



FrontEnd com React

Bootcamp Desenvolvedor Fullstack

Raphael Gomide

2021

FrontEnd com React

Bootcamp Desenvolvedor Fullstack

Raphael Gomide

© Copyright do Instituto de Gestão e Tecnologia da Informação.

Todos os direitos reservados.

Sumário

Capítulo 1. Introdução ao React	4
Considerações iniciais	4
React	4
Instalação e configuração	5
Características do React.....	7
Arquitetura do React	9
O arquivo App.js.....	10
Implementação simples com React – Projeto "react-intervalos"	11
Entendendo a implementação da aplicação	12
Capítulo 2. Class Components	15
Capítulo 3. Functional Components	17
Capítulo 4. React Hooks.....	19
O hook useState	19
O hook useEffect	20
Referências.....	23

Capítulo 1. Introdução ao React

Considerações iniciais

Prezado aluno, antes de abordarmos o assunto propriamente dito, peço para que utilize esta apostila como **referência** e não como o principal material do módulo. Como o Bootcamp é bastante prático, o principal e mais importante conteúdo situa-se nas **videoaulas**.

A orientação é que a apostila de módulos dos Bootcamps contenha aproximadamente 30 páginas. No Módulo 01, que também foi ministrado por mim – Raphael Gomide – a apostila é maior porque foi reaproveitada a partir da apostila oficial da disciplina de Fundamentos de Desenvolvimento Full Stack da pós-graduação em Desenvolvimento Full Stack do IGTI, onde a recomendação é que a mesma possua cerca de 60 páginas.

Outro detalhe importante são os exemplos desta apostila, que estão vinculados a projetos criados no CodeSandBox e não foram implementados nas videoaulas. Portanto, pode ser considerado um bom material complementar à apostila e ao módulo.

React

O React foi criado por colaboradores do Facebook e se denomina uma biblioteca JavaScript para construção de interfaces para o usuário. Foi inicialmente concebido para resolver um problema do Facebook de manter o estado das notificações que os usuários recebiam, que por muitas vezes não sincronizava corretamente.

A própria equipe de desenvolvimento do React o denomina como “a *JavaScript library for building user interfaces*”.

- Site oficial: <https://reactjs.org/>.

- Repositório no Github: <https://github.com/facebook/react>.

Instalação e configuração

O principal pré-requisito para a criação de apps com React é o Node.js, já que o React utiliza diversos de seus pacotes e necessita, por padrão, de diversas configurações para transpilação e empacotamento da aplicação. Assim, a configuração de um projeto React “do zero” não é nada trivial, em regra.

Para resolver esse problema, que poderia afastar entusiastas e principalmente novos desenvolvedores, foi criada uma ferramenta para simplificar o *scaffolding* de um novo projeto, denominada [create-react-app](#), também conhecida como CRA. Todos os projetos React deste módulo a utilizam. Para garantir uma melhor compatibilidade entre os projetos do professor e dos alunos – evitando assim bugs desnecessários de incompatibilidade – será sempre forçada a instalação da versão 3.4.1, que data de abril/2020. Para instalá-la já criando um projeto, execute o seguinte comando:

```
npx create-react-app --scripts-version 3.4.1 first-app
```

```
C:\igti  
λ npx create-react-app --scripts-version 3.4.1 first-app
```

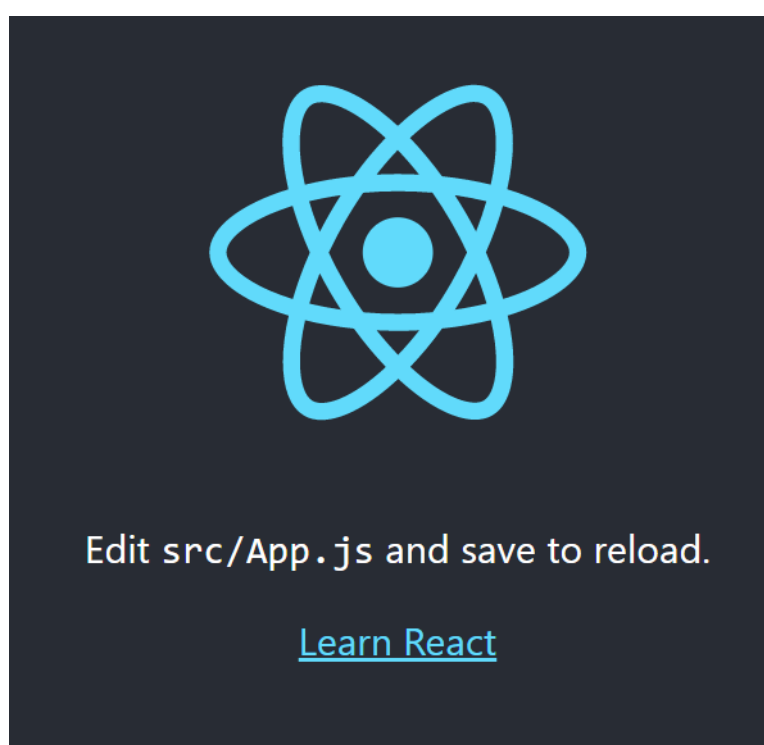
Instalação do *create-react-app*.

O comando acima faz o download de create-react-app, monta o *scaffolding* do projeto "first-app", faz o download de todas as dependências necessárias e, em seguida, descarta o pacote create-react-app (que de fato não é necessário para a continuidade do desenvolvimento em si).

Para a manutenção de pacotes, o React utiliza por padrão o [yarn](#), que é uma ferramenta que funciona como uma alternativa ao comando **npm**, que é nativo do Node.js. Para mais detalhes dos comandos do yarn acesse este [cheat sheet](#) (acesso em 29/07/2020). Um outro detalhe importante é que será utilizada a versão 1.x do

Yarn, já que a versão atual (2.x) foi totalmente remodelada. Portanto, pode-se afirmar que a versão 1.x é mais compatível, pelo menos por enquanto. As videoaulas iniciais demonstram como instalar o Yarn corretamente.

O servidor de desenvolvimento do React é executado, por padrão, na **porta 3000**. Para executar o seu projeto, acesse a pasta raiz do mesmo e escreva o seguinte comando em seu terminal de comandos: **yarn start**. Isso faz com que o servidor de desenvolvimento do React seja executado e o navegador padrão do Sistema Operacional seja inicializado em uma nova aba apontando para o endereço <http://localhost:3000>, onde há um projeto padrão do React inicializado.



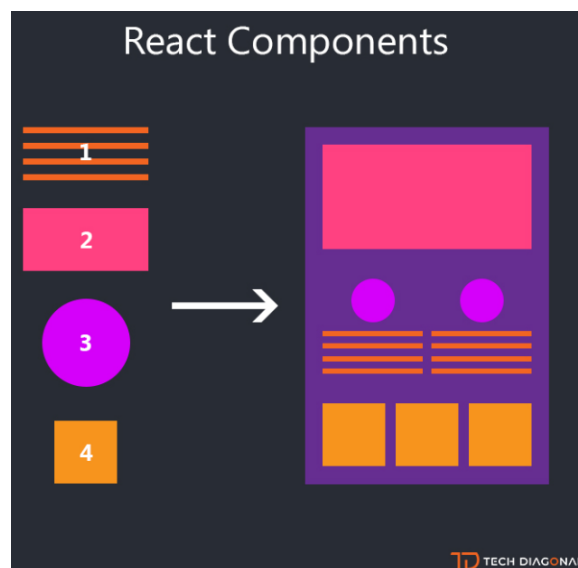
Aplicação inicial do create-react-app, versão 3.4.1.

Nas videoaulas será disponibilizado um projeto para servir de base para todos os demais projetos do módulo, evitando que o aluno tenha que seguir esses passos toda vez que for criar um novo projeto.

Características do React

O React se encontra atualmente na versão 16.x e foi totalmente reescrito internamente após a versão 15, sem afetar os projetos dos desenvolvedores.

Por ser baseado em componentes, o React permite muita reutilização de código. A figura abaixo ilustra este comportamento.



Reutilização de componentes no React.

Fonte: techdiagonal.com.

Perceba, na imagem acima, que os componentes **1** e **3** foram reutilizados duas vezes e que o componente **4** foi reutilizado três vezes.

A seguir são listadas algumas características importantes sobre o React.

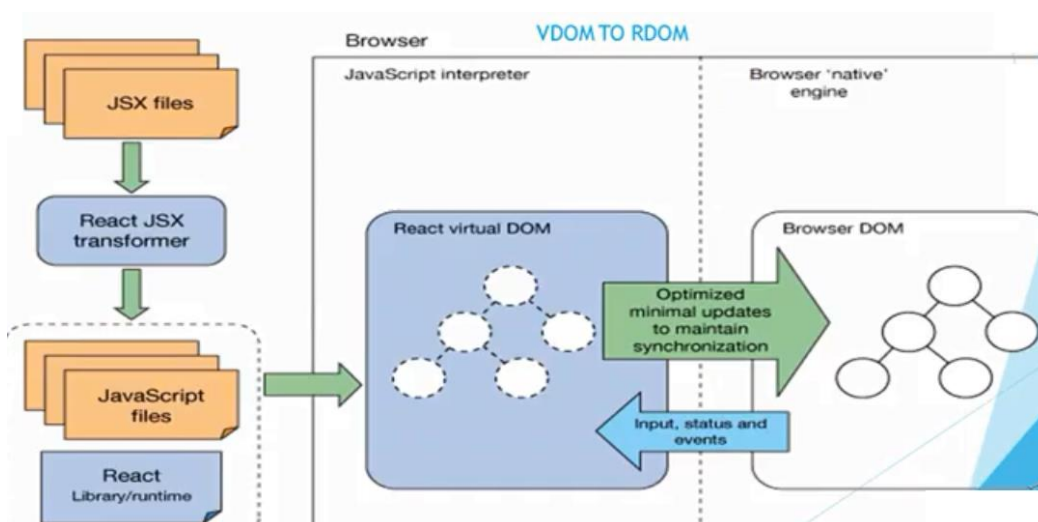
- **Componentizável:** o React também preza pela criação de componentes, assim como grande parte dos *frameworks* de JavaScript modernos. Componentes são blocos de código que, por serem altamente customizáveis, podem ser reutilizados em diversas partes de uma aplicação, onde cada instância do componente pode possuir o seu próprio estado e funcionalidades.

- **Declarativo:** seguindo os princípios do desenvolvimento reativo, a criação de componentes é bastante declarativa (ao invés de imperativa), o que faz com que o React **reaja** a mudanças no estado da aplicação de forma **eficiente**.
- **Virtual DOM:** em aplicações web, a manipulação do DOM é de responsabilidade do Virtual DOM no React, que cria uma estrutura em memória do DOM e só efetua as atualizações realmente necessárias, o que garante melhor desempenho e, conseqüentemente, melhor experiência do usuário, já que o app tende a ser mais fluido.
- **One-way data flow:** o React recomenda que concentremos o estado da aplicação no componente pai. Em regra, os filhos recebem dados do estado através de propriedades (**props**), que são, em regra, somente-leitura. Esse processo faz com que a lógica de alteração do estado se concentre em somente um componente, o que leva a menos *bugs* na aplicação. Os componentes filhos podem alterar o estado do componente pai através de eventos que, uma vez disparados, invocam funcionalidades do componente pai, que então possuem permissão para alterar o estado da aplicação. Entretanto, nada impede que componentes filhos também possuam estado.
- **Learn Once, Write Anywhere:** o React dá suporte ao desenvolvimento em diversas outras plataformas, com destaque para o React Native, que suporta por padrão *apps* nativos de dispositivos móveis com as plataformas Android e iOS. Ao entender bem o funcionamento do React para *web*, você aprende a desenvolver para outras plataformas mais facilmente.
- **JavaScript moderno (ES6+):** com o React, basta que o desenvolvedor aprenda JavaScript e os conceitos modernos do ES6+ e conheça a arquitetura do React. Isso permite mais flexibilidade e melhora a curva de aprendizado.
- **JSX (JavaScript XML):** o React suporta o **JSX** para facilitar a criação de componentes, tornando-os mais declarativos. A escrita com JSX torna o código de construção de componentes muito semelhante ao HTML. Entretanto, é necessária a utilização obrigatória do Babel para transpilar o código JSX,

tornando a aplicação compatível aos navegadores. Isso é feito automaticamente pelo **create-react-app**.

Arquitetura do React

A figura abaixo ilustra a arquitetura de aplicações com React:



Arquitetura de um projeto React.

Fonte: biznomy.com.

Analisando a imagem da esquerda para a direita e de cima para baixo, é possível perceber o seguinte:

1. É feita uma transformação (transpilação) de componentes feitos em JSX para código JavaScript, que junto ao código do React propriamente dito, são hospedados com a aplicação (em produção).
2. Durante a execução da aplicação, o React cria o VirtualDOM que monitora o DOM e só efetua a manipulação quando necessário e de forma eficiente.

O arquivo App.js

A figura a seguir mostra os principais componentes de App.js, que é criado automaticamente com o *create-react-app*, considerando a abordagem de **class components**.

```

1  import React, { Component } from 'react';  Importação de módulos
2  import logo from './logo.svg';             e assets
3  import './App.css';
4
5  class App extends Component {  Classes React devem herdar de Component
6    render() {                    Método de renderização de um componente
7      return (
8        <div className="App">
9          <header className="App-header">
10             <img src={logo} className="App-logo" alt="logo" />  Código JSX
11             <h1 className="App-title">Welcome to React</h1>
12           </header>
13           <p className="App-intro">
14             To get started, edit <code>src/App.js</code> and save to reload.
15           </p>
16         </div>
17       );
18     }
19   }
20
21   export default App;  Exportação do componente para ser utilizado
22                       externamente
  
```

Exemplo de escrita do arquivo App.js.

Algumas observações sobre a imagem acima:

- Perceba que o método *render* deve obrigatoriamente retornar algo. Esse retorno deve ser de **apenas um elemento**. Assim, para agrupar diversos elementos, pode ser utilizado um elemento *container*, como a **<div>**. O React possui o componente **React.Fragment**, que também faz esse papel. O **React.Fragment** pode também ser utilizado com os caracteres **<>** e **</>**.
- O trecho de código **src={logo}** indica que o caminho da imagem aponta para **'./logo.svg'**. Assim, utilizamos **{ }** para representar instruções JavaScript dentro de JSX.
- Perceba que o código JSX é bastante semelhante ao HTML. Uma das diferenças, por exemplo, é a utilização de **className** ao invés de **class**, já

que esta última é uma palavra reservada do JavaScript. É importante lembrar que JSX é JavaScript e não HTML.

- Para economizar texto e linhas, a linha 21 pode ser excluída e a linha 5 pode ser escrita como: ***export default class App...***

Implementação simples com React – Projeto "react-intervalos"

Este projeto visa mostrar a quantidade de números divisíveis por 2 a 9 com base em um número definido pelo usuário.

A figura a seguir mostra a implementação do projeto com React:

```

1  import React, { Component, Fragment } from 'react';
2
3  export default class App extends Component {
4    constructor() {
5      super();
6
7      this.state = {
8        currentValue: 1
9      };
10
11     this.divisors = [2, 3, 4, 5, 6, 7, 8, 9];
12   }
13
14   getDivisiveisPor(number) {
15     const numbers = [];
16     for (let i = 1; i <= this.state.currentValue; i++) {
17       if (i % number === 0) {
18         numbers.push(i);
19       }
20     }
21     return numbers;
22   }
23
24   render() {
25     return (
26       <Fragment>
27         <h3>Reatividade com intervalos de números - React</h3>
28         <div>
29           <div>
30             <label>
31               Contador
32               <input
33                 type="number"
34                 min="1"
35                 max="200"
36                 value={this.state.currentValue}
37                 onChange={event =>
38                   this.setState({ currentValue: event.target.value })
39                 }
40               />
41             </label>
42           </div>
43           <ul>
44             {this.divisors.map(divisor => {
45               return (
46                 <li key={divisor}>
47                   Números divisíveis por {divisor}:{' '}
48                   {this.getDivisiveisPor(divisor).map(number => {
49                     return <span key={number}>{number + ' '}</span>;
50                   })}
51                 </li>
52               );
53             })}
54           </ul>
55         </div>
56       </Fragment>
57     );
58   }
59 }
60

```

Implementação do projeto "react-intervalos"

Entendendo a implementação da aplicação

Na linha 1 é feita a importação da biblioteca do React, com destaque para Fragment, que será utilizada para retornar apenas um elemento no método render().

Nas linha 3 declaramos a classe `App` herdando de `Component`, que é uma classe do React. `Class Components` é uma das três formas básicas de se escrever componentes com React.

Nas linhas 4 a 12 criamos o construtor da classe que, como herda de `Component`, exige a invocação de `super()`. Além disso, criamos o estado da aplicação com `this.state` e instanciamos o atributo `this.divisors` que, sendo “fixo”, não necessita estar vinculado ao estado. Em geral, `this.state` recebe um objeto com um ou vários valores. O React trata `this.state` de forma especial, tornando-o reativo. No caso desta aplicação, o único dado que necessita ser realmente reativo é, de fato, `currentValue`.

Nas linhas 14 a 22 temos o método `getDivisiveisPor(number)`, cuja implementação consiste no cálculo dos divisores de determinado número.

Nas linhas 24 a 59 temos o principal método da aplicação, o `render()`, que vai fazer a renderização do conteúdo. Esse também é um método com tratamento especial do React. Perceba que `render()` retorna apenas um elemento (`<Fragment>`), que é um elemento *container* que agrupa todo o conteúdo com JSX.

Nas linhas 36 a 39 vinculamos o estado ao `value` do `<input>` e também definimos um novo valor do estado no evento `onChange` do `<input>`. Como o React não possui DSL (*Domain Specific Language*), devemos realizar esse tipo de implementação manualmente. Perceba na implementação de `onChange` que alteramos o estado de forma imutável através de `setState`, enviando um novo objeto como parâmetro. Isso acarreta em uma nova execução de `render()`, que então vai alterar os elementos que **observam** *state*, efetuando então a **reatividade**. Com o apoio do VirtualDOM, a execução de `render` é a mais eficiente possível.

Nas linhas 43 a 54 é feita a renderização dos elementos em tela, de forma reativa. Para isso, realizamos a implementação com ES6+ e a função `map`, que em essência realiza alguma transformação de dados e **retorna** um novo elemento (imutabilidade). No JSX, para invocarmos expressões JavaScript, basta utilizar a notação de *chaves* (*single braces*) - `{ }`. As cores das chaves facilitam a identificação do escopo de cada uma. Outro detalhe importante é a utilização de `key`, que é um

atributo recomendado pelo React que nos auxilia a reconciliar os elementos no VirtualDOM. Se não utilizarmos *key*, o React emite um alerta no console do navegador.

O código-fonte está disponível para visualização e estudo através do seguinte *link* - <https://codesandbox.io/s/react-intervalos-yutxu>. Faça testes incrementando e decrementando o input.

Nos próximos capítulos serão detalhados os três principais tipos de componentes que podem ser criados com o React, que são:

1. Class Components.
2. Functional Components.
3. Functional Components com Hooks.

Capítulo 2. Class Components

A abordagem de **Class Components** foi a primeira adotada pelo React e continua funcionando muito bem atualmente.

Com a criação dos **React Hooks** em 2018, a utilização de **Class Components** passou a ser considerada **verbosa** e **pouco declarativa** e, por isso, tende a ser descartada com o tempo, mesmo que a própria equipe do React tenha afirmado que vai manter o suporte para ambas abordagens, pelo menos a curto prazo. De qualquer forma, é possível afirmar que ainda vale a pena o seu estudo e utilização nas implementações de aplicações web com React, pois a tendência é que os componentes sejam migrados para a nova abordagem aos poucos.

Class Components baseiam-se em **classes** do JavaScript, herdam obrigatoriamente do objeto **Component** do React e possuem, essencialmente:

- **Construtor de classe:** aqui é geralmente definido o estado do componente com `this.state`, que pode ou não ser baseado em **props** (mais detalhes sobre **props** ainda serão vistos na apostila). A invocação de **super()**, que invoca instruções importantes de **Component**, é **obrigatória**.
- **Método render():** principal método de um **Class Component**, que realiza a renderização do componente em tela. Normalmente utiliza-se muito **JSX**, **Object Destructuring** e **Array.map** para se interpolar os dados.
- **Lifecycle methods:** são métodos opcionais de **Class Components** que capturam determinado evento no ciclo de vida do componente. Para mais detalhes sobre os lifecycle methods, acesse [este link](#) (acesso em 29/07/2020). Os principais lifecycle methods são:
 - **componentDidMount:** ocorre **após** o componente ser renderizado pela primeira vez, ou seja, acontece somente uma vez durante o ciclo de vida do componente. É um bom local para requisições HTTP únicas, por exemplo.

- **componentDidUpdate**: ocorre **após** toda e qualquer renderização subsequente do componente, ou seja, pode ocorrer diversas vezes durante o ciclo de vida do componente. É um bom local para a implementação de **efeitos colaterais** (manipulação manual do DOM, cálculos com base no objeto state, utilização de biblioteca de terceiros etc.).
- **componentWillUnmount**: ocorre **antes** do componente ser removido do DOM. Ocorre somente uma vez durante o ciclo de vida do componente. É o local adequado para remover eventListeners, utilizar clearInterval etc.

É muito importante assistir às videoaulas para praticar a criação e manutenção de class components, que é feita nos vídeos através de diversos exemplos práticos.

O seguinte [link](#) apresenta mais um exemplo prático de utilização de class components com lifecycle methods.



Projeto com class components e lifecycle methods

Fonte: codesandbox.io.

Capítulo 3. Functional Components

À medida em que uma aplicação React cresce, torna-se necessária a quebra da lógica em diversos componentes para facilitar a manutenção a longo prazo.

Quando esses componentes não manipulam o estado diretamente, é uma boa prática a adoção de **Functional Components**, ou seja, componentes baseados em **função**.

Esse tipo de componente é também considerado como “somente leitura”, pois os dados chegam através de propriedades (mais conhecidas como **props**), que são geralmente mantidas pelo componente pai, que pode então possuir **estado**. Assim, não há manipulação interna de estado em Functional Components (a não ser que seja adotada a estratégia dos **React Hooks**, que será vista no próximo capítulo). As **props** são utilizadas também para a comunicação entre componentes não só com dados, mas também com funções.

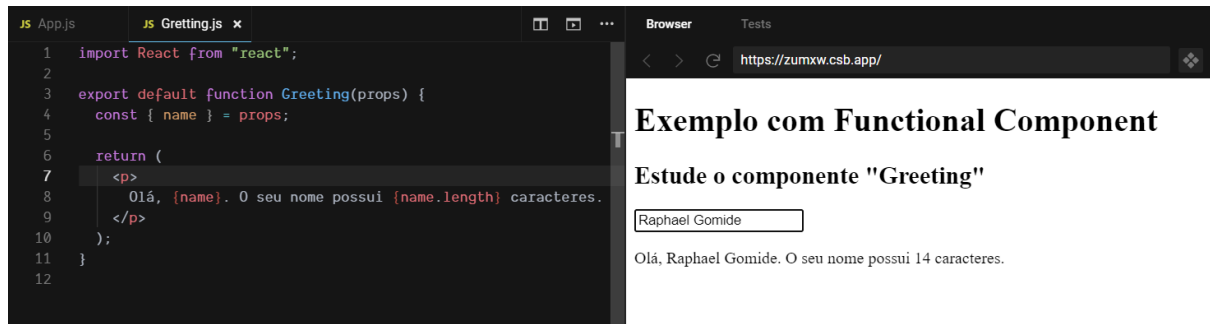
Algumas características importantes dos Functional Components:

- São baseados em **funções** e não mais em classes.
- **Não** possuem **lifecycle methods** por padrão.
- Podem possuir funções internas para abstrair melhor o código. Essas funções internas são mais conhecidas como **closures**, pois conseguem absorver o escopo externo do componente. Mais detalhes serão vistos nos tópicos sobre **React Hooks**.
- Manipulam *props*.
- Retornam **um** elemento JSX.

O seguinte [link](#) possui um projeto com um exemplo de Functional Component. Perceba que, no componente App — que é um class componente — há o objeto de estado com **name**. Esse valor é atualizado a partir do input e é enviado ao componente Greeting através da **prop "name"**. No arquivo **Greeting.js** há a definição

do Functional Component **<Greeting />** (que é invocado em App.js). Esse componente simplesmente lê o valor da **prop** e o utiliza na renderização do componente, adicionando também um cálculo com **name.length**.

Não deixe de assistir às videoaulas, onde há exemplos bem mais completos e complexos, além de exemplos de conversão de Class Components em Functional Components.



Projeto com utilização de Functional Component.

Fonte: codesandbox.io.

Capítulo 4. React Hooks

A funcionalidade dos **React Hooks** visa justamente unir a manipulação de **estado** dos **Class Components** com a **praticidade** dos **Functional Components**. Com isso, a escrita de componentes com **Hooks** torna-se ainda mais declarativa em comparação aos **Class Components**, e ao mesmo tempo possui as mesmas funcionalidades.

Existem diversos *hooks* que podem ser utilizados. Inclusive, o próprio desenvolvedor pode criar os seus *hooks*. Entretanto, para fins de simplicidade e para não fugir muito do escopo do módulo, serão apresentados somente os dois principais Hooks, que são **useState** e **useEffect**.

O hook **useState**

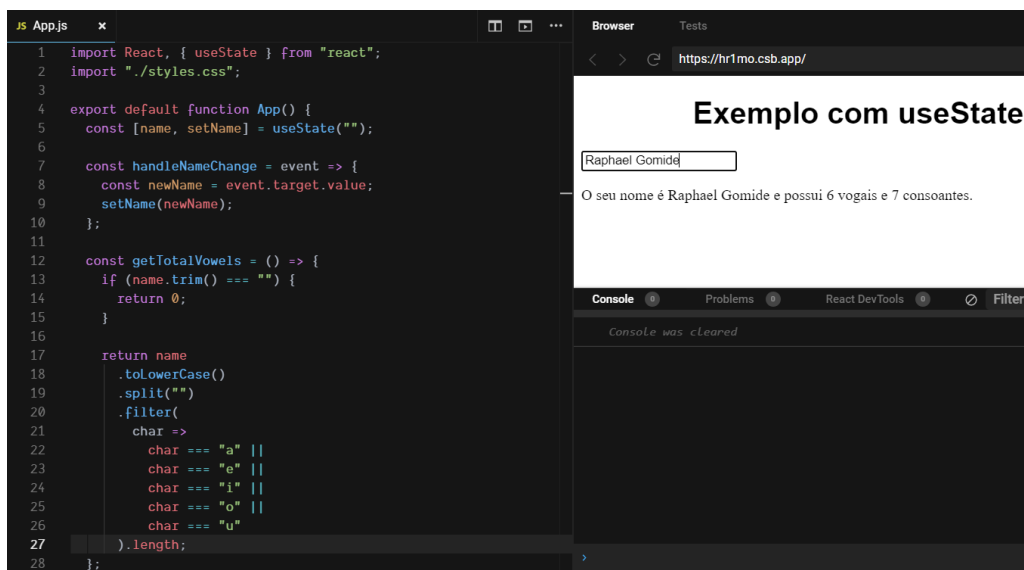
O hook **useState** visa prover as mesmas funcionalidades de **this.state** e **this.setState**, que existem nos **Class Components**. A grande vantagem é uma escrita mais declarativa e menos verbosa. Isso é feito com a utilização de **array destructuring** na declaração, que é uma funcionalidade do ES6+. A sintaxe padrão de uma declaração de **useState** é a seguinte:

```
const [variable, setVariable] = useState(0);
```

Assim, o desenvolvedor define alguma variável que vai ser monitorada pelo React (ex.: `name`), e escreve a linha acima – `const [name, setName] = useState('');`

O hook **useState** retorna por padrão um **array com dois elementos**. O primeiro elemento é o **valor inicial**, que é definido na **declaração** de **useState** e terá o **papel** de **this.state**. O segundo elemento é uma **função atualizadora**, que terá o papel de **this.setState**. Para simplificar a escrita, é muito comum a utilização de **array destructuring**, conforme o código acima. Além disso, **useState** pode ser inicializado com um valor padrão, como nos exemplos acima (0 e `' '`).

O seguinte [link](#) possui um projeto que utiliza useState, que pode ser visualizado na imagem abaixo:



```

1 import React, { useState } from "react";
2 import "./styles.css";
3
4 export default function App() {
5   const [name, setName] = useState("");
6
7   const handleChange = event => {
8     const newName = event.target.value;
9     setName(newName);
10  };
11
12  const getTotalVowels = () => {
13    if (name.trim() === "") {
14      return 0;
15    }
16
17    return name
18      .toLowerCase()
19      .split("")
20      .filter(
21        char =>
22          char === "a" ||
23          char === "e" ||
24          char === "i" ||
25          char === "o" ||
26          char === "u"
27      ).length;
28  };

```

Projeto com utilização do hook useState.

Fonte: codesandbox.io.

O hook useEffect

O hook **useEffect** é bastante poderoso e pode ser utilizado para a inserção/remoção de “efeitos colaterais” e visa também implementar as mesmas funcionalidades dos lifecycle methods **componentDidMount**, **componentDidUpdate** e **componentWillUnmount** dos **Class Components** com uma escrita muito mais declarativa.

Efeitos colaterais podem ser, em regra:

- Manipulação manual do DOM, como por exemplo a modificação de **document.title**.
- Inclusão/eliminação de eventListeners manuais.
- Inclusão/eliminação de intervals.

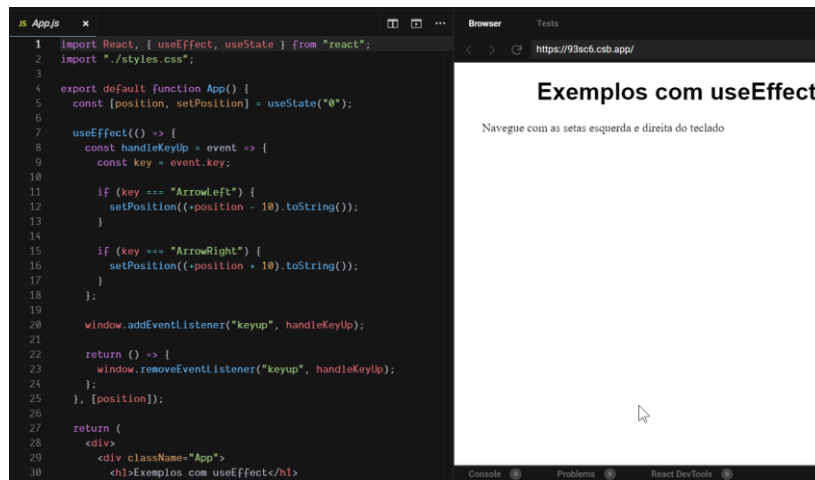
- Alteração de determinada variável de estado após a modificação de uma outra variável de estado.

Com **useEffect** não há mais o conceito de **montagem (mount)** e **atualização** do componente (**update**). O hook **useEffect** tem um modelo mental diferente dos métodos de ciclo de vida – a ideia principal de **useEffect** é **sincronizar** o DOM conforme os valores de **props** e **state**. Para mais informações, acesse [aqui](#).

O hook **useEffect** permite utilizar um parâmetro extra, conhecido como array de dependências (**dependency array** ou, simplesmente, **deps**):

- Quando não há o parâmetro, **useEffect** é invocado **após qualquer atualização** – semelhante ao método **componentDidUpdate** de Class Components.
- Quando o parâmetro é [] (array vazio), **useEffect** é invocado apenas uma vez – semelhante ao método **componentDidMount** de Class Components.
- Quando o parâmetro está preenchido com [state1, state2, prop1, prop2 etc.], **useEffect** é invocado após a atualização de estado de **qualquer uma** das variáveis.
- Quando há retorno na função o **useEffect** utiliza o retorno para eliminar recursos, semelhante ao **componentWillUnmount**.

O [seguinte](#) projeto tem um bom exemplo da utilização de **useEffect**. Verifique a imagem abaixo e não deixe de estudar as videoaulas, onde há exemplos mais complexos criados passo a passo.



```

1 import React, { useEffect, useState } from "react";
2 import "./styles.css";
3
4 export default function App() {
5   const [position, setPosition] = useState("0");
6
7   useEffect(() => {
8     const handleKeyUp = event => {
9       const key = event.key;
10
11       if (key === "ArrowLeft") {
12         setPosition((+position - 10).toString());
13       }
14
15       if (key === "ArrowRight") {
16         setPosition((+position + 10).toString());
17       }
18     };
19
20     window.addEventListener("keyup", handleKeyUp);
21
22     return () => {
23       window.removeEventListener("keyup", handleKeyUp);
24     };
25   }, [position]);
26
27   return (
28     <div>
29       <div className="App">
30         <h1>Exemplos com useEffect</h1>

```

Projeto com utilização do hook useEffect.

Fonte: codesandbox.io.

Referências

FACEBOOK OPEN SOURCE. *Getting Started*. 2020. Disponível em: <<https://reactjs.org/docs/getting-started.html>>. Acesso em: 14 dez. 2020.

MDN WEB DOCS. *Home*. 2020. Disponível em: <<https://developer.mozilla.org/en-US/>>. Acesso em: 14 dez. 2020.