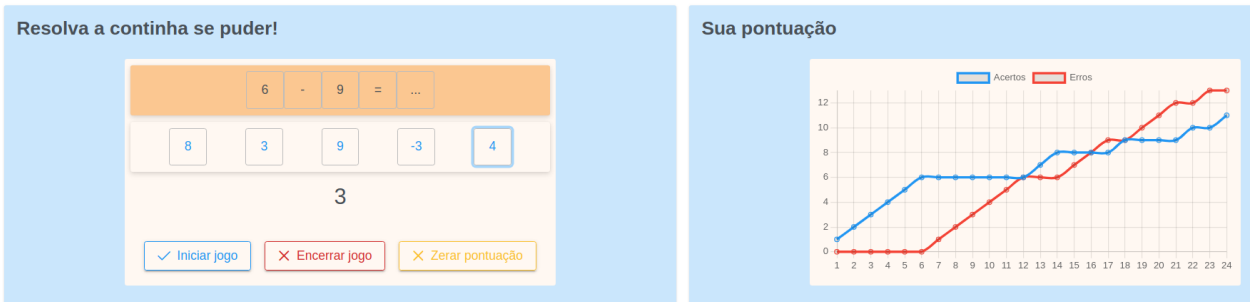
Professor Rodrigo Bossini
https://www.rodrigobossini.com.br

ReactJS
Ciclo de Vida de componentes

Exercícios

1. O exercício consiste no desenvolvimento de um jogo cuja tela principal se assemelha com aquela exibida pela Figura 1.1.

Figura 1.1



A aplicação irá, a cada intervalo de tempo previamente definido, exibir uma conta cujo resultado deverá ser calculado pelo jogador. Além daquilo já exibido pela Figura 1.1, ela também deverá ter botões extras com possíveis respostas para que o usuário possa escolher. Ela mostra, também, um gráfico com os números de acertos e erros ao longo do tempo.

Desafio: Fazer a implementação utilizando a biblioteca **PrimeReact**.

Produção: setembro de 2021

Link 1.1
https://www.primefaces.org/primereact/

O Link 1.2 leva à página da PrimeFlex. Ela traz um grid system e classes utilitárias, muito semelhante ao Bootstrap.

Link 1.2
https://www.primefaces.org/primeflex/

2 Solução

2.1 (Criação da aplicação, execução e instalação das dependências) Comece criando a aplicação com

npx create-react-app nome-do-app

Use

cd nome-do-app

para navegar até o diretório recém-criado e

code .

para abrir uma instância do VS Code vinculada ao diretório atual. Clique **Terminal >> New Terminal** no VS Code para utilizar um terminal embutido dele. Use

npm start

para colocar seu aplicativo em execução.

Abra um novo terminal do VS Code clicando no botão **+**, como mostra a Figura 2.1.1.

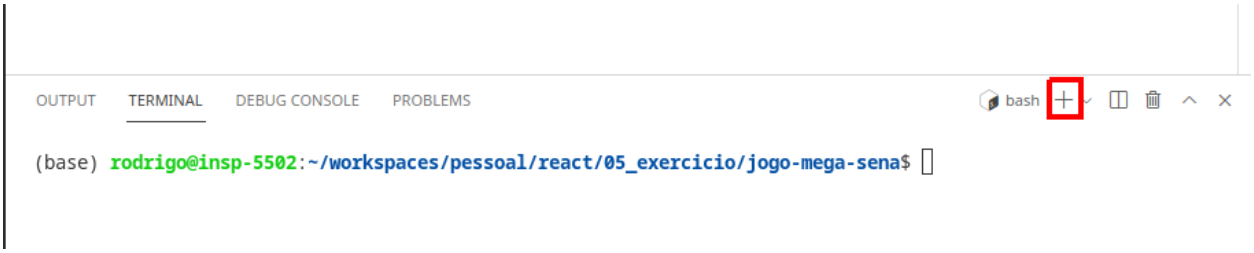


Figura 2.1.1

Apague todos os arquivos existentes na pasta **src**. A seguir, crie um arquivo chamado **index.js** nela. Seu conteúdo inicial aparece no Bloco de Código 2.1.1.

Bloco de Código 2.1.1

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';


export default class App extends Component {

  render() {
    return (
      <div>App</div>
    )
  }
}

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

Segundo a documentação do PrimeReact, que pode ser encontrada no Link 2.1.1, a sua instalação pode ser feita com os comandos a seguir.

Link 2.1.1
https://primefaces.org/primereact/showcase/#/setup

npm install primereact
npm install primeicons
npm install react-transition-group

Feitas as instalações, devemos importar os arquivos CSS desejados. Além dos arquivos referentes à biblioteca primereact e à biblioteca de ícones primeicons, também precisamos importar um tema. Visite a documentação uma vez mais para verificar a vasta lista de temas gratuitos disponíveis e escolher o que preferir. Veja o Bloco de Código 2.1.2.

Bloco de Código 2.1.2

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';
//esse você pode escolher
import 'primereact/resources/themes/saga-blue/theme.css';
import 'primereact/resources/primereact.min.css';
import 'primeicons/primeicons.css';


export default class App extends Component {
...
```

PrimeReact é uma biblioteca de componentes que não inclui um "grid system" para responsividade e classes utilitárias para espaçamento, cores etc. Esses detalhes são resolvidos pela **primeflex**. Sua documentação pode ser encontrada no Link 2.1.2.

Link 2.1.2
https://www.primefaces.org/primeflex/

A sua instalação pode ser feita com

npm install primeflex

Feita a sua instalação, importe seu arquivo css como exibe o Bloco de Código 2.1.3.

Bloco de Código 2.1.3

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';
import 'primereact/resources/themes/saga-blue/theme.css';
import 'primereact/resources/primereact.min.css';
import 'primeicons/primeicons.css';
import 'primeflex/primeflex.css';

export default class App extends Component {
...
```

Os gráficos da PrimeReact têm como dependência a biblioteca **chart.js** Ela pode ser instalada com

npm install chart.js

2.2 (Componente Cartao) Tanto o jogo quanto o gráfico são exibidos como conteúdos de cartões. Por isso, vamos definir um componente Cartao independente. Veja o Bloco de Código 2.2.1. Ele deve ser definido em um novo arquivo chamado **Cartao.js**, que deve ser criado na pasta **src**.

Bloco de Código 2.2.1

```
import React, { Component } from 'react';
import { Card } from 'primereact/card';
export default class Cartao extends Component {
  render() {
    return (
      <Card title={this.props.titulo} style={styles.card}>
        <div className={`${styles.inner} ${this.props.className}`}>{this.props.children}</div>
      </Card>
    );
  }
}

const styles = {
  card: {
    //função css var pega a cor associada à variável especificada
    backgroundColor: 'var(--blue-100)',
  },
  //borda arredondada, background laranja, largura, padding, margin
  inner: 'border-round bg-orange-50 w-8 p-2 m-auto',
};

Cartao.defaultProps = {
    titulo: 'Resolva a continha se puder!'
}
```

2.3 (Componente para exibir mensagem) O componente que exibe a mensagem para iniciar o jogo é um tanto simples. Veja a sua definição no Bloco de Código 2.3.1. Crie um arquivo chamado **Mensagem.js** na pasta **src** para escrevê-la.

Bloco de Código 2.3.1

```
import React, { Component } from 'react'

export default class Mensagem extends Component {
    render() {
        return (
            <div className={`${styles.texto} ${this.props.className}`}>
                {this.props.texto}
            </div>
        )
    }
}
const styles = {
    texto:
    //centraliza nos dois eixos, borda, background vermelho, sombra, altura
      'flex justify-content-center align-items-center border-round bg-red-100 shadow-2 h-4rem'
}
```

2.4 (Exibindo dois cartões e a mensagem com o grid system) O componente principal será responsável por exibir os dois cartões de maneira responsiva. Neste momento, vamos utilizar o grid system do PrimeFlex para fazer a exibição de dois cartões. O primeiro exibirá a mensagem. Veja o Bloco de Código 2.4.1. Estamos no arquivo **index.js**.

Bloco de Código 2.4.1

```
...
import Cartao from './Cartao';
import Mensagem from './Mensagem';
export default class App extends Component {
   render() {
   return (
      //grid conteúdo centralizado horizontalmente
     <div className='grid justify-content-center'>
      {/*12 colunas. 6 para telas grandes  */}
      <div className='col-12 lg:col-6'>
      {/* altura */}
        <Cartao className='h-18rem'>
        {/* garantindo altura para alternar entre mensagem e jogo */}
          <div className='h-12rem'>

            {/* centraliza e pega toda a altura que pode */}
             <div className='flex align-items-center h-full justify-content-center'>
               <Mensagem texto='Clique para iniciar' className='md:w-8 w-10' />
             </div>
          </div>
        </Cartao>
      </div>
      {/* 12 colunas. 6 para telas grandes */}
      <div className='col-12 lg:col-6'>
        <Cartao
          titulo="Sua pontuação"
          // altura igual à do outro
          className='h-18rem'>
        </Cartao>
      </div>
     </div>
   );
  }
}

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

2.5 (Botões) A aplicação exibe três botões: Iniciar Jogo, Encerrar Jogo e Zerar pontuação. Eles serão definidos como um único componente. Clique com o direito em **src** para criar um arquivo cujo nome será **Botoes.js**. O Bloco de Código 2.5.1 mostra a definição do componente.

Bloco de Código 2.5.1

```
import React, { Component } from 'react';
import {Button} from 'primereact/button'

export default class Botoes extends Component {
  render() {
    return (
      // usa a classe que chegou via props e a que ele mesmo define
      <div className={`${this.props.className} ${styles.botoes}`}>
        <div className="flex justify-content-evenly">
          <Button
            label='Iniciar jogo'
            className='p-button-raised  p-button-outlined'
            icon='pi pi-check'
            // função que será enviada via props
            onClick={this.props.fIniciar}
          />
          <Button
            label='Encerrar jogo'
            className='p-button-raised p-button-outlined p-button-danger'
            icon='pi pi-times'
            // função que será enviada via props
            onClick={this.props.fEncerrar}
          />
          <Button
            label='Zerar pontuação'
            className='p-button-raised p-button-outlined p-button-warning'
            icon='pi pi-times'
            // função que será enviada via props
            onClick={this.props.fZerar}
          />
        </div>

      </div>
    );
  }
}
const styles = {
    botoes: 'mt-5'
}
```

Ajuste o arquivo **index.js** para que os botões sejam exibidos, como no Bloco de Código 2.5.2.

Bloco de Código 2.5.2

```
mponent {
   render() {
   return (
      //grid conteúdo centralizado horizontalmente
    <div className='grid justify-content-center'>
      {/*12 colunas. 6 para telas grandes  */}
      <div className='col-12 lg:col-6'>
      {/* altura */}
        <Cartao className='h-18rem'>
        {/* garantindo altura para alternar entre mensagem e jogo */}
          <div className='h-12rem'>

            {/* centraliza e pega toda a altura que pode */}
             <div className='flex align-items-center h-full justify-content-center'>
               <Mensagem texto='Clique para iniciar' className='md:w-8 w-10' />
             </div>
          </div>
        <Botoes />
        </Cartao>
      </div>
      {/* 12 colunas. 6 para telas grandes */}
      <div className='col-12 lg:col-6'>
        <Cartao
          titulo="Sua pontuação"
          // altura igual à do outro
          className='h-18rem'>
        </Cartao>
      </div>
    </div>
   );
  }
}

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

2.6 (O componente Jogo) O componente Jogo será definido em um novo arquivo chamado **Jogo.js** que deve ser criado na pasta **src**. Ele resolve os seguintes problemas:

- Gera contas aleatórias, compostas por dois operandos e uma operação.
- Gera a resposta correta.
- Gera alternativas potencialmente incorretas.
- Faz a exibição do desafio e das alternativas durante cinco segundos, gerando uma nova combinação depois deste tempo.
- Exibe um timer de 5 segundos para cada desafio proposto.

(Código inicial e estado do Jogo) Seu código inicial é dado no Bloco de Código 2.6.1. Repare na definição de seu estado.

Bloco de Código 2.6.1

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

export default class Jogo extends Component {
  constructor(props) {
    super(props);
    this.state = {
      // coleção para armazenar as alternativas
      alternativas: Array(5).fill(undefined),
      // tempo de duração da exibição de cada desafio
      intervaloAtualizacao: 5000,
      //valor inicial do intervalo dentro do qual os valores serão gerados
      valorInicial: 1,
      //valor final do intervalo dentro do qual os valores serão gerados
      valorFinal: 10,
      //variável que controla o tempo que resta para o usuário resolver o desafio exibido
atualmente.
      tempoRestante: 5
    }
  }

  render() {
    return <div>Jogo</div>;
  }
}

//fora da classe
const styles = {};
```

(Timers) O jogo terá dois timers: um deles controla a troca do desafio a ser exibido para o usuário. O outro será para atualizar um contador segundo a segundo. Veja a sua definição no Bloco de Código 262.

Bloco de Código 2.6.2

```
export default class Jogo extends Component {
...

  timerGeral = null
  timerSegundoASegundo = null

…
}
```

(Objeto para armazenar os símbolos e as operações) Os desafios serão continhas de soma e subtração. Para cada desafio gerado, a aplicação precisa exibir o símbolo (+ ou -) apropriado e calcular o resultado, já que ele precisa constar na coleção de alternativas. O componente armazenará objetos JSON compostos pelo símbolo e pela respectiva operação, o que simplificará a geração de expressões aleatórias. Veja o Bloco de Código 2.6.3.

Bloco de Código 2.6.3

```
export default class Jogo extends Component {
 ...
  operacoes = [
    {simbolo: '+', operacao: (a, b) => a + b},
    {simbolo: '-', operacao: (a, b) => a - b},
  ]
…
}
```

(Gerando um desafio) Um desafio é uma conta no formato: **n1 op n2**. Por exemplo:

2 + 3

Assim, a geração de um desafio consiste em

- gerar o primeiro operando no intervalo desejado
- gerar o segundo operando no intervalo desejado
- sortear uma das operações e "guardar" o seu símbolo para futura exibição
- Calcular o resultado

A função do Bloco de Código 2.6.4 se encarrega disso.

Bloco de Código 2.6.4

```
export default class Jogo extends Component {
  ...
  gerarConta = () => {
    let n1 = Math.floor (Math.random() * this.state.valorFinal) + this.state.valorInicial;
    let n2 = Math.floor (Math.random() * this.state.valorFinal) + this.state.valorInicial;
    let oQueFazer = Math.floor(Math.random ()* this.operacoes.length)
    let simbolo = this.operacoes[oQueFazer]['simbolo']
    let resultado = this.operacoes[oQueFazer]['operacao'](n1, n2)
    return {n1, n2, simbolo, resultado}
  }
...
}
```

(Gerando as alternativas) Por padrão, a aplicação exibirá cinco alternativas para cada desafio. Evidentemente, a resposta certa deve estar incluída nas alternativas. A função que gera as alternativas opera da seguinte forma.

- Recebe a resposta certa como parâmetro.
- Constrói uma lista contendo a resposta certa.
- Gera mais quatro valores aleatórios dentro do intervalo de interesse e adiciona à lista, sem permitir valores duplicados.
- Devolve a lista.

A sua implementação aparece no Bloco de Código 2.6.5.

Bloco de Código 2.6.5

```
export default class Jogo extends Component {
...
  gerarAlternativas = (resultado) => {
    let aux = [resultado]
    while (aux.length < 5){
      let n = Math.floor (Math.random() * this.state.valorFinal) + this.state.valorInicial
      if (!aux.includes(n))
        aux.push(n)
    }
    return aux
  }
…
}
```

(Gerando um jogo completo) A geração de um jogo completo consiste em

- Gerar um desafio
- Gerar as alternativas
- Ajustar o estado do componente para que os novos valores sejam exibidos a cada ciclo de renderização.

No momento, a coleção de alternativas mantém a resposta correta sempre na primeira posição. É importante, portanto, gerar uma nova permutação para exibição. Para tal, utilizaremos uma biblioteca que se chama **underscore**. Veja a sua documentação no Link 2.6.1. Estamos interessados no método **shuffle**.

Link 2.6.1
https://underscorejs.org/#shuffle

A sua instalação pode ser feita com

npm install underscore

Feita a instalação, ela pode ser importada com import _ from 'underscore', como no Bloco de Código 2.6.6. Repare que usamos "_" (um underscore!) para acessar as funcionalidades da biblioteca. O nome pode ser qualquer um, no entanto.

Bloco de Código 2.6.6

```
import React, { Component } from 'react';
import _ from 'underscore'
export default class Jogo extends Component {
...
```

O Bloco de Código 2.6.7 gera um jogo como descrito.

Bloco de Código 2.6.7

```
export default class Jogo extends Component {
...
  gerarJogo = () => {
    this.setState({
      tempoRestante: 5
    })
    let {n1, n2, simbolo, resultado} = this.gerarConta()
    let alternativas = this.gerarAlternativas(resultado)
    this.setState({
      n1, n2, simbolo, resultado, alternativas: _.shuffle(alternativas), tempoRestante: 5
    })

  }
...
```

(Começando a exibir o componente no método render) Neste momento já temos condições de exibir o jogo. Começamos exibindo a região em que o desafio aparecerá. Veja o Bloco de Código 2.6.8.

Bloco de Código 2.6.8

```
export default class Jogo extends Component {
...
render() {
    const conta = (
      <div>
        <div className={styles.conta}>
          <div className={styles.valor}>{this.state.n1}</div>
          <div className={styles.valor}>{this.state.simbolo}</div>
          <div className={styles.valor}>{this.state.n2}</div>
          <div className={styles.valor}>=</div>
          <div className={styles.valor}>...</div>
        </div>
      </div>
    )
   return (
      <div>
        {conta}
      </div>
    )
}
}
//fora da classe
const styles = {
  // centraliza, borda, background laranja, sombra, altura
  conta:
    'flex justify-content-center align-items-center border-round bg-orange-200 shadow-2 h-4rem',
   // centraliza borda, altura e largura iguais
  valor:
    'flex justify-content-center align-items-center border-round border-1 border-400  h-3rem w-
3rem',
};
```

Ajuste o arquivo **index.js** como no Bloco de Código 2.6.8 para ver o resultado.

Bloco de Código 2.6.8

```
import Jogo from './Jogo';
export default class App extends Component {
   render() {
   return (
      //grid conteúdo centralizado horizontalmente
     <div className='grid justify-content-center'>
      {/*12 colunas. 6 para telas grandes  */}
      <div className='col-12 lg:col-6'>
      {/* altura */}
        <Cartao className='h-18rem'>
        {/* garantindo altura para alternar entre mensagem e jogo */}
          <div className='h-12rem'>

            {/* centraliza e pega toda a altura que pode */}
              {/* <div className='flex align-items-center h-full justify-content-center'>
                <Mensagem texto='Clique para iniciar' className='md:w-8 w-10' />
              </div> */}
              <Jogo/>
          </div>
          <Botoes />
        </Cartao>
      </div>
      {/* 12 colunas. 6 para telas grandes */}
      <div className='col-12 lg:col-6'>
        <Cartao
          titulo="Sua pontuação"
          // altura igual à do outro
          className='h-18rem'>
        </Cartao>
      </div>
     </div>
   );
  }
}
```

A seguir, ajustamos o arquivo **Jogo.js** para que as alternativas e o tempo restante também sejam exibidos. Veja o Bloco de Código 2.6.9.

Bloco de Código 2.6.9

```
export default class Jogo extends Component {
…
render() {
    const conta = (
      <div>
        <div className={styles.conta}>
          <div className={styles.valor}>{this.state.n1}</div>
          <div className={styles.valor}>{this.state.simbolo}</div>
          <div className={styles.valor}>{this.state.n2}</div>
          <div className={styles.valor}>=</div>
          <div className={styles.valor}>...</div>
        </div>
        </div>
    )
    const alternativas = (
        <div className={styles.alternativas}>
          {this.state.alternativas.map((alternativa, indice) => (
            <Button
              key={indice}
              className={`${styles.valor} ${styles.alternativa}`} label={alternativa?.toString()}
            />
          ))}
        </div>
    )
    const tempoRestante = (

        <div className={styles.tempoRestante}>
          {this.state.tempoRestante}
        </div>
    )

    return (
      <div>
        {conta}
        {alternativas}
        {tempoRestante}
      </div>
    )
  }
}
```

```
//fora da classe
const styles = {
  // centraliza, borda, background laranja, sombra, altura
  conta:
    'flex justify-content-center align-items-center border-round bg-orange-200 shadow-2 h-4rem',
    // centraliza, espaçamento uniforme, borda, sombra, altura margin top
  alternativas:
    'flex justify-content-evenly align-items-center border-round shadow-2 h-4rem mt-2',
    // centraliza borda, altura e largura iguais
  valor:
    'flex justify-content-center align-items-center border-round border-1 border-400  h-3rem w-
3rem',
    // outline no botão
  alternativa:
  'p-button-outlined',
  // centraliza, altura, tamanho da fonte
  tempoRestante:
    'flex justify-content-center align-items-center h-4rem text-3xl'
};
```

O resultado esperado aparece na Figura 2.6.1.

Figura 2.6.1

(Iniciando uma rodada) O algoritmo para iniciar uma nova rodada é o seguinte.

- Encerrar a execução dos dois timers
- Agendar a geração de novos jogos a cada cinco segundos, fazendo uma geração imediata
- Agendar a atualização do contador segundo a segundo

A função do Bloco de Código 2.6.10 implementa este algoritmo.

Bloco de Código 2.6.10

```
export default class Jogo extends Component {
  ...
  iniciarRodada = () => {
    // encerramos a execução dos dois timers
    clearInterval(this.timerGeral)
    clearInterval(this.timerSegundoASegundo)
    //uma função para registrar o timer das rodadas
    //ela dispara a geração do jogo imediatamente e agenda as demais
    //o timer segundo a segundo também é iniciado aqui
    let fire = (f, t) => {
      f()
      this.timerSegundoASegundo = setInterval(() => {
        this.setState({tempoRestante: this.state.tempoRestante - 1})
      },1000);
      return setInterval(f, t)
    }
    this.timerGeral = fire(this.gerarJogo, this.state.intervaloAtualizacao)
  }
...
```

A função **iniciarRodada** pode ser chamada no método **componentDidMount**, como destaca o Bloco de Código 2.6.11.

Bloco de Código 2.6.11

```
export default class Jogo extends Component {
  ...
componentDidMount(){
      this.iniciarRodada()
}
...
```

Teste o aplicativo uma vez mais. O timer segundo a segundo deve ser exibido, bem como o desafio e as alternativas.

(Encerrando o jogo: método componentWillUnmount) Quando o usuário clicar no botão para encerrar o jogo, a ideia é que o componente Jogo seja removido da DOM e que o componente Mensagem tome o seu lugar. Por isso, precisamos tomar o cuidado de encerrar a execução dos timers quando isso acontecer. Veja o Bloco de Código 2.6.12.

Bloco de Código 2.6.12

```
export default class Jogo extends Component {
…
  encerrar = () => {
    clearInterval(this.timerGeral)
    clearInterval(this.timerSegundoASegundo)
  }


 componentWillUnmount(){
   this.encerrar()
 }
…
}
```

2.7 (Ajustes no componente App e no componente Jogo) O componente principal é o responsável por decidir o que renderizar, em função das interações do usuário. Além disso, ele precisa manter informações sobre os acertos e erros ocorridos no jogo para alimentar o gráfico.

(O estado do componente App) A definição do estado do componente App aparece no Bloco de Código 2.7.1.

Bloco de Código 2.7.1

```
export default class App extends Component {
 state = {
   // jogo iniciado ou não?
   status: 'off',
   //número de acertos
   acertos: 0,
   //número de erros
   erros: 0,
   //número de tentativas
   contador: 0
 }
…
}
```

(Funções para alterar status e pontuação) O componente principal precisa que seu estado seja atualizado conforme o usuário interage com a aplicação. Como as interações são detectadas pelo componente Jogo, o componente principal define funções que alteram seu estado da maneira desejada e as entrega ao Jogo via props. Definiremos três funções:

- Uma função para atualizar o status do jogo (de on para off e vice-versa)
- Uma função para atualizar a pontuação (incrementar número de acertos ou erros e incrementar o contador)
- Uma função para zerar a pontuação

Veja o Bloco de Código 2.7.2.

Bloco de Código 2.7.2

```
export default class App extends Component {
...
  alterarStatus = (status) => {
    this.setState({ status });
  };
  atualizarPontuacao = (acertou) => {
    this.setState(
      acertou
        ? { acertos: this.state.acertos + 1, contador: this.state.contador + 1}
        : { erros: this.state.erros + 1, contador: this.state.contador + 1}
    );
  };
  zerarPontuacao = () => {
    this.setState({
      acertos: 0,
      erros: 0,
    });
  };
…
}
```

O componente principal as entrega ao componente Botoes via props, como no Bloco de Código 2.7.3.

Bloco de Còdigo 2.7.3

```
export default class App extends Component {
…
render() {
…
<Botoes
            fIniciar={() => this.alterarStatus('on')}
            fEncerrar={() => this.alterarStatus('off')}
            fZerar={() => this.zerarPontuacao()}
          />
...
```

A seguir, o componente passa a exibir Mensagem e Botoes de maneira mutuamente exclusiva. A exibição deles depende do status do jogo. Veja o Bloco de Código 2.7.4.

Bloco de Código 2.7.4

```
export default class App extends Component {
…
render() {
   return (
      //grid conteúdo centralizado horizontalmente
    <div className='grid justify-content-center'>
      {/*12 colunas. 6 para telas grandes  */}
      <div className='col-12 lg:col-6'>
      {/* altura */}
       <Cartao className='h-18rem'>
       {/* garantindo altura para alternar entre mensagem e jogo */}
        <div className='h-12rem'>
          {this.state.status === 'on' ? (
            <Jogo
              status={this.state.status}
              fAtualizarPontuacao={this.atualizarPontuacao}
            />
          ) : (
            <div className='flex align-items-center h-full justify-content-center'>
              <Mensagem texto='Clique para iniciar' className='w-6' />
            </div>
          )}
        </div>
        <Botoes
          fIniciar={() => this.alterarStatus('on')}
          fEncerrar={() => this.alterarStatus('off')}
          fZerar={() => this.zerarPontuacao()}
        />
       </Cartao>
      </div>
      {/* 12 colunas. 6 para telas grandes */}
      <div className='col-12 lg:col-6'>
       <Cartao
        titulo="Sua pontuação"
        // altura igual à do outro
        className='h-18rem'>
       </Cartao>
      </div>
    </div>
   );
```

Perceba que o componente Jogo recebe o status do jogo via props. Ele o utilizará em seu método **componentDidMount** de modo a iniciar uma rodada somente quando o status do jogo for igual a 'on'. Veja o Bloco de Código 2.7.5.

Bloco de Código 2.7.5

```
export default class Jogo extends Component {
…
componentDidMount(){
   //passado pelo componente principal
   if (this.props.status === 'on')
     this.iniciarRodada()
  }
...
```

Os botões que exibem as alternativas ainda não tem o tratamento de clique realizado. Quando um clique acontece neles, a pontuação do jogo precisa ser atualizada: se o valor existente no botão clicado for igual ao resultado armazenado no estado do componente, o usuário tem mais um acerto. Caso contrário, ele tem um erro. Além disso, iniciamos uma nova rodada sempre que o usuário clica para que ele tenha somente uma chance a cada rodada. Veja o Bloco de Código 2.7.6.

Bloco de Código 2.7.6

```
export default class Jogo extends Component {
…
render() {
…
   const alternativas = (
      <div className={styles.alternativas}>
        {this.state.alternativas.map((alternativa, indice) => (
          <Button
            key={indice}
            className={`${styles.valor} ${styles.alternativa}`} label={alternativa?.toString()}
            onClick={() => {
              this.iniciarRodada()
              this.props.fAtualizarPontuacao(this.state.resultado === alternativa)}
            }
          />
        ))}
      </div>
   )
  }
}
```

Para ter certeza de que tudo está funcionando, exiba os números de acerto e erro textualmente, como no Bloco de Código 2.7.7.

Bloco de Código 2.7.6

```
export default class App extends Component {
…
 render() {
   …
   <Botoes
            fIniciar={() => this.alterarStatus('on')}
            fEncerrar={() => this.alterarStatus('off')}
            fZerar={() => this.zerarPontuacao()}
          />
          {`Acertos: ${this.state.acertos}`}
          {`Erros: ${this.state.erros}`}
        </Cartao>
      </div>
   …
  }
}
```

Teste novamente o app clicando nos três botões e acertando e errando tentativas. O resultado esperado é parecido com o que exibe a Figura 2.7.1.

Figura 2.7.1

2.8 (Componente para exibição de um gráfico) O segundo cartão da aplicação exibe um gráfico de linha. Duas linhas são exibidas: uma delas mostra o número de acertos ao longo do tempo e a outra, o número de erros. Utilizaremos o componente Chart da PrimeReact para construir o gráfico. Veja a sua documentação no Link 2.8.1.

Link 2.8.1
https://www.primefaces.org/primereact/showcase/#/linechart

(Criação do componente e seu estado) Começamos criando um arquivo chamado **GraficoLinha.js** na pasta **src**. Seu código inicial aparece no Bloco de Código 2.8.1. Seu estado armazena duas informações: uma delas se refere ao grau de suavização da curva dos gráficos. A outra indica se a área sob as linhas do gráfico deve ser preenchida.

Bloco de Código 2.8.1

```
import React, { Component } from 'react';
import { Chart } from 'primereact/chart';
export default class GraficoLinha extends Component {
  constructor(props) {
    super(props);
    this.state = {
      //suavizar as curvas
      tension: 0.4,
      //preencher a área sob o gráfico?
      fill: false,
    };
  }
}
```

(Dados que caracterizam cada curva) Cada curva do gráfico será caracterizada por

- coleção de valores
- titulo
- cor

O Bloco de Código 2.8.2 mostra a definição de um objeto que armazena essas informações.

Bloco de Código 2.8.2

```
export default class GraficoLinha extends Component {
…
 colecoes = {
   acertos: {
     titulo: 'Acertos',
     dados: [],
     cor: '#2196F3',
   },
   erros: {
     titulo: 'Erros',
     dados: [],
     cor: '#F44336',
   },
  };
```

(E o eixo x?) Precisamos também de uma coleção de valores para representar o eixo x. Essa coleção é simplesmente 0, 1, 2, 3, 4..., pois representa o número de tentativas do usuário ao longo do tempo. Em função de cada valor desse, cada linha do gráfico mostra o número de acertos e o número de erros. Declaramos, portanto, a lista exibida pelo Bloco de Código 2.8.3.

Bloco de Código 2.8.3

```
export default class GraficoLinha extends Component {
  ...
  contador = [];
}
```

(A função de atualização das coleções do componente GraficoLinha) A cada interação do usuário, os componentes são atualizados, incluindo o gráfico. O componente principal entregará a ele, via props, dados referentes a:

- acertos
- erros
- número de tentativas
- e um booleano indicando se o gráfico precisa ser reiniciado, o que acontece quando ambos os números de acertos e erros são iguais a 0.

O componente GraficoLinha define uma função que atualiza as suas coleções de acordo com esses valores. Veja o Bloco de Código 2.8.4.

Bloco de Código 2.8.4

```
export default class GraficoLinha extends Component {
...
atualizarDados = () => {
   this.colecoes = {
     acertos: {
       titulo: 'Acertos',
       dados: this.props.zerar
         ? []
         : [...this.colecoes.acertos.dados, this.props.acertos],
       cor: '#2196F3',
     },
     erros: {
       titulo: 'Erros',
       dados: this.props.zerar
         ? []
         : [...this.colecoes.erros.dados, this.props.erros],
       cor: '#F44336',
     },
     tension: 0.4,
     fill: false,
   };
   this.contador = this.props.zerar ? [] : [...this.contador, this.props.contador];
 };
...
```

(A função render do componente GraficoLinha) A função render atualiza as coleções e devolve um objeto do tipo **Chart**, da PrimeReact. Veja a sua documentação para mais detalhes sobre as opções utilizadas. Muitas delas são definidas na documentação da Chart.js, que pode ser encontrada no Link 2.8.1. Os componentes da PrimeReact são apenas "wrappers" que simplificam o uso da Chartjs.

Link 2.8.1
https://www.chartjs.org/

A implementação da função render aparece no Bloco de Código 2.8.5.

Bloco de Código 2.8.5

```
export default class GraficoLinha extends Component {
...
render() {
   this.atualizarDados();
   return (
     <Chart
       options={{
         animation: {
           duration: 0,
         },
         scales: {
           y: {
             ticks: {
               stepSize: 1,
             },
           },
         },
       }}
       type='line'
       data={{
         labels: this.contador,
         datasets: [
           {
             label: this.colecoes.acertos.titulo,
             data: this.colecoes.acertos.dados,
             fill: this.state.fill,
             borderColor: this.colecoes.acertos.cor,
             tension: this.state.tension,
           },
           {
             label: this.colecoes.erros.titulo,
             data: this.colecoes.erros.dados,
             fill: this.state.fill,
             borderColor: this.colecoes.erros.cor,
             tension: this.state.tension,
           },
         ],
       }}
     />
   );
 }
…
}
```

(Utilizando o componente GraficoLinha no componente principal) O componente principal da aplicação passa a utilizar o componente GraficoLinha em seu segundo cartão. Veja o Bloco de Código 2.8.6. Ajuste também o componente para que ele deixe de exibir os números de acertos e erros textualmente.

Bloco de Código 2.8.6

```
export default class App extends Component {
render() {
   return (
      //grid conteúdo centralizado horizontalmente
     <div className='grid justify-content-center'>
      {/*12 colunas. 6 para telas grandes  */}
      <div className='col-12 lg:col-6'>
      {/* altura */}
        <Cartao className='h-18rem'>
        {/* garantindo altura para alternar entre mensagem e jogo */}
          <div className='h-12rem'>
            {this.state.status === 'on' ? (
               <Jogo
                 status={this.state.status}
                 fAtualizarPontuacao={this.atualizarPontuacao}
               />
             ) : (
               <div className='flex align-items-center h-full justify-content-center'>
                 <Mensagem texto='Clique para iniciar' className='w-6' />
               </div>
             )}
          </div>
          <Botoes
            fIniciar={() => this.alterarStatus('on')}
            fEncerrar={() => this.alterarStatus('off')}
            fZerar={() => this.zerarPontuacao()}
          />
           {/* comentar esse trecho */}
          {/* {`Acertos: ${this.state.acertos}`}
          {`Erros: ${this.state.erros}`} */}
        </Cartao>
      </div>
      {/* 12 colunas. 6 para telas grandes */}
      <div className='col-12 lg:col-6'>
        <Cartao
```

```
        titulo="Sua pontuação"
        // altura igual à do outro
        className='h-18rem'>
        <GraficoLinha
            acertos={this.state.acertos}
            erros={this.state.erros}
            contador={this.state.contador}
            zerar={(!this.state.acertos && !this.state.erros)}
            />
      </Cartao>
    </div>
  </div>
);
```

(Herdando de PureComponent) Note que, conforme o usuário clica nos botões, o gráfico gera um novo ponto, mesmo quando o jogo está parado. Isso ocorre pois seu ciclo de renderização dispara mesmo quando não há valores novos sendo entregues via props. Podemos ajustar isso fazendo com que a sua classe deixe de herdar de Component e passe a herdar de PureComponent. Veja o Bloco de Código 2.8.7.

Bloco de Código 2.8.7

```
import React, { PureComponent } from 'react';
import { Chart } from 'primereact/chart';
export default class GraficoLinha extends PureComponent {
...
```

Interaja novamente com a aplicação e veja os resultados.

2.9 (Implantando a aplicação no Github Pages) O Github Pages permite que aplicações compostas por arquivos estáticos (HTML, CSS e Javascript) sejam implantadas. Cada usuário Github tem direito a uma página referente a seu perfil e cada um de seus repositórios tem direito a uma página também. Neste seção, veremos como implantar a aplicação desenvolvida no Github Pages.

- Instale o pacote **gh-pages** com

npm install gh-pages --save-dev

- Adicione a propriedade **homepage** a seu arquivo **package.json**. Ela deve conter o link da sua página. O padrão é

http://usuario.github.io/nome-app

Exemplo:

https://professorbossini.github.io/pessoal_react_jogo_continhas/

A seguir, criamos os scripts **predeploy** e **deploy**, ambos no arquivo **package.json**. O resultado esperado é parecido com aquele que exibe o Bloco de Código 2.9.1.

Bloco de Código 2.9.1

```
{
 "homepage": "http://professorbossini.github.io/pessoal_react_jogo_continhas",
 "name": "jogo-continhas-de-matematica",
 "version": "0.1.0",
 "private": true,
 "dependencies": {
  "@testing-library/jest-dom": "^5.14.1",
  "@testing-library/react": "^11.2.7",
  "@testing-library/user-event": "^12.8.3",
  "chart.js": "^3.5.1",
  "primeflex": "^3.0.1",
  "primeicons": "^4.1.0",
  "primereact": "^6.5.1",
  "react": "^17.0.2",
  "react-dom": "^17.0.2",
  "react-scripts": "4.0.3",
  "react-transition-group": "^4.4.2",
  "underscore": "^1.13.1",
  "web-vitals": "^1.1.2"
 },
 "scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "eject": "react-scripts eject",
  "predeploy": "npm run build",
  "deploy": "gh-pages -d build"
 },
```

```
  "eslintConfig": {
    "extends": [
      "react-app",
      "react-app/jest"
    ]
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  },
  "devDependencies": {
    "gh-pages": "^3.2.3"
  }
}
```

Nota. A opção **-d** do pacote gh-pages vem de “dist”. Ela está sendo utilizada para indicar qual é o diretório em que se encontram os arquivos gerados no processo de build. Inspecione a estrutura da sua aplicação e verifique que há, de fato, um diretório chamado **build** na raiz. Se desejar, execute

npx gh-pages --help

para obter mais informações sobre esta ferramenta. Visite, também, a sua documentação. Ela pode ser encontrada por meio do Link 2.9.1.

Link 2.9.1
https://www.npmjs.com/package/gh-pages

A seguir, execute

Professor Rodrigo Bossini
https://www.rodrigobossini.com.br

npm run deploy

Produção: setembro de 2021

Referências

React – A JavaScript library for building user interfaces. 2021. Disponível em <https://reactjs.org/>. Acesso em agosto de 2021.