Este treinamento é uma compilação do conteúdo apresentado no site do react, focando nos principais conceitos e seus exemplos, em alguns trechos modifiquei um pouco as explicações para facilitar o entendimento delas e dos exemplos de código apresentados, realizei a inclusão de links e mais alguns tópicos que achei que poderiam melhorar a compreensão do conteúdo.

Autor: Gleryston Matos
GitHub: GlerystonMatos
Medium: Gleryston Matos
Linkedin: Gleryston Matos

Repositório do Treinamento: TreinamentoReact

Pré requisitos

É interessante ter alguns conhecimentos para ter um melhor entendimento dos assuntos abordados no treinamento.

01 - Conhecimento básico em HTML, CSS e JavaScript.

Durante o treinamento vou explicar um pouco sobre as tags HTML que forem sendo utilizadas, porém existe muito conteúdo disponível de forma gratuita na internet que pode ser consultado. Caso alguém não tenha um bom conhecimento, recomendo estudar um pouco o assunto.

HTML Tutorial

JavaScript tutorial

Em relação ao JavaScript, caso não sinta segurança sobre seus conhecimentos, a documentação do react recomenda um guia que pode ser acessado neste link.

02 - Conhecer funções, objetos, matrizes e classes.

Tendo conhecimento em outras linguagens de programação estes conceitos não serão problema.

03 - Alguns pontos do JavaScript merecem um pouco mais de atenção para não termos problemas durante as explicações.

Arrow functions

Use strict

Escopo de variáveis

Funções callback

Bind

Introdução

React é uma biblioteca JavaScript usada para construção de interfaces de usuários de forma fácil.

O react permite criar interfaces complexas a partir de pequenos e isolados trechos de código que chamamos de componentes.

O react foi pensado desde o início para ser adotado de forma gradual, podendo ser usado de acordo com a necessidade dos desenvolvedores.

Podemos adicionar interatividade com react em uma página HTML simples ou criar toda uma aplicação react complexa.

Neste treinamento vou tentar fazer um misto de teoria e prática criando exemplos para os conceitos que forem sendo apresentados.

Na documentação do react existe uma lista de cursos recomendados, sendo alguns deles gratuitos.

Eu pessoalmente recomendo a leitura da documentação, pois aqui estamos focando nos pontos mais importantes, então para se aprofundar nos assuntos que vamos discutir acredito ser uma boa opção.

Recomendo também o curso de react + firebase + bootstrap da udemy.

Preparando o ambiente

Existem editores online que podem ser utilizados para realizar os exemplos que serão criados no treinamento, alguns deles são:

01 - CodePen (Recomendado)
02 - CodeSandbox
03 - Glitch
04 - Stackblitz

Se preferir pode usar seu próprio editor de texto em seu ambiente local para desenvolver os exemplos, pessoalmente prefiro montar meu ambiente local.

Para isso vamos precisar primeiramente realizar a instalação da versão mais atual do Node.js

Não utilizaremos o Node.js diretamente, vamos usar algumas ferramentas instaladas juntamente com ele mais a frente no treinamento.

Após realizar a instalação do Node.js, podemos utilizar um editor como Sublime Text ou Visual Studio Code.

No meu caso irei usar o Visual Studio Code.

Agora vamos configurar o realce da sintaxe do editor, utilizando uma ferramenta chamada Babel.

Visual Studio Code

A seguir uma lista de atalhos úteis para se trabalhar no Visual Studio Code e alguns complementos que vão ajudar no desenvolvimento.

Recomendo a instalação imediata dos complementos, para podermos prosseguir.

Os atalhos são:

Inserir um comentário: SHIFT + ALT + A
Inserir um comentário: CTRL + ;
Identar o código: SHIFT + ALT + F
Seleção múltipla de linhas: CTRL + SHIFT + ALT + (↑ ou → ou ↓ ou ←)
Configurações: CTRL + SHIFT + P
Mover linha: ALT + para cima | para baixo

Os complementos são:

Babel JavaScript - Michael McDermott
Bookmarks - Alessandro Fragnani
Code Runner - Jun Han
Color Highlight - Sergii Naumov
CSS Formatter - Martin Aeschlimann
CSS Modules - clinyong
Debugger for Chrome - Microsoft
Docker - Microsoft
HTML Format - Mohamed Akram
Material Icon Theme - Philipp Kief
Pascal - Alessandro Fragnani (Opcional)
Pascal Formatter - Alessandro Fragnani (Opcional)
Rainbow Brackets - 2gua
React Native Snippet - Jundat95
React Native Tools - Microsoft
React-Native/React/Redux snippets for es6/es7 - EQuimper
Tabnine - TabNine
vscode-styled-components - Julien Poissonnier
XML Tools - Josh Jphnson

Adicionando o react a um site

Podemos usar o react em pequenas partes da nossa aplicação ou site, com poucas linhas de código e nenhuma ferramenta de build.

Primeiramente vamos adicionar o react a uma página HTML simples.

Vamos criar um arquivo chamado index.html com a seguinte estrutura.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Adicionando o react a um site</title>
  </head>
  <body>
  </body>
</html>
```

Em seguida vamos criar uma tag div vazia para poder exibir nosso componente react.

```
<div id="like_button_container"></div>
```

Para isso vamos adicionar um atributo HTML id único, para poder identificar a div no código JavaScript e poder exibir o componente dentro dela.

Podemos ter vários contêineres como esse dentro do corpo da nossa página.

Eles geralmente são vazios pois o react vai substituir todo e qualquer conteúdo existente dentro deles.

Agora vamos adicionar o react a nossa página, utilizando as seguintes tags.

```
<script src="https://unpkg.com/react@16/umd/react.development.js" crossorigin></script>
<script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js" crossorigin></script>
```

Essas tags são apenas para realização de testes e exemplos, porque antes de realizar o deploy para produção de um site com react temos que lembrar que o código JavaScript não minificado pode deixar a página significativamente mais lenta para os usuários.

Por este motivo, caso fossemos publicar nosso site teríamos de trocar o arquivo development.js por production.min.js porque o mesmo está minificado, ou seja, está otimizado para ser usado em produção.

Existem vários sites e ferramentas na internet que podem ser usados para minificar seus códigos JavaScript, um exemplo pode ser visto neste link.

Agora vamos adicionar o nosso componente a página.

```
<script src="like_button.js"></script>
```

O resultado final será o seguinte.

```
<!DOCTYPE html>
```

```html
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Adicionando o react a um site</title>
  </head>
  <body>
    <div id="like_button_container"></div>

    <script src="https://unpkg.com/react@16/umd/react.development.js" crossorigin></script>
    <script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js" crossorigin></script>

    <script src="like_button.js"></script>
  </body>
</html>
```

Agora vamos criar um arquivo chamado like_button.js, no mesmo diretório do nosso arquivo index.html.

Depois de criar o arquivo vamos adicionar o seguinte código ao arquivo: like_button.js
O código copiado é de um componente react chamado LikeButton, não se preocupem em entender o código copiado, no momento quero apenas mostrar a inclusão de um componente react em uma página HTML, futuramente vou explicar a composição de um componente react em seus mínimos detalhes.

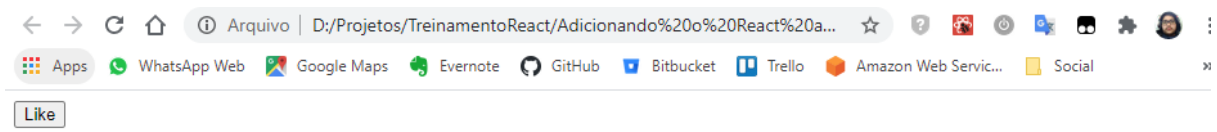
No momento vou comentar apenas sobre as duas últimas linhas do arquivo.

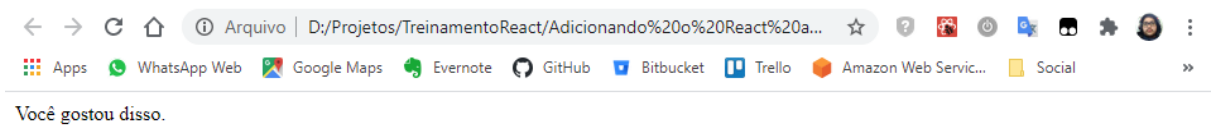
```
const domContainer = document.querySelector('#like_button_container');
ReactDOM.render(e(LikeButton), domContainer);
```

Nestas linhas o código localiza a div que foi adicionada na página HTML no passo anterior, e em seguida mostra o componente react dentro dela.

Após salvar tudo podemos abrir a página HTML no browser.

Será exibido um botão like, quando clicarmos nele será exibido um texto informando: Você gostou disso.





Acabamos de adicionar o nosso primeiro componente react a um site.

O código deste exemplo pode ser conferido aqui.

Reutilizando um componente

Podemos usar o mesmo componente em vários pontos da nossa página HTML.

Alterei o exemplo anterior exibindo o botão “Like” três vezes e passando algumas informações para ele através de atributos nas divs.

```
    <p>
      <div class="like_button_container" data-commentid="1"></div>
    </p>
    <p>
      <div class="like_button_container" data-commentid="2"></div>
    </p>
    <p>
      <div class="like_button_container" data-commentid="3"></div>
    </p>
```

Realizei algumas alterações no componente LikeButton exibindo qual o botão do comentário que foi clicado.

```
'use strict';

const e = React.createElement;

class LikeButton extends React.Component {
  constructor(props) {
    super(props);
    this.state = { liked: false };
  }

  render() {
    if (this.state.liked) {
      return 'Você gostou do número do comentário ' + this.props.commentID;
    }

    return e(
      'button',
      { onClick: () => this.setState({ liked: true }) },
      'Like'
    );
  }
}

document.querySelectorAll('.like_button_container')
```

```
.forEach(domContainer => {
  const commentID = parseInt(domContainer.dataset.commentid, 10);
  ReactDOM.render(
    e(LikeButton, { commentID: commentID }),
    domContainer
  );
});
```

O código do exemplo pode ser conferido aqui.

Criando um novo react app

Um toolchains é um conjunto de ferramentas de programação usadas para executar tarefas complexas.

Por este motivo existem toolchains para criar toda a estrutura de um projeto, para otimizar esse processo.

Existem vários toolchains recomendados pela documentação do react, entre eles eu escolhi usar o Create ceact app para criar o nosso projeto.

É possível criar o seu próprio toolchain do zero, se alguém tiver interesse existe um guia que explica o processo neste link.

O create react app é um ambiente confortável para aprender react, é a melhor maneira de começar um single-page application em react segundo a documentação.

Além de configurar o ambiente de desenvolvimento para utilizar as funcionalidades mais recentes do JavaScript, ele fornece uma experiência de desenvolvimento agradável, e otimiza o aplicativo para ser usado em produção.

Os únicos requisitos para utilização do mesmo são, ter instalados uma versão maior ou igual a 8.10 do Node.js e 5.6 do npm.

Para criar um novo projeto vamos usar o seguinte comando.

```
glery@GLERYSTON-PC MINGW64 /d/Projetos/TreinamentoReact/Criando um novo React App (master)
$ npx create-react-app my-app
```

O npx é um package runner que vem com o npm 5.2 ou superior.

Após a execução do comando, vai ser criado um projeto chamado my-app no diretório onde estamos atualmente no terminal, este processo pode demorar um pouco.

Após a criação do projeto podemos acessar a pasta usando o comando cd my-app e em seguida executar o projeto usando o comando npm start.

```
glery@GLERYSTON-PC MINGW64 /d/Projetos/TreinamentoReact/Criando um novo React App (master)
$ cd my-app/

glery@GLERYSTON-PC MINGW64 /d/Projetos/TreinamentoReact/Criando um novo React App/my-app (master)
$ npm start
```

O create react app não lida com back end nem banco de dados, ele apenas cria um projeto para frontend, podendo portanto usar qualquer backend da nossa escolha.

Por trás dos panos ele usa o Babel e Webpack, mas não é necessário saber nada sobre eles.

```
Compiled successfully!

You can now view my-app in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.11.9:3000

Note that the development build is not optimized.
To create a production build, use npm run build.
```

Depois de iniciar a aplicação podemos ver o resultado no browser, acessando os links informados no terminal.

Depois de finalizar nossa aplicação e fomos mandar para produção, podemos usar o comando npm run build para criar e mandar o build otimizado do aplicativo para a pasta build.

```
glery@GLERYSTON-PC MINGW64 /d/Projetos/TreinamentoReact/Criando um novo React App/my-app (master)
$ npm run build
```

Se vocês quiserem saber mais sobre o create react app, podem verificar o README ou o guia do usuário.

O código do exemplo pode ser conferido aqui.

Introduzindo ao JSX

O JSX é uma extensão de sintaxe para JavaScript, que lembra uma linguagem de template, mas que vem com todo o poder do JavaScript. É recomendado pela documentação do react a utilização dela para descrever como a interface do usuário deve ser.

O react não requer o uso do JSX. Porém, a maioria das pessoas acha prático como uma ajuda visual, quando estamos trabalhando com uma interface dentro do código em JavaScript.

Ele permite ao react mostrar mensagens mais úteis de erro e aviso.

Este é um exemplo usando apenas JavaScript.

```
const e = React.createElement;

// Exibe um "Gostar" <button>
return e(
  'button',
  { onClick: () => this.setState({ liked: true }) },
  'Gostar'
);
```

Agora o mesmo exemplo usando JSX.

```
// Exibe um "Gostar" <button>
return (
  <button onClick={() => this.setState({ liked: true })}>
    Gostar
  </button>
);
```

Podemos incluir qualquer expressão JavaScript válida dentro das chaves em JSX.
Por exemplo, 2 + 2, user.firstName, ou formatName(user) são todas expressões JavaScript válidas.

No próximo exemplo, vamos incorporar o resultado da chamada de uma função JavaScript, formatName(user), dentro de um elemento <h1>.

```
function formatName(user) {
  return user.firstName + ' ' + user.lastName;
}

const user = {
  firstName: 'Gleryston',
  lastName: 'Matos'
};

const element = (
  <h1>Olá, {formatName(user)}!</h1>
);

ReactDOM.render(element, document.getElementById('root'));
```

O código do exemplo pode ser conferido aqui.

JSX também é uma expressão

Depois da compilação, as expressões em JSX se transformam em chamadas normais de funções que retornam objetos JavaScript.

Significa que você pode usar JSX dentro de condições if e de laços for, atribuí-lo a variáveis, aceitá-lo como argumentos e retorná-los de funções:

```
function getGreeting(user) {
  if (user) {
    return <h1>Olá, {formatName(user)}!</h1>;
  }
  return <h1>Olá, Desconhecido.</h1>;
}
```

O código do exemplo pode ser conferido aqui.

Especificando atributos com JSX

Podemos usar aspas para especificar strings como atributos:

```
const element = <div tabIndex="0"></div>;
```

Também podemos usar chaves para incorporar uma expressão JavaScript em um atributo:

```
const element = <img src={user.avatarUrl}></img>;
```

O código do exemplo pode ser conferido aqui.

O react somente atualiza o necessário

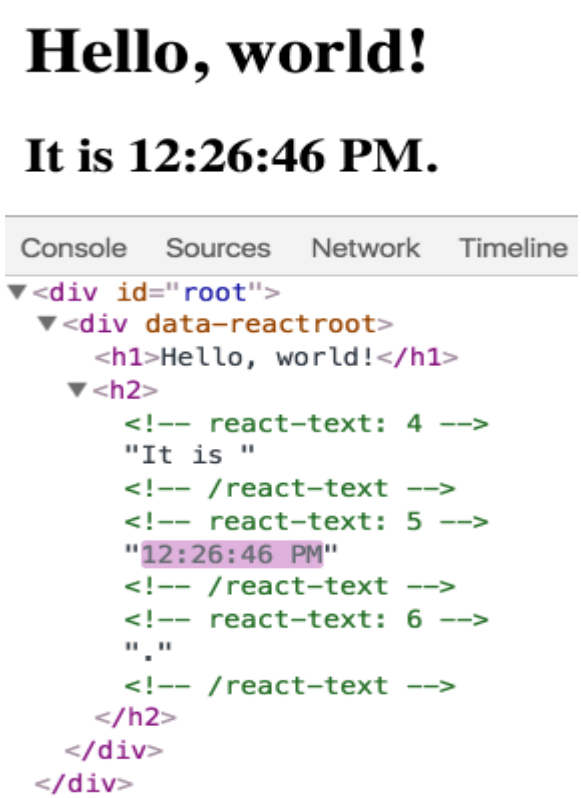
O react DOM compara o elemento novo e seus filhos com os anteriores e somente aplica as modificações necessárias no DOM para levá-lo ao estado desejado.

Vamos implementar um exemplo de um relógio.

```
function tick() {
  const element = (
    <div>
      <h1>Olá Mundo!</h1>
      <h2>Isto é {new Date().toLocaleTimeString()}.</h2>
    </div>
  );
  ReactDOM.render(element, document.getElementById('root'));
}
```

```
    setInterval(tick, 1000);
```

Podemos ver que apenas a informação da hora é atualizada inspecionando o exemplo com as ferramentas do navegador (F12):



Embora nós tenhamos criado um elemento descrevendo toda a estrutura da interface a cada segundo, apenas o nó de texto cujo conteúdo foi alterado é atualizado pelo react DOM.

Na nossa experiência, pensar em como a interface do usuário deve ficar a qualquer momento, em vez de como alterá-la ao longo do tempo, elimina vários possíveis bugs.

O código do exemplo pode ser conferido aqui.

Componentes e props

Os componentes nos permitem dividir a interface do usuário em partes independentes, reutilizáveis e pensar em cada uma delas individualmente.

Conceitualmente, componentes são como funções JavaScript.

Eles aceitam parâmetros arbitrários (que não tem uma definição exata), chamados props (que significa propriedades), e retornam elementos react que descrevem o que deve aparecer na tela.

Para entender melhor, as props são como um objeto, formado pelos campos que foram usados como propriedades, das tags que representam os componentes.

Componentes de função e classe

A maneira mais simples de definir um componente é escrever uma função JavaScript:

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

Essa função é um componente react válido porque aceita um único argumento de objeto props, com dados, e retorna um elemento react.

Nós chamamos esses componentes de componentes de função, porque são literalmente funções JavaScript.

Também podemos usar uma classe ES6 para definir um componente:

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

As duas formas definem o mesmo do ponto de vista do react.

O código do exemplo pode ser conferido aqui.

Renderizando um componente

Anteriormente, nós encontramos apenas elementos react que representam tags do DOM:

```
const element = <div />;
```

Porém, elementos também podem representar componentes definidos pelo usuário:

```
const element = <Welcome name="Gleryston" />;
```

Quando o react vê um elemento representando um componente definido pelo usuário, ele passa atributos JSX e componentes filhos para esse componente como um único objeto.

Chamamos esse objeto de props, como vimos anteriormente.

Por exemplo, esse código:

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

```
}

const element = <Welcome name="Gleryston" />;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

Renderiza Olá, Gleryston na página.

Recapitulando o que que está acontecendo nesse exemplo:

01 - Nós chamamos ReactDOM.render() com o elemento <Welcome name="Gleryston" />.

02 - O react chama o componente Welcome passando {name: 'Gleryston'} como props.

03 - Nosso componente Welcome retorna um elemento <h1>Olá, Gleryston</h1> como resultado.

04 - O react DOM atualiza de forma eficiente o DOM para corresponder <h1>Olá, Gleryston</h1>.

Uma observação, sempre iniciar os nomes dos componentes com uma letra maiúscula.

O react trata componentes começando com letras minúsculas como tags do DOM.

Por exemplo, representa uma tag div do HTML, mas <Welcome /> representa um componente e requer que Welcome esteja no escopo.

O código do exemplo pode ser conferido aqui.

O código do exemplo utilizando um componente de classe pode ser conferido aqui.

Compondo componentes

Componentes podem se referir a outros componentes em sua saída.

Isso permite usar a mesma abstração de componente para qualquer nível de detalhe.

Um botão, um formulário, uma caixa de diálogo, uma tela.

Por exemplo, nós podemos criar um componente App que renderiza Welcome várias vezes:

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

function App() {
```

```
    return (
      <div>
        <Welcome name="Gleryston" />
        <Welcome name="Tobias" />
        <Welcome name="Lola" />
      </div>
    );
  }

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

O código do exemplo pode ser conferido aqui.

Extraindo componentes

Nós não podemos ter medo de dividir componentes complexos em componentes menores.

Por exemplo, vamos analisar o seguinte componente:

```
  function Comment(props) {
    return (
      <div className="Comment">
        <div className="UserInfo">
          <img className="Avatar"
            src={props.author.avatarUrl}
            alt={props.author.name}
          />
          <div className="UserInfo-name">
            {props.author.name}
          </div>
        </div>
        <div className="Comment-text">
          {props.text}
        </div>
        <div className="Comment-date">
          {formatDate(props.date)}
        </div>
      </div>
    );
  }
```

O código do exemplo pode ser conferido aqui.

Ele aceita um autor que é um objeto, um texto como uma string e uma data, todos como props e descreve um comentário em um site de mídias sociais.

Esse componente pode ser complicado de alterar por causa de todo o aninhamento. Também é difícil reutilizar as partes individuais dele como o avatar.
Vamos quebrar ele em componentes menores.

Primeiro, vamos extrair o avatar da seguinte forma:

```
function Avatar(props) {
  return (
    <img
      className="Avatar"
      src={props.user.avatarUrl}
      alt={props.user.name}
    />
  );
}
```

O avatar não precisa saber que está sendo renderizado dentro de um comentário.

É por isso que vamos mudar a sua propriedade para um nome mais genérico, como usuário no lugar de autor.

É recomendável nomear as props a partir do ponto de vista do próprio componente ao invés do contexto em que ele está sendo usado.

Afinal são partes independentes da aplicação.

Agora nós podemos simplificar o comentário um pouco mais:

```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <Avatar user={props.author} />
        <div className="UserInfo-name">
          {props.author.name}
        </div>
      </div>
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
```

```
      </div>
    );
  }
```

Vamos extrair o componente informação do usuário, que renderiza um avatar ao lado do nome do usuário:

```
function UserInfo(props) {
  return (
    <div className="UserInfo">
      <Avatar user={props.user} />
      <div className="UserInfo-name">{props.user.name}</div>
    </div>
  );
}
```

Podemos simplificar o comentario ainda mais:

```
function Comment(props) {
  return (
    <div className="Comment">
      <UserInfo user={props.author} />
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

Extrair componentes pode parecer um trabalho pesado no começo, mas ter um grupo de componentes reutilizáveis compensa em aplicativos grandes.

Uma boa regra é que se uma parte da sua interface do usuário for usada várias vezes como um botão, painel, avatar ou for complexa o suficiente, é uma boa candidata a ser extraída para um componente separado.

O código do exemplo pode ser conferido aqui.

Props são somente leitura

Independente se você declarar um componente como uma função ou uma classe, ele nunca deve modificar seus próprios props.

Por exemplo a seguinte função:

```
function sum(a, b) {
  return a + b;
}
```

Essas funções são chamadas de puras, porque elas não tentam alterar suas entradas e sempre retornam o mesmo resultado para as mesmas.

Em comparação, essa função é impura porque altera sua própria entrada:

```
function withdraw(account, amount) {
  account.total -= amount;
}
```

React é bastante flexível mas tem uma única regra estrita.

Todos os componentes react têm que agir como funções puras em relação ao seus props, ou seja, não podemos alterar o valor dos props.

O código do exemplo pode ser conferido aqui.

Estado e ciclo de vida

Para explicar o estado e o ciclo de vida vamos usar o exemplo do relógio de uma das explicações anteriores.

Em elementos de renderização, nós aprendemos apenas uma maneira de atualizar a interface do usuário.

Nós chamamos React DOM.render() para mudar a saída renderizada.

Podemos ver isso no exemplo:

```
function tick() {
  const element = (
    <div>
      <h1>Olá Mundo!</h1>
      <h2>Isto é {new Date().toLocaleTimeString()}.</h2>
    </div>
  );
  ReactDOM.render(
    element,
    document.getElementById('root')
  );
```

```
}

setInterval(tick, 1000);
```

Agora vamos aprender como transformar o componente relógio de forma verdadeiramente reutilizável e encapsulada.

Ele irá configurar seu próprio temporizador e se atualizar a cada segundo.

Vamos começar encapsulando como o relógio parece:

```
function Clock(props) {
  return (
    <div>
      <h1>Olá Mundo!</h1>
      <h2>Isto é {props.date.toLocaleTimeString()}.</h2>
    </div>
  );
}

function tick() {
  ReactDOM.render(
    <Clock date={new Date()} />,
    document.getElementById('root')
  );
}

setInterval(tick, 1000);
```

Porém, falta um requisito crucial, o fato de que o relógio configurar um temporizador e atualizar a interface a cada segundo deve ser um detalhe de implementação do relógio, não devendo o usuário ter que fazer isso.

Nós queremos chamar o componente relógio e ele deve se atualizar.

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

Para implementá-lo, precisamos adicionar um state ao componente relógio.

O state do componente é similar às props, mas é privado e totalmente controlado pelo componente.

O código do exemplo pode ser conferido aqui.

Convertendo uma função para uma classe

Podemos converter um componente de função como relógio em um componente de classe em cinco etapas:
01 - Primeiro vamos criar uma classe ES6, com o mesmo nome, estendendo React.component.

02 - Vamos adicionar um único método vazio chamado render().

03 - Agora vamos mover o corpo da função para o método render().

04 - Vamos substituir as props por this.props no corpo de render().

05 - Por último vamos excluir a declaração da função vazia restante.

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Olá Mundo!</h1>
        <h2>Isto é {this.props.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

O relógio agora é definido como uma classe em vez de uma função.

O método render vai ser chamado toda vez que uma atualização acontecer, mas enquanto renderizamos o relógio no mesmo nó DOM, apenas uma única instância da classe relógio será utilizada.

Isso nos permite usar recursos adicionais, como o estado local e os métodos do ciclo de vida do componente.

O código do exemplo pode ser conferido aqui.

Adicionando estado local a uma classe

Vamos mover a data da props para o state em três passos:

01 - Primeiro vamos substituir this.props.date por this.state.date no método render();

02 - Agora vamos adicionar um construtor na classe que atribui a data inicial no this.state:

Temos que notar que nós passamos props para o construtor.

Componentes de classes devem sempre chamar o construtor com props, para receber os parâmetros passados pelo usuário.

03 - Vamos remover a props data do elemento relógio que é chamado para renderizar no react DOM.

Mais a frente, vamos adicionar o código do temporizador de volta ao próprio componente.

Depois de realizar as alterações vamos ter o seguinte resultado.

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      date: new Date()
    };
  }

  render() {
    return (
      <div>
        <h1>Olá Mundo!</h1>
        <h2>Isto é {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

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

Agora vamos fazer a configuração do próprio temporizador e atualizar a cada segundo.

O código do exemplo pode ser conferido aqui.

Adicionando métodos de ciclo de vida a classe

Em aplicações com muitos componentes, é muito importante limpar os recursos utilizados pelos componentes quando eles são destruídos.

Vamos configurar um temporizador sempre que o relógio for renderizado no DOM pela primeira vez.

Isso é chamado de mounting no react. (É o processo de renderização do componente na tela)

Também vamos limpar o temporizador sempre que o DOM produzido pelo relógio for removido.

Isso é chamado de unmounting no react. (É o processo de remoção do componente da tela)
Vamos declarar métodos especiais no componente de classe para executar alguns códigos quando um componente é montado e desmontado:

```
componentDidMount() {
}

componentWillUnmount() {
}
```

O método componentDidMount() é executado depois que a saída do componente é renderizada no DOM.

O que o torna um bom lugar para configurar um temporizador:

```
componentDidMount() {
  this.timerID = setInterval(
    () => this.tick(),
    1000
  );
}
```

Observem como nós salvamos o ID do temporizador em this.timerID.

Dentro do componente nós temos as props, o state e as propriedades da própria classe do componente.

As props e o state são ambos objetos JavaScript.

Podemos adicionar campos adicionais ao state e a classe do componente sempre que precisarmos.

Dessa forma temos três locais contendo informações em um componente, porém cada uma delas tem seu objetivo.

No caso das props e do state, apesar de ambos guardarem informações que influenciam no resultado da renderização, eles são diferentes por uma razão importante.

As props são passadas para o componente (como parâmetros de funções) e são configuradas pelo próprio react, não podendo ter seu valor alterado internamente.

O state é gerenciado de dentro do componente (como variáveis declaradas dentro de uma função).

Porém, sempre que o state muda, o componente responde renderizando novamente.

Se precisarmos armazenar alguma informação que não participe do fluxo de dados, ou seja, não irá disparar uma mudança no componente, sendo apenas necessária internamente como um ID do temporizador, podemos usar as propriedades da própria classe para isso.

Feita essa explicação vamos derrubar o temporizador no método do ciclo de vida componentWillUnmount():

```
componentWillUnmount() {
  clearInterval(this.timerID);
}
```

Agora vamos implementar um método chamado tick() que o componente relógio executará a cada segundo.

Ele vai usar this.setState() para agendar as atualizações para o estado local do componente.

```
tick() {
  this.setState({
    date: new Date()
  });
}
```

Agora o relógio bate a cada segundo.

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      date: new Date()
    };
  }

  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }

  componentWillUnmount() {
    clearInterval(this.timerID);
  }

  tick() {
    this.setState({
      date: new Date()
```

```
    });
  }

  render() {
    return (
      <div>
        <h1>Olá Mundo!</h1>
        <h2>Isto é {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

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

Agora vamos revisar rapidamente o que está acontecendo e a ordem na qual os métodos são chamados:

01 - Quando o componente relógio é passado para ReactDOM.render(), o React chama o construtor do componente relógio.

Como o relógio precisa exibir a hora atual, ele inicializa this.state com um objeto incluindo a hora atual.

Mais tarde, nós vamos atualizar este state.

02 - Em seguida o react chama o método render() do componente relógio.

É assim que o react aprende o que deve ser exibido na tela.

Em seguida o react atualiza o DOM para coincidir com a saída de renderização do relógio.

03 - Quando a saída do relógio é inserida no DOM, o react chama o método do ciclo de vida componentDidMount().

Dentro dele, o componente relógio pede ao navegador para configurar um temporizador para chamar o método tick() do componente uma vez por segundo.

04 - A cada segundo o navegador chama o método tick().

Dentro dele o componente relógio agenda uma atualização da interface chamando setState() com um objeto que contém a hora atual.

Graças à chamada do setState(), o método render() será diferente e, portanto, a saída de renderização incluirá a hora atualizada.

O react atualiza o DOM de acordo com as mudanças.
05 - Se o componente relógio for removido do DOM, o react chama o método do ciclo de vida componentWillUnmount() para que o temporizador seja interrompido.

O código do exemplo pode ser conferido aqui.

Usando o state corretamente

Existem três coisas que temos que saber sobre o state.

01 - Não modificar o state diretamente.

Neste exemplo mudamos o valor do state diretamente, porém esta alteração não renderizará o componente novamente.

```
// Errado
this.state.comment = 'Olá';
```

O correto é utilizar o setState();

```
// Correto
this.setState({comment: 'Olá'});
```

O único lugar onde podemos atribuir o state diretamente é no construtor.

02 - As atualizações do state podem ser assíncronas.

O react pode agrupar várias chamadas setState() em uma única atualização para melhorar o desempenho.

Como this.props e this.state podem ser atualizados de forma assíncrona, não podemos confiar nos seus valores para calcular o próximo state do componente.

Por exemplo, esse código pode falhar ao atualizar o contador.

```
// Errado
this.setState({
  counter: this.state.counter + this.props.increment,
});
```

Para consertá-lo, vamos usar uma segunda forma do setState() que aceita uma função ao invés de um objeto.

Essa função receberá o state anterior como o primeiro argumento e as props no momento em que a atualização for aplicada como o segundo argumento.

```
// Correto
this.setState((state, props) => ({
  counter: state.counter + props.increment
}));
```

Nós usamos uma arrow function nesse exemplo, mas também podemos usar como uma função regular.

```
// Correto
this.setState(function(state, props) {
  return {
    counter: state.counter + props.increment
  };
});
```

03 - Atualizações do state são mescladas.

Quando nós chamamos setState(), o react mescla o objeto que nós fornecemos ao state atual.
Por exemplo: nosso state pode conter várias variáveis independentes:

```
constructor(props) {
  super(props);
  this.state = {
    posts: [],
    comments: []
  };
}
```

Então podemos atualizá-los independentemente com chamadas separadas do setState():

```
componentDidMount() {
 fetchPosts().then(response => {
   this.setState({
     posts: response.posts
   });
 });

 fetchComments().then(response => {
   this.setState({
     comments: response.comments
   });
 });
}
```

O merge é superficial, então this.setState({comments}) deixa this.state.posts intacto, mas substitui completamente this.state.comments.

O código do exemplo pode ser conferido aqui.

Os dados fluem para baixo

Nem componentes pais ou filhos podem saber se um determinado componente é stateful ou stateless, e não devem se importar se ele é definido por uma função ou classe.

É por isso que o state é geralmente chamado de local ou encapsulado, não sendo acessível a nenhum componente que não seja o que o possui e o define.

Um componente pode escolher passar seu state como props para seus componentes filhos:

```
<FormattedDate date={this.state.date} />
```

O componente FormattedDate receberia o date em seus objetos e não saberia se ele veio do state de relógio, das props do relógio, ou se foi digitado manualmente:

```
function FormattedDate(props) {
  return <h2>It is {props.date.toLocaleTimeString()}.</h2>;
}
```

Isso é comumente chamado de fluxo de dados “top-down” ou “unidirecional”.

Qualquer state é sempre de propriedade de algum componente específico, e qualquer dado ou interface do usuário derivado desse state só pode afetar os componentes “abaixo” deles na árvore.

Se nos imaginarmos uma árvore de componentes como uma cascata de props, o state de cada componente é como uma fonte de água adicional que o une em um ponto arbitrário, mas também flui para baixo.

Para mostrar que todos os componentes estão isolados, podemos criar um componente App que renderiza três relógios.

```
function App() {
  return (
    <div>
      <Clock />
      <Clock />
      <Clock />
    </div>
  );
}
```

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

Cada relógio configura seu próprio temporizador e atualiza de forma independente.

Nos aplicativos react, se um componente é stateful ou stateless é considerado um detalhe de implementação do componente que pode mudar com o tempo.

Nós podemos usar componentes sem state dentro de componentes com state e vice-versa.

O código do exemplo pode ser conferido aqui.

Manipulando eventos

Manipular eventos em elementos react é muito semelhante a manipular eventos em elementos do DOM.

Existem algumas diferenças de sintaxe, sendo elas:

01 - Eventos em react são nomeados usando camelCase ao invés de letras minúsculas.
02 - Com o JSX nós passamos uma função como manipulador de eventos ao invés de um texto.

Por exemplo, com HTML:

```
<button onclick="activateLasers()">
  Ativar lasers
</button>
```

É um pouco diferente com react:

```
<button onClick={activateLasers}>
  Ativar lasers
</button>
```

Outra diferença é que nós não podemos retornar para evitar o comportamento padrão no react.

Nós devemos chamar preventDefault explícitamente.

Por exemplo, com HTML simples, para evitar que um link abra uma nova página, podemos usar:

```
<a href="#" onclick="console.log('O link foi clicado.'); return false">
  Clique Aqui
</a>
```

No react seria da seguinte forma:

```
function ActionLink() {
  function handleClick(e) {
    e.preventDefault();
    console.log('O link foi clicado.');
  }

  return (
    <a href="#" onClick={handleClick}>
      Clique Aqui
    </a>
  );
}
```

Aqui, ”e” é um synthetic event que é um agregador cross-browser que envolve os eventos nativos do navegador, através dele podemos acessar alguns atributos do evento.

O react define esses eventos sintéticos de acordo com a especificação W3C, então não precisamos nos preocupar com a compatibilidade entre navegadores.

Vocês podem acessar a página SyntheticEvent da documentação do react para saber mais.
Ao usar o react, geralmente não precisamos chamar addEventListener para adicionar um listener a um elemento no DOM depois que ele é criado.

Ao invés disso nós podemos apenas definir um listener quando o elemento é inicialmente renderizado.

Quando definimos um componente usando uma classe do ES6, um padrão comum é que um manipulador de eventos seja um método na classe.

Por exemplo, este componente Toggle renderiza um botão que permite ao usuário alternar entre os estados “ON” e “OFF”:

```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = { isToggleOn: true };

    // Aqui utilizamos o 'bind' para que o 'this' funcione dentro da nossa callback
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState(state => ({
      isToggleOn: !state.isToggleOn
```

```
      }));
    }

    render() {
      return (
        <button onClick={this.handleClick}>
          {this.state.isToggleOn ? 'ON' : 'OFF'}
        </button>
      );
    }
  }

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

Nós precisamos ter cuidado com o significado do this nos callbacks do JSX.

Em JavaScript, os métodos de classe não são vinculados por padrão.

Se nos esquecermos de fazer o bind de this.handleClick e passá-lo para um onClick, o this será undefined quando a função for realmente chamada.

Este não é um comportamento específico do react e sim uma parte de como funcionam as funções em JavaScript.

Geralmente, se nós nos referirmos a um método sem () depois dele, como onClick={this.handleClick}, nós devemos fazer o bind manual deste método.

Se ficar chamando “bind” fica incomodando, existem duas maneiras de contornar isso.

Se estivermos usando a sintaxe experimental de campos de classe pública, podemos usar campos de classe para vincular callbacks corretamente:

```
  class LoggingButton extends React.Component {
    // Essa sintaxe garante que o 'this' seja vinculado ao handleClick.
    // Atenção: essa é uma sintaxe *experimental*.
    handleClick = () => {
      console.log('isto é:', this);
    }

    render() {
      return (
        <button onClick={this.handleClick}>
          Clique em mim
```

```
          </button>
        );
      }
    }
```

Essa sintaxe é habilitada por padrão no create react app.

Se nós não estivermos usando a sintaxe de campos de classe, podemos usar uma arrow function como callback:

```
    class LoggingButton extends React.Component {
      handleClick() {
        console.log('isto é:', this);
      }

      render() {
        // Essa sintaxe garante que o 'this' seja vinculado ao handleClick.
        return (
          <button onClick={() => this.handleClick()}>
            Clique em mim
          </button>
        );
      }
    }
```

O problema com esta sintaxe é que um callback diferente é criado toda vez que o LoggingButton é renderizado.

Na maioria dos casos, tudo bem. No entanto, se esse callback for passado para componentes inferiores através de props, esses componentes poderão fazer uma renderização extra.

Geralmente é recomendado pela documentação que seja feita a vinculação no construtor ou a sintaxe dos campos de classe para evitar esse tipo de problema de desempenho.

O código do exemplo pode ser conferido aqui.

Passando argumentos para manipuladores de eventos

Dentro de uma estrutura de repetição, é comum querer passar um parâmetro extra para um manipulador de eventos.

Por exemplo, se id é o ID de identificação da linha, qualquer um dos dois a seguir vai funcionar:

```
    <button onClick={(e) => this.deleteRow(id, e)}>Deletar linha</button>
    <button onClick={this.deleteRow.bind(this, id)}>Deletar linha</button>
```

As duas linhas acima são equivalentes e usam arrow functions e Function.prototype.bind respectivamente.

Em ambos os casos, o argumento e representando o evento do react será passado como segundo argumento após o ID.

Com uma arrow function, nós temos que passá-lo explicitamente. Mas com o bind outros argumentos adicionais serão automaticamente encaminhados.

O código do exemplo pode ser conferido aqui.

Renderização condicional

Em react, podemos criar componentes distintos que encapsulam o comportamento que nós precisamos.

Então, podemos renderizar apenas alguns dos elementos, dependendo do estado da nossa aplicação.

Renderização condicional em react funciona da mesma forma que condições funcionam em JavaScript.

Usando operadores de JavaScript como if ou operador condicional para criar elementos representando o estado atual, e deixamos o react atualizar a interface para respondê-los.

Por exemplo, vamos observar esses dois componentes:

```
function UserGreeting(props) {
  return <h1>Bem vindo de volta!</h1>;
}

function GuestGreeting(props) {
  return <h1>Por favor inscreva-se.</h1>;
}
```

Vamos criar um componente Greeting (Cumprimento) que mostra um dos outros dois componentes se o usuário estiver logado ou não:

```
function UserGreeting(props) {
  return <h1>Bem vindo de volta!</h1>;
}

function GuestGreeting(props) {
  return <h1>Por favor inscreva-se.</h1>;
}
```

```
function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  if (isLoggedIn) {
    return <UserGreeting />;
  }
  return <GuestGreeting />;
}

ReactDOM.render(
  // Tente mudar para isLoggedIn = {true}:
  <Greeting isLoggedIn={false} />,
  document.getElementById('root')
);
```

O código do exemplo pode ser conferido aqui.

Variáveis de elementos

Podemos usar variáveis para guardar elementos. Isto pode nos ajudar a renderizar condicionalmente parte do componente enquanto o resto do resultado não muda.

Por exemplo, podemos observar esses dois novos componentes representando os botões de Logout e Login:

```
function LoginButton(props) {
  return (
    <button onClick={props.onClick}>
      Login
    </button>
  );
}

function LogoutButton(props) {
  return (
    <button onClick={props.onClick}>
      Logout
    </button>
  );
}
```

Nesse exemplo, nós vamos criar um componente stateful chamado LoginControl.

O componente vai renderizar o LoginButton ou LogoutButton dependendo do estado atual.

Ele também vai renderizar Greeting do exemplo anterior:

```jsx
class LoginControl extends React.Component {
  constructor(props) {
    super(props);
    this.handleLoginClick = this.handleLoginClick.bind(this);
    this.handleLogoutClick = this.handleLogoutClick.bind(this);
    this.state = {isLoggedIn: false};
  }

  handleLoginClick() {
    this.setState({isLoggedIn: true});
  }

  handleLogoutClick() {
    this.setState({isLoggedIn: false});
  }

  render() {
    const isLoggedIn = this.state.isLoggedIn;
    let button;

    if (isLoggedIn) {
      button = <LogoutButton onClick={this.handleLogoutClick} />;
    } else {
      button = <LoginButton onClick={this.handleLoginClick} />;
    }

    return (
      <div>
        <Greeting isLoggedIn={isLoggedIn} />
        {button}
      </div>
    );
  }
}

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

Declarar uma variável e usar uma declaração condicional if é uma ótima maneira de renderizar um componente, mas às vezes podemos querer usar uma sintaxe mais curta.

Existem algumas maneiras para utilizar condições inline em JSX.

O código do exemplo pode ser conferido aqui.

If inline com o operador lógico &&

Nós podemos incorporar qualquer expressão em JSX encapsulando em chaves, incluindo o operador lógico && do JavaScript.

Isto pode ser muito conveniente para incluir um elemento condicionalmente:

```
function Mailbox(props) {
  const unreadMessages = props.unreadMessages;
  return (
    <div>
      <h1>Olá!</h1>
      {unreadMessages.length > 0 &&
        <h2>
          Você tem {unreadMessages.length} mensagens não lidas.
        </h2>
      }
    </div>
  );
}

const messages = ['React', 'Re: React', 'Re:Re: React'];
ReactDOM.render(
  <Mailbox unreadMessages={messages} />,
  document.getElementById('root')
);
```

Isto funciona porque em JavaScript, true && expressão são sempre avaliadas como expressão, e false && expressão são sempre avaliadas como false.

Portanto, se a condição é true, o elemento logo depois do && irá aparecer no resultado.

Se o elemento é false, react irá ignorá-lo.

O código do exemplo pode ser conferido aqui.

If-Else inline com operador condicional

Outro método para renderizar elementos inline é utilizar o operador condicional do JavaScript condição ? true : false.

Neste exemplo, nós vamos utilizar para renderizar condicionalmente um pequeno bloco de texto.

```
      render() {
        const isLoggedIn = this.state.isLoggedIn;
        return (
          <div>
            O usuário é <b> {isLoggedIn ? 'atualmente' : 'não'} </b> conectado.
          </div>
        );
      }
```

Pode também ser usado para expressões mais longas, embora o que está acontecendo seja menos óbvio:

```
      render() {
        const isLoggedIn = this.state.isLoggedIn;
        return (
          <div>
            {isLoggedIn
              ? <LogoutButton onClick={this.handleLogoutClick} />
              : <LoginButton onClick={this.handleLoginClick} />
            }
          </div>
        );
      }
```

Assim como em JavaScript, vocês podem decidir o estilo mais apropriado com base na sua opinião e na equipe.

Vocês tem que se lembrar que toda vez que as condições se tornam muito complexas, pode ser um bom momento para extrair componentes.

O código do exemplo pode ser conferido aqui.

Evitando que um componente seja renderizado

Em casos raros nós podemos ter a necessidade de que um componente se esconda ainda que ele tenha sido renderizado por outro componente.

Para fazer isso, basta retornar null ao invés do resultado renderizado.

No próximo exemplo, o WarningBanner é renderizado dependendo do valor da prop chamada warn. Se o valor da prop é false, o componente não é renderizado:

```
    function WarningBanner(props) {
      if (!props.warn) {
        return null;
```

```
    }

    return (
      <div className="warning">
        Aviso!
      </div>
    );
  }

class Page extends React.Component {
  constructor(props) {
    super(props);
    this.state = {showWarning: true};
    this.handleToggleClick = this.handleToggleClick.bind(this);
  }

  handleToggleClick() {
    this.setState(state => ({
      showWarning: !state.showWarning
    }));
  }

  render() {
    return (
      <div>
        <WarningBanner warn={this.state.showWarning} />
        <button onClick={this.handleToggleClick}>
          {this.state.showWarning ? 'Hide' : 'Show'}
        </button>
      </div>
    );
  }
}

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

Retornar null do método render de um componente não afeta a ativação dos métodos do ciclo de vida do componente.

Por exemplo, o método componentDidUpdate ainda será chamado.

O código do exemplo pode ser conferido aqui.

Listas e chaves

Antes de iniciarmos, vamos ver como transformamos listas em JavaScript.

Com base neste exemplo, nós usamos a função map() para receber um array de números e dobrar o valor de cada um deles.

Em seguida o novo array é retornado pela função map() é atribuído para a variável doubled e é impressa no console:

```
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map((number) => number * 2);
console.log(doubled);
```

Esse código vai imprimir o valor [2, 4, 6, 8, 10] no console.

No react, transformar arrays em listas de elementos é praticamente idêntico a isso.

O código do exemplo pode ser conferido aqui.

Renderizando múltiplos componentes

Podemos criar coleções de elementos e adicioná-las no JSX usando chaves {}.

No próximo exemplo, percorremos pelo array numbers usando a função map() do JavaScript.

Depois retornamos um elemento <li> para cada item.

Por último, atribuímos o array de elementos que é retornado para listItems:

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <li>{number}</li>
);
```

Adicionamos todo o array listItems dentro de um elemento <ul> e renderizamos ele no DOM:

```
ReactDOM.render(
  <ul>{listItems}</ul>,
  document.getElementById('root')
);
```

Este exemplo mostra uma lista não ordenada de números entre 1 e 5.

O código do exemplo pode ser conferido aqui.

Componente de lista básico

Geralmente nós vamos renderizar listas dentro de um componente.

Podemos refatorar o exemplo anterior em um componente que aceita um array de números e retorna uma lista de elementos.

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li>{number}</li>
  );
  return (
    <ul>{listItems}</ul>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```

Ao executar esse código, nós podemos perceber um aviso de que uma chave deve ser definida para os itens da lista.

Key é um atributo string especial que nós precisamos definir ao criar listas de elementos.

Vamos analisar os motivos pelos quais isso é importante.

Vamos atribuir uma key aos itens da nossa lista dentro de numbers.map() e resolver o valor da chave que está em falta.

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li key={number.toString()}>
      {number}
    </li>
  );
  return (
    <ul>{listItems}</ul>
  );
}
```

O código do exemplo pode ser conferido aqui.

Chaves

As chaves ajudam o react a identificar quais itens sofreram alterações, foram adicionados ou removidos.

As chaves devem ser atribuídas aos elementos dentro do array para dar uma identidade estável aos elementos:

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <li key={number.toString()}>
    {number}
  </li>
);
```

A melhor forma de escolher uma chave é usar uma string que identifica unicamente um item da lista dentre os demais.

Na maioria das vezes nós usamos os campos IDs dos nossos dados como chave:

```
const todoItems = todos.map((todo) =>
  <li key={todo.id}>
    {todo.text}
  </li>
);
```

Quando nós não possuirmos nenhum ID estável para os itens renderizados, podemos usar o índice do item como chave em último recurso:

```
const todoItems = todos.map((todo, index) =>
  // Apenas faça isso caso os itens não possuam IDs estáveis
  <li key={index}>
    {todo.text}
  </li>
);
```

Não é recomendado o uso de índices para chave se a ordem dos itens pode ser alterada.

Isso pode impactar de forma negativa o desempenho e poderá causar problemas com o estado do componente.

Vocês podem ler o artigo escrito por Robin Pokorny para uma explicação mais aprofundada dos impactos negativos de se usar um índice como chave.

Se nós não atribuirmos uma chave de forma explícita para os itens de uma lista, então o react vai utilizar os índices como chave por padrão.
Aqui vocês podem ver uma explicação mais aprofundada sobre o porquê o uso das chaves é necessário caso vocês estejam interessado em aprender mais sobre isso.

O código do exemplo pode ser conferido aqui.

Extraindo componentes com chaves

As chaves apenas fazem sentido no contexto do array que está encapsulando os itens.

Por exemplo, se nós extrairmos um componente ListItem, nós devemos deixar as chaves nos elementos <ListItem /> ao invés de deixar no elemento <li> dentro de ListItem.

Exemplo: Uso Incorreto de Chaves

```
function ListItem(props) {
  const value = props.value;
  return (
    // Errado! Não há necessidade de definir a chave aqui:
    <li key={value.toString()}>
      {value}
    </li>
  );
}

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    // Errado! A chave deveria ser definida aqui:
    <ListItem value={number} />
  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```

Exemplo: Uso Correto de Chaves

```
function ListItem(props) {
  // Correto! Não há necessidade de definir a chave aqui:
  return <li>{props.value}</li>;
}

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    // Correto! A chave deve ser definida dentro do array.
    <ListItem key={number.toString()} value={number} />
  );

  return (
    <ul>
      {listItems}
    </ul>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```

Por via de regra, os elementos dentro de uma função map() devem especificar chaves.

O código do exemplo pode ser conferido aqui.

Chaves devem ser únicas apenas entre elementos irmãos

Chaves usadas nos arrays devem ser únicas entre seus elementos irmãos. Contudo elas não precisam ser únicas globalmente. Podemos usar as mesmas chaves ao criar dois arrays diferentes:

```
function Blog(props) {
  const sidebar = (
    <ul>
      {props.posts.map((post) =>
        <li key={post.id}>
          {post.title}
        </li>
      )}
    </ul>
```

```
    );

    const content = props.posts.map((post) =>
      <div key={post.id}>
        <h3>{post.title}</h3>
        <p>{post.content}</p>
      </div>
    );

    return (
      <div>
        {sidebar}
        <hr />
        {content}
      </div>
    );
  }

  const posts = [
    {id: 1, title: 'Olá Mundo', content: 'Bem-vindo ao aprendizado do React!'},
    {id: 2, title: 'Instalação', content: 'Você pode instalar o React a partir do npm.'}
  ];

  ReactDOM.render(
    <Blog posts={posts} />,
    document.getElementById('root')
  );
```

As chaves servem como uma dica para o React. Mas elas não são passadas para os componentes. Se nós precisarmos do mesmo valor em um componente, devemos definir ele explicitamente como uma prop com um nome diferente:

```
  const content = posts.map((post) =>
    <Post
      key={post.id}
      id={post.id}
      title={post.title} />
  );
```

No exemplo acima, o componente Post pode acessar props.id. Mas, não pode acessar props.key.

O código do exemplo pode ser conferido aqui.

Incluindo map() no JSX

Nos exemplos acima declaramos uma variável listItems separada e adicionamos ela no JSX:

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <ListItem key={number.toString()}
              value={number} />
  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}
```

O JSX permite incluir qualquer expressão dentro de chaves, então podemos adicionar o resultado do map() diretamente:

```
function NumberList(props) {
  const numbers = props.numbers;
  return (
    <ul>
      {numbers.map((number) =>
        <ListItem key={number.toString()}
                  value={number} />
      )}
    </ul>
  );
}
```

Às vezes isso resulta em um código mais limpo. Mas esse padrão também pode ser confuso. Como em JavaScript, depende de você decidir se vale a pena extrair uma variável para aumentar a legibilidade. Temos que nos lembrar de que se o corpo da função map() tiver muitos níveis, poderá ser um bom momento para extrair um componente.

O código do exemplo pode ser conferido aqui.

Formulários

Os elementos de formulário HTML funcionam de maneira um pouco diferente de outros elementos DOM no React, porque os elementos de formulário mantêm naturalmente um estado interno.

Por exemplo, este formulário em HTML puro aceita um único nome:

```
<form>
```

```
    <label>
      Nome:
      <input type="text" name="name" />
    </label>
    <input type="submit" value="Enviar" />
  </form>
```

Esse formulário tem o comportamento padrão do HTML de navegar para uma nova página quando o usuário enviar o formulário.

Se quisermos esse comportamento no react, ele simplesmente funciona. Mas na maioria dos casos, é conveniente ter uma função JavaScript que manipula o envio de um formulário e tem acesso aos dados que o usuário digitou nos inputs.

O modo padrão de fazer isso é com uma técnica chamada “componentes controlados” (controlled components).

O código do exemplo pode ser conferido aqui.

Componentes controlados

Em HTML, elementos de formulário como <input>, <textarea> e <select> normalmente mantêm seu próprio estado e o atualiza baseado na entrada do usuário. Em React, o estado mutável é normalmente mantido na propriedade state dos componentes e atualizado apenas com setState().

Podemos combinar os dois fazendo o estado do react ser a “única fonte da verdade”. Assim, o componente react que renderiza um formulário também controla o que acontece neste formulário, nas entradas subsequentes do usuário.

Um input cujo o valor é controlado pelo react dessa maneira é chamado de “componente controlado” (controlled component).

Por exemplo, se quisermos que o exemplo anterior registre o nome quando ele for enviado, podemos escrever o formulário como um componente controlado:

```
  class NameForm extends React.Component {
    constructor(props) {
      super(props);
      this.state = {value: ''};

      this.handleChange = this.handleChange.bind(this);
      this.handleSubmit = this.handleSubmit.bind(this);
    }

    handleChange(event) {
      this.setState({value: event.target.value});
```

```
    }

    handleSubmit(event) {
      alert('Um nome foi enviado: ' + this.state.value);
      event.preventDefault();
    }

    render() {
      return (
        <form onSubmit={this.handleSubmit}>
          <label>
            Nome:
            <input type="text" value={this.state.value} onChange={this.handleChange} />
          </label>
          <input type="submit" value="Enviar" />
        </form>
      );
    }
  }
```

Como o atributo value é definido no nosso <input type="text">, o valor exibido sempre será o mesmo de this.state.value, fazendo com que o estado do react seja a fonte da verdade.

Como o handleChange é executado a cada tecla pressionada para atualizar o estado do react, o valor exibido será atualizado conforme o usuário digita.

Com um componente controlado, o valor da entrada é sempre direcionado pelo estado react. Embora isso signifique que nós precisamos digitar um pouco mais de código, agora também podemos passar o valor para outros elementos da interface do usuário ou redefini-los de outros manipuladores de eventos.

O código do exemplo pode ser conferido aqui.

Tag textarea

Em HTML, o texto de um elemento <textarea> é definido por seus filhos:

```
<textarea>
  Apenas algum texto em uma área de texto
</textarea>
```

Em react, em vez disso, o <textarea> usa um atributo value.

Desta forma, um formulário usando um <textarea> pode ser escrito de forma muito semelhante a um formulário que usa um input de linha única:

```
class EssayForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: 'Por favor, escreva uma dissertação sobre o seu elemento DOM favorito.'
    };

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    alert('Uma dissertação foi enviada: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Dissertação:
          <textarea value={this.state.value} onChange={this.handleChange} />
        </label>
        <input type="submit" value="Enviar" />
      </form>
    );
  }
}
```

Observem que this.state.value é inicializado no construtor, para que o textarea comece com algum texto.

O código do exemplo pode ser conferido aqui.

Tag select

Em HTML, <select> cria uma lista suspensa (drop-down). Por exemplo, esse HTML cria uma lista suspensa de sabores:

```
<select>
  <option value="laranja">Laranja</option>
```

```
    <option value="limao">Limão</option>
    <option selected value="coco">Coco</option>
    <option value="manga">Manga</option>
  </select>
```

Podemos notar que a opção “coco” é selecionada por padrão, por causa do atributo selected. No react, em vez de usar este atributo selected, usamos um atributo value na raiz da tag select.

Isso é mais conveniente em um componente controlado, porque nós só precisamos atualizá-lo em um só lugar.

Por exemplo:

```
    class FlavorForm extends React.Component {
      constructor(props) {
        super(props);
        this.state = {value: 'coco'};

        this.handleChange = this.handleChange.bind(this);
        this.handleSubmit = this.handleSubmit.bind(this);
      }

      handleChange(event) {
        this.setState({value: event.target.value});
      }

      handleSubmit(event) {
        alert('Seu sabor favorito é: ' + this.state.value);
        event.preventDefault();
      }

      render() {
        return (
          <form onSubmit={this.handleSubmit}>
            <label>
              Escolha seu sabor favorito:
              <select value={this.state.value} onChange={this.handleChange}>
                <option value="laranja">Laranja</option>
                <option value="limao">Limão</option>
                <option value="coco">Coco</option>
                <option value="manga">Manga</option>
              </select>
            </label>
            <input type="submit" value="Enviar" />
          </form>
```

```
        );
      }
    }
```

No geral, isso faz com que as tags <input type="text">, <textarea> e <select> funcionem de forma muito semelhante - todos eles aceitam um atributo value que nós podemos usar para implementar um componente controlado.

Observação:

Nós podemos passar um array para o atributo value, permitindo que nós possamos selecionar várias opções em uma tag select:

```
<select multiple={true} value={['B', 'C']}>
```

O código do exemplo pode ser conferido aqui.

Tag de entrada de arquivos

Em HTML, o <input type="file"> permite ao usuário escolher um ou mais arquivos de seu dispositivo para serem enviados para um servidor ou manipulados por JavaScript através da File API.

```
<input type="file" />
```

Como seu valor é de somente leitura, ele é um componente não controlado do react. Esses são discutidos junto a outros componentes não controlados no guia avançado da documentação do react.

O código do exemplo pode ser conferido aqui.

Manipulando múltiplos inputs

Quando nós precisamos manipular múltiplos inputs controlados, nós podemos adicionar um atributo name a cada elemento e deixar a função manipuladora escolher o que fazer com base no valor de event.target.name.

Por exemplo:

```
class Reservation extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      isGoing: true,
      numberOfGuests: 2
    };
```

```
    this.handleInputChange = this.handleInputChange.bind(this);
  }

  handleInputChange(event) {
    const target = event.target;
    const value = target.name === 'isGoing' ? target.checked : target.value;
    const name = target.name;

    this.setState({
      [name]: value
    });
  }

  render() {
    return (
      <form>
        <label>
          Estão indo:
          <input
            name="isGoing"
            type="checkbox"
            checked={this.state.isGoing}
            onChange={this.handleInputChange} />
        </label>
        <br />
        <label>
          Número de convidados:
          <input
            name="numberOfGuests"
            type="number"
            value={this.state.numberOfGuests}
            onChange={this.handleInputChange} />
        </label>
      </form>
    );
  }
}
```

Observem como usamos a sintaxe ES6 nomes de propriedades computados para atualizar a chave de estado correspondente ao nome de entrada fornecido:

```
    this.setState({
      [name]: value
    });
```

É equivalente a este código no ES5:

```
var partialState = {};
partialState[name] = value;
this.setState(partialState);
```

Além disso, como o setState() automaticamente mescla um estado parcial ao estado atual, nós podemos chamá-lo apenas com as partes alteradas.

O código do exemplo pode ser conferido aqui.

Valor nulo em um input controlado

A especificação de uma prop value em um componente controlado impede que o usuário altere a entrada, a menos que nós desejemos. Se nós especificarmos uma prop value, mas o input ainda for editável, nós podemos ter acidentalmente definido o value como undefined ou null.

O código a seguir demonstra isso. (O input é bloqueado no início, mas torna-se editável após um tempo.)

```
ReactDOM.render(<input value="hi" />, document.getElementById('root'));

setTimeout(function() {
  ReactDOM.render(<input value={null} />, document.getElementById('root'));
}, 1000);
```

O código do exemplo pode ser conferido aqui.

Alternativas para componentes controlados

Às vezes pode ser tedioso usar componentes controlados, porque nós precisamos escrever um manipulador de eventos para cada maneira que seus dados podem mudar e canalizar todo o estado do input através de um componente react.

Isso pode se tornar particularmente irritante quando nós estamos convertendo uma base de código preexistente para o react ou integrando um aplicativo react com uma biblioteca que não seja baseada em react.

Nessas situações, podemos verificar os componentes não controlados, uma técnica alternativa para implementar formulários de entrada.

Soluções completas

Se vocês quiserem uma solução completa, incluindo validação, manter o controle dos campos visualizados e lidar com o envio de formulários, o Formik é uma das escolhas mais populares. No entanto, ele é construído sobre os mesmos princípios de componentes controlados e gerenciamento de estado - portanto, não negligencie o aprendizado desses conceitos.

Elevando o state

Com frequência, a modificação de um dado tem que ser refletida em vários componentes.

É recomendado elevar o state compartilhado ao elemento pai comum mais próximo. Vamos ver como isso funciona na prática.

Nessa seção, vamos criar uma calculadora de temperatura que calcula se a água ferveria em determinada temperatura.

Vamos iniciar com um componente chamado BoilingVerdict (Veredito fervente). Ele recebe a temperatura por meio da prop celsius e retorna uma mensagem indicando se a temperatura é suficiente para a água ferver.

```
function BoilingVerdict(props) {
  if (props.celsius >= 100) {
    return <p>A água ferveria.</p>;
  }
  return <p>A água não ferveria.</p>;
}
```

A seguir, criaremos um componente chamado Calculator. Ele renderiza um <input> que recebe a temperatura e a mantém em this.state.temperature.

Além disso, ele renderiza o BoilingVerdict de acordo com o valor atual do input.

```
class Calculator extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {temperature: ''};
  }

  handleChange(e) {
    this.setState({temperature: e.target.value});
  }

  render() {
```

```
        const temperature = this.state.temperature;
        return (
          <fieldset>
            <legend>Informe a temperatura em Celsius:</legend>
            <input
              value={temperature}
              onChange={this.handleChange} />
            <BoilingVerdict
              celsius={parseFloat(temperature)} />
          </fieldset>
        );
      }
    }
```

O código do exemplo pode ser conferido aqui.

Adicionando um segundo input

Nosso novo requisito é que, além de um input para grau Celsius, também tenhamos um input para grau Fahrenheit e que ambos estejam sincronizados.

Podemos iniciar extraindo um componente TemperatureInput do componente Calculator.

Vamos adicionar uma nova prop scale que pode ser tanto "c" como "f":

```
    const scaleNames = {
      c: 'Celsius',
      f: 'Fahrenheit'
    };

    class TemperatureInput extends React.Component {
      constructor(props) {
        super(props);
        this.handleChange = this.handleChange.bind(this);
        this.state = {temperature: ''};
      }

      handleChange(e) {
        this.setState({temperature: e.target.value});
      }

      render() {
        const temperature = this.state.temperature;
        const scale = this.props.scale;
        return (
```

```
        <fieldset>
          <legend>Informe a temperatura em {scaleNames[scale]}:</legend>
          <input value={temperature}
                 onChange={this.handleChange} />
        </fieldset>
      );
    }
  }
```

Agora podemos modificar o Calculator para renderizar dois inputs de temperatura separados:

```
class Calculator extends React.Component {
  render() {
    return (
      <div>
        <TemperatureInput scale="c" />
        <TemperatureInput scale="f" />
      </div>
    );
  }
}
```

Agora nós temos dois inputs. Porém, quando a temperatura é inserida em um deles, o outro não atualiza. Isso contraria nosso requisito: queremos que eles estejam sincronizados.

Também não podemos renderizar o BoilingVerdict a partir do Calculator, porque esse não conhece a temperatura atual já que ela está escondida dentro do TemperatureInput.

O código do exemplo pode ser conferido aqui.

Codificando funções de conversão

Primeiro, vamos criar duas funções para converter de Celsius para Fahrenheit e vice-versa:

```
function toCelsius(fahrenheit) {
  return (fahrenheit - 32) * 5 / 9;
}

function toFahrenheit(celsius) {
  return (celsius * 9 / 5) + 32;
}
```

Essas duas funções convertem números. Vamos criar outra função que recebe uma string temperature e uma função de conversão como argumentos e retorna uma string.

Vamos usá-la para calcular o valor de um input com base no outro input.

Retorna uma string vazia quando temperature for inválido e manter o retorno arredondado até a terceira casa decimal.

```
function tryConvert(temperature, convert) {
  const input = parseFloat(temperature);
  if (Number.isNaN(input)) {
    return '';
  }
  const output = convert(input);
  const rounded = Math.round(output * 1000) / 1000;
  return rounded.toString();
}
```

Por exemplo, tryConvert('abc', toCelsius) retorna uma string vazia e tryConvert('10.22', toFahrenheit) retorna '50.396'.

O código do exemplo pode ser conferido aqui.

Elevando o state

Atualmente, ambos os componentes TemperatureInput mantém, de modo independente, o valor em seu state local.

```
class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {temperature: ''};
  }

  handleChange(e) {
    this.setState({temperature: e.target.value});
  }

  render() {
    const temperature = this.state.temperature;
    const scale = this.props.scale;
    return (
      <fieldset>
        <legend>Informe a temperatura em {scaleNames[scale]}:</legend>
        <input value={temperature}
               onChange={this.handleChange} />
      </fieldset>
```

```
        );
      }
    }
```

Porém, queremos que esses dois inputs estejam sincronizados um com o outro. Quando o input de Celsius for atualizado, o input de Fahrenheit deve mostrar a temperatura convertida e vice-versa. No react, o compartilhamento do state é alcançado ao movê-lo para o elemento pai comum aos componentes que precisam dele. Isso se chama “elevar o state” (state lift).

Vamos remover o state local do TemperatureInput e colocá-lo no Calculator.

Se o Calculator é dono do state compartilhado, ele se torna a “fonte da verdade” para a temperatura atual em ambos os inputs. Ele pode instruir ambos a terem valores que são consistentes um com o outro. Já que as props de ambos os TemperatureInput vem do mesmo componente pai, Calculator, os dois inputs sempre estarão sincronizados.

Vamos ver o passo a passo de como isso funciona.

Primeiro, vamos substituir this.state.temperature pôr this.props.temperature no componente TemperatureInput.

Por hora, vamos fingir que this.props.temperature já existe. Apesar de que, no futuro, teremos que passá-la do Calculator:

```
        // Antes: const temperature = this.state.temperature;
        const temperature = this.props.temperature;
```

Sabemos que props são somente leitura. Quando a temperature estava no state local, o TemperatureInput podia simplesmente chamar this.setState() para modificá-lo. Porém, agora que a temperature vem do elemento pai como uma prop, ó TemperatureInput não tem controle sobre ela.

No react, isso é comumente solucionado ao tornar um componente comum em um “componente controlado”. Assim como o <input> do DOM aceita ambas as props value e onChange, o componente personalizado TemperatureInput também pode aceitar ambas as props temperature e onTemperatureChange do Calculator, seu componente pai.

Agora, quando o TemperatureInput quiser atualizar sua temperatura, ele executa this.props.onTemperatureChange:

```
      handleChange(e) {
        // Antes: this.setState({temperature: e.target.value});
        this.props.onTemperatureChange(e.target.value);
      }
```

Observação:

O nome das props temperature ou onTemperatureChange não possui nenhum significado especial. Elas poderiam ter quaisquer outros nomes, tais como value e onChange, o que é uma convenção comum.

A prop onTemperatureChange será fornecida juntamente com a prop temperature pelo componente pai Calculator. Esse irá cuidar das alterações ao modificar seu próprio state local, fazendo com que ambos os inputs sejam renderizados com novos valores.
Vamos conferir a nova implementação do componente Calculator em breve.

Antes de mergulhar nas alterações do Calculator, vamos recapitular as alterações no componente TemperatureInput.

Nós removemos o state local dele, então ao invés de ler this.state.temperature, agora lemos this.props.temperature.

Ao invés de chamar this.setState() quando quisermos fazer uma alteração chamamos this.props.onTemperatureChange() que será fornecido pelo Calculator:

```
class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange(e) {
    // Antes: this.setState({temperature: e.target.value});
    this.props.onTemperatureChange(e.target.value);
  }

  render() {
    // Antes: const temperature = this.state.temperature;
    const temperature = this.props.temperature;
    const scale = this.props.scale;
    return (
      <fieldset>
        <legend>Informe a temperatura em {scaleNames[scale]}:</legend>
        <input value={temperature}
               onChange={this.handleChange} />
      </fieldset>
    );
  }
}
```

Agora é a vez do componente Calculator.

Vamos armazenar no state local os valores temperature e scale referentes ao input atual.

Eles representam o state que “foi elevado” dos inputs e servirá como “fonte da verdade” para ambos.

É a representação mínima de todos os dados necessários para conseguir renderizar ambos os inputs.

Por exemplo, se informarmos 37 no input de Celsius, o state do componente Calculator será:

```
{
  temperature: '37',
  scale: 'c'
}
```

Posteriormente, se editarmos o input Fahrenheit para ser 212, o state do componente Calculator será:

```
{
  temperature: '212',
  scale: 'f'
}
```

Poderíamos ter armazenado o valor de ambos os inputs mas isso não é necessário. É suficiente armazenar o valor do input recentemente alterado e da escala que ele representa. Podemos assim inferir o valor do outro input com base nos valores atuais de temperature e scale.

Os inputs ficam sincronizados porque seus valores são calculados tendo como base o mesmo state:

```
class Calculator extends React.Component {
  constructor(props) {
    super(props);
    this.handleCelsiusChange = this.handleCelsiusChange.bind(this);
    this.handleFahrenheitChange = this.handleFahrenheitChange.bind(this);
    this.state = {temperature: '', scale: 'c'};
  }

  handleCelsiusChange(temperature) {
    this.setState({scale: 'c', temperature});
  }

  handleFahrenheitChange(temperature) {
    this.setState({scale: 'f', temperature});
  }

  render() {
    const scale = this.state.scale;
    const temperature = this.state.temperature;
```

```
        const celsius = scale === 'f' ? tryConvert(temperature, toCelsius) : temperature;
        const fahrenheit = scale === 'c' ? tryConvert(temperature, toFahrenheit) : temperature;

        return (
          <div>
            <TemperatureInput
              scale="c"
              temperature={celsius}
              onTemperatureChange={this.handleCelsiusChange} />
            <TemperatureInput
              scale="f"
              temperature={fahrenheit}
              onTemperatureChange={this.handleFahrenheitChange} />
            <BoilingVerdict
              celsius={parseFloat(celsius)} />
          </div>
        );
      }
    }
```

Agora, tanto faz qual input for editado, this.state.temperature e this.state.scale no componente Calculator serão atualizados. Um dos inputs recebe o valor como está, preservando o que foi informado pelo usuário e o valor do outro input é sempre calculado com base no primeiro.

Vamos recapitular o que acontece quando um input é editado:

01 - O react executa a função especificada como onChange no <input> do DOM. Nesse caso, esse é o método handleChange do componente TemperatureInput.

02 - O método handleChange do componente TemperatureInput executa this.props.onTemperatureChange() com o novo valor desejado. Suas props, incluindo onTemperatureChange, foram fornecidas pelo seu componente pai, o Calculator.

03 - Quando renderizado previamente, o componente Calculator especificou que onTemperatureChange do TemperatureInput de Celsius é o método handleCelsiusChange do Calculator, e que onTemperatureChange do TemperatureInput de Fahrenheit é o método handleFahrenheitChange do Calculator. Então um desses dois métodos do Calculator será executado dependendo de qual input for editado.

04 - Dentro desses métodos, o componente Calculator pede ao react para ser renderizado novamente ao chamar this.setState() com o novo valor da temperatura e da escala do input recém editado.

05 - O react executa o método render do componente Calculator para aprender como a interface deveria ficar. Os valores de ambos os inputs são recalculados com base na temperatura e escala atuais. A conversão de temperatura é realizada aqui.

06 - O react executa o método render dos dois componentes TemperatureInput com suas novas props especificadas pelo Calculator. O react aprende como a interface do usuário deve ficar.

07 - O react executa o método render do componente BoilingVerdict, passando a temperatura em Celsius como prop.

08 - O react DOM atualiza o DOM com o veredito e com os valores de input desejáveis. O input que acabamos de editar recebe seu valor atual e o outro input é atualizado com a temperatura após a conversão.

Toda edição segue os mesmos passos então os inputs ficam sincronizados.

O código do exemplo pode ser conferido aqui.

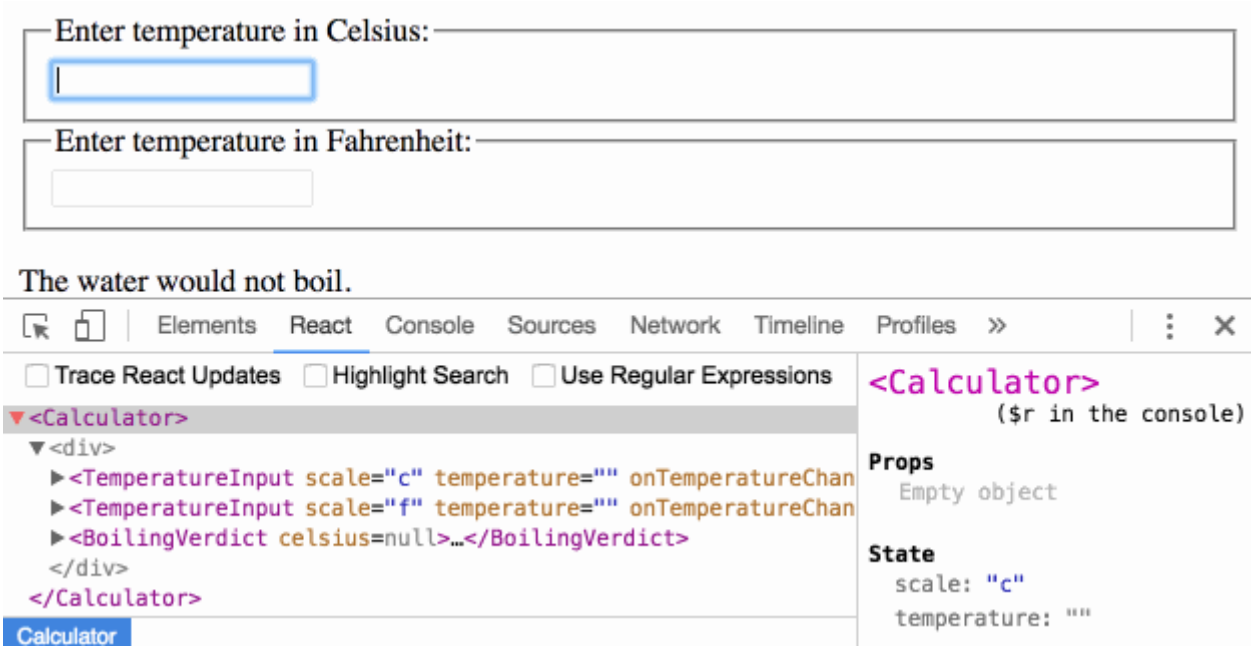
Lições aprendidas

Deve haver uma única “fonte da verdade” para quaisquer dados que sejam alterados em uma aplicação react. Geralmente, o state é adicionado ao componente que necessita dele para renderizar. Depois, se outro componente também precisar desse state, nós precisamos elevá-lo ao elemento pai comum mais próximo de ambos os componentes. Ao invés de tentar sincronizar o state entre diferentes componentes, nós devemos contar com o fluxo de dados de cima para baixo.

Elevar o state envolve escrever mais código de estrutura do que as abordagens de two-way data bind, mas como benefício, demanda menos trabalho para encontrar e isolar erros. Já que o state “vive” em um componente e somente esse componente pode alterá-lo, a área de superfície para encontrar os erros é drasticamente reduzida. Além disso, é possível implementar qualquer lógica personalizada para rejeitar ou transformar o input do usuário.

Se alguma coisa pode ser derivada tanto das props como do state, ela provavelmente não deveria estar no state. Por exemplo, ao invés de armazenar ambos celsiusValue e fahrenheitValue, armazenamos somente o valor da última temperature editada e o valor de scale. O valor do outro input pode sempre ser calculado com base nessas informações no método render(). Isso permite limpar ou arredondar o valor no outro input sem perder precisão no valor informado pelo usuário.

Quando nós vermos algo de errado na interface do usuário, nós podemos utilizar o react developer tools para inspecionar as props e subir a árvore de elementos até encontrar o componente responsável por atualizar o state. Isso permite que nós encontremos a fonte dos erros:


Enter temperature in Celsius:
Enter temperature in Fahrenheit:
The water would not boil.
Elements
React
Console
Sources
Network
Timeline
Profiles
Trace React Updates
Highlight Search
Use Regular Expressions
<Calculator>

<TemperatureInput scale="c" temperature="" onTemperatureChan
<TemperatureInput scale="f" temperature="" onTemperatureChan
<BoilingVerdict celsius=null>…</BoilingVerdict>

</Calculator>
Calculator
<Calculator>
($r in the console)
Props
Empty object
State
scale: "c"
temperature: ""