Front-end: React

React - Aula 05

Prof. MSc. Kelson Almeida

Agenda

- HTTP com React
  - Criando servidor HTTP fake
  - Resgatando dados (GET)
  - Enviando dados (POST)

JSON Server

- JSON Server é uma biblioteca que permite criar rapidamente uma API RESTful fake (simulada) usando apenas um arquivo JSON.
- Muito útil para devs frontend que precisam testar e prototipar seus aplicativos antes de implementar o backend real.

```
CSS

npm install json-server
```

JSON Server

- Nos exemplos ao lado temos a instalação do pacote JSON Server através do comando:
  - npm install json-server
- É preciso que no arquivo “package.json” adicionemos a propriedade “server” em “scripts”.
- Assim criaremos um “watch” do json-server para o arquivo data/db.json
- O servidor json-server é inicializado em outro terminal através do comando:
  - npm run server

```
CSS

npm install json-server
```

JSON Server

- No arquivo data/db.json, nós criamos um json que irá abstrair um possível banco de dados do qual queremos extrair os dados a serem retornados via API. (fake back-end)
- Nestes exemplos, o json-server será inicializado na porta 3000. E um endpoint http://localhost:3000/products ficará disponível para consulta.

Vamos praticar?

- Através do seu projeto React crie uma RESTful api fake que retorne uma lista de Alunos, onde cada aluno terá nome, email e curso.
- Faça um get nessa lista de alunos, onde o mesmo retornará a lista desses alunos.
- Mostre o resultado em um cliente HTTP.

Resgatando dados [GET]

- Vamos entender um pouco sobre o hook *useEffect*
- Este hook é utilizado para executar um efeito (como **chamada de API**, alteração no DOM ou atualização de estado).
- O *useEffect* tece dois argumentos:
  - Uma função que contém o efeito que você deseja executar.
  - Um array de dependências opcional que informa ao React quando o efeito deve ser executado.

Resgatando dados [GET]

- No exemplo ao lado criamos uma variável de estado “products” com um valor inicial vazio [].
- Em seguida usamos o *useEffect* para buscar os dados da API e atualizar o estado de “products”.
- Dentro da função assíncrona passada dentro do *useEffect* esperamos a resposta usando “await”, em seguida convertemos a resposta em um objeto JSON usando o .json e atualizamos o estado de products.
- O array presente na linha 19 controla quando o efeito será executado, se passarmos um array vazio, como foi o caso, o efeito será executado apenas uma vez, imediatamente após a montagem do componente. Isso é útil para buscar os dados da API apenas 1 vez, quando o componente é renderizado pela primeira vez.

```
1  import { useState, useEffect } from "react"
2
3  const url = "http://localhost:3000/products"
4
5  const ResgatandoDados = () => {
6    const [products, setProducts] = useState([])
7
8    // 1 - resgatando dados
9    useEffect(() => {
10     async function fetchData() {
11
12       const res = await fetch(url)
13
14       const data = await res.json()
15
16       setProducts(data)
17     }
18     fetchData()
19   }, [])
```

Resgatando dados [GET]

- Tá, prof. Mas e esses “await” aí no código?
- “await” é uma palavra reservada do JavaScript que pode ser usada dentro de funções assíncronas para aguardar a conclusão de uma operação assíncrona antes de continuar a execução do código.
- No nosso exemplo o “await” é usado para esperar a resposta da API, assim garantindo que teremos uma resposta antes de proceder com o processo de busca dos produtos na tela.

```
1  import { useState, useEffect } from "react"
2
3  const url = "http://localhost:3000/products"
4
5  const ResgatandoDados = () => {
6    const [products, setProducts] = useState([])
7
8    // 1 - resgatando dados
9    useEffect(() => {
10     async function fetchData() {
11
12       const res = await fetch(url)
13
14       const data = await res.json()
15
16       setProducts(data)
17     }
18     fetchData()
19   }, [])
```

Resgatando dados [GET]

- E o que é uma operação assíncrona?
- É uma operação que ocorre em paralelo com a execução do código.
- Ou seja, não depende do término de outras operações.
- Por isso precisamos tratar com *await* para garantir que teremos a *response* que desejamos na operação e assim só continuar a operação quando a busca finalizar.

```
1  import { useState, useEffect } from "react"
2
3  const url = "http://localhost:3000/products"
4
5  const ResgatandoDados = () => {
6    const [products, setProducts] = useState([])
7
8    // 1 - resgatando dados
9    useEffect(() => {
10     async function fetchData() {
11
12       const res = await fetch(url)
13
14       const data = await res.json()
15
16       setProducts(data)
17     }
18     fetchData()
19   }, [])
```

Vamos praticar?

- Crie uma página que retorne a lista de alunos que a sua API fake está retornando.
- Retorne esses dados organizados em uma tabela (table).

Adicionando Dados [POST]

- A função handleSubmit é executada quando o formulário é submetido. O preventDefault impede que a página seja recarregada ao formulário ser enviado.
- Em seguida o productToAdd é o objeto literal que posteriormente será convertido em JSON
- Note que o “response” tem uma chamada com await apontando para um “fetch” que executa o POST para salvar o formulário.

```
1  import { useState } from "react"
2
3  const url = "http://localhost:3000/products"
4
5  const AdicionandoDados = () => {
6
7      const [name, setName] = useState("")
8      const [price, setPrice] = useState("")
9
10     const handleSubmit = async (e) => {
11       e.preventDefault()
12
13       const productToAdd = {
14         name: name,
15         price: price
16       }
17
18       const response = await fetch(url, {
19         method: 'POST',
20         headers: {
21           "Content-Type": "application/json"
22         },
23         body: JSON.stringify(productToAdd)
24       })
25
26     }
```

Adicionando Dados [POST]

- A função handleSubmit é executada quando o formulário é submetido. O preventDefault impede que a página seja recarregada ao formulário ser enviado.
- Em seguida o productToAdd é o objeto literal que posteriormente será convertido em JSON
- Note que o “response” tem uma chamada com await apontando para um “fetch” que executa o POST para salvar o formulário.

```
33      const response = await fetch(url, {
34        method: 'POST',
35        headers: {
36          "Content-Type": "application/json"
37        },
38        body: JSON.stringify(productToAdd)
39      })
40
41      const addedProducts = await response.json()
42
43      setProducts((prevProducts) => [...prevProducts, addedProducts])
44
45      setName("")
46      setPrice("")
47
48    }
```

Adicionando Dados [POST]

- A função handleSubmit é executada quando o formulário é submetido. O *preventDefault* impede que a página seja recarregada ao formulário ser enviado.
- Em seguida o *productToAdd* é o objeto literal que posteriormente será convertido em JSON
- Note que o “response” tem uma chamada com await apontando para um “fetch” que executa o POST para salvar o formulário.

```
28    return (
29      <div>
30        <form onSubmit={handleSubmit}>
31          <label>
32            Nome:
33            <input type='text' name='name' value={name} onChange={(e) => setName(e.target.value)} />
34          </label>
35          <br/>
36          <label>
37            Preço:
38            <input type='number' name='price' value={price} onChange={(e) => setPrice(e.target.value)} />
39          </label>
40          <br/>
41          <input type='submit' value='Salvar' />
42
43        </form>
44      </div>
45    )
```

Vamos praticar?

- Crie um formulário que adicione novos alunos nessa API fake e a lista (a tabela) atualize automaticamente.
- A mesma página carrega a lista de alunos e tem o formulário para salvar novos alunos.