Minicurso IFTech 2019.2

Título: O ronco do V8
Resumo: Já imaginou utilizar uma única tecnologia para desenvolver sua aplicação, desde o seu frontend até a persistência em um banco de dados? Isso possível com o poderoso javascript. Neste curso iremos desenvolver uma aplicação utilizando javascript em todas essas fases. Com React, NodeJS e MongoDB.
Pré-requisito: Conhecimento de JS
Orientadora: Aline Morais

O que é V8?
É um engine Javascript de código aberto criada pelo Projeto Chromium para os navegadores Google Chrome e Chromium. O criador do projeto é Lars Bak. Ela também é usada no lado do servidor (NodeJS, MongoDB).

Mas o que é uma engine Javascript?
É um programa ou um interpretador que executa código Javascript.

Objetivo deste minicurso: construir uma aplicação web usando a arquitetura orientada a recursos, REST. Ela será uma rede social de DEVs, onde o usuário vai acessar com seu username do github e dará link e deslike em DEVs.

Steps:

1. Um bem vindo ao minicurso O Ronco do V8 (server.js)
    a. Clonar o projeto
    b. Criar um server (Express, GET)

```
const express = require('express');
const app = express();

app.get('/', (req, res) => {
   // return res.json({ message: `Hellooooo ${req.query.name}`});
   return res.send(`Bem vindo ao mundo JS, ${req.query.name}`);
});
```

    c. Ouvir uma porta, para todas as redes

```
app.listen(8081, '0.0.0.0');
```

    d. Abrir no Chrome localhost:8081
    e. Testar com HTTPie

Boas vindas

```
http localhost:8081 username==elileal
```

f. Usar JSON na resposta

```
app.get('/', (req, res) => {
   return res.json({ message: `Bem vindo ao mundo JS,
${req.query.name}`});
});
```

2. Rotas (routes.js)
    a. Explicar a estrutura de pastas (src)
    b. Criar o arquivo de rotas (Express)

```
const express = require('express')
const routes = express.Router()

routes.get('/', (req, res) => {
   return res.json({ message: `Bem vindo ao mundo JS,
${req.query.name}`});
});

module.exports = routes
```

3. Conectar com o banco (server.js)

```
const mongoose = require('mongoose');
const databaseURI = 'mongodb://mongo:27017/sect2019'
mongoose.connect(
   databaseURI,
   {
       useNewUrlParser: true,
       autoIndex: false
   })
   .then(result => {
       console.log('MongoDB Conectado');
   })
   .catch(error => {
       console.error(`[SERVER_ERROR] MongoDB initial connection
errors:`, error);
   })
```

4. Body JSON (Conceito de API Rest)

```
app.use(express.json())
```

5. Models (models/Dev.js)
    a. MVC - Model é a abstração da nossa tabela do banco de dados, a view é em React, controller são as regras de negócio.

https://copy-paste.online/

```
const { Schema, model } = require('mongoose')

const DevSchema = new Schema({
   name: {
       type: String,
       required: true,
   },
   user: {
       type: String,
       required: true,
   },
   bio: String,
   avatar: {
       type: String,
       required: true,
   },
   likes: [{
       type: Schema.Types.ObjectId,
       ref: 'Dev',
   }],
   dislikes: [{
       type: Schema.Types.ObjectId,
       ref: 'Dev',
   }],

}, {
   timestamps: true,
});

module.exports = model('Dev', DevSchema);
```

6. Controllers (controllers/DevController.js)

```
const Dev = require('../models/Dev')

module.exports = {
   store(req, res){
```

```
        return res.json({ ok: true});
    }
}
```

routes.js

```
const DevController =require('./controllers/DevController')
routes.post('/devs', DevController.store)
```

DevController.store

```
const { username } = req.body;
const response = await
axios.get(`https://api.github.com/users/${username}`)
return res.json(response.data);
```

Axios é assíncrono (async await)

7. Persistindo (DevController.store)

```
const { name, bio, avatar_url: avatar} = response.data
const dev = await Dev.create({
            name,
            user: username,
            bio,
            avatar
        })
return res.json(dev);
```

Short sintaxe

Não permitir usuários repetidos

```
const userExists = await Dev.findOne({ user:username})

        if (userExists){
            return res.json(userExists)
        }
```

O controller não pode ter mais métodos que Index, Show, Store, Update, Delete

```
Index: Lista os recursos
Show: Mostrar um único recurso
Store: Para armazenar
Update e Delete
```

LikeController.store

```
const Dev = require('../models/Dev')

module.exports = {
   async store(req, res){



       return res.json(loggedDev)


   }
}
```

Cadastrar user

```
http POST localhost:8081/devs username=elileal
```

Listar users

```
   http localhost:8081/devs user:<user._id>
```

Preciso do desenvolvedor que está dando o like e o que está recebendo o like

Nova rota

```
http://localhost:3333/devs/<devID>/likes
```

routes.js

```
routes.post('/devs/:devId/likes', LikeController.store)
```

Usando o header, são informações enviadas que não tem haver com o recurso que estamos manipulando, sem corpo.
Como não existe autenticação enviamos o usuário (logado) no header
Como parâmetro <devID> o usuário que receberá o like

```
const { user } = req.headers
const { devId } = req.params
const loggedDev = await Dev.findById(user)
const targetDev = await Dev.findById(devId)
```

```
if (!targetDev){
      return res.status(400).json({error: 'Dev not exists'})
}

loggedDev.likes.push(targetDev._id)
await loggedDev.save()
```

Vamos criar um controller de dislikes para não apareçam mais os usuários que ele deu deslikes

```
const Dev = require('../models/Dev')

module.exports = {
   async store(req, res){
       const { user } = req.headers
       const { devId } = req.params

       const loggedDev = await Dev.findById(user)
       const targetDev = await Dev.findById(devId)

       if (!targetDev){
           return res.status(400).json({error: 'Dev not exists'})
       }

       loggedDev.dislikes.push(targetDev._id)
       await loggedDev.save()

       return res.json(loggedDev)

   }
}
```

Visualizar os dados no mongo express

Nova rota para listar os likes

routes.js

```
routes.get('/devs', DevController.index)
```

DevController.index
Três condições para listagem dos usuários, não são: o próprio usuário logado, os usuários que ele deu like ou dislike

```
async index(req,res){
       const { user } = req.headers
       const loggerDev = await Dev.findById(user)

       const users = await Dev.find({
```

```
            $and: [
                { _id : {$ne:user}},
                { _id : {$nin: loggerDev.likes}},
                { _id : {$nin: loggerDev.dislikes}},
            ],
        })
        return res.json(users);
    },
```

Liberando o acesso para referência cruzada (CORS)
server.js

```
const cors = require('cors');
```

Antes das rotas

```
app.use(cors())
```

REACT

1. Deixar o backend rodando
2. No git já existe um projeto react pronto, mas você pode criar um projeto com o comando, já tiver o node instalado na sua máquina:

```
$ npx create-react-app frontend
```

3. Editar o App.js dentro <p>

Bem vindo ao mundo JS

4. Pasta public, o arquivo index.html serve para o React inserir o conteúdo
    a. View source code (chrome)
    b. Desabilitar o JS (Inspecionar/[...]configurações/Debugger/Disable Javascript) + F5. Toda a aplicação é gerada pelo JS e não pelo HTML
    c. IMPORTANTE

```
<div id="root"></div>
```

5. Deletar

```
<link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
```

assim como o arquivo na pasta source

```
<link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
```

+ o arquivo: serve para construção de PWA

```
<meta name="theme-color" content="#000000" />
```

cor na barra de endereços do navegador android

Retirar comentários também

6. Título: DevCom

7. Como o react embutiu esse code na div?
    a. index.js: arquivo de entrada da aplicação, sempre o react inicializar vai procurar por ele.
    b. Retirar o service worker
    c. Analisando o que restou:
        i. React - principal lib a ser importada
        ii. React DOM - Como o ambiente é Web usá-se ela. Versão do react que lida com a DOM, árvore de elementos. Para criar, deletar elementos esse lib é essencial.
        iii. Render:

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

Pega o arquivo indicado (<App />) e coloca na div#root
Poder ser qualquer html

```
ReactDOM.render(<h1>Bem vindo ao mundo JS</h1>,
document.getElementById('root'));
```

Normalmente se usa o render apenas uma vez dentro da aplicação, ele serve, pode-se dizer, pra cadastrar o componente global da aplicação.

8. O que é um componente?
    a. Basicamente é uma função que retorna um conteúdo html (ver App.js)

```
function App() {
 return (
   <h1>Bem vindo ao mundo JS</h1>
 );
}
```

Pode-se colocar qualquer html que será renderizado no div#root
Inclusive pode-se importa outros componentes (cenas dos próximos capítulos)

9. Limpar o projeto, deletar: serviceWorker.js | index.css | limpar o App.css | App.test.js
10. Importa a logo e outros materiais para a pasta src/assets
    a. O import é por dentro do Java Script

```
import logo from './assets/logl.svg'
```

```
<img src="logo"/>
```

Assim ele coloca uma string no src

```
<img src={logo} alt="DevCom"/>
```

As chaves indica que é um código JS, pode-se assim fazer if ou algum cálculo

11. Componentes
    a. Criar a pasta src/pages
    b. Cada página na aplicação será um componente

c. Login.js, sempre importar o React em todo componente
d. Todo componente será uma função.

```
function Login() {
   return (
       <img src={logo} alt="DevCom" />
   );
}
```

e. Exportar o componente

```
export default function Login() {
```

f. importar o Login para App

```
import Login from './pages/Login'
function App() {
 return (
   <Login />
 );
}
```

O nome do componente será uma Tag HTML
Essa divisão auxilia na modulação do código quando fomos usar o roteamento
Com isso também é possível criar um css para cada componente

```
import './Login.css';
```

Quando é só um arquivo que não precisa nomear a importação fica assim

g. Estilos globais para a aplicação

App.css

```
* {
 margin: 0;
 padding: 0;
 outline: 0;
 box-sizing: border-box;
}

html, body, #root {
 height: 100%;
}

body {
 background: #f5f5f5;
}
```

```
body, input, button {
 font-family: Arial, Helvetica, sans-serif;
}
```

h. Criando a estrutura do Login
   i. div.login-container: ao redor da estrutua
   ii. Obs: sempre usar className, apesar do React também entender class, mas esta é uma palavra reservada do JS
   iii. Estilizando com Login.css https://copy-paste.online/

```
.login-container {
   height: 100%;
   display: flex;
   justify-content: center;
   align-items: center;
}

.login-container form {
   width: 100%;
   max-width: 300px;
   display: flex;
   flex-direction: column;
}

.login-container form input {
   margin-top: 20px;
   border: 1px solid #ddd;
   border-radius: 4px;
   height: 48px;
   padding: 0 20px;
   font-size: 16px;
   color: #666;
}

.login-container form input::placeholder {
   color: #999;
}

.login-container form button {
   margin-top: 10px;
   border: 0;
```

```
    border-radius: 4px;
    height: 48px;
    font-size: 16px;
    background: #DF4723;
    font-weight: bold;
    color: #FFF;
    cursor: pointer;
}
```

12. Roteamento
    a. react-router-dom: ela está disponível para vários sistemas, mas como estamos na web usamos o dom.
    b. src/routes.js:
        i. import

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

        ii. BrowserRouter: é o roteamento no browser. A rota fica separada por uma barra (localhost:3001/devs). Existem outros tipo como o hash router que adiciona o /#.
        iii. Route: objeto que fará o roteamento da aplicação
        iv. As rotas também são componentes, logo são funções exportadas. E retornam um conteúdo html, como chamamos de JSX, nome da sintaxe que nem é JS, nem é HTML.

```
      <BrowserRouter>
           <Route>


           </Route>
       </BrowserRouter>
```

        v. Cada página terá um Route
        vi. Criar a pa ǵ ina Main.js e importar em routes.js

```
import React from 'react';

export default function Main(){
    return (
        <h1>Bem vindo ao mundo JS</h1>
    );
}
```

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

        vii. Criando a rota de Login e Main

```
      <BrowserRouter>
           <Route path="/" component={Login} />
           <Route path="/main" component={Main} />
       </BrowserRouter>
```

viii. App.js: importar as rotas

```
import Routes from './routes'

function App() {
 return (
   <Routes />
 );
}
```

Quando executar no browser o /main não vai funcionar, pois essa lib react-router-dom não verificar por default a exatidão do path, só verifica se ele começa com o que foi passado. Pra resolver isso usa-se o exact na primeira rota.

13. State: precisamos pegar os dados digitados
    a. State: é toda e qualquer informação que o componente vai manipular, que ele precise modificar, acessar.

Login.js

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

b. Inicialização: vazio

```
const username = useState('');
```

Se quiser inicializar com outro valor, basta colocar dentro da função.
Além do valor da variável, ela também retorna uma função em formato de array, que serve para modificar o valor de username. E para acessar o valor da variável, chame username

```
const [username, setUsername] = useState('');
```

c. value e onChange: value é o valor do input, já o onChange é uma função nativa do javasript que dispara toda vez nas alterações do input.
Ela retorna um evento. Usando o e.target.value você pode capturar o valor.

```
        <input
                placeholder="Digite seu usuário no GitHub"
                value={username}
                onChange={e => setUsername(e.target.value)}
            />
```

d. Função dentro de função: para manipular os dados

Como estamos usando um <form> ele tem uma função nativa chamada onSubmit que é disparada assim que o form é enviado. Essa função passa um evento, podemos usá-la para manipular o input.

```
          <form onSubmit={handleSubmit}>
```

E para prevenir o comportamento padrão do form que é de enviar um POST para uma página, usamos uma função que bloqueia esse comportamento default

```
const handleSubmit = (e) => {
       e.preventDefault();
   }
```

Podemos com isso, usar um console.log para exibir o username, isso é visto na ferramenta de desenvolvedor do Chrome (F12)

Aprendemos até agora dois conceitos do React: Componentes e Estados

14. Propriedades: são os atributos que passamos para cada uma das Tags. No React é possível passar propriedades pro meus componentes.
    a. Como enviar? Basta no seu componente, para como atributo/valor

App.js

```
<Login x={x}/>
```

    b. Como pegar? O objeto props é sempre passado de componente para componente, basta acessar a nossa propriedade através dele. Ou desestruturar e pegar a propriedade através dele.

```
export default function Login(props) {
   console.log(props.x);
```

ou desestruturando

```
export default function Login({x}) {
   console.log(x);
```

    c. history: herdado em todas as rotas. E serve para fazer a navegação das páginas

```
export default function Login({history}) {

   const [username, setUsername] = useState('');

   const handleSubmit = (e) => {
       e.preventDefault();
       history.push('/main');
   }
```

Ele vai empilhar (push) as páginas

Sucesso! Aprendemos os três conceitos do React: componentes, estados e propriedades

15. Chamada a API (axios): do retorno vamos anotar o id (userLogged) para depois poder utilizar para fazer, por exemplo a listagem do devs
    a. src/services: para configurar a nossa api

Passamos um objeto para configurar

```
import axios from 'axios';

const api = axios.create({
   baseURL: 'http://localhost:3001'
});

export default api;
```

    b. importando a api

```
import api from '../services/api'
```

    c. rota post: passando apenas o que não foi informado na configuração e como segundo parâmetro, enviamos os dados que iriam no corpo da requisição

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

       const response = await api.post('/devs', {
           username,
       });
       history.push('/main');
   }
```

Usando a short syntax do Ecma Script 6 (ES6)

    d. Podemos verificar o response no console
    e. Como a única informação que nos interessa é o id, pegamos ele de dentro de:

```
    const { _id } = response.data;
```

    f. Passando o _id para página Main. React Routes permite o envio de parâmetros como fizemos no NodeJs.

       Primeiro trocamos o path da rota Main

```
         <Route path="/devs/:id" component={Main} />
```

Depois em Login trocamos:

```
      history.push(`/dev/${_id}`);
```

A URL fica com o id do usuário:

```
http://localhost:3001/dev/5dddc768a1bd06001d6375b7
```

g. Recuperando o id na Main

```
  <h1>{match.params.id}</h1>
```

h. Tudo estático

```
import React from 'react';

import logo from '../assets/logo.svg'

import like from '../assets/like.svg'
import dislike from '../assets/dislike.svg'

export default function Main({ match }) {
   return (
       <div className="main-container">
           <img src={logo} alt="DevCom" />
           <ul>
               <li>
                   <img
src="https://avatars0.githubusercontent.com/u/23192463?v=4" alt="" />
                   <footer>
                       <strong>Eliabe Leal</strong>
                       <p>Software Engineer at @sesdsgovpb, programming
lover and musician in his spare time.</p>
                   </footer>
                   <div className="buttons">
                       <button type="button">
                           <img src={dislike} alt="dislike" />
                       </button>
                       <button type="button">
                           <img src={like} alt="like" />
                       </button>
                   </div>
               </li>
           </ul>
       </div>
```

```
    );
}
```

i. Repetindo para estilizar: vamos repetir o li para termos conteúdo
j. Criando o Main.css https://copy-paste.online/

```
.main-container {
    max-width: 980px;
    margin: 0 auto;
    padding: 50px 0;
    text-align: center;
}

.main-container ul {
    list-style: none;
    display: grid;
    grid-template-columns: repeat(3, 1fr);
    grid-gap: 30px;
    margin-top: 50px;
}

.main-container ul li {
    display: flex;
    flex-direction: column;
}

.main-container ul li img {
    max-width: 100%;
    border-radius: 5px 5px 0 0;
}

.main-container ul li footer {
    flex: 1;
    background: #FFF;
    border: 1px solid #EEE;
    padding: 15px 20px;
    text-align: left;
    border-radius: 0 0 5px 5px;
}

.main-container ul li footer strong {
```

```
    font-size: 16px;
    color: #333;
}

.main-container ul li footer p {
    font-size: 14px;
    line-height: 20px;
    color: #999;
    margin-top: 5px;
}

.main-container ul li .buttons{
    margin-top: 10px;
    display: grid;
    grid-template-columns: repeat(2, 1fr);
    grid-gap: 10px;
}

.main-container ul li .buttons button {
    height: 50px;
    box-shadow: 0 2px 2px 0 rgba(0, 0, 0, 0.05);
    border: 0;
    border-radius: 4px;
    background: #FFF;
    cursor: pointer;
}

.main-container ul li .buttons button:hover img {
    transform: translateY(-5px);
    transition: all 0.2s;
}

.main-container .empty {
    font-size: 32px;
    color: #999;
    font-weight: bold;
    margin-top: 300px;
}

.match-container{
```

```
    position: absolute;
    top: 0;
    left: 0;
    right: 0;
    bottom: 0;
    display: flex;
    flex-direction: column;
    justify-content: center;
    align-items: center;
    background: rgba(0, 0, 0, 0.8)
}

.match-container .avatar {
    width: 200px;
    height: 200px;
    border-radius: 50%;
    border: 5px solid #FFF;
    margin: 30px 0;
}

.match-container strong {
    font-size: 32px;
    color: #FFF;
}

.match-container p {
    margin-top: 10px;
    font-size: 20px;
    line-height: 30px;
    max-width: 400px;
    color: rgba(255, 255, 255, 0.8)
}

.match-container button {
    border: 0;
    background: none;
    font-weight: bold;
    color: rgba(255, 255, 255, 0.8);
    font-size: 18px;
    margin-top: 30px;
```

```
    cursor: pointer;
}
```

k. useEffect: é uma função que é chamada após a renderização do componente. Recebe dois parâmetros, o primeiro é a função que vamos executar quando ele for chamado e o segundo é quando que vamos executar essa função. Podemos informar variáveis em um array (dependências do useEffect), que sempre que forem alteradas, a nossa função será chamada, se o array for vazio, nossa função será chamada apenas uma vez em nosso componente.

   Essa função vai buscar os dados dos devs, para exibi-los em tela.

```
export default function Main({ match }) {
   useEffect(() => {

   }, [match.params.id]);
   return (
```

   O React não recomenda usar async await nos hooks (funções disparadas em determinados momentos da aplicação) eles são assíncronos. Para isso, é aconselhável usar outra função dentro dele. Chamando a própria logo abaixo.

```
useEffect(() => {
       async function loadUsers() {
           const response = await api.get('/devs', {
               headers: {
                   user: match.params.id
               }
           })
       }
       loadUsers();
   }, [match.params.id]);
```

Podemos visualizar o retorno no console (response.data)

l. Estados de novo: Como precisamos armazenar esse retorno, vamos utilizar o conceito de estado.

A inicialização desse estado deve ser um array vazio, por ele vai armazenar vários usuários.

```
  const [users, setUsers] = useState([]);
```

Basta agora pegarmos o response e passarmos ele para o nosso estado, isso vai garantir que sempre que esse estado mudar, tudo que depender dele será atualizado.

```
const [users, setUsers] = useState([]);
   useEffect(() => {
       async function loadUsers() {
           const response = await api.get('/devs', {
               headers: {
                   user: match.params.id
               }
           })

           setUsers(response.data);
       }
       loadUsers();
   }, [match.params.id]);
```

m. Sendo dinâmico

Vamos usar um JS dentro da <ul> para percorrer (.map) nossos users
A sintaxe permite que nós possamos abstrair o return de uma arrow function

```
                {users.map(user => {
                    return
                  })}
```

ou

```
                {users.map(user => (

                  ))}
```

Agora é só inserir dentro da função o nosso li e alterar as propriedades que setamos estaticamente.

```
{users.map(user => (
                    <li>
                        <img src={user.avatar} alt={user.name} />
                        <footer>
                            <strong>{user.name}</strong>
                            <p>{user.bio}</p>
                        </footer>
                        <div className="buttons">
```

```
                            <button type="button">
                                <img src={dislike} alt="dislike" />
                            </button>
                            <button type="button">
                                <img src={like} alt="like" />
                            </button>
                        </div>
                    </li>
                ))}
```

No console há um erro, pois falta a propriedade key no primeiro elemento logo após o map. Isso é utilizado para o React gerenciar o elementos, pra ele saber qual é qual, se caso precise mudar ele lugar. Senão toda vez ele tem que renderizar a lista do zero, isso não performático.

Tem que ser um valor único do usuário (id)

n. Likes e Dislikes: toda função que é gerada a partir da interação do usuário, recomenda-se começar com handle

```
async function handleLike(id){
       console.log('like', id)
   }
   async function handleDislike(id){
       console.log('dislike', id)
   }
```

essa função será chamada assim que clicarmos no botão

```
<button type="button" onClick={handleDislike}>
                                <img src={dislike} alt="dislike" />
                            </button>
```

Passando assim nossa função receberá um objeto do evento

```
   <button type="button" onClick={handleDislike(user._id)}>
                                <img src={dislike} alt="dislike" />
                            </button>
```

Dessa forma recebemos o id do usuário

Quando rodarmos esse código, você verá que, sem clicar no botão, a função foi chamada. Isso ocorre, porque quando o React chegar nessa linha ele já irá executar essa função, ele não vai passar a referência dela como parâmetro para o onClick. Podemos fazer um hack (de haquear) que é só declarar uma nova função por volta.

```
<button type="button" onClick={() => handleDislike(user._id)}>
                                <img src={dislike} alt="dislike" />
                            </button>
```

Quando um usuário dá dislike em outro, chamados a rota de dislike

```
async function handleDislike(id){
       await api.post(`/devs/${id}/dislike`)
   }
```

Mas quando enviar para essa rota ela também espera receber um header, sendo que o método post tem como segundo parâmetro o corpo da requisição. Então enviamos no terceiro.

```
async function handleDislike(id) {
       await api.post(`/devs/${id}/dislike`, null, {
           headers: { user: match.params.id },
       })
   }
```

Para ver se deu certo vamos no Chrome e em Network que marca todas as requisições. Quando é enviado, ele gera duas requisições, uma é pra ver se o Chrome tem permissão para realizar essa tarefa e a segunda é a de fato.

Para o card desaparecer, foi preciso dar um F5, mas vamos mudar isso.

Vamos usar o state de users, para remover o usuário que demos dislike.

```
   setUsers(users.filter(user => user._id != id));
```

Importante: nunca mexer na variável users diretamente, sempre usar o setUsers. Mesmo ele sendo um array, eu não posso usar um push, splice (pra remover uma posição), sempre setUsers

Usando um IF no React

```
   {users.length > 0 ? (
               <ul>
                   {users.map(user => (
                       <li key={user._id}>
                           <img src={user.avatar} alt={user.name} />
                           <footer>
                               <strong>{user.name}</strong>
```

```
                                <p>{user.bio}</p>
                            </footer>
                            <div className="buttons">
                                <button type="button" onClick={() =>
handleDislike(user._id)}>
                                    <img src={dislike} alt="dislike" />
                                </button>
                                <button type="button" onClick={() =>
handleLike(user._id)}>
                                    <img src={like} alt="like" />
                                </button>
                            </div>
                        </li>
                    ))}
                </ul>
            ) : <div className="empty"> Acabou! :(</div>}
```