**1.Configurando o Ambiente**

**Criando a estrutura do projeto**

Crie a pasta src e dentro, crie arquivo **App.jsx**

*import React from 'react';*

*function App(){*

*return <h1>Helloo World</h1>*

*}*

crie o arquivo **index.jsx**

<!DOCTYPE html>

<html *lang*="en">

<head>

<meta *charset*="UTF-8">

<meta *http-equiv*="X-UA-Compatible" *content*="IE=edge">

<meta *name*="viewport" *content*="width=device-width, initial-scale=1.0">

<title>Projeto</title>

</head>

<body>

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

<script *src*="../dist/bundle.js"></script>

</body>

</html>

*// usamos esse i****d****="root" para mostrar todos os componentes na tela dentro dessa div, padrão do react*

**Criando o arquivo json**

As orientações abaixo devem ser feitas no terminal basta estar na pasta do projeto

Ex: C:\Users\ADMIN\Desktop\Projeto> **"comando no terminal é escrito aqui "**

para criar um arquivo json mostrando as dependências e versões use o comando

**yarn init -y**

adicione o react

**yarn add react**

adicionada a biblioteca react com os módulos e dependências react

**yarn react-dom**

A biblioteca *react-dom* permite que react se comunique com a árvore de elementos html para que ele saiba como criar elementos em tela e deletar etc "

**Configurando Babel**

adicionando e configurando babel:

**yarn add @babel/core @babel/cli @babel/preset-env -D**

babel ferramenta que recodifica código para que todo navegador entenda.

criar arquivo nomeado como **babel.config.js**

digite o código a seguir dentro:

***module*.*exports*={**

**presets:['@babel/preset-env',**

**['@babel/preset-react',{**

**runtime:'automatic'**

**}]**

**]**

**};**

comando de execução babel:

**yarn babel src/index.js --out-file dist/bundle.js**

esse comando acima escolhe o arquivo a ser recodificado e o diretório a ser criado para novo código

o babel não entende html ou jsx , então inserimos um novo recurso.

**yarn add @babel/preset-react -D**

agora o babel consegue recodificar jsx para diversos navegadores entenderem.

o babel está configurado.

**Configurando webpack**

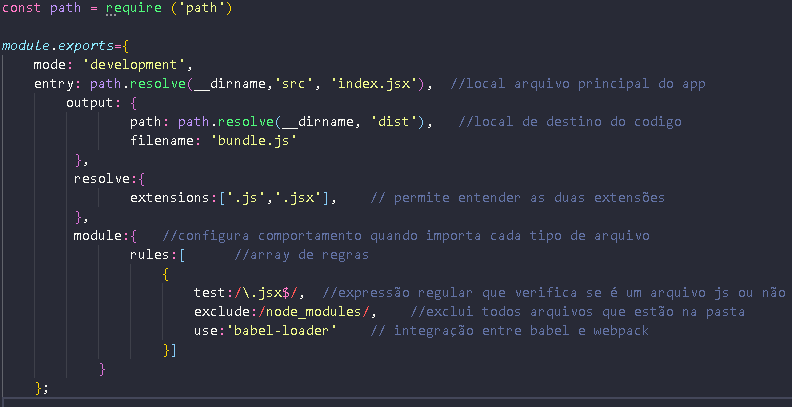
webpack permite configurar e ensinar a aplicação tratar cada tipo de arquivo,convertendo a tipos o que navegador entenda.

instalar e configurar:

**yarn add webpack webpack-cli -D**

Crie o arquivo de configuração **webpack.config.js**

imagem com código:

****

código:

const path = require ('path')

*module*.*exports*={

mode: 'development',

entry: path.resolve(\_\_dirname,'src', 'index.jsx'),

output: {

path: path.resolve(\_\_dirname, 'dist'),

},

resolve:{

extensions:['.js','.jsx'],

},

module:{

rules:[

{

test:/\.jsx$/,

exclude:/node\_modules/,

use:'babel-loader'

}]

}

};

terminal:

**yarn add babel-loader -D**

**yarn webpack**

**Servindo Html Estático**

Para injetar js em html automaticamente.

**yarn add html-webpack-plugin -D**

Permite injetar o bundle no arquivo html

acrescentar no **webpack.config.js**

**const HtmlWebpackPlugin = require('html-webpack-plugin')**

**plugins:[**

**new HtmlWebpackPlugin({**

**template: path.resolve(\_\_dirname,'public','index.html')**

**})**

**],**

O arquivo está atualmente assim.

**const path = require ('path')**

**const HtmlWebpackPlugin = require('html-webpack-plugin')**

**const path = require ('path')**

**const HtmlWebpackPlugin = require('html-webpack-plugin')**

***module*.*exports*={**

**mode: 'development',**

**entry: path.resolve(\_\_dirname,'src', 'index.jsx'),**

**output: {**

**path: path.resolve(\_\_dirname, 'dist'),**

**filename: 'bundle.js'**

**},**

**resolve : {**

**extensions:['.js','.jsx'],**

**},**

**plugins:[**

**new HtmlWebpackPlugin({**

**template: path.resolve(\_\_dirname,'public','index.html')**

**})**

**],**

**module:{**

**rules:[**

**{**

**test:/\.jsx$/,**

**exclude:/node\_modules/,**

**use:'babel-loader'**

**}]**

**}**

**};**

***module*.*exports*={**

**mode: 'development',**

**entry: path.resolve(\_\_dirname,'src', 'index.jsx'),**

**output: {**

**path: path.resolve(\_\_dirname, 'dist'),**

**filename: 'bundle.js'**

**},**

**resolve : {**

**extensions:['.js','.jsx'],**

**},**

**plugins:[**

**new HtmlWebpackPlugin({**

**template: path.resolve(\_\_dirname,'public','index.html')**

**})**

**],**

**module:{**

**rules:[**

**{**

**test:/\.jsx$/,**

**exclude:/node\_modules/,**

**use:'babel-loader'**

**}]**

**}**

**};**

**Webpack Dev Server**

**yarn add webpack-dev-server -D**

Permite que webpack execute as alterações e faça as modificações sem que precisemos usar o comando **yarn webpack**

Acrescentar a config no **webpackconfig**

Mais uma propriedade :

**devServer:{**

**contentBase: path.resolve(\_\_dirname,'public')**

**},**

basta executar o comando

**yarn webpack serve**

agora toda alteração está sendo atualizada automaticamente e fica disponível no endereço <http://localhost:8080/>

const path = require ('path')

const HtmlWebpackPlugin = require('html-webpack-plugin')

*module*.*exports*={

mode: 'development',

entry: path.resolve(\_\_dirname,'src', 'index.jsx'),

output: {

path: path.resolve(\_\_dirname, 'dist'),

filename: 'bundle.js'

},

resolve : {

extensions:['.js','.jsx'],

},

devServer:{

contentBase: path.resolve(\_\_dirname,'public')

},

plugins:[

**new** HtmlWebpackPlugin({

template: path.resolve(\_\_dirname,'public','index.html')

})

],

module:{

rules:[

{

test:/\.jsx$/,

exclude:/node\_modules/,

use:'babel-loader'

}]

}

};

**Utilizando source maps**

Agora será configurado source maps para ajudar a mostrar o erro de forma correta.

devtool é uma configuração podendo conter opção 'eval-source-map’

**Ambiente Dev e Produção**

Essas configurações a seguir são para definir ambiente de dev ou produção, um padrão adotado neste conteúdo.

**const isDevelopement = process.env.NODE\_ENV != 'production';**

***module*.*exports*={**

**mode: isDevelopement ? 'development': 'production',**

**devtool: isDevelopement ? 'eval-source-map' : 'source-map',**

O arquivo webpack.config.js deve estar do seguinte modo

**const path = require ('path')**

**const HtmlWebpackPlugin = require('html-webpack-plugin')**

**const isDevelopement = process.env.NODE\_ENV != 'production';**

***module*.*exports*={**

**mode: isDevelopement ? 'development': 'production',**

**devtool: isDevelopement ? 'eval-source-map' : 'source-map',**

**entry: path.resolve(\_\_dirname,'src', 'index.jsx'),**

**output: {**

**path: path.resolve(\_\_dirname, 'dist'),**

**filename: 'bundle.js'**

**},**

**resolve : {**

**extensions:['.js','.jsx'],**

**},**

**devServer:{**

**contentBase: path.resolve(\_\_dirname,'public')**

**},**

**plugins:[**

**new HtmlWebpackPlugin({**

**template: path.resolve(\_\_dirname,'public','index.html')**

**})**

**],**

**module:{**

**rules:[**

**{**

**test:/\.jsx$/,**

**exclude:/node\_modules/,**

**use:'babel-loader'**

**},]**

**}**

**};**

Configurando variáveis de ambiente.

**yarn add cross-env -D**

No **packagejson** vamos configurar os comandos scripts.

**"scripts" : {**

**"dev": "webpack serve",**

**"build" : "cross-env NODE\_ENV= production webpack"**

**},**

**Importando arquivos Css**

Agora temos que configurar o projeto para entender arquivos css,

então no **webpackconfig** criaremos uma nova regra em ***rules***.

**yarn add style-loader css-loader -D**

**rules:[**

**{**

**test:/\.jsx$/,**

**exclude:/node\_modules/,**

**use:'babel-loader'**

**},**

**{ //essa é a nova regra**

**test:/\.css$/,**

**exclude:/node\_modules/,**

**use:['style-loader' ,'css-loarder'],**

**}**

**]**

Assim podemos importar arquivos css que serão entendidos e recodificados.

**Utilizando SASS**

o Sass é um pré-processador de css permitindo funcionalidades adicionais no css.

**yarn add node-sass -D**

atualize o arquivo webpack.config.js

substitua essa configuração pela existente

{

test:/\.scss$/,

exclude:/node\_modules/,

use:['style-loader' ,'css-loader','sass-loader'],

}

imagens abaixo:

Webpack.Config.js ficar de acordo com imagem abaixo :

****

imagem da estrutura do projeto:

**2.Conceitos importantes**

Trabalhando com React existem 3 conceitos muito importantes, que são: **componente, propriedade e estado,** a seguir veremos esses conceitos.

**2.1 Primeiro componente react**

Na pasta **src** crie uma pasta chamada **components** e dentro da mesma crie um arquivo chamado **RepositoyList.jsx**

Neste arquivo digite o código básico de um arquivo jsx ou js.

export function RepositoryList(){

return(

);

}

Agora adicionaremos uma ***section*** contendo uma classe para editarmos em css, dentro da *s****ecti****on* vamos ter um título <h1>, uma lista de repositórios.

export function RepositoryList(){

return(

<section *className*="repository-list">

<h1>Lista de repositórios</h1>

<ul>

<li>

<strong>unform</strong>

<p>Forms in ReactJs</p>

<a *href*="">Acessar repositório</a>

</li>

<li>

<strong>unform</strong>

<p>Forms in ReactJs</p>

<a *href*="">Acessar repositório</a>

</li>

<li>

<strong>unform</strong>

<p>Forms in ReactJs</p>

<a *href*="">Acessar repositório</a>

</li>

<li>

<strong>unform</strong>

<p>Forms in ReactJs</p>

<a *href*="">Acessar repositório</a>

</li>

</ul>

</section>

);

}

}

No arquivo App vamos colocar nosso componente **repositorylist** dentro do ***return***, e importar **repositorylist** no topo do arquivo , veja código a seguir:

import React from 'react';

import { RepositoryList } from './components/RepositoryList';

import './Styles/Global.scss';

export function App(){

return (

<RepositoryList></RepositoryList>

)

}

Se executarmos o comando **yarn dev**  no terminal vamos iniciar nossa aplicação disponível no endereço <http://localhost:8080/> :



Na pasta de **Styles** no arquivo **Global.scss** vamos manter a seguinte configuração:

\*{

margin: 0;

padding: 0;

box-sizing: border-box;

}

body{

font:16px "Helvetica Neve", Arial, sans-serif;

color: #121214

}

Voltando para nosso arquivo **repositorylist.jsx** um detalhe interessante que podemos fazer em componentes é a transição de html para **javascripit** de uma maneira muito natural.  
  
Se criarmos uma variável chamada **repositoryName** recebendo um nome, podemos inserir no html através do javascript, ou seja, uma interpolação, veja exemplo abaixo:

const repositoryName= 'unform';

export function RepositoryList(){

return(

<section *className*="repository-list">

<h1>Lista de repositorios</h1>

<ul>

<li>

<strong>{repositoryName}</strong>

<p>Forms in ReactJs</p>

<a *href*="">Acessar repositório</a>

</li>

Pronto criamos um componente novo.

**2.2 Propriedades no react**

Para entendermos o conceito de propriedade vamos criar um novo componente chamado **repositoryItem.jsx**

**export function RepositoryItem(){**

**return(**

**<li>**

**<strong>unform</strong>**

**<p>Forms in ReactJs</p>**

**<a *href*="">Acessar repositório</a>**

**</li>**

**);**

**}**

Observe que pegamos o conteúdo <li> do arquivo **repositoryList** que estava dentro do <ul>

no arquivo **repositoryList** faremos mudanças também:

import { RepositoryItem } from "./RepositoryItem";

export function RepositoryList(){

return(

<section *className*="repository-list">

<h1>Lista de repositorios</h1>

<ul>

<*RepositoryItem*></*RepositoryItem*>

<*RepositoryItem*></*RepositoryItem*>

<*RepositoryItem*></*RepositoryItem*>

<*RepositoryItem*></*RepositoryItem*>

</ul>

</section>

);

}

Observe que transformamos o <li> em um componente, quando temos algo que se repete inúmeras vezes na nossa aplicação, na maioria das vezes, faz sentido transformar em componente.

O conceito de propriedade é quando o componente pai(**RepositoryList**) envia informações para componente filho(**RepositoryItem**).

import { RepositoryItem } from "./RepositoryItem";

export function RepositoryList(){

return(

<section *className*="repository-list">

<h1>Lista de repositorios</h1>

<ul>

<*RepositoryItem* *repository*="unform2"/>

<*RepositoryItem*/>

<*RepositoryItem*/>

<*RepositoryItem*/>

</ul>

</section>

);

}

Agora no **repositoryItem** podemos acessar a informação como exemplo abaixo:

export function RepositoryItem(*props*){

return(

<li>

<strong>{*props*.repository}</strong>

<p>Forms in ReactJs</p>

<a *href*="">Acessar repositório</a>

</li>

);

}

Props é padrão no react quando a função possui parâmetro herdado.  
Dentro de **props** possui as propriedades colocadas anteriormente no componente pai, **props.repository**

Quando tem muitas propriedades podemos criar um objeto:  
  
**repositoryList.jsx**

import { RepositoryItem } from "./RepositoryItem";

const repository={

name:'unform',

description:'Forms in React',

link:'https://www.google.com'

}

export function RepositoryList(){

return(

<section *className*="repository-list">

<h1>Lista de repositorios</h1>

<ul>

<*RepositoryItem* *repository*={repository}/>

<*RepositoryItem*/>

<*RepositoryItem*/>

<*RepositoryItem*/>

</ul>

</section>

);

}

dessa forma nosso código fica mais organizado.

agora precisamos alterar o arquivo **repositoryItem.jsx**

**e**xport function RepositoryItem(*props*){

return(

<li>

<strong>{*props*.repository.name}</strong>

<p>Forms in ReactJs</p>

<a *href*="">Acessar repositório</a>

</li>

);

}

Perceba que agora conseguimos navegar nas propriedades e no objeto escolhendo a variável desejável.

Agora que sabemos usar a propriedade e o objeto, vamos atualizar nossos arquivos.

**repositoryList.jsx**

import { RepositoryItem } from "./RepositoryItem";

const repository={

name:'unform',

description:'Forms in React',

link:'https://www.google.com'

}

export function RepositoryList(){

return(

<section *className*="repository-list">

<h1>Lista de repositórios</h1>

<ul>

<*RepositoryItem* *repository*={repository}/>

<*RepositoryItem* *repository*={repository}/>

<*RepositoryItem* *repository*={repository}/>

<*RepositoryItem* *repository*={repository}/>

</ul>

</section>

);

}

**repositoryItem.jsx**

export function RepositoryItem(*props*){

return(

<li>

<strong>{*props*.repository.name}</strong>

<p>{*props*.repository.description}</p>

<a *href*={*props*.repository.link}>Acessar Repositório</a>

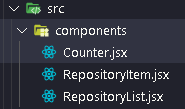
</li>

);

}

**2.3 Estado do componente**

De forma simples para entender o que é estado vamos criar um componente chamado **Counter.jsx**

****

export function Counter(){

return(

<div>

<h2>0</h2>

<button *type*="button">Increment</button>

</div>

);

}

Agora vamos fazer alterações no nosso arquivo **App.jsx**

import React from 'react';

import { Counter } from './components/Counter';

import { RepositoryList } from './components/RepositoryList';

import './Styles/Global.scss';

export function App(){

return (

<>

<*RepositoryList*/>

<*Counter*/>

</>

);

}

Estamos usando fragmento <></> ele é necessário e não aparece no código do navegador.

PARA MELHOR ENTENDIMENTO:



Essa pequena ilustração simples ajuda a entender a forma que os componentes estão, sua hierarquia de forma visual.

Agora vamos criar a função que faz o incremento no botão, a função a princípio só irá exibir no console uma mensagem.

export function Counter(){

function increment(){

console.log('Incrementing');

}

return(

<div>

<h2>0</h2>

<button *type*="button" *onClick*={increment}>

Increment

</button>

</div>

);

}

export function Counter(){

let counter = 0 ;

function increment(){

counter += 1 ;

}

return(

<div>

<h2>0</h2>

<button *type*="button" *onClick*={increment}>

Increment

</button>

</div>

);

}

esse código seria comum funcionar em programação orientada objeto, porém na web precisamos usar um recurso que salva um valor e altera esse valor quando necessário, chamamos de estado, veremos código abaixo:

O nome desse recurso é useState() é uma função que retorna um array contendo dois valores diferentes o valor atual, e valor a ser inserido, usamos um recurso chamado desestruturação podendo separar esses dois valores em suas respectivas variáveis.

import {useState} from 'react';

export function Counter(){

const [counter, setCounter] = useState(0);

function increment(){

setCounter(counter + 1) ;

}

return(

<div>

<h2>{counter}</h2>

<button *type*="button" *onClick*={increment}>

Increment

</button>

</div>

);

}

Pronto aprendemos nosso primeiro estado.

**2.4 A imutabilidade no react**

Conceito de imutabilidade na prática é você manter a informação original criar uma cópia e depois acrescentar uma informação desejada, ou seja,

no caso de um vetor :

usuarios= [‘vinicius’ , ‘Yan’ , ‘André’ , ‘Alex’]

novousuario=[...usuarios, ‘Pedro’]

O próprio useState aplica esse conceito de imutabilidade e ao longo do conteúdo isso vai ficando mais claro.

**2.5 Fast Refresh no webpack**

Essa ferramenta permite que alteremos nossa aplicação e ao atualizar mantém os valores de estado, ou seja, em fase de desenvolvimento podemos testar e alterar a aplicação sem perder dados e valores que usamos para teste , EX: um e-commerce usamos informações como produtos valores e etc , essa informações são mantidas até mesmo os resultados dos cálculos.

Congifurando:

<https://github.com/pmmmwh/react-refresh-webpack-plugin>

# if you prefer yarn

**yarn add -D @pmmmwh/react-refresh-webpack-plugin react-refresh**

Abra o terminal vscode e coloque comando e execute .

Abra nosso arquivo webpack.config.js

Criaremos a constante do refresh na mesma linha.

const ReactRefreshWebpackPlugin require('@pmmmwh/react-refresh-webpack-plugin')

Agora como é um plugin inserimos a configuração no plugin:

plugins:[

isDevelopement && **new** ReactRefreshWebpackPlugin(),

**new** HtmlWebpackPlugin({

template: path.resolve(\_\_dirname,'public','index.html')

})

].filter(*Boolean*),

Se estivermos em ambiente de desenvolvimento ele vai executar o :

&& **new** ReactRefreshWebpackPlugin(),

se não, ele retorna false e não executa, e false não é um retorno de plugin válido , vai dar erro e para resolver esse problema usamos um recurso .Filter(Boolean)

plugins:[

isDevelopement && **new** ReactRefreshWebpackPlugin(),

**new** HtmlWebpackPlugin({

template: path.resolve(\_\_dirname,'public','index.html')

})

].filter(*Boolean*),

Dessa forma remove o retorno false de dentro do plugin permitindo seu funcionamento e uso de condicional no plugin webpack.

Agora acrescentaremos a configuração hot:true em devserver :

devServer:{

contentBase: path.resolve(\_\_dirname,'public'),

hot:true,

},

E a última configuração:

module:{

rules:[

{

test:/\.jsx$/,

exclude:/node\_modules/,

use:{

loader:'babel-loader}',

options:{

plugins:[

isDevelopement && require.resolve('react-refresh/babel')

].filter(*Boolean*)

}

},

},

ampliamos nossa configuração adicionando regra usando babel e plugin refresh juntamente com babel.

Nesta parte deu erro no meu projeto , algum bug relacionado a usuarios do windows ou regras de pastas.

Para sanar fiz seguintes passos:

Roda o seu repositório dentro do disco local C

Não em usuários.

Só cola a pasta no disco local C

E dá um **yarn**

Depois **yarn dev**

**Webpack.config.js**

**const path = require ('path');**

**const HtmlWebpackPlugin = require('html-webpack-plugin');**

**const ReactRefreshWebpackPlugin = require('@pmmmwh/react-refresh-webpack-plugin');**

**const isDevelopement = process.env.NODE\_ENV !== 'production';**

***module*.*exports*={**

**mode: isDevelopement ? 'development': 'production',**

**devtool: isDevelopement ? 'eval-source-map' : 'source-map',**

**entry: path.resolve(\_\_dirname,'src', 'index.jsx'),**

**output: {**

**path: path.resolve(\_\_dirname, 'dist'),**

**filename: 'bundle.js'**

**},**

**resolve : {**

**extensions:['.js','.jsx'],**

**},**

**devServer:{**

**contentBase: path.resolve(\_\_dirname,'public'),**

**hot: true,**

**},**

**plugins:[**

**isDevelopement && new ReactRefreshWebpackPlugin(),**

**new HtmlWebpackPlugin({**

**template: path.resolve(\_\_dirname,'public','index.html')**

**})**

**].filter(*Boolean*),**

**module:{**

**rules:[**

**{**

**test:/\.jsx$/,**

**exclude:/node\_modules/,**

**use:{**

**loader:'babel-loader',**

**options:{**

**plugins:[**

**isDevelopement && require.resolve('react-refresh/babel'),**

**].filter(*Boolean*)**

**},**

**},**

**},**

**{**

**test:/\.scss$/,**

**exclude:/node\_modules/,**

**use:['style-loader' ,'css-loader','sass-loader'],**

**},**

**],**

**},**

**};**

3.Chamadas HTTP

3.1 Estilização da listagem

Agora que sabemos o que é estado e propriedades, podemos excluir o arquivo **counter.jsx,** e apagar o conteúdo dentro do return do arquivo **App.jsx**.

import React from 'react';

import { RepositoryList } from './components/RepositoryList';

import './styles/global.scss';

export function App(){

return (

<*RepositoryList*/>

);

}

Agora vamos estilizar nosso projeto, crie na pasta **styles** um arquivo chamado **repositories.scss**

Escreva esse código no arquivo **repositorylist.jsx** para importar o estilo que vamos configurar.

**import '../styles/repositories.scss';**

No arquivo **repositories.scss** vamos estilizar com esse código a seguir:

section*.repository-list*{

margin:40px;

h1{

margin-bottom:16px;

}

ul{

list-style:none;

li{

& + li{

margin-top:20px;

}

p{

font-size:14px;

color:#444;

margin-top:8px;

}

a{

display: inline-block;

margin-top:16px;

text-decoration:none;

color:#8257e6

}

}

}

}

3.2 Utilizando useEffect

o Github possui uma API <https://api.github.com/users/viniciuspollogomes>, essa API fornece informações de usuários e repositórios.

vamos usar um repositório da Rocketseat, <https://api.github.com/orgs/rocketseat/repos>

Nós criaremos um estado para armazenar as informações desse repositório , faremos isso no arquivo **RepositoryList.jsx** usando **useState().**

**export function RepositoryList(){**

**const[repositories, setRepositories] = useState([]);**

**return(**

Não esqueça de importar o **useState.**

**import {useState,UseEffect} from 'react';**

importamos o **useEffect** pois usaremos também.

**useEffect(()=>{} , []);**

Essa função ela recebe dois parâmetros dentro dos parênteses,

O primeiro parâmetro recebe uma função. () =>{}

O segundo parâmetro ele espera uma variável.  
  
O useEffect ele vai executar uma ação sempre que a variável colocada no segundo parâmetro sofrer alteração.

useEffect(()=>{

} , []);

Essa é a indentação correta, dentro dessa função no espaço que não contém informação colocaremos a ação que nossa função realizará sempre que o segundo parâmetro sofrer alteração.

useEffect(()=>{

fetch('https://api.github.com/orgs/rocketseat/repos')

.then(*response* => *response*.json())

.then(*data*=> console.log(*data*))

} , []);

**fetch** busca os dados com a api, quando o fetch me devolver essa resposta em **. then** vou converter essa resposta para **json**

.then(*response* => *response*.json())

Quando a resposta para **json** terminar de ser convertida, você vai ter no segundo **.then** os dados sendo mostrados no console para testar.

.then(*data*=> console.log(*data*))

para próximo tópico mantenha o seu effect salvando a variável **data** por meio da função **setRepositories**

**useEffect(()=>{**

**fetch('https://api.github.com/orgs/rocketseat/repos')**

**.then(*response* => *response*.json())**

**.then(*data*=> setRepositories(*data*))**

**} , []);**

De forma resumida nós usamos o **useState** para salvar estado da nossa variável e atualiza na tela conforme seja alterada, e combinamos com **useEffect** que executa uma função uma única vez ou toda vez que a variável no segundo parâmetro for alterada.

3.3 Listando Repositórios

No arquivo **RepositoryList** vamos começar apagando a variável

**const repository={**

**name:'unform',**

**description:'Forms in React',**

**link:'https://www.google.com'**

**}**

Apague esse código também :

<ul>

<*RepositoryItem* *repository*={repository}/>

<*RepositoryItem* *repository*={repository}/>

<*RepositoryItem* *repository*={repository}/>

<*RepositoryItem* *repository*={repository}/>

</ul>

coloque este código no lugar:

<ul>

{repositories.map(

*repository*=>{

return<*RepositoryItem* *key*={*repository*.name}*repository*={*repository*}/>

})

}

</ul>

Esse código acima ele está usando a função map que percorre repositórios e consegue retornar.

injetamos nosso repositório da api github no nosso componente , as informações são propriedades e serão usadas.

Atualize o **RepositoryItem.jsx**

export function RepositoryItem(*props*){

return(

<li>

<strong>{props.repository.name}</strong>

<p>{props.repository.description}</p>

<a *href*={props.repository.html\_url}>Acessar Repositório</a>

</li>

);

}

4. Typescript

4.1 Configurando o typescript

Instalando **typescript:**

**yarn add typescript -D**

**yarn tsc --init**

Agora o typescript será inicializado em nosso projeto, ou seja, será gerado um arquivo **tsconfig.json** em nossa aplicação.

Esse arquivo possui inúmeras configurações, algumas habilitadas outras estão em comentário.

deixarei as configurações que iremos usar ou criar.

**"lib": ["dom", "dom.iterable","esnext"],**

geralmente as configurações já estão escritas como comentário basta reutilizar o texto.

**"allowJs": true,**

**"allowSyntheticDefaultImports": true,**

**"skipLibCheck": true,**

**"esModuleInterop": true,**

**"forceConsistentCasingInFileNames": true**

**"moduleResolution": "node",**

**"resolveJsonModule":true,**

**"isolatedModules":true,**

**"noEmit": true,**

**"jsx": "react-jsx",**

as configurações a seguir você deve remover por enquanto:

**"target": "es5",**

**"module": "commonjs",**

Agora acrescente no final do objeto :

{

"compilerOptions": {

//AQUI É ONDE FICA NOSSAS CONFIGURAÇÕES

},

"include": ["src"] ← adicione apenas essa linha

}

Agora temos que configurar babel.config.js e acrescentar as configurações do typescropt.  
  
Digite no terminal:

**yarn add @babel/preset-typescript -D**

O arquivo **babel.config.js**:

*module*.*exports*={

presets:['@babel/preset-env',

'@babel/preset-typescript',

['@babel/preset-react',{

runtime:'automatic'

}]

]

};

Agora vamos editar o **webpack.config.js:**

**rules:[**

**{**

**test:/\.(j|t)sx$/, //Nessa parte ele vai entender jsx e tsx**

**exclude:/node\_modules/,**

**use:{**

**loader:'babel-loader',**

**options:{**

**plugins:[**

**isDevelopement && require.resolve('react-refresh/babel'),**

**].filter(*Boolean*)**

**},**

**},**

Altere também essa parte :

resolve : {

extensions:['.js','.jsx','.ts','.tsx'],

},

Mudança Final:

entry: path.resolve(\_\_dirname,'src', 'index.tsx'),

Não esqueça de mudar formato do arquivo index.jsx para .tsx,

Execute no terminal:  
  
**yarn add @types/react-dom -D**

pronto.

4.2 Componentes com typescript

Mude o formato do arquivo **App.jsx** para **.tsx**

Mude o formato do arquivo **RepositoryItem.jsx** para **.tsx**

Vai dar erro e corrigiremos aplicando o **typescript** pois a **props** é **any** ,

interface *RepositoryItemProps*{

repository:{

name: *string*;

description: *string*;

html\_url: *string*;

}

}

export function RepositoryItem(*props*: *RepositoryItemProps*){

return(

<li>

<strong>{*props*.repository.name}</strong>

<p>{*props*.repository.description}</p>

<a *href*={*props*.repository.html\_url}>Acessar Repositório</a>

</li>

);

}

Em **RepositoryList.jsx** mude o formato para **.tsx** e use esse código :

import { RepositoryItem } from "./RepositoryItem";

import {useState, useEffect} from 'react';

import '../styles/repositories.scss';

interface *Repository*{ //Tipagem

name: *string*;

description: *string*;

html\_url: *string*;

}

export function RepositoryList(){

//Estado é do tipo Repository

const[repositories, setRepositories] = useState<*Repository*[]>([]);

useEffect(()=>{

fetch('https://api.github.com/orgs/rocketseat/repos')

.then(*response* => *response*.json())

.then(*data*=> setRepositories(*data*))

} , []);

return(

<section *className*="repository-list">

<h1>Lista de repositorios</h1>

<ul>

{repositories.map(

*repository*=>{

return<*RepositoryItem* *key*={*repository*.name} *repository*={*repository*}/>

})

}

</ul>

</section>

);

}

Eu tive dificuldades de explicar textualmente sobre **typescript**, porém buscando recursos como vídeos sobre **typescript** em **React** você tenha um entendimento maior sobre as vantagens e recursos feitos na prática.

A dica que dou para o autodidata é pesquisar cada recurso desconhecido, entendê-lo mais a fundo pela curiosidade e tornar isso um hábito.