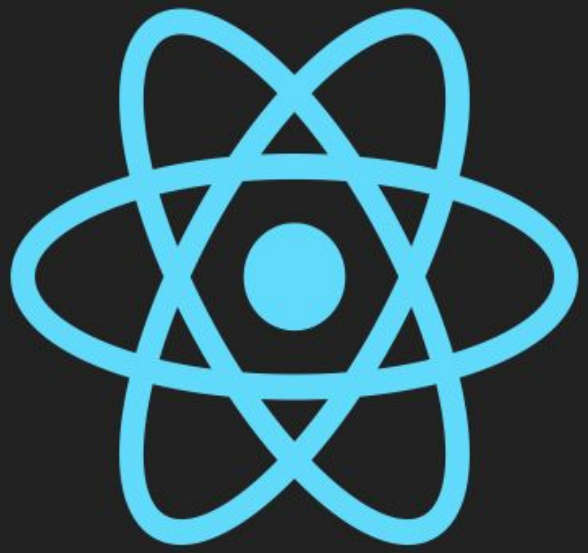
Desenvolvimento WEB



REACT JS

O que vamos aprender?

- Recapitulação Javascript - ES6
- O que é react
- Criação de projetos com React
- Componentização
- Criação de interfaces dinâmicas
- Renderização condicional
- Manipulação de estados
- Consumo de APIs - AXIOS
- Deploy de uma aplicação React no Heroku
- Funções Assíncronas
- Styled Components
- Gerenciadores de Pacotes

Avaliações

- Avaliação conceito (10 pontos)
  - Formulário de Auto Avaliação
- Avaliação teórica (30 pontos)
  - Formulário de múltipla escolha
- Projeto final em grupo (60 pontos)
  - Desenvolvimento de uma aplicação frontend com consumo de API

ES6 Javascript

Aula 1

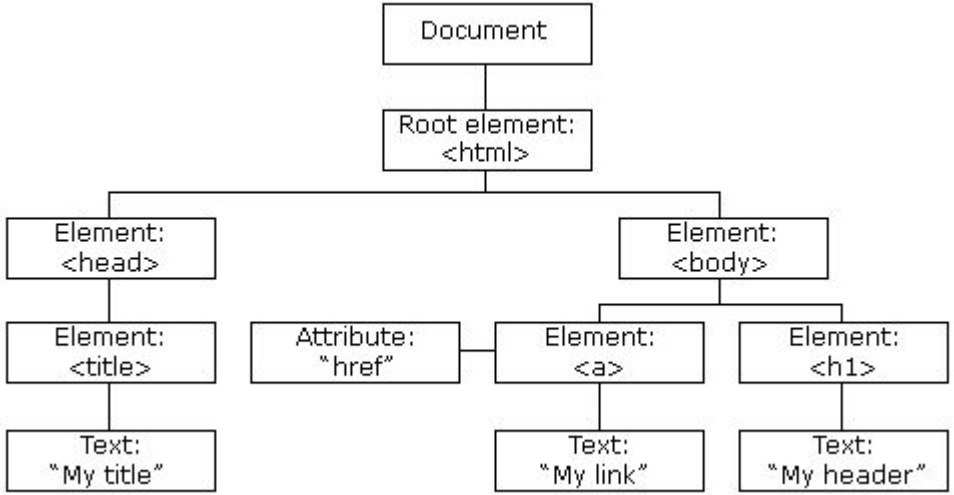
article.html.js
module.exports = (scope) => '<article>
<header>
</header>
${require('./tags.html.js')(scope)}

${scope.body}

DOM: a página no mundo Javascript

Quando uma página da web é carregada, o navegador cria um modelo de objeto de documento (do inglês **D**ocument **O**bject **M**odel) da página.

O modelo HTML DOM é construído como uma árvore de objetos:



DOM: a página no mundo Javascript

Com o modelo de objeto, o JavaScript obtém todo o poder de que precisa para criar HTML dinâmico, sendo capaz de:

- Alterar todos os elementos HTML na página
- Alterar todos os atributos HTML na página
- Alterar todos os estilos CSS na página
- Remover elementos e atributos HTML existentes
- Adicionar novos elementos e atributos HTML
- Reagir a todos os eventos HTML existentes na página
- Criar novos eventos HTML na página

JavaScript HTML DOM Document

O objeto HTML DOM **document** é o proprietário de todos os outros objetos em sua página da web.

Se você deseja acessar qualquer elemento em uma página HTML, você sempre começa acessando o objeto **document**.

Veremos alguns exemplos de como podemos usar o objeto **document** para acessar e manipular HTML

Funções e os eventos do DOM

Os métodos/funções HTML DOM são ações que você pode executar (em elementos HTML).

Propriedades HTML DOM são valores (de elementos HTML) que você pode definir ou alterar.

Veremos alguns exemplos à seguir.

Funções e os eventos do DOM

Alguns exemplos de métodos :

Método	Descrição
document.getElementById(*id*)	Encontra um elemento por id de elemento
element.setAttribute*(attribute, value)*	Altera o valor do atributo de um elemento HTML
document.createElement(*element*)	Cria um elemento HTML
document.getElementById(*id*).onclick = function(){*code*}	Adicionando código de manipulador de eventos a um evento onclick

Funções e os eventos do DOM

Alguns exemplos de propriedades :

Propriedade	Descrição
element.innerHTML = *new html content*	Alterar o HTML interno de um elemento
element.attribute = new value	Altere o valor do atributo de um elemento HTML
element.style.*property = new style*	Altere o estilo de um elemento HTML

Tipos de dados

Vamos conhecer alguns tipos de dados existentes em JavaScript

- String
- Number
- Boolean
- Undefined e Null
- Object
- Array

String

String é uma cadeia de caracteres. Podem ser representados através das aspas simples (‘ ’), aspas duplas (“ ”) ou template strings (` `).

As Template Strings permitem que trabalhemos com strings multi-linhas e utilizemos expressões de linguagem utilizando a formatação ${}.

```
var nome = "Carlos";
var frase = `${nome} é um cara legal.`;
//Carlos é um cara legal.
```

Number

São tipos de dados os quais conseguimos manipular de forma numérica.

```
//Numbers
33 //Int (Inteiro)
15.6 //Float (Real)
NaN // Not a Number
Infinity //Infinito
```

Boolean

True ou False. Muito utilizados em funções condicionais, entre outros.

Undefined e Null

Undefined e null são dois tipos de caracteres muito confundidos no JavaScript, porém, entender a diferença entre eles é crucial durante o desenvolvimento das nossas aplicações.

- Undefined: Valor indefinido - Algo que não existe
- Null: Valor nulo - Objecto que não tem nada dentro dele

Object

Um objeto é composto de Propriedades/Atributos e Funcionalidades/Métodos. É necessária uma atenção especial neste tipo de dado pois ele estará presente frequentemente em nosso cotidiano.

```
const pessoa = {
    name: "Abner",
    idade: 30,
    endereco: {
        cidade: "Petrópolis",
        estado: "Rio de Janeiro"
    },
    gritar: function(){
        console.log("AAAAAAAAAAAAAAA")
    }
}
```

Object

Acessando Propriedades/Atributos e Funcionalidades/Métodos.

```
const pessoa = {
    nome: "Abner",
    idade: 30,
    endereco:{
        cidade: "Petrópolis",
        estado: "Rio de Janeiro"
    },
    gritar: function() {
        console.log("AAAAAAAAAAAAAAAAA");
    }
}

console.log(pessoa.nome)
console.log(pessoa.endereco.cidade)
pessoa.gritar()
```

Array (Vetores)

É uma lista, ou seja um agrupamento de dados. Podem receber qualquer tipo de dados dentro de si;

```
const meuArray = ["Banana", "Maçã", "Mamão"]

const meuArray = [10, 256, "Abner", true, {nome: "Carlos", idade: 26}, 15.8]
```

Funções Anônimas

Em casos como esse, onde não há um outra parte do código onde queremos referenciar uma função e ela será apenas referenciada ao invés de chamar a função, podemos usar o conceito de função anônima, já criando a função no lugar onde antes apenas indicamos seu nome. Por exemplo :

```
var inputTamanho = document.getElementById("inputTamanho");
var outputTamanho = document.getElementById("outputTamanho");

inputTamanho.oninput = function(){
    outputTamanho.value = inputTamanho.value;
};
```

Manipulando Strings

Uma variável que armazena um string faz muito mais que isso! Ela permite, por exemplo, consultar o seu tamanho e realizar transformações em seu valor. Por exemplo :

```
var palavraTosca = 'Catapora'
palavraTosca.length; // Tamanho da string
palavraTosca.replace("pora", "pimba");
```

Manipulando Strings

Assim como em Java, podemos converter uma String para inteiro ou ponto flutuante usando o método parseInt e parseFloat :

```
var textoInteiro = "10";
var inteiro = parseInt(textoInteiro, 10);
var textoFloat = "10.22";
var float = parseFloat(textoFloat)
```

Manipulando Números

Números, assim como strings, também são imutáveis. O exemplo abaixo altera o número de casas decimais com a função toFixed. Esta função retorna uma string, mas, para ela funcionar corretamente, seu retorno precisa ser capturado:

```
var milNumber = 1000;
var milString = milNumber.toFixed(2); //Recebe o retorno da função
console.log(milString); //Imprime a string "1000.00"
```

Arrays em Javascript

A utilização de arrays em javascript não é muito diferente do que foi visto em Portugol ou Java. Os pontos interessantes são que, por javascript não ser tipado, podemos armazenar valores de tipos diferentes em vetores no Javascript.

```
var variosTipos = ["Mamona", 10, [1,2]];
console.log(variosTipos[0]);
```

Arrays em Javascript

Para adicionar elementos ao vetor, podemos utilizar a função **push** que adiciona um elemento na última posição do array ou adicionar direto em um índice selecionado :

```
var palavras = ["RSW", "Ensino"];
palavras.push("Inovação");
palavras[9] = "Criatividade";
console.log(palavras);
```

Arrays em Javascript são semelhantes às ArrayLists em Java

Arrays em Javascript

Manipulando Arrays

- .length() => Permite ver o comprimento do array
- .join(“,”) => Junta todos os elementos do array em uma string
- .pop() => Remove o último elemento do array
- .shift() => Remove o primeiro elemento do array
- .push() => Adiciona um elemento ao final do array
- .indexOf() => Localiza um elemento no array
- .find() => Localiza um elemento no array

Arrays em Javascript

Concatenando Arrays

```
var meuArray = [1,2,3]
var meuArray2 = [6,7,8]
var meuArrayConcatenado = [...meuArray, 4, 5, ...meuArray2, 9, 10, 11]
console.log(meuArrayConcatenado)
```

ES6

Laços de Repetição e Condicionais

Todas as estruturas de laços de repetição e condicionais que vimos em disciplinas passadas funcionam em javascript. Apenas alguns exemplos de sintaxe:

```
while(contador <= 10){
    //código a ser repetido
}
```

```
if(condicao){
    //código a ser executado
}
```

```
for(/*variável de controle*/;/* condição */; /* pós execução*/){
    // código a ser repetido
}
```

Funções Temporais

Em JavaScript, podemos criar um timer para executar um trecho de código após um certo tempo, ou ainda executar algo de tempos em tempos.

A função **setTimeout** permite que agendemos alguma função para execução no futuro e recebe o nome da função a ser executada e o número de milissegundos a esperar:

```
function executar(){
    console.log("Executou...")
}

setTimeout(executar, 1000);
```

Funções Temporais

Se for um código recorrente, podemos usar o setInterval que recebe os mesmos argumentos mas executa a função indefinidamente de tempos em tempos:

```
function executar(){
    console.log("Executou...")
}

setInterval(executar, 1000);
```

É uma função útil para, por exemplo, implementar um banner rotativo, apresentado no exercício à seguir.

Desestruturação no JavaScript

Desestruturação consiste em você extrair do objeto apenas as propriedades que precisa. Muito útil para objetos que possuem muitas propriedades.

```
const {nome, idade, endereco, gritar} = pessoa;
const {cidade} = endereco;

console.log(nome)
console.log(idade)
console.log(cidade)
gritar()
```

ES6

Desestruturação no JavaScript

Funciona também com arrays, porém como o array utiliza [ ] a desestruturação fica dessa maneira:

```
var letras = ["A","B","C","D"]
const[primeiro, segundo] = letras
```

```
var letras = ["A","B","C","D"]
const[primeiro,, terceiro] = letras
```

ES6

Spread & Rest Operator...

Ambos são bem parecidos e podem gerar muita confusão porém, existem suas diferenças. Spread tem a ideia de “espalhar” os dados, enquanto o Rest tem a ideia de “pegar o resto dos dados”. Este operador é muito útil para manipulação de Arrays, Objetos e também na implementação de funções mais dinâmicas.

...Spread

```
var meuArray = [1,2,3]
var meuArray2 = [6,7,8]
var meuArrayConcatenado = [...meuArray, 4, 5, ...meuArray2, 9, 10, 11]
console.log(meuArrayConcatenado)
```

ES6

Spread & Rest Operator...

...Rest

```
var nomes = ["João", "Juliana", "Alexandre", "Roberto"];
const [primeiro, ...resto] = nomes;

console.log(primeiro)
console.log(resto)
```

ES6

Spread & Rest Operator...

Aplicação em objetos ...Spread

```
const pessoa = {
    nome: "Abner",
    idade: 30,
    endereco:{
        cidade: "Petrópolis",
        estado: "Rio de Janeiro"
    },
    gritar: () => {
        console.log("AAAAAAAAAAAAAAAAA");
    }
}

const pessoa2 = {...pessoa, idade: 18}
console.log(pessoa2)
```

ES6

Spread & Rest Operator...

Aplicação em objetos ...Rest

```
const pessoa = {
    nome: "Abner",
    idade: 30,
    endereco:{
        cidade: "Petrópolis",
        estado: "Rio de Janeiro"
    },
    gritar: () => {
        console.log("AAAAAAAAAAAAAAAAA");
    }
}

const { nome, ...resto} = pessoa;
console.log(nome)
console.log(resto)
```

ES6

Spread & Rest Operator...

Aplicação em funções ...Spread

```
function somarNumeros(n1, n2){
    console.log(n1 + n2)
}

var numeros = [2, 2]
somarNumeros(...numeros)
```

ES6

Spread & Rest Operator...

Aplicação em funções ...Rest

```
function nomesComRest(...nomes){
    console.log(nomes)
}

nomesComRest("João", "Juliana", "Alexandre", "Roberto")
nomesComRest("João", "Juliana")
```

ES6

Arrow Functions =>

Uma expressão arrow function possui uma sintaxe mais curta quando comparada a uma expressão de função.

```
ola = _ => 'Olá'
```

ES6

Arrow Functions =>

```
let ola = function () {
    return 'Olá!'
}
```

```
ola = _ => 'Olá'
```

ES6

Método Map

- O método **map()** invoca a função callback passada por argumento para cada elemento do Array e devolve um novo Array como resultado.

```
const registros = [
    { name: 'JAVA', aplicacao: 'Back-end'},
    { name: '.NET', aplicacao: 'Back-end'},
    { name: 'REACT', aplicacao: 'Front-end'},
    { name: 'SQL', aplicacao: 'Banco de Dados'},
    { name: 'REACT-NATIVE', aplicacao: 'Front-end'},
    { name: 'HTML', aplicacao: 'Front-end'},
    { name: 'CSS', aplicacao: 'Front-end'},
    { name: 'JAVASCRIPT', aplicacao: 'Front-end'},
]

const novoArray = registros.map((t, index) => {
    return `Nome: ${t.name} | Aplicação: ${t.aplicacao} | Posição: ${index}`;
})

console.log(novoArray)
```

ES6

Método Filter

- O método **filter()** cria um novo array com todos os elementos que passaram no teste implementado pela função fornecida.

```
const tecnologias = ['JAVA', '.NET', 'REACT', 'SQL', 'REACT-NATIVE', 'HTML', 'CSS', 'JAVASCRIPT']
const tecnologiasFiltradas = tecnologias.filter((tecnologia) => {
    return tecnologia.toLowerCase().includes('j')
})
console.log(tecnologiasFiltradas);
```

```
const numeros = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
const novoArray = numeros.filter(num => num > 5)
console.log(novoArray);
```

ES6

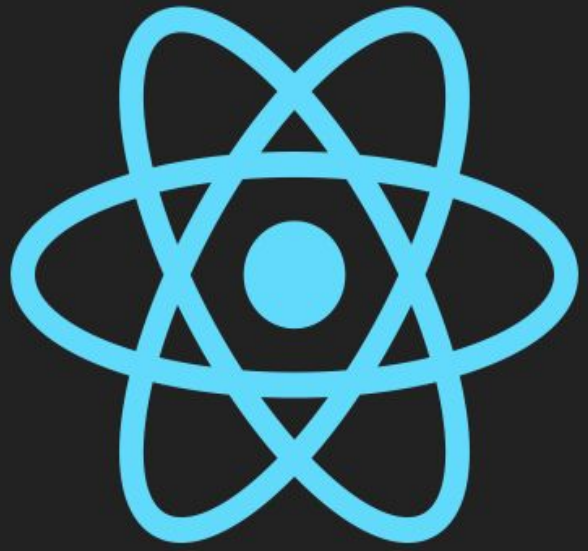
Método Reduce

- O método **reduce()** executa uma função **reducer** (fornecida por você) para cada elemento do array, resultando num único valor de retorno.

```
const numeros = [1, 2, 3, 4, 5]
const result = numeros.reduce((acumulador, valorCorrente) => acumulador + valorCorrente);
console.log(result);
```

ES6

Apresentação



REACT JS

História

- Biblioteca JavaScript (React.js ou ReactJS) criada em 2013.
- Foco em criar interfaces de usuário (frontend) em páginas web.
- É mantido pelo Facebook, Instagram, outras empresas e uma comunidade de desenvolvedores individuais.
- Em 2015, o Facebook anunciou o módulo de React Native, que em conjunto com o React, possibilita o desenvolvimento de aplicativos para Android e IOS
- Utiliza componentes de interface de usuário nativos de ambas plataformas, sem precisar recorrer ao HTML.
- Em uma pesquisa de 2018 sobre hábitos de desenvolvedores do site Stack Overflow, o React foi a terceira biblioteca ou framework mais citado o pelos usuários e desenvolvedores profissionais, ficando atrás somente do Node.js e Angular, respectivamente.

O que é React?

- De acordo com a própria documentação, o React nada mais é do que uma biblioteca JavaScript para criar interfaces de usuário.
- Quando falamos em React, devemos sempre trazer em mente dois conceitos fundamentais da linguagem. **Componentização** e **Manipulação de estados (states)**
- Com React é possível criar **componentes** encapsulados que gerenciam seu próprio **estado.** Como a lógica do componente é escrita em JavaScript e não em templates, podemos facilmente passar diversos tipos de dados a longo da nossa aplicação e ainda manter o estado fora do DOM.

Por que o React?

- Simples
- Fácil de aprender
- Abordagem nativa
- Fácil testabilidade
- Componentização
- Comunidade Ativa
- Facilidade e diversidade de instalações de pacotes
- Criação de SPAs (Single Page Applications)

Algumas aplicações que utilizam React:

- Facebook
- Uber
- Instagram
- WhatsApp
- Khan Academy
- Airbnb
- Dropbox
- Flipboard
- Netflix
- Paypal

Como o React funciona?

SPA

- Melhor experiência do usuário
- Performance
- Responsabilidade do client-side
- Facilidade de manutenção

Como o React funciona?

- JSX - Linguagem permite escrever o HTML dentro do Javascript.
- Babel - converte o código para uma versão da linguagem que o processador entenda. No React, converte o JSX em JavaScript através da transpilação.
- Webpack
  - Utiliza todo o código gerado pelo babel e cria bundle(um pacote) com um único código para o browser. Para cada tipo de arquivo, o código é convertido de uma maneira diferente.
  - Loaders ensinam a fazer as importações de arquivos de diferentes fontes (CSS, imagens, etc...)
  - Live Reload do Webpack Server

Gerenciador de pacotes

- **npm**: Gerenciador de pacotes padrão do Node.js
- **yarn**: Gerenciador de pacotes criado pelo Facebook
- Repositório onde ficam armazenados os pacotes
- Cliente que permite o envio / download de código do repositório.

Gerenciador de pacotes

Instalação do yarn (OPCIONAL)

- Método 1: Através do instalador contido neste link.
- Método 2: Através do Chocolatey. Para isso, primeiro é necessário instalar o Chocolatey na sua máquina. Você pode fazer a instalação do Chocolatey através das instruções contidas neste link. Depois, basta executar o comando.

  choco install yarn

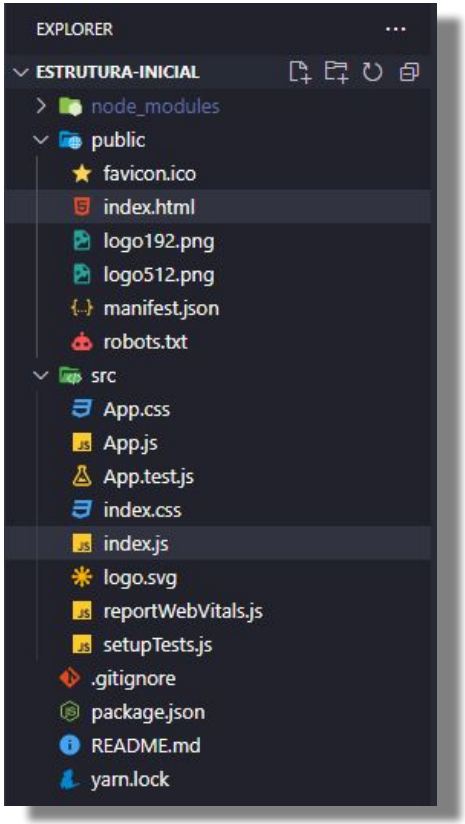
Estrutura Inicial do React

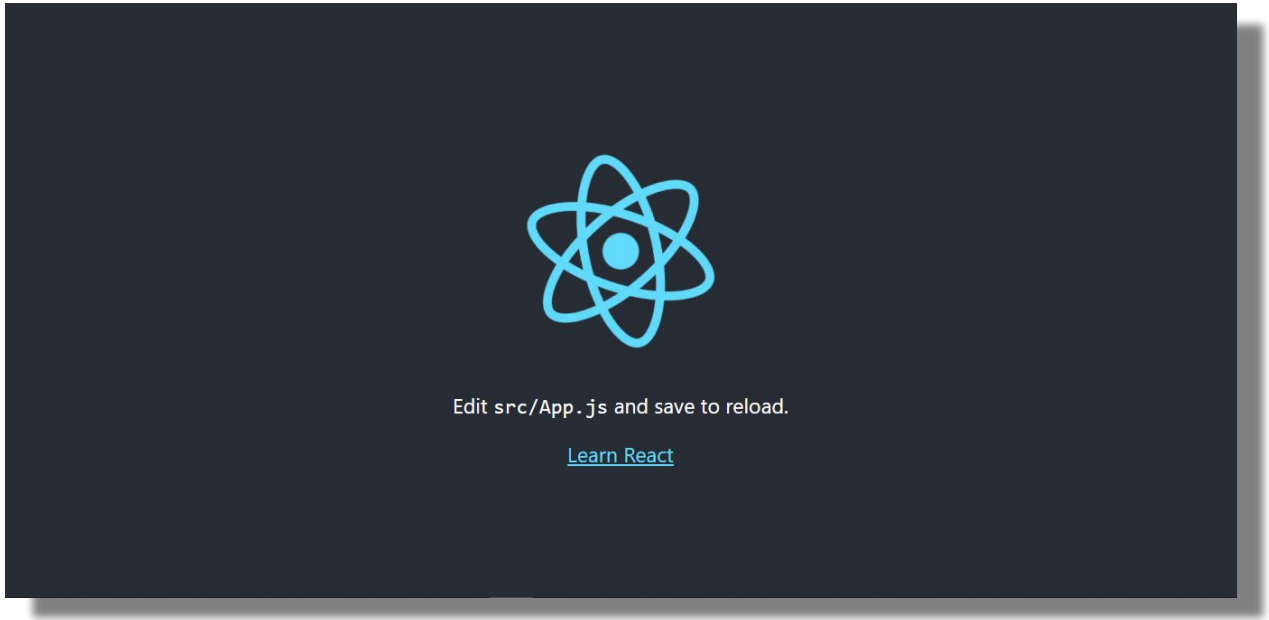
Com o auxílio do npx, package runner do npm, iremos executar um comando que nos propiciará a configuração inicial de uma aplicação react. No terminal, vamos executar o comando npx create-react-app nome-do-app.

Instalação dos pacotes e configurações iniciais de um projeto react

```
npm
Microsoft Windows [versão 10.0.19042.1288]
(c) Microsoft Corporation. Todos os direitos reservados.

C:\Users\AbnerJóia\Documents\Aulas-Residencia\ReactWEB\Exemplos-criação-slide\Estrutura-Inicial>npx create-react-app est
rutura-inicial
npx: installed 67 in 4.612s

Creating a new React app in C:\Users\AbnerJóia\Documents\Aulas-Residencia\ReactWEB\Exemplos-criação-slide\Estrutura-Inic
ial\estrutura-inicial.

Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts with cra-template...

yarn add v1.22.10
[1/4] Resolving packages...
[2/4] Fetching packages...
info fsevents@1.2.13: The platform "win32" is incompatible with this module.
info "fsevents@1.2.13" is an optional dependency and failed compatibility check. Excluding it from installation.
info fsevents@2.3.2: The platform "win32" is incompatible with this module.
info "fsevents@2.3.2" is an optional dependency and failed compatibility check. Excluding it from installation.
[3/4] Linking dependencies...
warning "react-scripts > @typescript-eslint/eslint-plugin > tsutils@3.20.0" has unmet peer dependency "typescript@>=2.8.
0 || >= 3.2.0-dev || >= 3.3.0-dev || >= 3.4.0-dev || >= 3.5.0-dev || >= 3.6.0-dev || >= 3.6.0-beta || >= 3.7.0-dev || >=
 3.7.0-beta".
[##------------------------------------------------------------------------------------------------------------] 638/34288
```

Estrutura Inicial do React



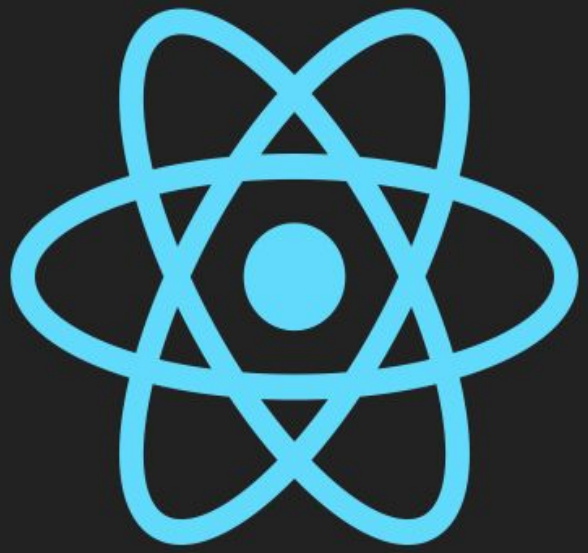


Estrutura Inicial do React

Após a criação do projeto react e de conhecermos sua estrutura inicial e como ele trabalha podemos startar nosso projeto e ver o resultado gerado no navegador deste projeto inicial.

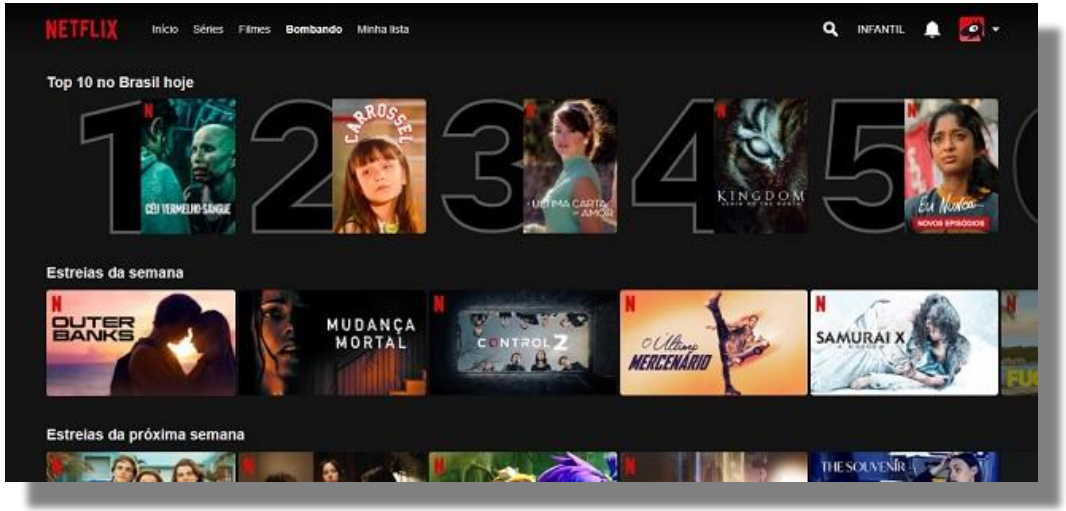
Para isso utilizamos o comando npm start ou yarn start no cmd dentro do diretório do projeto.

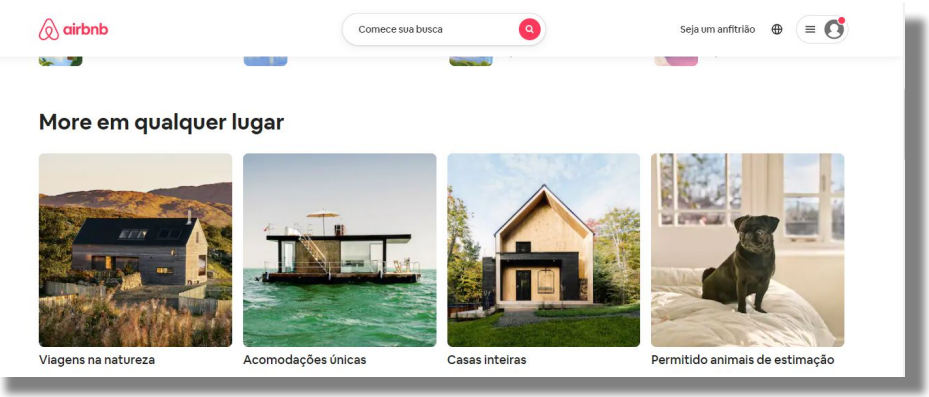
Componentes



REACT JS

Componentes





Principais Conceitos Práticos

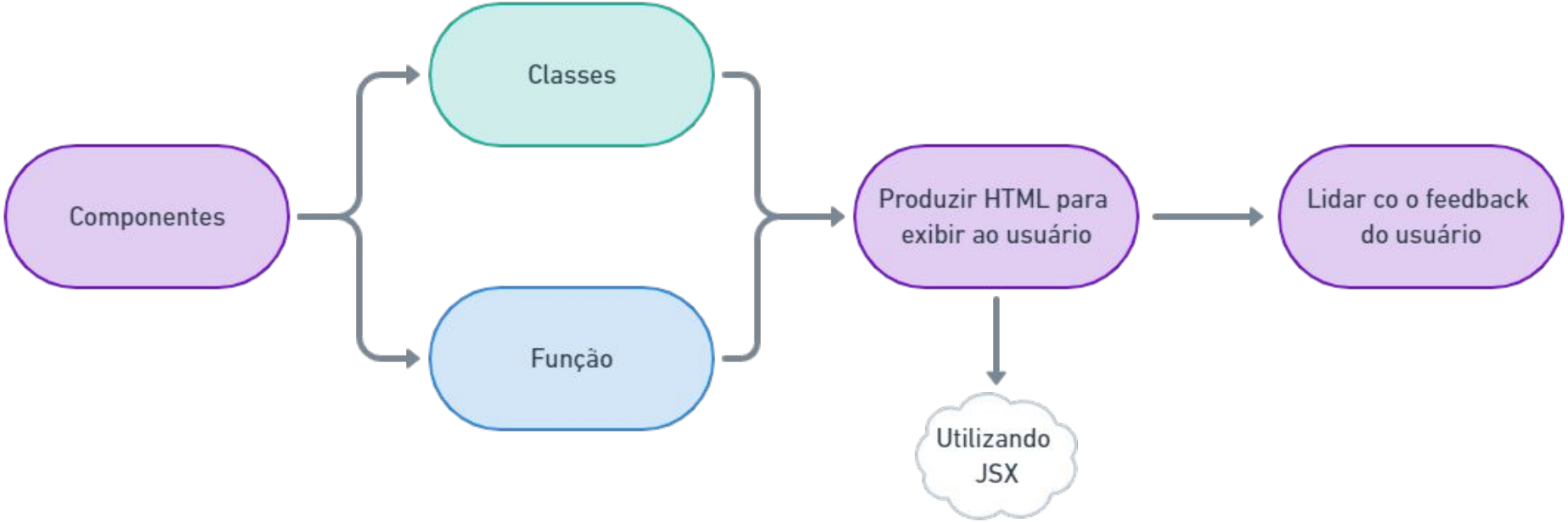
- Componentes de Classe
  - Utiliza o método render()
  - Permite a utilização de states
  - Permite o uso de métodos de ciclo de vida(lifecycle methods)
- Componentes de Função
  - Forma mais simples de definir um componente.
  - Não mantém states
  - Não permite a utilização de lifecycle methods

```
function App(){
  return <h1>Hello World</h1>
}
```

```
class App extends React.Component {
  render(){
    return <h1>Hello World!</h1>
  }
}
```

```
const App = () => {
  return <h1>Hello World</h1>
}
```

Componentes



Componentes de Classe - States

- Gerenciados de dentro do componente (Como variáveis declaradas dentro de uma função)
- Diferentemente de variáveis comuns, só podem ter seu valor alterado através do setState();
- Quando o state muda, o componente responde renderizando novamente;
- O setState() é **assíncrono.** O React intencionalmente “espera” até todos os componentes terem chamado setState() em seus manipuladores de evento antes de começar a renderizar novamente. Isso aumenta performance por evitar renderizações desnecessárias.

Componentes de Classe - States

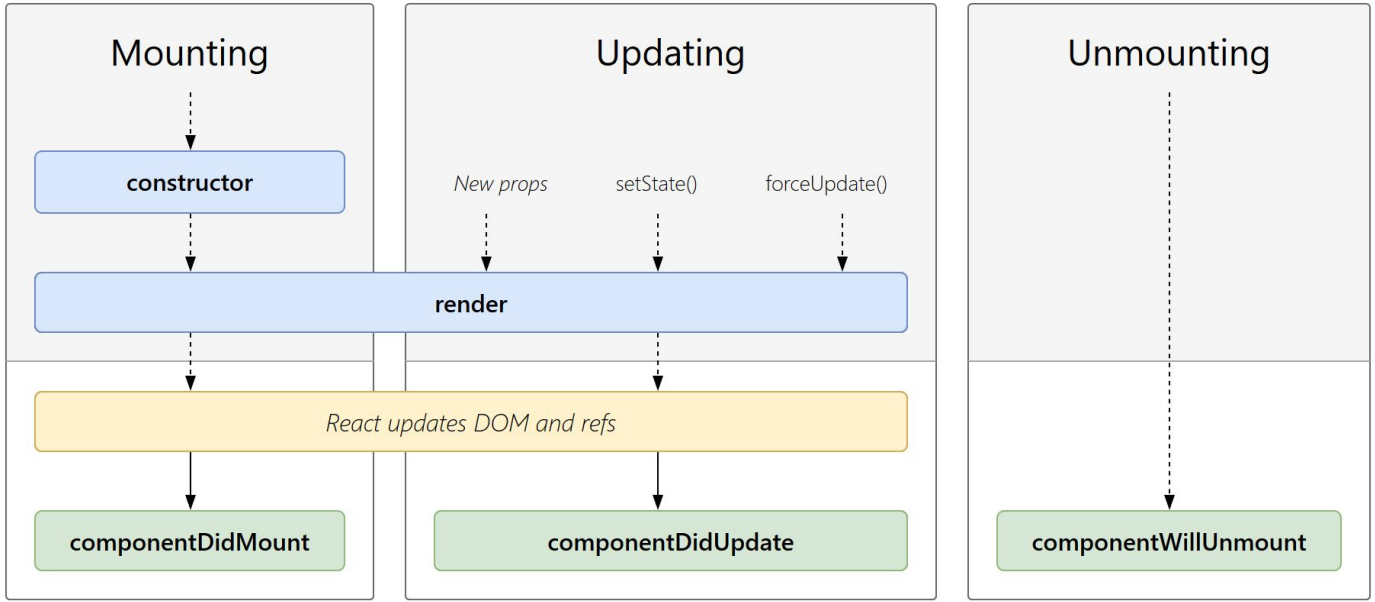
Aplicação utilizando componentes de classe

```
import React from "react";

class App extends React.Component {
  state = {
    nome: "",
  };

  handleClick = () => {
    if(this.state.nome === 'Fulano'){
      this.setState({nome: ''})
    }else{
      this.setState({nome: 'Fulano'})
    }
  }

  render() {
    return (
      <>
        <div>Hello {this.state.nome}</div>
        <button onClick={this.handleClick}>Click aqui</button>
      </>
    );
  }
}

export default App;
```

Componentes de Classe - Métodos de Ciclo de Vida (Lifecycle Methods)

- Cada componente do React tem um ciclo de vida que você pode monitorar e manipular durante suas três fases principais.
- As três fases são: Montagem, Atualização e Desmontagem.
- **Montagem**: inserção de elementos no DOM.
- **Atualização**: Como o nome já diz, acontece quando um componente é atualizado (mudança no state ou props do componente).
- **Desmontagem**: remoção de elementos no DOM

Componentes de Classe - Métodos de Ciclo de Vida (Lifecycle Methods)



Componentes de Classe - Métodos de Ciclo de Vida (Lifecycle Methods)

- **componentDidMount**: É executado quando todos os componentes da tela são renderizados corretamente. **Geralmente** é utilizado para fazer requisições a APIs.

```
componentDidMount() {
  axios
    .get(`https://api.github.com/users/${this.props.user}`)
    .then((response) => {
      this.setState((state) => (state = response.data));
    });
}
```

- **componentDidUpdate**: É executado quando um componente é atualizado (quando um state é alterado, por exemplo). Tem, como seus dois primeiros argumentos, as props anteriores e o state anterior.

```
componentDidUpdate(prevProps, prevState) {
  if (this.state.name !== prevState.name) {
    console.log(
      `State alterado de ${prevState.name} para ${this.state.name}`
    );
  }
}
```

Componentes de Classe - Métodos de Ciclo de Vida (Lifecycle Methods)

- **componentWillUnmonting** é executado no momento em que o componente é removido do DOM “Retirado da tela”

Props

- Quando o React vê um elemento representando um componente definido pelo usuário, ele passa atributos JSX e componentes filhos para esse componente como um único objeto. Nós chamamos esse objeto de “props”.
- Utilizamos as props geralmente para passar informações armazenadas em states para componentes filhos, porém podemos passar qualquer tipo de objeto para estes componentes.

```
import React from 'react';
import ReactDOM from 'react-dom';

const Welcome = (props) => {
  return <h1>Hello, {props.name}</h1>;
}

ReactDOM.render(
  <React.StrictMode>
    <Welcome name="Abner Jóia" />
  </React.StrictMode>,
  document.getElementById('root')
);
```

Componentes de Função

- Componentes funcionais eram utilizados somente quando não havia necessidade de se manipular um state ou de fazer uso dos lifecycle methods
- Com a criação dos Hooks, agora é possível fazer uso de ambas ferramentas citadas acima em componentes funcionais
- A utilização dessas ferramentas é um pouco diferente através dos Hooks, então vamos dar uma olhada.

Componentes equivalentes no ponto de vista do React

```
class Welcome extends React.Component() {
    render(){
        return <h1>Hello, {this.props.name}</h1>
    }
}
```

```
function Welcome(props) {
    return <h1>Hello, {props.name}</h1>
}
```

Componentes de Função

- States: Os states deverão ser importados como useState de dentro do pacote ‘react’. Para isso, utilizaremos as chaves para fazer uma importação não-default.
- Lifecycle Methods (componentDidMount e componentDidUpdate): Os Lifecycle Methods deverão ser importados como useEffect de dentro do pacote ‘react’. Para isso, utilizaremos as chaves para fazer uma importação não-default.

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

Componentes de Função - Hooks

Sintaxe

useState

```
const[state, setState] = useState(initialState);
```

useEffect

```
useEffect(() => {
    //Efeito
}, [Dependencia])
```

OBS: Se o array de dependência do useEffect permanecer vazio, ele será executado apenas uma vez! (comportamento semelhante ao componentDidMount)

Comunicação direta

- Comunicação direta se refere quando temos a necessidade de passar informação do componente pai para o componente filho.
- Passamos essas informações através das props

```
import React from "react";
import ComponenteFilho from "../ComponenteFilho";

function ComponentePai(){
    return(
        <ComponenteFilho nome="Gabriel" idade={20} nerd={true}/>
    )
}

export default ComponentePai;
```

```
import React from "react";

function ComponenteFilho(props) {
  return (
    <>
      <span>{props.nome} </span>
      <span>{props.idade} </span>
      <span>{props.nerd ? "Verdadeiro" : "Falso"}</span>
    </>
  );
}

export default ComponenteFilho;
```

Comunicação Indireta

- A comunicação indireta ocorre quando temos a necessidade de passarmos informações do componente filho para o componente pai.
- Aplicação no componente Pai.

```
import React, {useState} from "react";
import IndiretoFilho from "../IndiretoFilho";

function IndiretoPai() {
    const [pai , setPai] = useState({nome: '?', idade: 0, nerd: false})

    function informacoesDoPai(nome, idade, nerd){
        setPai({nome: nome, idade: idade, nerd: nerd});
    }

  return (
    <>
      <span>{pai.nome} </span>
      <span><strong>{pai.idade}</strong> </span>
      <span>{pai.nerd ? 'Verdadeiro' : 'Falso'} </span>
      <IndiretoFilho funcaoInformacoes={informacoesDoPai} />
    </>
  );
}

export default IndiretoPai;
```

Comunicação Indireta

- Componente filho

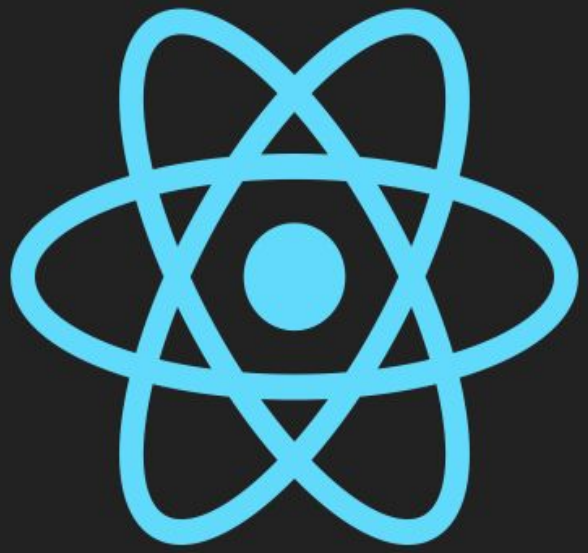
```
import React from "react";

function IndiretoFilho(props) {
  return (
    <>
      <h3>Filho</h3>
      <button onClick={() => props.funcaoInformacoes('João', 60, true)}>Enviar Informações</button>
    </>
  );
}

export default IndiretoFilho;
```

Componente Controlado

- Um componente controlado se dá quando um state está atrelado ao input, fazendo com que o resultado apresentado em tela seja reflexo da evolução de seu estado React. Fazendo o estado React ser a “única fonte da verdade”

```
function Input() {
  const [nome, setNome] = useState("Inicial");

  function handleChange(e) {
    setNome(e.target.value);
  }

  return (
    <div>
      <label>Nome: </label>
      <input type="text" value={nome} onChange={(e) => handleChange(e)} />
    </div>
  );
}
```

Navegação



REACT JS

Proposta da aula

- Construção aplicativos web com react
  - Renderização condicional
  - React Router
- Estilizando Componentes
  - Estilização Inline
  - Styled Components

Renderização Condicional

- Pode-se criar componentes distintos
- Renderizar apenas alguns dos elementos
- Tudo isso conforme o estado da sua aplicação
- Funciona da mesma forma que condições em Javascript
- Usando operadores como if para atualizar a UI desejada

Renderização Condicional

Considere estes dois componentes:

```
function SaudacaoUsuario(){
    return <h1>Bem-vindo de volta!</h1>
}
```

```
function SaudacaoVisitante(){
    return <h1>Por favor registre-se.</h1>
}
```

Renderização Condicional

Aplicando condição ternária

```
function RenderizacaoCondicional() {
  const [isLogado, setIslogado] = useState(false);

  function logar() {
    setIslogado(!isLogado);
  }

  return (
    <>
      {isLogado ? <SaudacaoUsuario /> : <SaudacaoVisitante />}
      <button onClick={() => logar()}>{isLogado ? 'Logoff' : 'Login'}</button>
    </>
  );
}
```

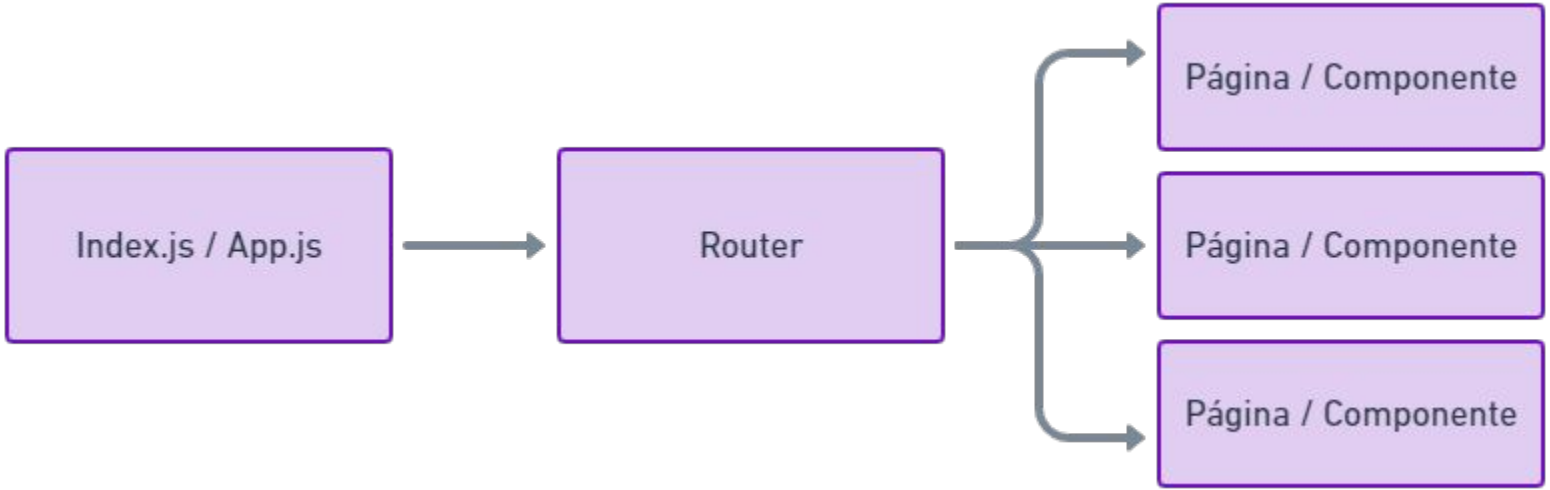
Renderização Condicional

Aplicação com função

```
function RenderizacaoCondicional() {
  const [isLogado, setIslogado] = useState(false);

  function logar() {
    setIslogado(!isLogado);
  }

  function validarLoginLogoff(props) {
    if (props) {
      return <SaudacaoUsuario />;
    }
    return <SaudacaoVisitante />;
  }

  return (
    <>
      {validarLoginLogoff(isLogado)}
      <button onClick={() => logar()}>{isLogado ? "Logoff" : "Login"}</button>
    </>
  );
}
```

React Router

O React Router permite criarmos rotas para navegação na aplicação permitindo carregar uma Página / Componente específico.



React Router

Como funciona?

Definimos as rotas baseada nas URLs da aplicação.

No navegador a aplicação React utiliza uma rota padrão http://localhost:3000 para acessar a página principal.

Ao criar uma nova página / componente podemos definir que a rota para a exibição deste componente será por exemplo: http://localhost:3000/sobre.

Baseado na rota da URL o React router irá renderizar o componente respectivo a rota estabelecida.

React Router

Podemos configurar as Rotas em qualquer lugar da nossa aplicação, porém é aconselhável a configuração nos arquivos Index.js, App.js ou em um componente específico para controlar as Rotas.

Para configurar as rotas da aplicação utilizaremos um pacote específico para esse tipo de implementação que é o React-Router-Dom.

Comando de instalação:

```
npm install --save react-router-dom@5.3.0
```

Documentação: https://reactrouter.com/web/guides/quick-start

React Router

Agora que já temos o pacote instalado vamos configurar as nossas rotas.

Primeira etapa é importar o pacote BrowserRouter.

Depois podemos englobar todas as páginas que terão rotas estabelecidas.

funcionou?

```
import React from "react";
import { BrowserRouter } from "react-router-dom";

import Home from "./pages/Home";
import QuemSomos from "./pages/QuemSomos";

function App() {
  return (
    <BrowserRouter>
      <Home />
      <QuemSomos />
    </BrowserRouter>
  );
}

export default App;
```

React Router

Precisamos importar mais dois pacotes do react-router-dom Switch e Route.

Switch, receberá vários componentes Route e dado o caminho que for passado na URL um deles será renderizado.

```
import React from "react";
import { BrowserRouter, Switch, Route } from "react-router-dom";

import Home from "./pages/Home";
import QuemSomos from "./pages/QuemSomos";

function App() {
  return (
    <BrowserRouter>
      <Switch>
        <Route path="/" exact component={Home}/>
        <Route path="/quemsomos" component={QuemSomos}/>
      </Switch>
    </BrowserRouter>
  );
}
```

React Router

- O path=”/” combinado com o atributo exact garantem que, se a rota tiver somente “/” ele vai ser renderizado.
- Isso evita conflitos com as outras rotas que possuam o valor passado no path.
- Podemos ainda passar uma rota default para página não encontradas após a última rota.
- Uma rota que representa a página 404 do nosso sistema.
  - <Route path=’*’ component={ComponenteDePagina404} />
- Agora é possível acessar: http://localhost:3000/QuemSomos e visualizar nosso componente.

React Router

Exemplo de aplicação página 404 Not Found

```
import React from "react";
import { BrowserRouter, Switch, Route } from "react-router-dom";

import Home from "./pages/Home";
import QuemSomos from "./pages/QuemSomos";
import NotFound from "./pages/PaginaNotFound";

function App() {
  return (
    <BrowserRouter>
      <Switch>
        <Route path="/" exact component={Home}/>
        <Route path="/quemsomos" component={QuemSomos}/>
        <Route path="*" component={NotFound} />
      </Switch>
    </BrowserRouter>
  );
}

export default App;
```

React Router

- Podemos navegar entre páginas sem precisar dar refresh
- Para isso utilizaremos outro componente do React Router Dom, Link
  - import { Link } from ‘react-router-dom’
- E usar dentro do JSX o componente dessa forma:
  - <Link to=”/quemsomos”>Ir para a página Quem Somos</Link>

```
import React from "react";
import { Link } from "react-router-dom";

function Home() {
  return (
    <>
      <h1>HOME</h1>
      <Link to="/quemsomos">Ir para a página Quem Somos</Link>
    </>
  );
}

export default Home;
```

React Router

- Outra forma de navegar é utilizar um Hook do react router dom chamado useHistory

```
import React from "react";
import { useParams, useHistory } from "react-router-dom";

import "./style.css";

function QuemSomos() {
  const { nome } = useParams();
  const history = useHistory();

  function handleClick(){
    history.push('/')
  }

  return (
    <>
      <h1 className="usuario-logado">Usuário logado: {nome}</h1>
      <h1>Quem Somos</h1>
      <button onClick={handleClick}>Ir Para Home</button>
    </>
  );
}

export default QuemSomos;
```

React Router

- Podemos passar parâmetros nas rotas enviando informações de uma página para outra.
- Precisamos informar na rota que será passado um parâmetro.

```
import React from "react";
import { BrowserRouter, Switch, Route } from "react-router-dom";

import Home from "./pages/Home";
import QuemSomos from "./pages/QuemSomos";
import NotFound from "./pages/PaginaNotFound";

function App() {
  return (
    <BrowserRouter>
      <Switch>
        <Route path="/" exact component={Home}/>
        <Route path="/quemsomos/:nome" component={QuemSomos}/>
        <Route path="*" component={NotFound} />
      </Switch>
    </BrowserRouter>
  );
}

export default App;
```

React Router

- Na página Home baseado em nosso exemplo informamos o parâmetro a ser passado para a página Quem somos.

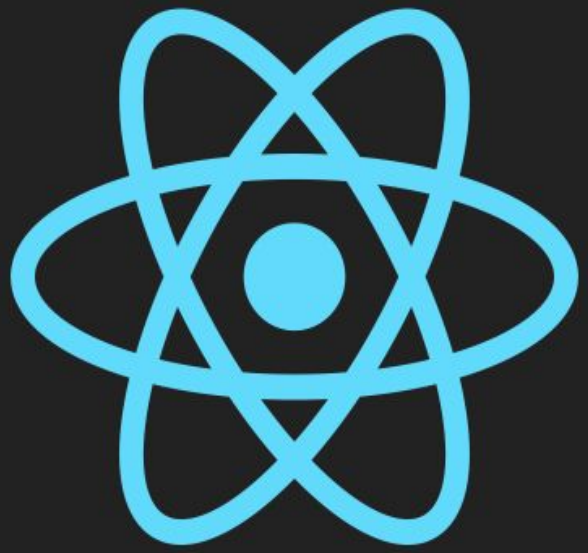
```
import React, {useState} from "react";
import { Link } from "react-router-dom";

function Home() {
    const [usuario] = useState({
        nome: 'Abner',
        idade: 30
    })

  return (
    <>
      <h1>HOME</h1>
      <Link to={`/quemsomos/${usuario.nome}`}>Ir para a página Quem Somos</Link>
    </>
  );
}

export default Home;
```

React Router

- Na página Quem Somos precisamos recuperar o parâmetro que está vindo na URL para exibirmos em tela ou manipular o valor.
- Utilizamos um Hook que importamos do react-router-dom o useParams
- Conseguimos obter a variável passada através da desestruturação.

```
import React from "react";
import { useParams } from "react-router-dom";

function QuemSomos() {
  const {nome} = useParams();

  return (
    <>
      <h1>Usuário logado: {nome}</h1>
      <h1>Quem Somos</h1>
    </>
  );
}

export default QuemSomos;
```

Estilização



REACT JS

Estilização Inline

- A propriedade style é usada para adicionar estilos em tempo de renderização dinamicamente.

```
<h1 style={{backgroundColor: 'blue', color: 'white', textAlign: 'center'}}>Usuário logado: {nome}</h1>
<h1>Quem Somos</h1>
```

```
const styleInline = {
  backgroundColor: "blue",
  color: "white",
  textAlign: "center",
};

return (
  <>
    <h1 style={styleInline}>Usuário logado: {nome}</h1>
    <h1>Quem Somos</h1>
  </>
);
```

Estilização CSS Externo

- Uma outra possibilidade é referenciar uma folha de estilo externa CSS.
- Estamos trabalhando com JSX, estamos desenvolvendo tudo no mundo Javascript e por esse motivo ao declarar uma classe em algum elemento existe uma pequena diferença na nomenclatura. class é uma palavra reservada do Javascript então quando quisermos referenciar uma classe de estilos utilizamos a nomenclatura className.

```
import "./style.css";

function QuemSomos() {
  const { nome } = useParams();

  return (
    <>
      <h1 className="usuario-logado">Usuário logado: {nome}</h1>
      <h1>Quem Somos</h1>
    </>
  );
}
```

Styled Components

- Precisaremos instalar outro pacote do React
  - npm install styled-components --save

```
import styled from 'styled-components';

export const Button = styled.button`
    cursor: pointer;
    background: transparent;
    font-size: 1rem;
    color: palevioletred;
    border: 2px solid palevioletred;
    margin: 0 1rem;
    padding: 0.25rem 1rem;
    transition: 0.5s all ease-out;
    &:hover{
        background-color: palevioletred;
        color: white;
    }
`
```

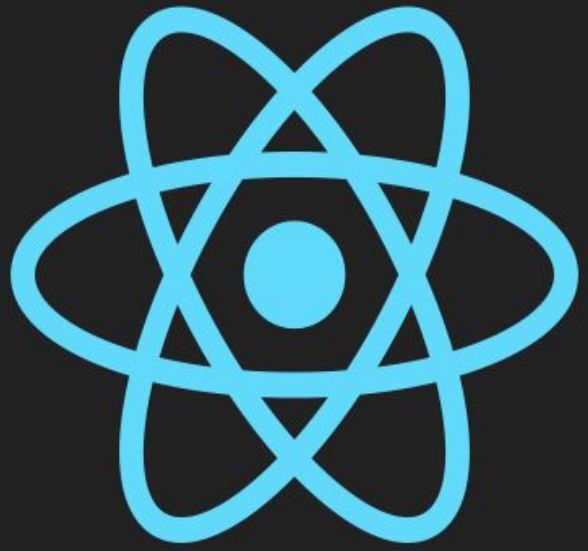
Styled Components

- Podemos criar componentes de estilo substituindo os elementos HTML por componentes que já carregam o próprio estilo.
- Com styled-components é possível receber props de estilo e alterar algum estilo de acordo com as props.

```
<Link to={`/quemsomos/${usuario.nome}`}>
  <Button cor='blue'>Ir para a página Quem Somos</Button>
</Link>
```

```
import styled from 'styled-components';

export const Button = styled.button`
    cursor: pointer;
    background: transparent;
    font-size: 1rem;
    color: ${props => props.cor};
    border: 2px solid palevioletred;
    margin: 0 1rem;
    padding: 0.25rem 1rem;
    transition: 0.5s all ease-out;
    &:hover{
        background-color: palevioletred;
        color: white;
    }
`
```

AXIOS



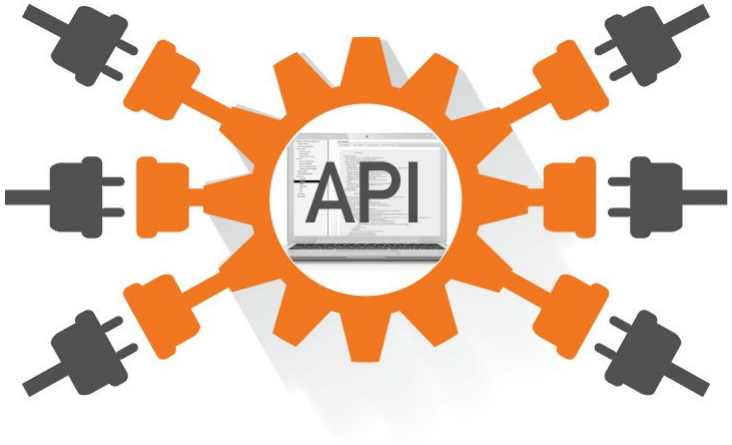
REACT JS

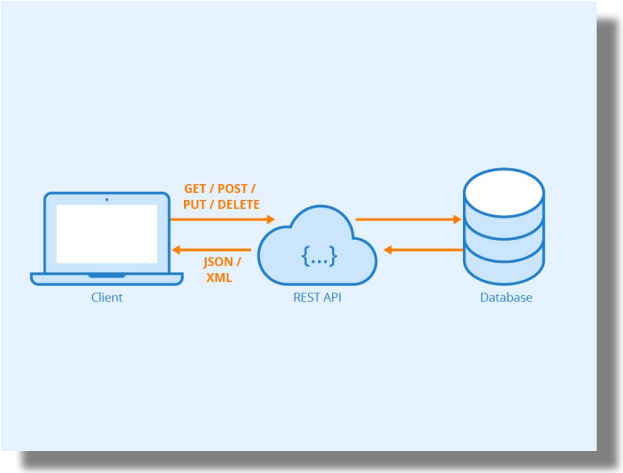
Proposta da Aula

- Trabalhando com APIs (APIs Calls)
  - Relembrando o conceito de APIs Calls
- Funções assíncronas
  - Async Await e Then
- Axios
  - Pacote de instalação
  - Configuração
  - Consumo de APIs com axios

Relembrando o conceito de API Calls

É possível fazer requisições por meio de API. Isso significa que você pode enviar ou receber informações para/de outro aplicativo, e os dados ou atributos capturados podem ser usados para exibir conteúdos, integrar aplicações ou sincronizar dados entre bancos de dados para enriquecer informações.





Funções Assíncronas

- Funções assíncronas: São funções que não acontecem ao mesmo passo do tempo de execução. Quando uma função assíncrona é chamada, ela retorna uma Promise, ou seja, uma promessa de retorno;
- Se eu declarar uma variável que receberá uma resposta de uma função assíncrona e logo em seguida tentar acessá-la, ela poderá não conter este valor neste momento, o que pode trazer um comportamento indesejável a nossa aplicação.

Async Await e Then

- Para que aguardemos a resposta de uma função assíncrona, podemos utilizar métodos Async Await ou Then().
- Desta forma, poderemos trabalhar com a segurança de que já recebemos o valor da nossa Promise

```
const getData = async () => {
  const response = await axios.get("https://api.github.com/users/Abjoia");
  setUsuario(response.data);
};
```

```
const getData = () => {
  axios
    .get("https://api.github.com/users/Abjoia")
    .then((response) => setUsuario(response.data));
};
```

Utilizando o Axios

- Com o Axios instalado, podemos setar as configurações iniciais para utilizá-lo em nossa aplicação. Para isso, dentro de src, vamos criar uma pasta chamada Services e um arquivo chamado api.js dentro dela.

- Dentro deste arquivo, vamos importar o axios e configurar nossa baseURL.

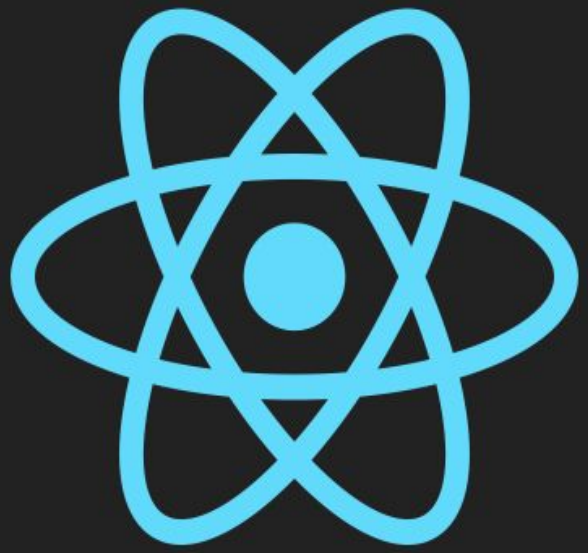
```
import axios from "axios";

const api = axios.create({
    baseURL: 'https://api.github.com/',
})

export default api;
```

Utilizando o Axios

- Agora basta importarmos o arquivo api.js em qualquer lugar da nossa aplicação onde queremos fazer um requisição à API declarada na baseURL.
- Exemplo:

```
import React, { useState, useEffect } from "react";
import api from "./services/api"

function App() {
  const [usuario, setUsuario] = useState({ login: "", name: "" });

  useEffect(() => {
    const user = 'Facebook';
    const getData = async () => {
      const response = await api.get(`users/${user}`);
      setUsuario(response.data);
    };
    getData();
  }, []);
```

Context API



REACT JS

Context API

- Em uma aplicação típica do React, os dados são passados de cima para baixo (de pai para filho) via props, mas esse uso pode ser complicado para certos tipos de props (como preferências locais ou tema de UI), que são utilizadas por muitos componentes dentro da aplicação. Contexto (context) fornece a forma de compartilhar dados como esses, entre todos componentes da mesma árvore de componentes, sem precisar passar explicitamente props entre cada nível.

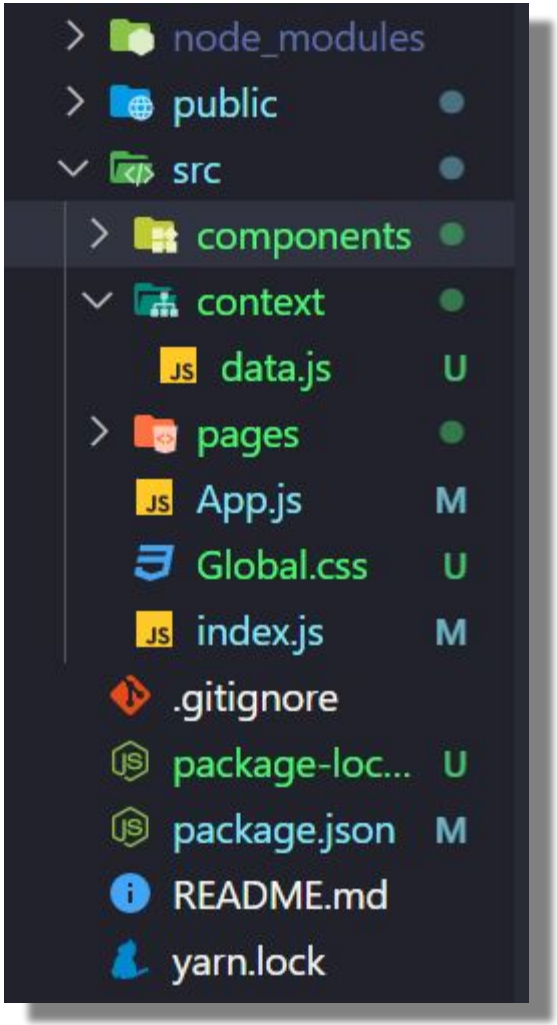
Context API

- Contexto (context) é indicado para compartilhar dados que podem ser considerados “globais” para a árvore de componentes do React. Usuário autenticado ou o idioma preferido, são alguns casos comuns.
- Usando contexto, nós podemos evitar passar props através de elementos intermediários.

Context API

Então como criamos um context?

- Devemos criar um diretório dentro de src chamado context para manter a organização e dentro dela criamos o arquivo de contexto.
- Como será um arquivo de dados não teremos JSX, por esse motivo criamos com a extensão js



Context API

- Dentro de data.js definimos o estado do contexto que serão os dados expostos a nível global na aplicação
- Iniciamos o contexto através do React.createContext()

```
import React from "react";

export const estadoContexto = {
    nome: "Abner"
}
export const DataContext = React.createContext(null);
```

Context API

- Agora precisamos englobar toda aplicação para que os dados sejam acessíveis pelos componentes.
- Os componentes mais alto nível que possuímos são App.js e index.js podemos aplicar em qualquer um deles.
- O DataContext contém uma propriedade Provider que precisamos utilizar passando o estadoContexto em value

```
import Home from "./pages/Home";
import Sobre from "./pages/Sobre";

import {DataContext, estadoContexto} from "./context/data";

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

function App() {
  return (
    <DataContext.Provider value={estadoContexto}>
      <BrowserRouter>
        <Switch>
          <Route path="/" exact component={Home} />
          <Route path="/sobre" component={Sobre} />
        </Switch>
      </BrowserRouter>
    </DataContext.Provider>
  );
}

export default App;
```

Context API

- Agora basta importar o contexto no componente para acessar os dados globais.

```
import React, {useContext} from "react";
import {DataContext} from '../../context/data'

function Display(){
    const {nome} = useContext(DataContext);
    return(
        <h1>{nome}</h1>
    )
}

export default Display;
```

Context API

- Podemos deixar a implementação do context mais interessante e adequada combinando com um componente. aumentando o poder da context API
- Com essa refatoração o contexto vai ficar assim =>

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

export const estadoContexto = {
    nome: "Abner"
}

export const DataContext = React.createContext(null);

function Contexto(props){
    const [nome, setNome] = useState(estadoContexto);

    function handleSetNome(e){
        setNome(e.target.value)
    }

    return(
        <DataContext.Provider value={{nome, handleSetNome}}>
            {props.children}
        </DataContext.Provider>
    )
}

export default Contexto;
```

Context API

- No App.js importamos o componente ao invés de importar o DataContext e o estadoContext
- O contexto engloba toda aplicação todos elementos filho de contexto são enviados por props automaticamente, que podem ser recuperados com props.children no componente contexto

```
import Home from "./pages/Home";
import Sobre from "./pages/Sobre";

import Contexto from "./context/data";

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

function App() {
  return (
    <Contexto>
      <BrowserRouter>
        <Switch>
          <Route path="/" exact component={Home} />
          <Route path="/sobre" component={Sobre} />
        </Switch>
      </BrowserRouter>
    </Contexto>
  );
}

export default App;
```

Context API

- No componente contexto retornamos DataContext.Provider englobando todos os filhos de context, logo englobando toda a aplicação.
- Essa implementação permite uma organização maior no código e facilita na criação de estados.

```
function Contexto(props){
    const [nome, setNome] = useState(estadoContexto);

    function handleSetNome(e){
        setNome(e.target.value)
    }

    return(
        <DataContext.Provider value={{nome, handleSetNome}}>
            {props.children}
        </DataContext.Provider>
    )
}

export default Contexto;
```

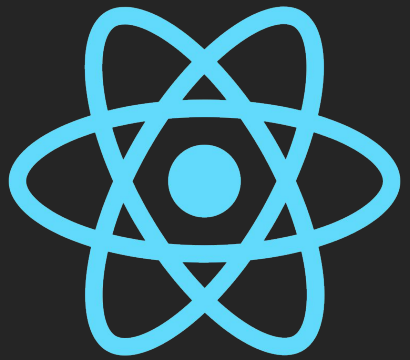
Context API

- Os componentes continuam usando os dados do contexto da mesma forma

```
import React, {useContext} from "react";
import {DataContext} from '../../context/data'

function Display(){
    const {nome} = useContext(DataContext);
    return(
        <h1>{nome}</h1>
    )
}

export default Display;
```

```
import React, {useContext} from 'react'
import {DataContext} from '../../context/data'

function Input(){
    const {nome, handleSetNome} = useContext(DataContext)
    return(
        <label>
            Nome:
            <input type="text" value={nome || ''} onChange={(e) => handleSetNome(e)}/>
        </label>
    )
}

export default Input;
```

Deploy

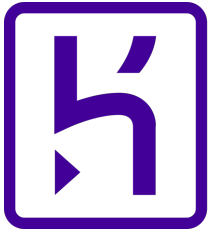


REACT JS

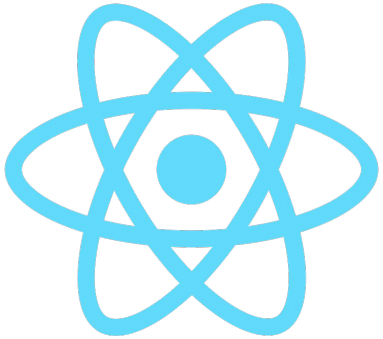
Deploy

O que será necessário?

- Git
- Uma conta no Heroku
- CLI do Heroku

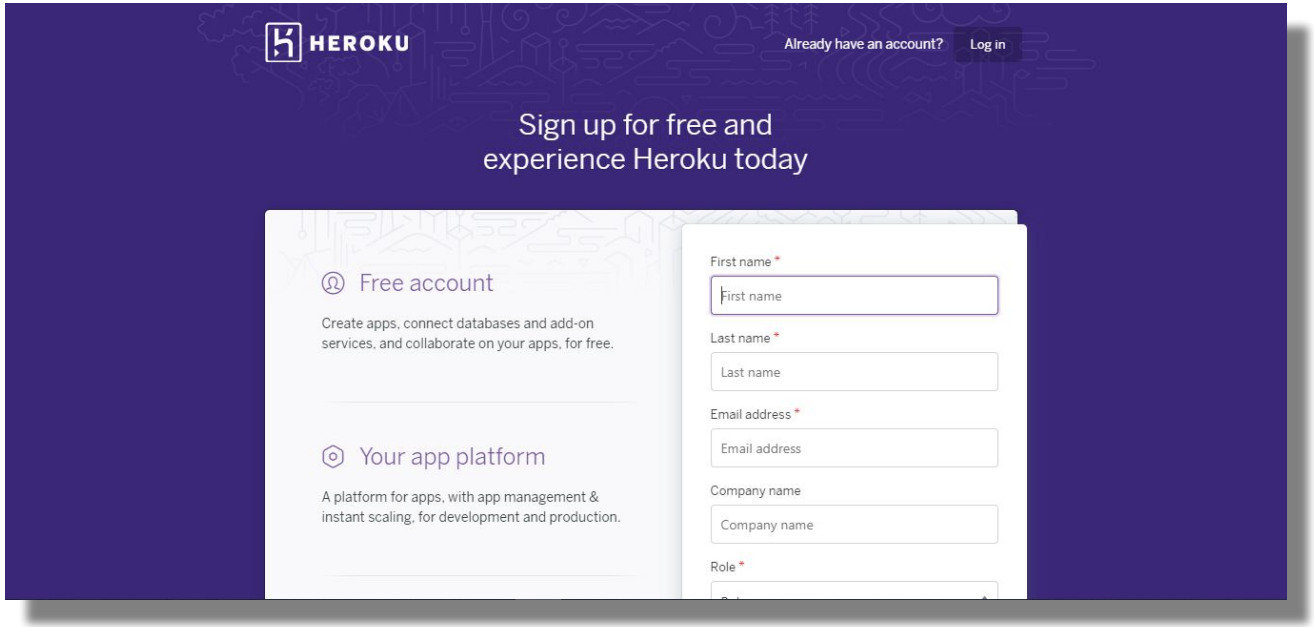


HEROKU



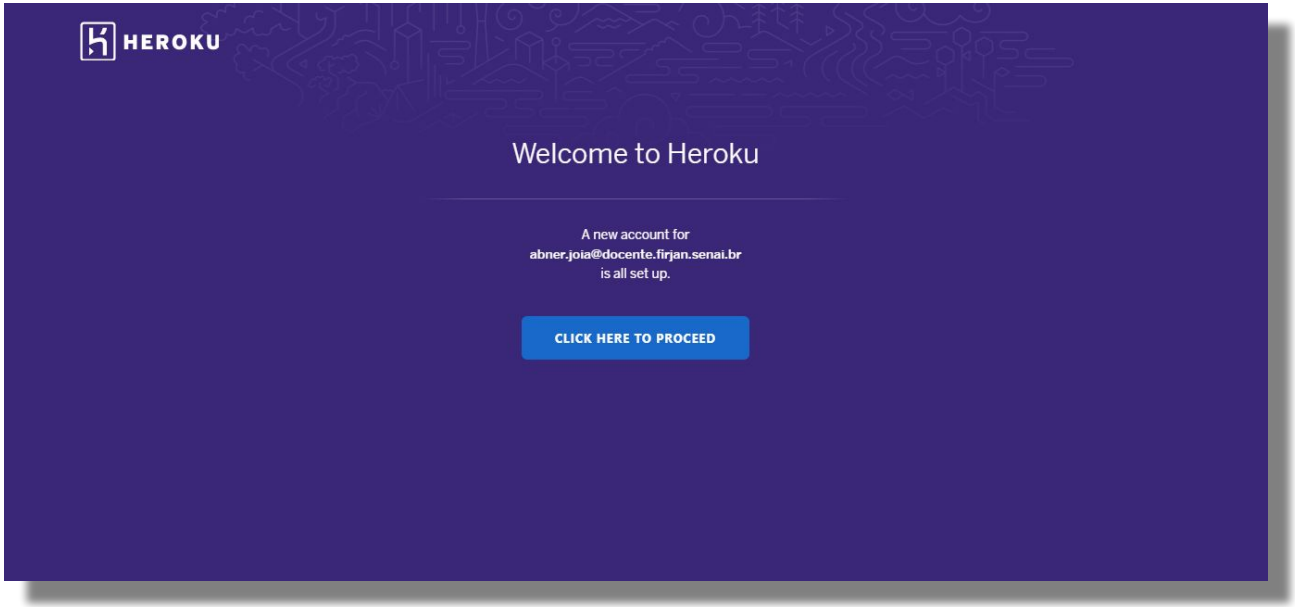
Heroku

- Criar conta no heroku https://signup.heroku.com/



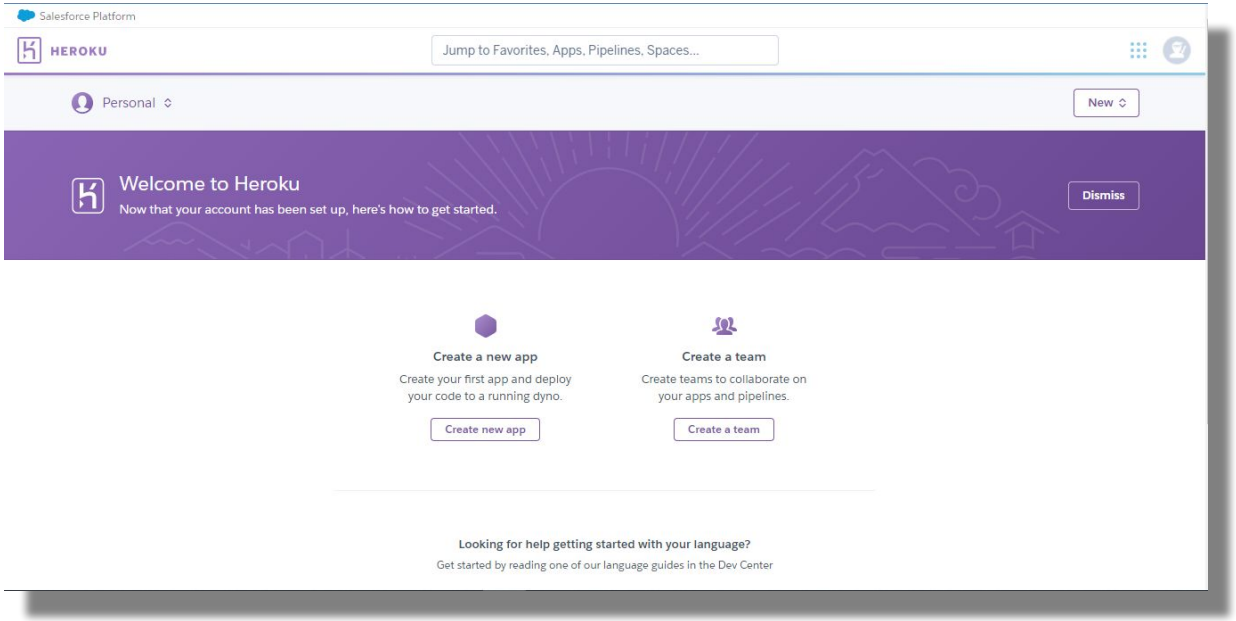
Heroku

- Depois de confirmar a conta pelo email e criar a senha você já terá uma conta ativada no heroku



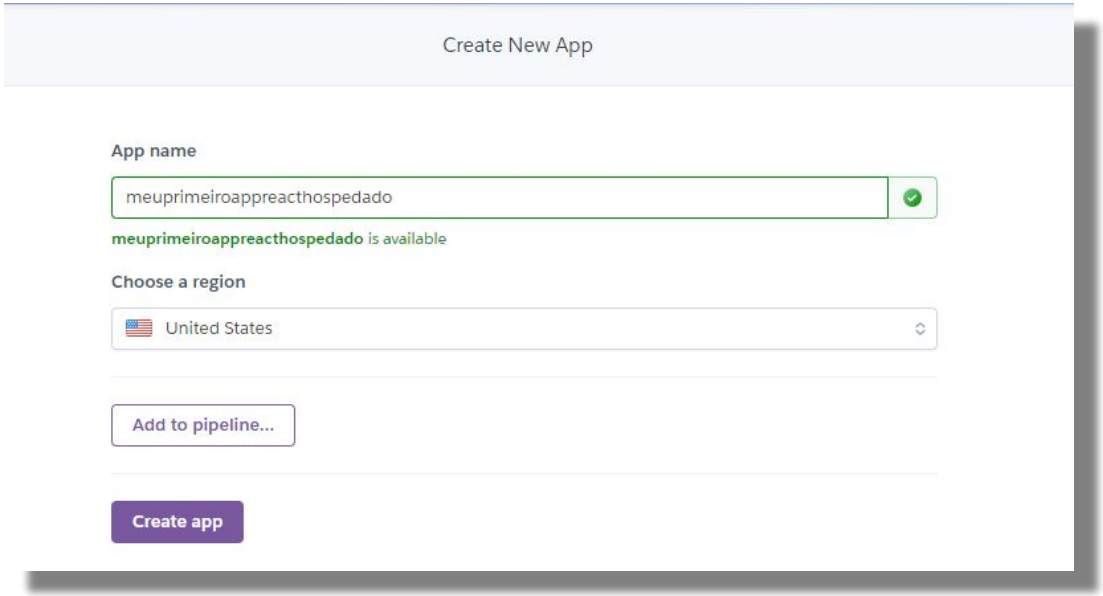
Heroku

- Acessando sua conta vá até dashboard



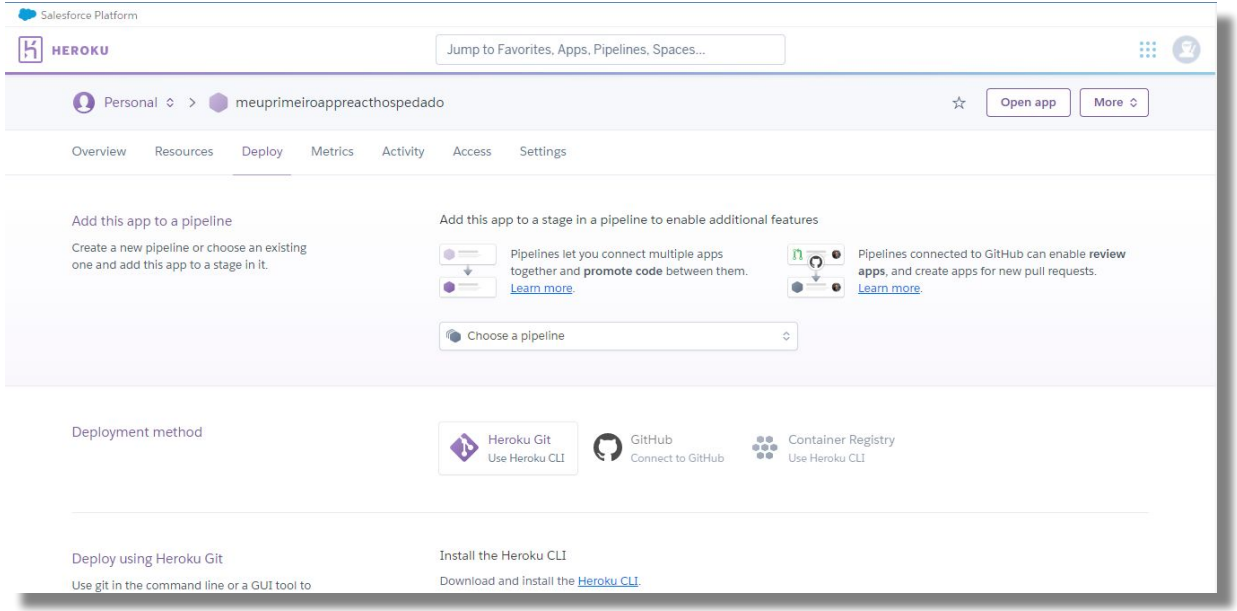
Heroku

- Create new app



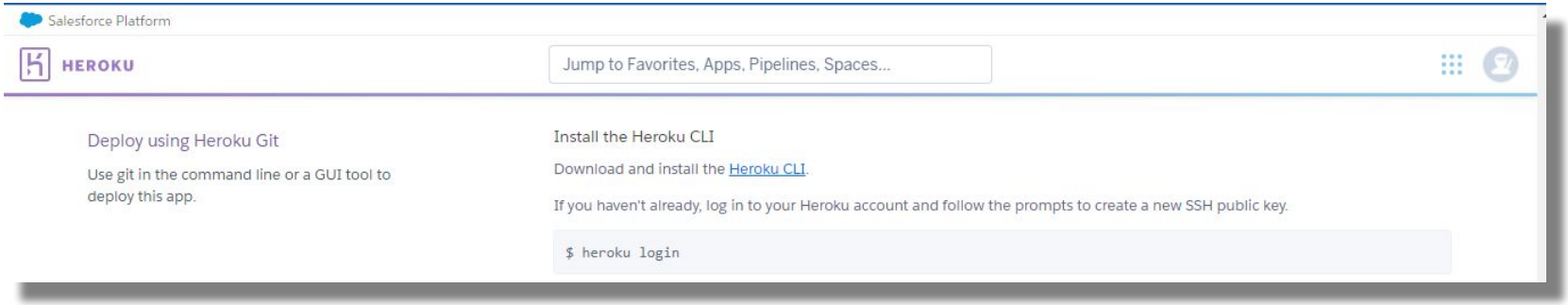
Heroku

- Esta tela apresenta todas as informações de sua aplicação no Heroku
- Em Deploy temos alguns metodos e o passo a passo para realizar o Deploy da nossa aplicação.



Heroku

- O método que vamos utilizar é o Heroku Git
- Precisamos baixar Heroku CLI



Heroku

- Depois da instalação do Heroku CLI seguimos com os próximos passos
- Todos os comandos sendo utilizados no GitBash

If you haven't already, log in to your Heroku account and follow the prompts to create a new SSH public key.

```
$ heroku login
```

Create a new Git repository

Initialize a git repository in a new or existing directory

```
$ cd my-project/
$ git init
$ heroku git:remote -a meuprimeiroappreacthospedado
```

Deploy your application

Commit your code to the repository and deploy it to Heroku using Git.

```
$ git add .
$ git commit -am "make it better"
$ git push heroku master
```

You can now change your main deploy branch from "master" to "main" for both manual and automatic deploys, please follow the instructions here.

Heroku

- heroku login

```
MINGW64:/c/Users/AbnerJóia/Documents/Aulas-Residencia/ReactWEB/Exem...

AzureAD+AbnerJóia@T2M0110 MINGW64 ~/Documents/Aulas-Residencia/ReactWEB/Exemplos-c
riação-slide/context/contextexemplo (main)
$ heroku login
heroku: Press any key to open up the browser to login or q to exit:
Opening browser to https://cli-auth.heroku.com/auth/cli/browser/b3a4a631-98b4-4895
-bc11-1544574fc60c?requestor=SFMyNTY.g2gDbQAAAA0xNzAuMjMzLjUxLjUwbgYAiFOsH3OBYgABU
YA.EhKqlRR-OK-1JlqYPXQbkrqNRSQN2A25uYVrdAC9j_U
heroku: Waiting for login...
Logging in... done
Logged in as abner.joia@docente.firjan.senai.br
```

Heroku

- seguindo os próximos passos...
- Quando criamos um projeto React utilizando create-react-app automaticamente ele inicia um repositório GIT local. Por esse motivo não existe a necessidade de executarmos git init

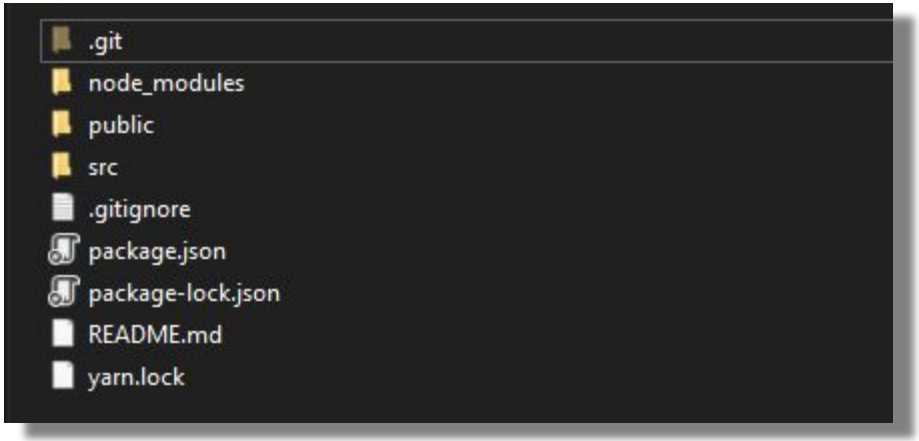
Create a new Git repository

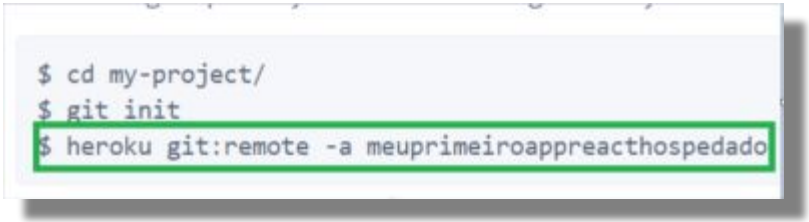
Initialize a git repository in a new or existing directory

```
$ cd my-project/
~~$ git init~~
$ heroku git:remote -a meuprimeiroappreacthospedado
```

Heroku

- O que precisamos é no diretório raiz do projeto React onde existe a pasta .git, abrir o git bash neste nível e executar heroku git:remote -a <nomedaaplicacao no heroku>





Heroku

- Agora nosso repositório local já está relacionado com nossa aplicação no Heroku

```
MINGW64:/c/Users/AbnerJóia/Documents/Aulas-Residencia/ReactWEB/Exem...

AzureAD+AbnerJóia@T2M0110 MINGW64 ~/Documents/Aulas-Residencia/ReactWEB/Exemplos
-criação-slide/context/contextexemplo (main)
$ heroku git:remote -a meuprimeiroappreacthospedado
set git remote heroku to https://git.heroku.com/meuprimeiroappreacthospedado.git

AzureAD+AbnerJóia@T2M0110 MINGW64 ~/Documents/Aulas-Residencia/ReactWEB/Exemplos-c
riação-slide/context/contextexemplo (main)
$
```

Heroku

- Podemos seguir com o deploy da nossa aplicação

Deploy your application

Commit your code to the repository and deploy it to Heroku using Git.

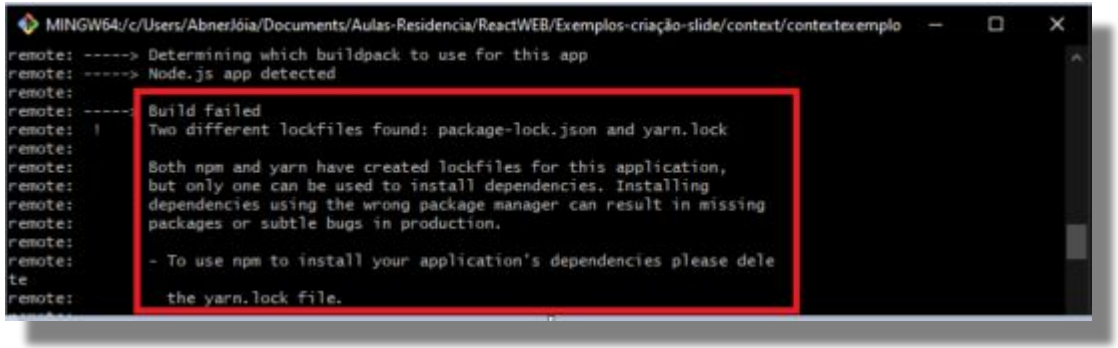
```
$ git add .
$ git commit -am "make it better"
$ git push heroku master
```

OBS: o heroku informa o comando com base na branch master porém, o novo padrão do github utiliza a branch main como a principal. Caso a branch principal do seu repositório for main deve substituir o apontamento de master para main

```
AzureAD+AbnerJóia@T2M0110 MINGW64 ~/Documents/Aulas-Residencia/ReactWEB/Exemplos-
criação-slide/context/contextexemplo (main)
$ git push heroku main
```

Heroku

- Caso seu projeto possua o yarn.lock possivelmente ocorrerá um erro de build e seu deploy será rejeitado.



Heroku

- Realizando a remoção precisamos commitar a modificação

```
MINGW64:/c/Users/AbnerJóia/Documents/Aulas-Residencia/ReactWEB/Exe...
AzureAD+AbnerJóia@T2M0110 MINGW64 ~/Documents/Aulas-Residencia/ReactWEB/Exemplos-
riação-slide/context/contextexemplo (main)
$ git status
On branch main
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    yarn.lock

no changes added to commit (use "git add" and/or "git commit -a")

AzureAD+AbnerJóia@T2M0110 MINGW64 ~/Documents/Aulas-Residencia/ReactWEB/Exemplos-
emplo (main)
$ git add .

AzureAD+AbnerJóia@T2M0110 MINGW64 ~/Documents/Aulas-Residencia/ReactWEB/Exemplos-
emplo (main)
$ git commit -m"removendo yarn.lock"
[main 37cc63f] removendo yarn.lock
 1 file changed, 11465 deletions(-)
 delete mode 100644 yarn.lock

AzureAD+AbnerJóia@T2M0110 MINGW64 ~/Documents/Aulas-Residencia/ReactWEB/Exemplos-
criação-slide/context/contextexemplo (main)
$
```

Heroku

- Após o commit precisamos apenas repetir o comando git push heroku master
- Depois do ajuste o build e o deploy devem ser realizados sem problemas
- Este seria o endereço de nossa aplicação https://meuprimeiroappreacthospedado.herokuapp.com

```
remote: -----> Build succeeded!
remote: -----> Discovering process types
remote:        Procfile declares types     -> (none)
remote:        Default types for buildpack -> web
remote:
remote: -----> Compressing...
remote:        Done: 65.4M
remote: -----> Launching...
remote:        Released v3
remote:        https://meuprimeiroappreacthospedado.herokuapp.com/ deployed to He
roku
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/meuprimeiroappreacthospedado.git
 * [new branch]      main -> main
```