Objetivo:

- Criar um projeto React;
- Estrutura do arquivo index.html, index.js e componente root;
- Expressões JSX;
- Objeto state do componente;
- Objeto props;
- Objeto context;
- Redux;
- Ciclo de vida do componente;
- HTTP Request.

Instruções: Antes de começar você precisa ter instalado o Node.js e NPM (Node Package Manager). Para criar a estrutura do projeto React usaremos a ferramenta create-react-app, ela é recomendada para criar o frontend no formato de SPA (Single-Page Application, isto é, uma única página como o Gmail e Facebook).

Utilize o comando a seguir para verificar se você possui a ferramenta create-react-app instalada:

```
create-react-app -V
```

Caso não tenha ela instalada, então utilize o comando a seguir para instalar:

```
npm install -g create-react-app
```

O npx é uma ferramenta de pacote disponível no npm 5.2+. Verifique se você tem ela instalada:

```
npx -v
```

Para mais detalhes acesse https://create-react-app.dev/docs/getting-started/.

i. Criar um projeto React

No prompt do CMD acesse a pasta que você deseja criar o projeto e digite o comando a seguir:

```
npx create-react-app exemplo1react
```

O comando criará a pasta exemplo1react e colocar nela os arquivos de uma aplicação React, assim como é mostrado a seguir.

Observação:

- A ferramenta create-react-app utiliza Babel e webpack (https://webpack.js.org/concepts/). Babel é um transpiler usado principalmente para converter o código ECMAScript 2015+ (ES6 é a 6ª versão da ECMAScript) em uma versão compatível com versões anteriores do JavaScript dos navegadores (https://babeljs.io/docs/en/). Para mais detalhes https://code-trotter.com/web/understand-the-different-javascript-modules-formats/.

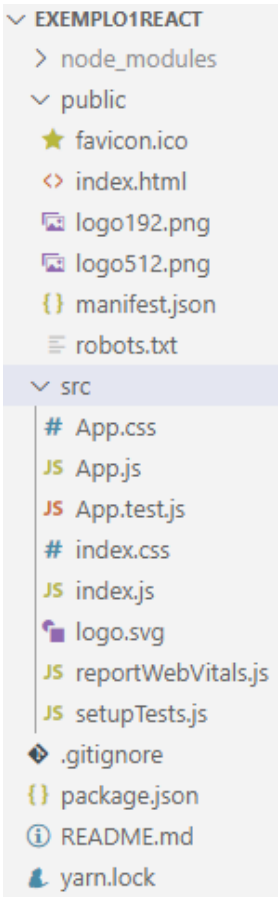
Para subir o servidor acesse a pasta exemplo1react no CMD e digite o comando a seguir:

`npm start` ou `yarn start`

Normalmente o servidor subirá na porta 3000, isto é, http://localhost:3000.

Arquivos do projeto React:

- node_modules: arquivos do Node.js e as dependências do projeto;
- public: arquivos estáticos do projeto. O React renderiza os componentes no arquivo index.html;
- src: estão os arquivos vinculados aos componentes que o React renderizará no arquivo index.html.



ii. Estrutura do arquivo index.html, index.js e componente root

a) O arquivo public/index.html precisa ter a estrutura de um documento HTML e uma marcação com o identificador root, pois será nessa marcação que o React renderizará o componente App (App é o componente root da aplicação). Substitua o código do seu arquivo index.html pelo código mostrado a seguir, pois o objetivo é ter o mínimo de código necessário.

```
<!DOCTYPE html>
<html lang="pt-BR">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <title>Exemplo React</title>
  </head>
  <body>
    <div id="root"></div>
  </body>
</html>
```

b) No arquivo src/index.js está a chamada do método render. Ele recebe como 1º parâmetro o componente a ser renderizado e como 2º parâmetro a marcação do arquivo public/index.html que receberá o conteúdo do 1º parâmetro. Como exemplo, substitua o código do arquivo src/index.js pelo mostrado a seguir. Observe que será colocado `<div>oi</div>` na marcação `<div id="root"></div>`:

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

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

Observação:

- O parâmetro `<div>oi</div>` é um texto e deveria estar entre aspas, porém o código do arquivo src/index.js é processado pelo JSX – mecanismo que converte marcações HTML em elementos React;
- JSX (JavaScript XML) é uma extensão da linguagem JavaScript, baseada na ES6, que traduz HTML em JavaScript em tempo de execução.

Substitua o código do seu arquivo src/index.js pelo código mostrado a seguir, pois o objetivo é ter o mínimo de código necessário.

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

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

c) No arquivo src/index.js a marcação `<App />` é usada para indicar que o componente App será renderizado na `<div id="root"></div>` do arquivo public/index.html.

O componente App está no arquivo src/App.js, substitua o código desse arquivo pelo código a seguir.

```
import React from 'react';

function App() {
  return <div>Bom dia</div>;
}

export default App;
```

```
import React from 'react';

class App extends React.Component {
  render(){
    return <div>Boa tarde</div>;
  }
}

export default App;
```

Componentes React são como funções que retornam HTML, veja que a função `App()` retorna código HTML. Para o arquivo JavaScript ser um componente React ele precisa importar a classe React (`import React from 'react'`) e exportar a função (`export default App`).

Um componente React pode ser escrito usando a notação de função ou classe, assim como está no exemplo anterior. Na notação de classe é necessário estender a classe Component e implementar o método render, pois será o método render que retornará o código HTML.

Observação: Pode excluir os seguintes arquivos:

- Na pasta public: logo192.png e logo512.png;
- Na pasta src: App.test.js, serviceWorker.js, setupTests.js, reportWebVitals.js e logo.svg.

Remover as referências para as imagens logo192.png e logo512.png no arquivo public/manifest.json. Ao final o arquivo terá o seguinte conteúdo:

```
{
  "short_name": "React App",
  "name": "Create React App Sample",
  "icons": [
    {
      "src": "favicon.ico",
      "sizes": "64x64 32x32 24x24 16x16",
      "type": "image/x-icon"
    }
  ],
  "start_url": ".",
  "display": "standalone",
  "theme_color": "#000000",
  "background_color": "#ffffff"
}
```

iii. Expressões JSX

É necessário usar chaves para colocar expressão JavaScript dentro de marcação HTML:

```
let idade = 21;
const marcacao = <div>A sua idade é {idade + 1}</div>;
```

Duas ou mais marcações precisam estar envolvidos por uma marcação pai, por este motivo as marcações a seguir apresentam erro de parsing no JSX:

```
const marcacoes = <div>marcação</div><div>marcação</div>;
```

O correto é envolver elas por uma marcação pai:

```
const marcacoes = <div><div>marcação</div><div>marcação</div></div>;
```

Quando as marcações são escritas em mais de uma linha recomenda-se colocar elas entre parênteses:

```
const marcacoes = (
    <div>
      <div>1marcação</div>
      <div>2marcação</div>
    </div>
  );
```

Todas as marcações precisam ser fechadas, então a marcação a seguir está incorreta:

```
const entrada = <input type='text'>;
```

O correto é

```
const entrada = <input type='text' />;
```

iv. Objeto state do componente

O React fornece os objetos state, props e context.

O objeto state é usado para manter propriedades que podem ser acessadas em todo o componente. Para adicionar propriedades no state usamos a função useState. A função useState recebe como parâmetro de entrada o valor inicial da

propriedade, no exemplo a seguir as propriedades nome e idade foram inicializadas com os valores Ana e 20, respectivamente.

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

export default function App() {
  const [nome, setNome] = useState('Ana');
  const [idade, setIdade] = useState(20);

  return <div>A {nome} possui {idade} anos</div>;
}
```

Observações:

- Uma propriedade do estado não deve ser modificada diretamente, por exemplo, nome = "Ana Maria". Para modificar uma propriedade do estado usamos a função devolvida na definição do estado, por exemplo, setNome("Ana Maria");
- A cada modificação no estado o componente é renderizado novamente;
- O objeto state estará disponível apenas dentro do próprio componente. Podemos passar um estado para outro componente utilizando o objeto props (veremos isso posteriormente);
- As chamadas para modificar o state são assíncronas (https://pt-br.reactjs.org/docs/faq-state.html#why-doesnt-react-update-thisstate-synchronously).

No exemplo a seguir o valor inicial será colocado na variável nome e usaremos a função setNome associada ao evento onChange do elemento HTML input para alterar a propriedade nome do estado.
Como a função só pode retornar um elemento então envolvemos as marcações numa marcação <React.Fragment>. Porém usamos a sintaxe curta, ou seja, na notação de tags vazias (https://pt-br.reactjs.org/docs/fragments.html#short-syntax).

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

export default function App() {
  let [nome, setNome] = useState("Ana");
  const [idade, setIdade] = useState(20);

  return (
    <>
      <div>
        <label>Nome</label>
        <input value={nome} onChange={e => setNome(e.target.value)} />
      </div>
      <div>
        A {nome} possui {idade} anos
      </div>
    </>
  );
```

```
}
```

v. Objeto props

O React fornece os objetos state, props e context.

props (abreviação de ”properties”) e state são ambos objetos JavaScript. Apesar de ambos guardarem informações que influenciam no resultado da renderização, eles são diferentes por uma razão importante: props são passados para o componente (como parâmetros de funções), enquanto state é gerenciado de dentro do componente (como variáveis declaradas dentro de uma função).

Como exemplo, crie o arquivo src/Lista.js. Nele colocaremos o componente Lista que será chamado a partir do componente App, ou seja, o componente Lista será filho do componente App.

A seguir tem-se a interface da aplicação a ser codificada como exemplo. Os campos de entrada estão no componente App e a lista está no componente Lista. Além disso, podemos excluir um item clicando com o botão direito do mouse.

Nome Lúcia
Idade 19
A Lúcia possui 19 anos
Cadastrar

- Ana - 21
- Pedro - 17
- Lúcia - 19

Para implementar essa aplicação temos de colocar o array numa propriedade do state (aqui chamada de dados) e passar a variável dados como propriedade para o componente Lista em `<Lista dados={dados} remove={remove} />`. Observe que passamos também a função remove como propriedade.

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

export default function App() {
  const [nome, setNome] = useState("Ana");
  const [idade, setIdade] = useState(20);
  const [dados, setDados] = useState([]);

  const handleSubmit = e => {
    e.preventDefault(); // evita o evento reload da página
    // cria uma cópia do array e faz um append com o objeto criado
    setDados([...dados,{nome,idade}]);
  }

  const remove = (e,index) => {
    e.preventDefault(); // evita o evento popupmenu
    const temp = [...dados]; //cria uma cópia do array
    temp.splice(index, 1);  //remove o elemento que está na posição index
```

```
    setDados(temp); //atualiza a propriedade dados
  }

  return (
    <>
      <form onSubmit={handleSubmit}>
      <div>
        <label>Nome</label>
        <input value={nome} onChange={e => setNome(e.target.value)} />
      </div>
      <div>
        <label>Idade</label>
        <input value={idade} onChange={e => setIdade(e.target.value)} />
      </div>
      <div>
        A {nome} possui {idade} anos
      </div>
      <div>
        <button>Cadastrar</button>
      </div>
      </form>
      <Lista dados={dados} remove={remove} />
    </>
  );
}
```

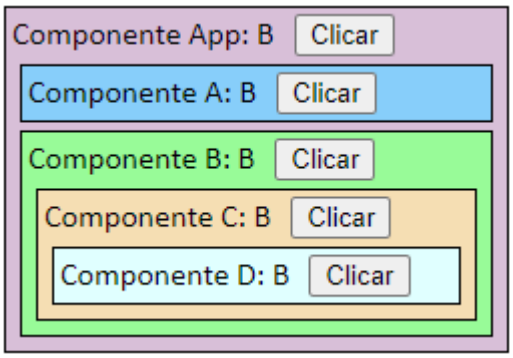
As propriedades são recebidas como parâmetro no objeto props do componente filho.

Observação: um componente filho só é capaz de acessar membros do componente pai que estão no objeto props. Como exemplo, dentro do componente Lista não é possível acessar o método handleSubmit do componente App.

```
export default function Lista(props) {
  return (
    <ul>
      {props.dados.map((item, index) => (
        <li key={index} onContextMenu={e => props.remove(e,index)}>
          {item.nome} - {item.idade}
        </li>
      ))}
    </ul>
  );
}
```

vi. Objeto context

Considere como exemplo a aplicação ao lado. Nela os componentes A e B estão dentro do componente App, o componente C está dentro do componente B e o componente D está dentro do componente C. Ao clicar em um botão todas as mensagens são atualizadas. Para fazer a comunicação fluir por toda a árvore de componentes precisaríamos definir um estado no componente App e passar ele via props para os componentes filhos e esses por sua vez teriam de passar sucessivamente as propriedades para seus filhos.



Uma solução simples é definir um contexto (context) e compartilhar dados entre todos os componentes da mesma árvore de componentes sem precisar passar explicitamente props entre cada nível (https://pt-br.reactjs.org/docs/context.html).

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 prop através de elementos intermediários.

Contexto é usado principalmente quando algum dado precisa ser acessado por muitos componentes em diferentes níveis. Use contexto moderadamente uma vez que isto pode dificultar a reutilização de componentes.

Para reproduzir o exemplo crie os arquivos mostrados na estrutura ao lado.

No arquivo src/Conexto.js definimos o contexto através da função createContext do React. Esse contexto precisa ser importado por cada um dos componentes da árvore que desejam usar o objeto contexto.

Código do arquivo src/Contexto.js:

```
import { createContext } from "react";

const Clicado = createContext();

export default Clicado;
```

No componente App colocamos dentro da marcação `<Clicado.Provider value={{ botao, setBotao }}> </Clicado.Provider>` os componentes que poderão usar o contexto. As propriedades do objeto Context são passados pela propriedade `value` da marcação `<Clicado.Provider>`.

Observe que os valores da propriedade Context são, na prática, propriedades do objeto state do componente App. Desta forma, quando um componente aninhado usar a função setBotao, na prática ele estará chamando a função setBotao do componente App.

Código do arquivo src/App.js:

```
import { useState } from "react";
import A from "./A";
import B from "./B";
import Clicado from "./Contexto";
import "./estilos.css";

export default function App() {
  const [botao, setBotao] = useState("App");

  return (
    <Clicado.Provider value={{ botao, setBotao }}>
      <div className="app">
        <span>Componente App: {botao}</span>
        <button onClick={() => setBotao("App")}>Clicar</button>
        <A />
        <B />
      </div>
    </Clicado.Provider>
  );
}
```

Código do arquivo src/A.js:

```
import { useContext } from "react";
import Clicado from "./Contexto";
import "./estilos.css";

export default function A() {
  const { botao, setBotao } = useContext(Clicado);

  return (
    <div className="aa">
      <span>Componente A: {botao}</span>
      <button onClick={() => setBotao("A")}>Clicar</button>
    </div>
  );
}
```

Código do arquivo src/B.js:

```
import { useContext } from "react";
import Clicado from "./Contexto";
import C from "./C";
import "./estilos.css";

export default function B() {
  const { botao, setBotao } = useContext(Clicado);

  return (
```

```
    <div className="bb">
      <span>Componente B: {botao}</span>
      <button onClick={() => setBotao("B")}>Clicar</button>
      <C />
    </div>
  );
}
```

Código do arquivo src/C.js:

```
import { useContext } from "react";
import Clicado from "./Contexto";
import D from "./D";
import "./estilos.css";

export default function C() {
  const { botao, setBotao } = useContext(Clicado);

  return (
    <div className="cc">
      <span>Componente C: {botao}</span>
      <button onClick={() => setBotao("C")}>Clicar</button>
      <D />
    </div>
  );
}
```

Código do arquivo src/D.js:

```
import { useContext } from "react";
import Clicado from "./Contexto";
import "./estilos.css";

export default function D() {
  const { botao, setBotao } = useContext(Clicado);

  return (
    <div className="dd">
      <span>Componente D: {botao}</span>
      <button onClick={() => setBotao("D")}>Clicar</button>
    </div>
  );
}
```

Código do arquivo src/estilos.css:

```
body {
  font-family: calibri;
}
```

```
button {
  margin-left: 10px;
}
.app {
  background-color: Thistle;
  border: 1px solid black;
  padding: 3px;
  margin: 4px;
}
.aa {
  background-color: LightSkyBlue;
  border: 1px solid black;
  padding: 3px;
  margin: 4px;
}
.bb {
  background-color: PaleGreen;
  border: 1px solid black;
  padding: 3px;
  margin: 4px;
}
.cc {
  background-color: Wheat;
  border: 1px solid black;
  padding: 3px;
  margin: 4px;
}
.dd {
  background-color: LightCyan;
  border: 1px solid black;
  padding: 3px;
  margin: 4px;
}
```

vii. Redux

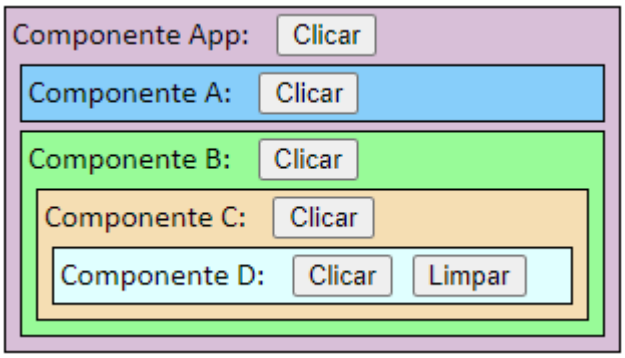
O Redux é uma biblioteca de gerenciamento do objeto state. O seu uso é aconselhável em situações específicas:

- Existem propriedades do state que precisam ser usadas em diferentes componentes do aplicativo. O Redux é uma alternativa ao objeto Context;
- O estado é atualizado com frequência;
- A lógica para atualizar o estado pode ser complexa;
- O aplicativo tem uma base de código de médio ou grande porte e pode ser trabalhado por muitas pessoas;
- É necessário saber como o estado está sendo atualizado ao longo do tempo.

Para mais detalhes acesse https://redux.js.org/faq/general#when-should-i-use-redux.

Para mais detalhes de como usar o Redux acesse https://redux.js.org/introduction/getting-started#redux-toolkit-example.

Como exemplo iremos substituir o objeto Context da aplicação do tópico anterior para utilizar Redux e obter o mesmo resultado. Ao lado tem-se a interface da aplicação.



Utilize o comando a seguir para adicionar o pacote Redux Tookit no nosso projeto:

```
npm i @reduxjs/toolkit
```

Utilize o comando a seguir para adicionar o pacote React Redux no nosso projeto (https://www.npmjs.com/package/react-redux):

```
npm i react-redux
```

Para reproduzir o exemplo crie os arquivos mostrados na estrutura ao lado e utilize os códigos mostrados a seguir.

No arquivo src/store.js definimos a store através da função configureStore do Redux Tookit. A função configureStore recebe como parâmetro um JSON com a propriedade reducer. O reducer disponível em origemSlice.reducer define o estado e a lógica de atualização do estado.

Código do arquivo src/store.js:

```
import { configureStore } from "@reduxjs/toolkit";
import { origemSlice } from "./origemSlice";

export default configureStore({
  reducer: origemSlice.reducer,
});
```

A função createSlice cria o estado na propriedade initialState e define a lógica de atualização na propriedade reducers. Neste exemplo, o estado possui apenas a propriedade value e as funções setOrigem e clear para modificar o estado.
A função createSlice recebe um estado inicial, um objeto com as funções reducers e um “slice name” (https://redux-toolkit.js.org/api/createslice)

Código do arquivo src/origemSlice.js:

```
import { createSlice } from "@reduxjs/toolkit";

export const origemSlice = createSlice({
  name: "origem",
  initialState: {
```

```
    value: "",
  },
  reducers: {
    setOrigem: (state, action) => {
      state.value = action.payload;
    },
    clear: (state) => {
      state.value = "";
    },
  },
});

export const { setOrigem, clear } = origemSlice.actions;
```

O componente <Provider> disponibiliza o armazenamento Redux para qualquer componente aninhado que precise acessar o armazenamento Redux (https://react-redux.js.org/api/provider).

Código do arquivo src/index.js:

```
import React from "react";
import ReactDOM from "react-dom";
import App from "./App";

import { Provider } from "react-redux";
import store from "./store";

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById("root")
);
```

A única forma de mudar o estado é criando uma ação (action – um objeto que descreve o que aconteceu) e despachá-lo (dispatch). No código a seguir a função dispatch chama a função setOrigem do reducer, o parâmetro “App” estará disponível na propriedade action.payload do reducer.

Código do arquivo src/App.js:

```
import A from "./A";
import B from "./B";
import "./estilos.css";
import { useSelector, useDispatch } from "react-redux";
import { setOrigem } from "./origemSlice";

export default function App() {
  const origem = useSelector((state) => state.value);
  const dispatch = useDispatch();

  return (
```

```
    <div className="app">
      <span>Componente App: {origem}</span>
      <button onClick={() => dispatch(setOrigem("App"))}>Clicar</button>
      <A />
      <B />
    </div>
  );
}
```

Código do arquivo src/A.js:

```
import "./estilos.css";
import { useSelector, useDispatch } from "react-redux";
import { setOrigem } from "./origemSlice";

export default function A() {
  const origem = useSelector((state) => state.value);
  const dispatch = useDispatch();

  return (
    <div className="aa">
      <span>Componente A: {origem}</span>
      <button onClick={() => dispatch(setOrigem("A"))}>Clicar</button>
    </div>
  );
}
```

Código do arquivo src/B.js:

```
import C from "./C";
import "./estilos.css";
import { useSelector, useDispatch } from "react-redux";
import { setOrigem } from "./origemSlice";

export default function B() {
  const origem = useSelector((state) => state.value);
  const dispatch = useDispatch();

  return (
    <div className="bb">
      <span>Componente B: {origem}</span>
      <button onClick={() => dispatch(setOrigem("B"))}>Clicar</button>
      <C />
    </div>
  );
}
```

Código do arquivo src/C.js:

```
import D from "./D";
import "./estilos.css";
import { useSelector, useDispatch } from "react-redux";
import { setOrigem } from "./origemSlice";

export default function C() {
  const origem = useSelector((state) => state.value);
  const dispatch = useDispatch();

  return (
    <div className="cc">
      <span>Componente C: {origem}</span>
      <button onClick={() => dispatch(setOrigem("C"))}>Clicar</button>
      <D />
    </div>
  );
}
```

Código do arquivo src/D.js:

```
import "./estilos.css";
import { useSelector, useDispatch } from "react-redux";
import { setOrigem, clear } from "./origemSlice";

export default function D() {
  const origem = useSelector((state) => state.value);
  const dispatch = useDispatch();

  return (
    <div className="dd">
      <span>Componente D: {origem}</span>
      <button onClick={() => dispatch(setOrigem("D"))}>Clicar</button>
      <button onClick={() => dispatch(clear())}>Limpar</button>
    </div>
  );
}
```

viii. Ciclo de vida do componente

Cada componente React possui um ciclo de vida que podemos acessar através de operações (funções). O ciclo de vida está vinculado as fases de montagem (mounting – criar o componente), atualização (updating – alterações nas propriedades dos objetos state ou props) e desmontagem (unmounting – excluir o componente) (https://pt-br.reactjs.org/docs/react-component.html).

A fase mouting

A montagem significa colocar os elementos no DOM (Document Object Model). Como a página é um objeto no navegador e esse objeto é chamado de document no JavaScript do navegador, então os elementos HTML são transformados em objetos JavaScript para serem colocados no objeto document do navegador.

O React possui os seguintes métodos que são chamadas na seguinte ordem durante a montagem do componente:

1. constructor() é chamado quando o componente é criado. Nele colocamos o código para setar o objeto state e outras propriedades. O construtor é chamado somente uma vez na criação do componente;
2. static getDerivedStateFromProps() é chamado após o construtor e antes do método render. Nele colocamos o código para alterar as propriedades do objeto state usando o objeto props. Ele recebe como parâmetro os objetos props e state, e retorna, obrigatoriamente, propriedades para o objeto state. O método getDerivedStateFromProps é chamado a cada alteração em propriedades do state. Esse método é estático;
3. render() é usado para renderizar o HTML do componente no DOM. O método render é chamado a cada alteração em alguma propriedade do state;
4. componentDidMount() é chamado após o render, ou seja, ele é invocado após o componente estar renderizado no navegador. Como exemplo, nesse método colocamos o código para fazer uma requisição no servidor e carregar o resultado numa propriedade do state. Ao fazermos assim, o componente é exibido para o usuário e, posteriormente, os dados são carregados. O método componentDidMout é chamado somente uma vez por componente.

A fase updating

O componente é atualizado quando ocorre a modificação de alguma propriedade dos objetos state ou props. O React possui os seguintes métodos que são chamados na seguinte ordem durante a atualização do componente:

1. getDerivedStateFromProps()
2. shouldComponentUpdate() retorna true/false. Ao retornar false ele impede que os objetos state e props do componente não sejam atualizados;
3. render()
4. getSnapshotBeforeUpdate() é chamado antes das atualizações nos objetos state ou props. Para incluir o método getSnapshotBeforeUpdate é necessário também incluir o método componentDidUpdate;
5. componentDidUpdate() é chamado após as atualizações nos objetos state ou props.

A fase unmounting

O componente é removido do DOM. O React possui o seguinte método que é chamado quando o componente é removido:

1. componentWillUnmount() é chamado imediatamente antes do componente ser removido.

No React os componentes podem ser criados usando classes (componente classe) e funções (componentes funcionais). Ao longo desta aula estamos estudando apenas React Hooks, isto é, componentes criados usando funções. Desta forma, os métodos do ciclo de vida do componente não estarão disponíveis diretamente.

Hooks: são funções que permitem fazer a ligação com os recursos do state e ciclo de vida do React a partir de componentes funcionais (componentes criados usando funções). Vale lembrar que o state é uma propriedade da classe e o ciclo de vida está nos métodos da classe (componentDidMount, componentDidUpdate e componentWillUnmount), isto é, esses membros não podem ser simplesmente colocados no corpo de uma função. Observações:

1. O Hooks permite que você use React sem classes;
2. As funcionalidades codificadas no construtor e nos métodos do ciclo de vida do "componente classe" podem ser codificadas de forma equivalente no Hooks usando funções Hooks, assim como useState e useEffect:
    - construtor: é usado para iniciar as propriedades do estado no componente classe. No Hooks a função useState é usada para criar estados;

- render: é usado para retornar as marcações do componente. No Hooks o render é o corpo da própria função retornado pela instrução return;
- componentDidMount, componentDidUpdate e componentWillUnmount: são os métodos chamados respectivamente após o componente ser renderizado, após o estado (state) sofrer alteração, e ao finalizar o componente. No Hook essas operações podem ser implementadas usando a função de efeito colateral useEffect.

3. Hooks não funcionam dentro de classes, ou seja, não podemos usar as funções useState e useEffect no corpo de uma classe;
4. Não podemos usar Hooks dentro de um componente classe, mas podemos misturar classes e funções com Hooks na árvore do componente.

Para mais detalhes https://pt-br.reactjs.org/docs/hooks-intro.html.

Exemplo usando React Hooks: nesse exemplo fazemos uso da função de efeito colateral useEffect para ter acesso aos eventos mounting e unmounting do ciclo de vida do componente e update nas propriedades do state.

O 1º parâmetro da useEffect é a função a ser executada cada vez que o state sofrer qualquer alteração. Porém, ao passar um array vazio como 2º parâmetro, então ela será chamada somente ao criar o componente. Por exemplo:

```
useEffect(() => console.log("App - mounting"), []);
```

Para a função useEffect ser chamada ao destruir o componente ela precisa retornar uma função. Por exemplo:

```
useEffect(() => {
  return () => console.log("Lista - unmounting");
}, []);
```

A função useEffect pode receber como 2º parâmetro os estados que ela irá monitorar. No exemplo a seguir ela será disparada somente quando a propriedade nomes sofrer alteração:

```
useEffect(() => console.log("nomes - update"), [nomes]);
```

Porém, se não fornecermos o 2º parâmetro, então o efeito será disparado ao mudar qualquer propriedade do objeto state. Por exemplo:

```
useEffect( () => console.log("Update state") );
```

Código do arquivo src/App.js:

```
import { useEffect, useState } from "react";
import Lista from "./Lista";

export default function App() {
  const [nome, setNome] = useState("");
  const [nomes, setNomes] = useState([]);

  useEffect(() => console.log("App - mounting"), []);
  useEffect(() => console.log("nome - update"), [nome]);
  useEffect(() => console.log("nomes - update"), [nomes]);

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

```
    setNomes([...nomes, nome]);
  };

  const remove = (e, index) => {
    e.preventDefault();
    const temp = [...nomes];
    temp.splice(index, 1);
    setNomes(temp);
  };

  return (
    <>
      <form onSubmit={handleSubmit}>
        <label>Nome</label>
        <input value={nome} onChange={(e) => setNome(e.target.value)} />
        <button>Cadastrar</button>
      </form>
      {nomes.length > 0 && <Lista nomes={nomes} remove={remove} />}
    </>
  );
}
```

Código do arquivo src/Lista.js:

```
import { useEffect } from "react";

export default function Lista(props) {
  useEffect(() => console.log("Lista - mounting"), []);
  useEffect(() => {
    return () => console.log("Lista - unmounting");
  }, []);
  return (
    <ul>
      {props.nomes.map((item, index) => (
        <li key={index} onContextMenu={(e) => props.remove(e, index)}>
          {item}
        </li>
      ))}
    </ul>
  );
}
```

ix. HTTP Request

A API Fetch fornece uma interface JavaScript para acessar e manipular requisições e respostas HTTP. A API fornece o método global fetch() para fazer requisições assíncronas na rede, ou seja, o método retorna uma promise. Para mais detalhes acesse https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch.

O exemplo a seguir faz uso do método fetch para fazer uma requisição no webservice da ViaCEP (http://viacep.com.br). O webservice retorna em diferentes formatos, mas aqui utilizamos apenas o retorno em JSON, por exemplo, a URL https://viacep.com.br/ws/12247014/json/ retornará

```
{
  "cep": "12247-014",
  "logradouro": "Avenida Cesare Mansueto Giulio Lattes",
  "complemento": "",
  "bairro": "Eugênio de Mello",
  "localidade": "São José dos Campos",
  "uf": "SP",
  "ibge": "3549904",
  "gia": "6452",
  "ddd": "12",
  "siafi": "7099"
}
```

A URL https://viacep.com.br/ws/12247099/json/ retornará:

```
{
  "erro": true
}
```

Código do arquivo src/App.js:

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

export default function App() {
  const [cep, setCep] = useState("");
  const [resposta, setResposta] = useState("");

  const handleSubmit = (e) => {
    e.preventDefault();
    const url = `https://viacep.com.br/ws/${cep}/json/`;
    fetch(url)
      .then((response) => response.json()) //converte a resposta para JSON
      .then((json) => {
        const saida = [];
        for (let key in json) {
          saida.push(
            <div key={key}>
              {key}: {json[key]}
            </div>
          );
        }
        setResposta(saida);
      });
  };

  return (
    <>
      <form onSubmit={handleSubmit}>
        <label>Nome</label>
        <input value={cep} onChange={(e) => setCep(e.target.value)} />
        <button>Buscar</button>
```

```
      </form>
      <div>{resposta}</div>
    </>
  );
}
```

O pacote axios (https://www.npmjs.com/package/axios) também é usado para processar requisições HTTP, e ele também retorna um objeto Promise. O pacote axios possui vantagens sobre a fetch e uma delas e converter automaticamente a resposta para JSON. Para mais detalhes sobre a diferença entre fetch e axios acesse https://medium.com/trainingcenter/axios-ou-fetch-765e5db9dd59 e https://stackoverflow.com/questions/40844297/what-is-difference-between-axios-and-fetch.

Para fazer uso do pacote axios é necessário adicionar ele no projeto:

```
npm i axios
```

Assim como o exemplo anterior, o código a seguir faz uma requisição no webservice da ViaCEP, mas aqui foi utilizado axios.

Código do arquivo src/App.js:

```
import axios from "axios";
import { useState } from "react";

export default function App() {
  const [cep, setCep] = useState("");
  const [resposta, setResposta] = useState("");

  const handleSubmit = (e) => {
    e.preventDefault();
    const url = `https://viacep.com.br/ws/${cep}/json/`;
    axios
      .get(url)
      .then(({ data }) => {
        //o objeto JSON está na propriedade data da resposta
        const saida = [];
        for (let key in data) {
          saida.push(
            <div key={key}>
              {key}: {data[key]}
            </div>
          );
        }
        setResposta(saida);
      });
  };

  return (
    <>
```

```
      <form onSubmit={handleSubmit}>
        <label>Nome</label>
        <input value={cep} onChange={(e) => setCep(e.target.value)} />
        <button>Buscar</button>
      </form>
      <div>{resposta}</div>
    </>
  );
}
```