return (  
    <div>  
      <h2 style={n < 20 ?  
        ({ color: "black", backgroundColor: "red" })  
        :  
        ({ color: "red", backgroundColor: "black" })  
      }>Css dinâmico</h2>  
    </div>  
  )  
}
```

05 classes dinâmico

- Podemos também aplicar lógica para **mudar a classe de CSS de um elemento**;
- Também utilizaremos o **if ternário**;
- Essa abordagem é **mais interessante que o CSS inline**;
- Pois as classes estarão isoladas no arquivo de CSS, resolvendo o problema de organização de código;
- Vamos ver na prática!

```
7
8   const hasTitle = false
9
10  return (
11    <div>
12      <h3 className={hasTitle ? "title" : "noTitle"}>Este título vai ter classe dinâmica</h3>
13    </div>
14  )
15
```

modulo04 > src > App.css > .noTitle

```
1  .title {
2    color: pink;
3    background-color: black;
4  }
5
6  .noTitle {
7    color: purple;
8    background-color: aquamarine;
9  }
```

06 css modules

- O **CSS Modules** é um recurso de CSS **scoped**;
- Ou seja, ele vai ser **exclusivo do componente**;
- O nome do arquivo é: **Componente.module.css**;
- Precisamos importá-lo também no componente;
- Vamos ver na prática!

modulo04 > src > components > Title.jsx > Title

```
1  import styles from "./Title.module.css"
2
3  export function Title() {
4    return (
5      <div>
6        <h1 className={styles.title}>Meu Título</h1>
7      </div>
8    )
9  }
```

modulo04 > src > components > Title.module.css > .title

```
1  .title {
2    color: yellow;
3    background-color: darkblue;
4  }
```

Módulo 05

01 Criando form

- No React vamos também utilizar a **tag form** para formulários;
- As labels dos inputs contém o atributo **htmlFor**, que deve ter o valor do nome do input;
- **Não utilizamos action**, pois o processamento será feito de form assíncrona;
- Vamos criar um form!

```
export function MyForms() {
  return (
    <div>
      <form>
        <div>
          <label htmlFor="name">Nome:</label>
          <input type="text" name="name" placeholder="Digite seu nome" />
        </div>

        <input type="submit" value="Enviar" />
      </form>
    </div>
  )
}
```

02 Label envolvendo input

- Em React um padrão comum é a **tag label envolvendo o input**;
- Isso faz com que o atributo for se torne **opcional**;
- **Simplificando nossa estrutura de HTML**, sem perder a semântica;
- Vamos ver isto na prática!

```
<label>
  <span>E-mail:</span>
  <input type="email" name="email" placeholder="Digite seu e-mail" />
</label>
```

03 Gerenciamento de dados no input

Para manipular os valores dos inputs vamos utilizar o **hook useState**;
Ou seja, podemos armazenar na variável e utilizar o **set para alterar o valor**;

Vamos criar uma função para alterar o valor no evento **onChange**;

Deixando nosso código fácil de trabalhar nas próximas etapas: como envio dos dados para BD e validação;

Vamos ver isto na prática!

```
export function MyForms() {
  const [name, setName] = useState();
  const [email, setEmail] = useState();

  function handleName(event) {
    setName(event.target.value)
  }

  return (
    <div>
      <form>
        <div>
          <label htmlFor="name">Nome:</label>
          <input type="text" name="name" placeholder="Digite seu nome" onChange={handleName} />
        </div>
      </form>
    </div>
  )
}
```


04 Alterando os state inline

- Quando temos vários inputs podemos **realizar a manipulação de forma mais simples**;
- Basicamente criamos uma **função inline no onChange**;
- Ela vai **alterar o valor do state** com o método set, da mesma forma que a função isolada;
- Vamos ver isto na prática!

```
<label htmlFor= name >nome:</label>
<input
  type="text"
  name="name"
  placeholder="Digite seu nome"
  onChange={ (e) => setName(e.target.value)} />
</div>
```

05 Envio de form

- Para enviar um form vamos utilizar o evento **onSubmit**;
- **Ele chamará uma função**, e nesta devemos lembrar de parar a submissão com o **preventDefault**;
- Nesta etapa podemos realizar validações, envio de form para o servidor, reset de form e outras ações;
- Vamos ver isto na prática!

```
function handleSubmit(e) {
  e.preventDefault(); // para o envio
  console.log("enviando o formulário");
  alert(`Nome: ${name} email: ${email}`)
}

return (
  <div>
    <form onSubmit={handleSubmit}>
    <div>
```

06 controlled inputs

- **Controlled inputs** é um recurso que nos permite mais flexibilidade nos forms de React;
- Precisamos apenas **igualar o valor ao state**;
- Um uso muito comum: formulários de edição, que os dados vem do back-end, conseguiremos preencher o input mais facilmente;
- Vamos ver isto na prática!

```
<input
  type="text"
  name="name"
  placeholder="Digite seu nome"
  value={name}
  onChange={handleName} />
</div>

<label>
  <span>E-mail:</span>
  <input
    type="email"
    name="email"
    placeholder="Digite seu e-mail"
    value={email}
    onChange={(e) => setEmail(e.target.value)}
  />
</label>
```

07 resetando formulários

- Com o controller inputs limpar o form será **fácil**;
- Basta **atribuir um valor de uma string vazia aos states** e pronto!
- Isso será feito após o envio, em formulários que o usuário precisa preencher novamente;
- Vamos ver isto na prática!

```
function handleSubmit(e) {
  e.preventDefault(); // para o envio
  console.log("enviando o formulário");
  alert(`Nome: ${name} email: ${email}`)

  setEmail('');
  setName('');
}
```

08 Textarea no react

- O textarea **pode ser considerado um input de texto** normal;
- Utilizaremos o **value** para alterar o state inicial;
- E o evento **onChange** para modificar o valor do state;
- Vamos ver isto na prática!

```
<textarea
  name="bio"
  placeholder="Descrição do usuário"
  value={bio}
  onChange={(e) => setBio(e.target.value)}
></textarea>
```

09 Select no react

- O select também será **muito semelhante** aos outros inputs;
- Quando temos a alteração de um valor o **evento onChange** pode captar isso;
- O value também pode atribuir qual **option** estará selecionada;
- Vamos ver isto na prática!

```
<label>
  <span>Função do Sistema</span>
  <select name="role" onChange={(e) => setRole(e.target.value)} value={role}>
    <option value="user">Usuário</option>
    <option value="editor">Editor</option>
    <option value="admin">Adm</option>
  </select>
</label>
```

Módulo 06

Object.key

Características, como por exemplo:

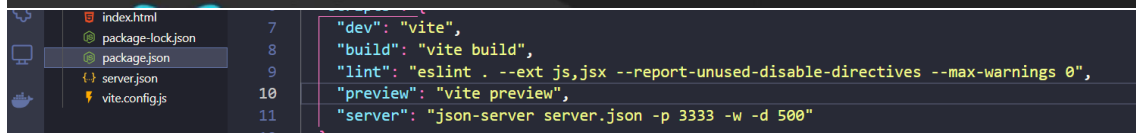
`Object.keys` é um método em JavaScript que retorna um array contendo as propriedades enumeráveis de um objeto passado como parâmetro. No código apresentado, o `Object.keys` é utilizado para obter as categorias disponíveis no objeto `words`, que é importado do arquivo `words.js`. Com isso, é possível escolher aleatoriamente uma categoria e uma palavra dessa categoria para o jogo, por meio das funções `pickWordAndCategory` e `startGame`, respectivamente.

Se for usar uma função na dependência de monitoramento do react js ai vc usa o call-back

Módulo 07

01 criando projeto com json server npm i json-server

- O **JSON server** é um pacote npm;
- Ele **simula uma API**, e isso nos possibilita fazer requisições HTTP;
- Vamos aprender a **integrar este recurso com o React**;
- Podemos entender isso como uma etapa de preparação para APIs reais;
- Ou seja, atingir o mesmo resultado mas sem precisar de uma estrutura no back-end;
- Vamos criar um projeto e instalar o JSON server;



```
{
  "dev": "vite",
  "build": "vite build",
  "lint": "eslint . --ext js,jsx --report-unused-disable-directives --max-warnings 0",
  "preview": "vite preview",
  "server": "json-server server.json -p 3333 -w -d 500"
}
```

02 importando do useeffect

- O **useEffect** faz com que determinada ação seja executada apenas uma vez;
- Isso é interessante pois os componentes estão sempre se **re-renderizando**, então precisamos ter **ações únicas** às vezes;
- O useEffect ainda possui um **array de dependências**, que deve conter os dados que ativem a execução da função de forma automática;
- O useEffect estará presente sempre nas **requisições assíncronas!**

03 resgatando dados da api

- Para trazer os dados vamos ter que utilizar vários recursos;
- Primeiramente ter um local para salvá-los (**useState**);
- Renderizar a chamada a API apenas uma vez (**useEffect**);
- Um meio de fazer a requisição assíncrona (**Fetch API**);
- Vamos ver isto na prática!

```

function App() {
  const [products, setProducts] = useState([]);
  const url = "http://localhost:3333/products";

  async function fetchData() {
    const res = await fetch(url)
    const data = await res.json();
    setProducts(data)
  }

  useEffect(() => {
    fetchData()
  }, [])

  console.log(products)

  return (
    <div className='App'>
      <h1>lista de produtos</h1>
      <ul>
        {products.map((products) => {
          return (
            <li key={products.id}>{products.name} - R$: {products.price}</li>
          )
        })}
      </ul>
    </div>
  )
}

```

04 adicionando dados com react

- Para adicionar um item vamos precisar resgatar os dados do form com o **useState**;
- Reunir eles em uma **função após o submit** e enviar um request de **POST** para a nossa API;
- O processo é bem parecido com o de resgate de dados, mas agora estamos **enviando dados**;
- Vamos ver isto na prática!

```

const url = "http://localhost:3333/products";

function App() {
  const [products, setProducts] = useState([]);
  const [name, setName] = useState("");
  const [price, setPrice] = useState("");

  const fetchData = async() => {
    const res = await fetch(url)
    const data = await res.json();
    setProducts(data)
  }

  const handleSubmit = async(e) => {
    e.preventDefault();

    const product = { name, price };

    const res = await fetch(url, {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify(product),
    });
  }
}

```



```

<div className='App'>
  <h1>lista de produtos</h1>
  <div className='addProduct'>
    <form onSubmit={handleSubmit}>
      <label> Name:
        <input type="text" value={name} name="name" onChange={(e) => setName(e.target.value)} />
      </label>

      <label> Preço $:
        <input type="number" value={price} name="price" onChange={(e) => setPrice(e.target.value)} />
      </label>

      <input type="submit" value="enviar" />
    </form>
  </div>
</div>

```

05 carregamento de dados dinâmico

Se a requisição foi feita com sucesso, podemos **adicionar o item a lista após o request**;

Isso torna nossa aplicação mais **performática**;

Utilizaremos o **set do useState** para isso;

Vamos ver isto na prática!

```

const handleSubmit = async(e) => {
  e.preventDefault();

  const product = { name, price };

  const res = await fetch(url, {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify(product),
  });

  const addedProduct = await res.json();

  setProducts(state => [...state, addedProduct])

  setName("");
  setPrice("");
}

```

6 custom Hook para resgatar de dados

É normal dividir funções que podem ser reaproveitadas em hooks;

Esta técnica é chamada de **custom hook**, e vamos criar um para o resgate de dados;

Os hooks geralmente ficam na **pasta hooks**;

Devemos utilizar o padrão **useName**;

Basicamente criamos uma função e exportamos ela;

Vamos ver isto na prática!

```

src > hooks > useFetch.jsx > useFetch
1  import { useState, useEffect } from "react"
2
3  export const useFetch = (url) => {
4    const [data, setDate] = useState(null);
5
6    const fetchData = async() => {
7      const res = await fetch(url);
8      const json = await res.json();
9      setDate(json);
10   }
11
12   useEffect(() => {
13     fetchData();
14   }, [url]);
15
16   return { data };
17 }

```

```

8
9  function App() {
10    const { data } = useFetch(url);
11
12    <ul>
13      {data && data.map((products) => {
14        return (
15          <li key={products.id}>{products.name} - R$: {products.price}</li>
16        )
17      })}
18    </ul>

```

07 refatorando o post

- Podemos **utilizar o mesmo hook** para incluir uma etapa de POST;
- **Vamos criar um novo useEffect** que mapeia uma outra mudança de estado;
- Após ela ocorrer executamos a adição de produto;
- **Obs:** nem sempre reutilizar um hook é a melhor estratégia;
- Vamos ver isto na prática!

```

src > hooks > useFetch.jsx > useFetch > httpRequest
1  import { useState, useEffect } from "react"
2
3  export const useFetch = (url) => {
4    const [data, setDate] = useState(null);
5    const [config, setConfig] = useState(null);
6    const [method, setMethod] = useState(null);
7    const [callFetch, setCallFetch] = useState(false);
8
9    const fetchData = async() => {
10     const res = await fetch(url);
11     const json = await res.json();
12     setDate(json);
13   }
14
15   const httpRequest = async () => {
16     if (method === "POST") {
17       let fetchOptions = [url, config];
18
19       const res = await fetch(...fetchOptions);
20       const json = await res.json();
21
22       setCallFetch(json);
23     }
24   }
25

```

```

const httpConfig = (data, method) => {
  if (method === "POST") {
    setConfig({
      method,
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify(data)
    })
  }

  setMethod(method);
}

useEffect(() => {
  fetchData();
}, [url, callFetch]);

useEffect(() => {
  httpRequest();
}, [config, method, url]);

return { data, httpConfig };
}

```

```

function App() {
  const { data, httpConfig } = useFetch(url);

  const [name, setName] = useState("");
  const [price, setPrice] = useState("");

  const handleSubmit = async(e) => {
    e.preventDefault();

    const product = { name, price };
    httpConfig(product, "POST");

    setName("");
    setPrice("");
  }
}

```

08 estados de loading

- Quando fizermos requisições para APIs **é normal que haja um intervalo de loading entre a requisição e o recebimento** da resposta;
- Podemos fazer isso no nosso **hook** também;
- **Identificar quando começa e termina** este estado;
- Vamos ver isto na prática!

```

const [isLoading, setIsLoading] = useState(false);
const [loading, setLoading] = useState(false);

```

```

const fetchData = async() => {
  setLoading(true);
  const res = await fetch(url);
  const json = await res.json();
  setDate(json);
  setLoading(false);
}

```

```

return { data, httpConfig, loading };

```

```

{loading ?
  <p>Carregando</p>
  :
  <ul>
    {data && data.map((products) => {

```

09 loading no post

- Podemos bloquear ações indevidas em outras requests também, **como no POST**;
- Uma ação interessante é **remover a ação de adicionar outro item** enquanto o request ainda não finalizou;
- Vamos ver isto na prática!

```
<input type="submit" value="enviar" disabled={loading} />
```

```
const httpRequest = async () => {  
  setLoading(true);
```

10 tratando error

Podemos tratar os erros das requisições por meio de um **try catch**;

Além de pegar os dados do erro, também podemos **alterar um state para imprimir um elemento se algo der errado**;

Desta maneira conseguimos **prever vários cenários** (dados resgatados, carregamento e erro);

Vamos ver isto na prática!

```
const fetchData = async() => {  
  try {  
    setLoading(true);  
    const res = await fetch(url);  
    const json = await res.json();  
    setDate(json);  
  } catch(err) {  
    console.log(err.message);  
  } finally {  
    setLoading(false);  
  }  
}
```