

Por Mateus Silva



React **Hooks** **Handbook**

Sumário

Sobre o autor	2
Sobre o eBook	3
O que são hooks?	4
Reutilização de lógica.....	5
Componentes muito grandes	6
Uma confusão chamada classes	6
Depois da tempestade... os hooks	7
State em componentes funcionais	8
Pra onde foram os lifecycle methods?	10
Nova forma de usar a Context API	13
useReducer	16
Performance e memorização.....	19
Evitando renderizações desnecessárias	22
Removendo cálculos pesados de dentro do render	24
Acessando elementos na DOM por referências	25
useImperativeHandle e forwardRef	27
useEffect vs useLayoutEffect.....	30
Custom Hooks	32
useDebugValue	34
Conclusão	36



Sobre o autor



Muito prazer, eu sou o Mateus Silva, um cara apaixonado por tecnologia, programação e autor deste eBook!

Nos últimos anos venho me dedicando exclusivamente ao estudo e trabalho com JavaScript, mais especificamente com a stack Node, React e React Native.

Atualmente meu foco está voltado a compartilhar meu conhecimento com a comunidade e ensinar estas tecnologias, pois acredito serem as melhores escolhas dentre as opções que temos disponíveis hoje no mercado.

Se você quiser saber um pouco mais sobre mim e sobre meu trabalho, vou deixar aqui em baixo o meu canal no YouTube e também meu perfil no Instagram, onde conseguimos ter um contato mais próximo, e inclusive, ao final desta leitura, me manda uma DM por lá me contando o que achou do eBook, se te ajudou ou não, enfim... feedbacks são sempre bem-vindos!



Sobre o eBook

Neste eBook vamos abordar a assinatura, sintaxe, casos de uso e tudo o que você precisa saber sobre todos os React Hooks.

Além disso, no rodapé de todas as páginas você encontrará um ícone do YouTube que quando clicado vai te levar para um vídeo onde eu mostro na prática como usar o hook do capítulo que você estiver lendo.

Eu espero que você aproveite e que ao terminar de ler esteja se sentindo confortável pra trabalhar com qualquer um dos hooks disponíveis, e claro, use este eBook e a Playlist no YouTube como um material de consulta sempre que precisar.

Vamos lá?



O que são hooks?

Falando de código, os hooks são apenas funções JavaScript (que sempre começam com a palavra **use**) implementadas na versão 16.8 do React, em 2019, e que são totalmente compatíveis com versões anteriores da biblioteca, ou seja, *no breaking changes here!*

A grande diferença é que estas funções nos permitem usar funcionalidades que antes existiam apenas em classes — como states e lifecycles — agora também em componentes funcionais. Recomendo que você primeiro leia o conteúdo da página antes de partir para o vídeo.

Três características importantes que precisamos saber sobre os hooks são: primeiro, que eles funcionam apenas em componentes funcionais, segundo, que eles devem ser invocados sempre no escopo principal do seu componente (fora de ifs e functions, por exemplo) e terceiro, que eles não podem estar atrelados a nenhuma condicional para serem executados.

Mas você talvez esteja se perguntando: por que eu usaria isso se já tenho estas funcionalidades nas minhas classes? Quais são os benefícios?

Pra isso, vamos primeiro entender as dores...



Reutilização de lógica

Até antes dos hooks, se quiséssemos compartilhar a mesma lógica entre vários componentes, teríamos que escolher entre usar um High Order Component (HOCs) ou Render Props, por exemplo.

Um dos problemas de ambos é que se fossemos implementar um destes conceitos em um componente que já existe, precisaríamos reorganizá-lo para suportar este novo formato.

Além disso, em casos mais complexos, temos o que chamamos de Wrapper Hell, que é o que acontece quando temos muitos aninhamentos de componentes e... bom, o resultado no nosso Dev Tools é algo mais ou menos assim:

```
▼ <Animate animation={enter: enter(), leave: leave(), appe
  ▼ <AnimateChild key="rc_animate_1507887864748" ref=ref()
    ▼ <DOMWrap key="rc_animate_1507887864748" style={} tag='
      ▼ <ul style={} className="ant-menu ant-menu-inline ant
        ▼ <Connect(TestError) key=".$71" ref=chainedFunction
          ▼ <TestError connectionId="c18323cd-c841-430f-90a2-
            ▼ <MenuItem key="71" connectionId="c18323cd-c841-
              ▼ <Tooltip title="" placement="right" overlayCla
                ▼ <Tooltip ref=ref() title="" placement="right"
                  ▼ <Trigger ref=fn() popupClassName="ant-menu-
                    ▼ <MenuItem connectionId="c18323cd-c841-430
                      ▼ <li style={paddingLeft: 48} className="
                        ▶ <Icon type="frown-o">...</Icon>
                          PCS_MISSING
                        ▼ <Connect(ImportPlugin) url="shader">
                          ▼ <ImportPlugin url="shader" dispatch=
                            ▼ <Connect(Plugin) id="76de5df9-ee25-
                              ▼ <Plugin id="76de5df9-ee25-4404-bf
                                ▼ <div style={color: "rgba(0, 0, 0
                                  ▼ <Connect(Element) key="ce6785d
                                    ▼ <Element id="ce6785d9-790a-4e
                                      ▼ <Connect(GroupElement) hand
                                        ▼ <GroupElement handleEvent=
                                          ▼ <Fields handleEvent=fn()
                                            ▼ <div style={width: "100
                                              ▶ <Connect(Row) key="5c
                                              ▶ <Connect(Row) key="30
                                              ▶ <Connect(Row) key="66
                                              ▶ <Connect(Row) key="72
                                              ▼ <Connect(Row) key="cc
                                                ▼ <Row id="ccc62da8-6e
                                                  ▼ <div style={margin
                                                    <div style={flex
```



Componentes muito grandes

Outro problema que enfrentávamos era que a lógica de uma feature, em alguns casos, ficava espalhada em vários métodos do ciclo de vida dos componentes.

Um exemplo claro disso são os `eventListeners`, onde, por exemplo, inicializávamos dentro do `componentDidMount`, ouvíamos por alterações no `componentDidUpdate` para comparar os valores e ver se não seria necessário recriar os nossos listeners e no final de tudo, se não quiséssemos ter problemas na nossa aplicação, precisaríamos remover todos os listeners criados em mais um lifecycle, desta vez no `componentWillUnmount`.

Este é apenas um cenário que pode parecer simples, mas imagine-o se repetindo com várias funcionalidades (não apenas com `eventListeners`) dentro de um componente muito grande e você vai entender o real problema que isso traz para a legibilidade e manutenção do código.

Uma confusão chamada classes

Primeiro de tudo, quando criávamos um componente novo precisávamos pensar: será que este componente vai receber um state em algum momento? E a partir desta resposta, decidíamos se iríamos o declarar como um class ou um functional component.

No caso de estarmos errados e por exemplo, o escrevêssemos como um functional component e posteriormente precisássemos implementar um state nele, este componente precisaria ser reescrito em forma de classe.

Outro ponto é: você realmente entende o que é a variável `this`? E o seu escopo dentro de uma classe? A diferença de seu valor dentro de uma função declarada como uma arrow function e da “forma tradicional” com a keyword function? Como funciona o binding dessa variável?

Além destes pontos citados a cima, as classes também estavam atrapalhando o time de desenvolvimento do React na implementação do hot reload... problema atrás de problema!



Depois da tempestade... os hooks

Bom, agora que já discurremos a cerca dos problemas que enfrentávamos no React antes dos hooks, hora de vê-los funcionando na prática!



Importante

Nos exemplos estarei usando apenas o nome dos hooks (useState, useEffect, etc.). Todos estes hooks são importados de dentro do pacote do React, desta forma:

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




State em componentes funcionais


Este é provavelmente o hook que você mais vai usar.

Como o próprio nome já indica, ele nos dá o “poder” de usar states em componentes funcionais.

A função `useState` recebe um parâmetro (o valor inicial do estado ou uma função que retorna este valor) e retorna um array com duas posições, sendo a primeira delas o valor do estado e a segunda uma função que será responsável por alterar o valor deste estado. Vamos ver um exemplo pra ficar mais claro:

```
function Profile() {  
  const [name, setName] = useState('Mateus');  
  
  return (  
    <input  
      value={name}  
      onChange={e => setName(e.target.value)}  
    />  
  );  
}
```

 state

 função responsável por alterar o state

Uma característica deste hook é que podemos, na função que atualiza o state (`setName`, no exemplo acima), enviar uma função como argumento e nesta função, capturar o valor atual do state (`name`) da mesma forma que fazíamos com o `setState` nas classes.

Esta funcionalidade é muito útil quando precisamos atualizar um estado baseado em seu valor anterior (principalmente em hooks com arrays de dependência, que veremos mais a frente), como por exemplo, em uma função de toggle onde a cada clique o estado alterna entre `true` e `false`, como neste exemplo:



Clique para assistir o vídeo sobre `useState`

```
function Wallet() {  
  const [showBalance, setShowBalance] = useState(true);  
  
  return (  
    <button  
      onClick={() => setShowBalance((prevState) => !prevState)}  
    >  
      Toggle Balance  
    </button>  
  );  
}
```

No primeiro click, **prevState** receberá o valor **true** e alterará o valor de **showBalance** para **false**, depois receberá **false** e setará como **true** novamente e assim sucessivamente por conta da negação (!).



Pra onde foram os lifecycle methods?

Ok, aprendemos a criar os nossos estados e isso já é demais! Mas... pra onde foram os lifecycle methods? Cadê o `componentDidMount`? `componentWillUnmount`? `componentDidUpdate` (e poderíamos passar o resto do dia aqui listando todos os métodos disponíveis)?

Bom, de forma direta e reta, eles foram reduzidos a apenas um hook: o **useEffect**.

O **useEffect** também é uma função, porém, diferente do **useState**, esta não retorna nenhum valor, precisamos apenas invocá-la no corpo do componente.

É um bom lugar para fazer requisições à APIs como fazíamos no `componentDidMount`, por exemplo.

Seu uso é muito simples: ele recebe apenas dois parâmetros: uma função de efeito, e um array de dependências que pode receber variáveis ou funções e sempre que o valor de uma dessas variáveis sofre uma alteração ou uma das função deste array é remontada a função de efeito (primeiro parâmetro) é executada novamente.

Vale lembrar também que toda função e variável (state, prop, ou definida no corpo do componente) que estiver sendo usada dentro da função de efeito deve estar listada no array de dependências.

Vamos para os exemplos:

```
function UserInfo(props) {  
  useEffect(() => {  
    console.log(`O id do usuário mudou para: ${props.userID}`);  
  }, [props.userID])  
}
```



função de efeito



array de dependências, sempre que `props.userID` mudar, a função de efeito será executada novamente

Uma outra característica do **useEffect** é que sua função de efeito não pode ser uma async function, logo, se precisarmos executar este tipo de código dentro dela temos estas duas opções:



Clique para assistir o vídeo sobre useEffect

```
function UserInfo(props) {
  // criando e executando uma função dentro do useEffect...
  useEffect(() => {
    async function loadUserData() {
      await services.getUserByID(props.userID);
    }

    loadUserData();
  }, [props.userID])

  // ou com uma IIFE (Immediately Invoked Function Expression)
  useEffect(() => {
    (async () => {
      await services.getUsers(props.userID);
    })();
  }, [props.userID])
}
```

Mas e se quisermos executar uma requisição à API apenas uma vez, quando o componente é montado em tela (semelhante ao componentDidMount)?

A resposta é muito simples: é só deixar o array de dependências vazio:

```
function UsersList() {
  const [users, setUsers] = useState([]);

  useEffect(() => {
    (async () => {
      const { data } = await services.getUsers();
      setUsers(data.users);
    })();
  }, []); // array vazio, função de efeito executa apenas no primeiro render
}
```

Não paramos por aí... se por algum motivo você quiser que esta função execute toda vez que o componente renderizar é só passar a função de efeito e não passar nada como segundo argumento do useEffect:



```
function App() {
  useEffect(() => {
    console.log('Componente renderizou novamente ... ');
  });
}
```

Honestamente não vejo muitos casos onde você vai precisar disso, mas... pelo menos agora você sabe que pode fazer.

O que vimos até agora serve pra substituir, basicamente, todos os métodos de ciclo de vida das classes, mas faltou um: cadê o `componentWillUnmount`?

Pra termos a mesma funcionalidade dele, ou seja, uma função que executa logo antes do componente sair da tela, basta retornarmos uma função dentro do **useEffect**:

```
function App() {
  useEffect(() => {
    console.log('Componente montou');

    return () => console.log('Componente vai desmontar ... ');
  }, []);
}
```

Muito útil pra removermos eventListeners, por exemplo:

```
function App() {
  useEffect(() => {
    document.addEventListener('scroll', handleScroll);

    return () => document.removeEventListener('scroll', handleScroll);
  }, []);

  function handleScroll(event) {
    console.log('Window scroll ... ', event);
  }
}
```



Nova forma de usar a Context API

Diga adeus ao render props!

Depois da chegada dos hooks, trabalhar com a Context API do React é uma das coisas mais legais de se fazer, na minha opinião.

Até então, nós precisávamos usar uma técnica chamada render props dentro de um componente Consumer sempre que quiséssemos acessar os valores de um contexto. Esta técnica é uma das responsáveis por um dos três problemas que os hooks vieram pra resolver que vimos lá no começo do eBook: o Wrapper Hell .

Vamos apenas relembrar como era o uso da Context API antes da chegada dos hooks:

```
const ThemeContext = createContext();

function App() {
  return (
    <ThemeContext.Provider value={{ mode: 'dark' }}>
      <Button />
    </ThemeContext.Provider>
  );
}

function Button() {
  return (
    <ThemeContext.Consumer>
      {theme => (
        <button className={`_${theme.mode}`}>Themed Button</button>
      )}
    </ThemeContext.Consumer>
  );
}
```

Neste exemplo temos um caso simples, o problema aparece mesmo quando precisamos acessar vários contextos ao mesmo tempo, aí temos uns monstros assim:



Clique para assistir o vídeo sobre useContext

```

const AuthContext = createContext();
const OrgContext = createContext();
const ThemeContext = createContext();

function App() {
  return (
    <AuthContext.Provider value={{ authenticated: true }}>
      <OrgContext.Provider value={{ name: 'DevAcademy' }}>
        <ThemeContext.Provider value={{ mode: 'dark' }}>
          <Button />
        </ThemeContext.Provider>
      </OrgContext.Provider>
    </AuthContext.Provider>
  );
}

function Button() {
  return (
    <AuthContext.Consumer>
      {auth => (
        <OrgContext.Consumer>
          {org => (
            <ThemeContext.Consumer>
              {theme => (
                <button className={theme.mode === 'dark' && 'dark-mode'}>
                  {auth.authenticated ? `Logout from ${org.name}` : 'Login'}
                </button>
              )}
            </ThemeContext.Consumer>
          )}
        </OrgContext.Consumer>
      )}
    </AuthContext.Consumer>
  );
}

```

Claro que aqui dei um exemplo que talvez não seja tão convencional mas que, se fosse necessário, teria que ser feito desta forma. Te convenci que isso era, de fato, um problema?

Então agora vamos reescrever este mesmo código usando o hook useContext:



Clique para assistir o vídeo sobre useContext

```

const AuthContext = createContext();
const OrgContext = createContext();
const ThemeContext = createContext();

function App() {
  return (
    <AuthContext.Provider value={{ authenticated: true }}>
      <OrgContext.Provider value={{ name: 'DevAcademy' }}>
        <ThemeContext.Provider value={{ mode: 'dark' }}>
          <Button />
        </ThemeContext.Provider>
      </OrgContext.Provider>
    </AuthContext.Provider>
  );
}

function Button() {
  const auth = useContext(AuthContext);
  const theme = useContext(ThemeContext);
  const org = useContext(OrgContext);

  return (
    <button className={theme.mode === 'dark' && 'dark-mode'}>
      {auth.authenticated ? `Logout from ${org.name}` : 'Login'}
    </button>
  );
}

```

Muito mais simples, né?

O primeiro ponto a destacar é que a forma como declaramos os nossos contextos não mudou, ou seja, ainda precisamos da função `createContext` e do componente `Provider` por volta de quem vai acessar seus valores.

A mudança mesmo está na forma em que consumimos os contextos: basta invocar a função `useContext` enviando como único argumento o contexto que queremos pegar informações e o hook retorna todos os valores pra gente, sem precisar de render props nem nada do tipo.



Clique para assistir o vídeo sobre `useContext`

useReducer

Talvez esse nome te lembre alguma coisa... pensou em Redux, né? E de fato eles são muito semelhantes (pra não dizer iguais): disparamos uma action e uma função reducer faz a alteração no estado baseado no tipo de action que foi disparada.

Normalmente utilizamos este hook em duas situações: quando nosso código tem uma lógica muito complexa e precisa alterar vários estados ao mesmo tempo, ou quando precisamos que o estado de um componente seja alterado a partir de um filho que está em níveis muito baixos. Neste caso, normalmente nós enviaríamos uma função de callback via props componente por componente até chegar onde gostaríamos (prop drilling), para então, neste componente, executar a função de callback e desencadear as alterações no estado de seu pai. Porém, com o **useReducer**, temos um ganho de performance (pequeno, mas existe) quando enviamos a função dispatch via Context ao invés de um callback via props.

A função useReducer, assim como o useState, retorna um array com duas posições: na primeira o state e na segunda a função dispatch, que será a responsável por disparar as actions.

Partindo agora para sua assinatura, ela pode receber até 3 argumentos, nessa ordem: primeiro a função reducer e depois o estado inicial — que são obrigatórios e são o que você vai usar em quase 100% dos casos — e um terceiro argumento, este opcional, que é uma função que será executada apenas no primeiro render do componente e serve para transformar o estado inicial. Esta função recebe como único argumento o valor que foi informado no estado inicial (segundo argumento passado em useReducer) e retorna um novo valor. Por sua vez, o useReducer assumirá este valor retornado como seu estado inicial, e não mais o valor que foi passado como seu segundo argumento (provavelmente você nunca vai usar isso, mas vamos entender seu funcionamento mesmo assim).

Eu sei, é bastante informação e pode parecer um pouco confuso, mas é bem simples, na verdade.

Vamos aos exemplos que tudo ficará mais claro:



```


const initialState = 1;
function initializer(initialValue) {
  return { count: initialValue };
}


function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      throw new Error('Invalid action type');
  }
}


function App() {
  const [state, dispatch] = useReducer(reducer, initialState, initializer);


  return (
    <div>
      <h1>Valor atual: {state.count}</h1>
      <button onClick={() => dispatch({ type: 'increment' })}>+</button>
      <button onClick={() => dispatch({ type: 'decrement' })}>-</button>
    </div>
  );
}


```

 valor inicial do state

 função que será executada no primeiro render, receberá o valor de `initialState` como argumento e o retorno desta função será o estado inicial do reducer

 função reducer é quem será executada quando a função `dispatch` for invocada ela recebe como argumento o `state` atual e a `action` (valor que foi enviado dentro da função `dispatch` quando invocada)

 `state` é o estado que você já conhece (sempre que mudar... re-render)

 função `dispatch`: recebe um objeto e faz a função `reducer` ser executada, atualizar o `state` e consequentemente fazer um re-render no componente

 exemplo de chamada da função `dispatch`



Clique para assistir o vídeo sobre `useReducer`

Neste snippet tivemos uma “visão geral” do funcionamento do hook, mas vamos trazer isso pra um exemplo da vida real:

```
const initialState = {
  isLoading: true,
  hasError: false,
  users: [],
};

function reducer(state, action) {
  switch (action.type) {
    case 'success':
      return {
        isLoading: false,
        hasError: false,
        users: action.payload.users,
      };
    case 'error':
      return {
        isLoading: false,
        hasError: true,
        users: [],
      };
    default:
      throw new Error('Invalid action type');
  }
}

function Users() {
  const [state, dispatch] = useReducer(reducer, initialState);

  useEffect(() => {
    (async () => {
      try {
        const { data } = await services.getUsers();
        dispatch({ type: 'success', payload: { users: data } });
      } catch (err) {
        dispatch({ type: 'error' });
      }
    })();
  }, []);
}
```



Clique para assistir o vídeo sobre useReducer

Performance e memorização

Vamos imaginar um cenário onde criamos um componente Button que recebe apenas uma função onClick como propriedade, teríamos algo mais ou menos assim:

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

Parece tudo bem, mas se esta função faz uma alteração de state no componente pai, o nosso componente Button vai ser renderizado novamente a cada clique, o que não deveria acontecer, afinal de contas o Button só precisa da função onClick pra existir e se ela não foi alterada, ele não precisa ser renderizado novamente. Vamos ver isso na prática:



```
function App() {
  const [state, setState] = useState('');

  function handleClick () {
    console.log('State updated');
    setState(Math.random());
  }

  return (
    <Button onClick={handleClick} />
  );
}

function Button(props) {
  console.log('Button re-rendered');
  return (
    <button onClick={props.onClick}>
      Clique
    </button>
  );
}
```

Clicando 5 vezes neste botão o resultado no console foi esse:

```
State updated
Button re-rendered
State updated
Button re-rendered
State updated
Button re-rendered
State updated
Button re-rendered
State updated
Button re-rendered
>
```



Clique para assistir o vídeo sobre useCallback

O Button é renderizado a cada novo clique devido a alteração de state em seu pai, quando, na verdade, ele só precisaria ser renderizado novamente se o onClick fosse alterado.

Se você já é um pouco mais experiente em React talvez esteja pensando: aí é só usar o React.memo ou um PureComponent e o problema está resolvido, não é?

Bom, na verdade não.

Antes de mais nada, se você não conhece o React.memo, ele é uma função do React que memoriza um componente e o faz sofrer novas renderizações apenas se suas props forem alteradas e não sempre que seu pai é renderizado. É bem fácil usá-lo, e no caso do nosso Button, ele ficaria assim:

```
const Button = React.memo((props) => {  
  console.log('Button re-rendered');  
  return (  
    <button onClick={props.onClick}>  
      Clique  
    </button>  
  );  
});
```

Mas se testarmos este código teremos exatamente o mesmo resultado de antes: Button sofrendo re-renders a cada novo clique.

E por que isso acontece?

Primeiro, vamos dar um passo atrás: em componentes de classe sempre que um estado ou uma prop sofre uma alteração o método render é executado novamente, certo?

Em componentes funcionais não existe um método render, a função que é executada sempre que o componente sofre um re-render é o próprio componente.

Dito isso, sempre que o Button é clicado e altera o state do componente App, a função App() é executada novamente e, consequentemente, a função handleClick é re-criada, o que quer dizer que tem uma nova referência na memória, e é por isso que mesmo usando o React.memo caímos naquele problema de múltiplos renders desnecessários no nosso Button.

E aí entra o hook que abordaremos a seguir.



Evitando renderizações desnecessárias

O hook que veremos agora é o `useCallback`. Ele serve para memorizar funções e prevenir que elas sejam criadas a cada novo render do componente.

Este hook recebe apenas 2 argumentos: o primeiro é a função que queremos memorizar e o segundo é um array de dependências que, assim como no `useEffect`, sempre que um dos valores deste array mudar fará o hook executar novamente e, conseqüentemente, recriar a função (e mais uma vez: todos os valores externos (props, states, etc.) que forem usados dentro da função memorizada devem estar declarados no array de dependências).

```
const handleClick = useCallback(() => {  
  // regra de negócio  
}, []);
```



função que será memorizada



array de dependências, mesmo funcionamento do `useEffect`



Clique para assistir o vídeo sobre `useCallback`

Agora vamos ver como aquele nosso exemplo do Button ficaria com o **useCallback**:

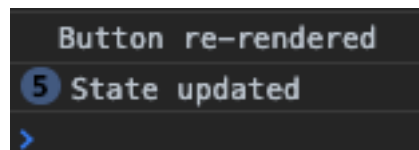
```
function App() {
  const [state, setState] = useState('');

  const handleClick = useCallback(() => {
    console.log('State updated');
    setState(Math.random());
  }, []);

  return (
    <Button onClick={handleClick} />
  );
}

const Button = React.memo((props) => {
  console.log('Button re-rendered');
  return (
    <button onClick={props.onClick}>
      Clique
    </button>
  );
});
```

E o resultado que temos no console após os mesmos 5 cliques agora é este:

A screenshot of a console window with a dark background. It shows two log entries. The first entry is 'Button re-rendered' in a light gray font. The second entry is '5 State updated' in a light blue font, with a blue circle containing the number '5' to its left. Below the second entry is a blue prompt character '>'.

```
Button re-rendered
5 State updated
>
```

Button renderizado apenas uma vez!

E talvez você esteja cogitando usar o **useCallback** em **todas** as funções que declarar no seu componente pra memorizar tudo e não ficar as re-criando a cada render. Acertei né?

Direto ao ponto: não é necessário. A recomendação da própria documentação oficial do React diz que o ideal é usar o **useCallback** apenas nestes casos de otimização de componentes filhos (seu próprio nome já diz, **useCallback** e não **useFunction**).

Memorização pode, inclusive, custar mais caro, ou seja: o tiro pode sair pela culatra. Então fica aqui a dica pra sua carreira como dev: não tente fazer otimizações prematuras.



Clique para assistir o vídeo sobre **useCallback**

Removendo cálculos pesados de dentro do render

Como mencionei aqui em cima, nas classes sempre que um state ou uma prop é alterada o método render é executado. Em decorrência deste fato, devemos sempre evitar colocar dentro deste método cálculos pesados (matemáticos, manipulação de datas, etc.) que não precisam ser recalculados em cada novo render.

Em classes este problema é facilmente resolvido fazendo estes cálculos e tratamentos de dados mais pesados fora do método render. Mas e nos componentes funcionais, que como vimos anteriormente, executam todo o seu corpo a cada novo render? Como isolamos estes cálculos?

Aí entra o useMemo. A assinatura dele é igual à do useCallback: recebe uma função e um array de dependências que funciona da mesma forma que já vimos anteriormente (só pra reforçar: sempre que um valor mudar o hook executa novamente e se passarmos um array vazio executa apenas no primeiro render). A diferença aqui é que ao invés de retornar a função passada no primeiro parâmetro como no useCallback, o useMemo, por sua vez, executa esta função e retorna o seu resultado:

```
function App() {  
  const [state, setState] = useState(Math.PI);  
  
  const memoizedValue = useMemo(() => {  
    return veryComplexMathCalc(state);  
  }, [state]);  
  
  return <h1>Valor memorizado: {memoizedValue}</h1>;  
}
```



variável que receberá o retorno da função



função que será executada e retornará o valor a ser memorizado



array de dependências



Clique para assistir o vídeo sobre useMemo

Acessando elementos na DOM por referências

Em alguns casos precisamos acessar elementos através de referências na nossa árvore (DOM) — como por exemplo, para pegar valores de inputs quando estamos trabalhando com uncontrolled components, ou então forçar focus, blur, este tipo de coisa.

Nas classes basta passar uma propriedade `ref`, que recebe uma função com o elemento como seu único argumento (`el`, no exemplo a baixo) e salvamos este elemento em uma propriedade na variável `this` da classe:

```
<input ref={el => this.input = el} />
```

Mas nos componentes funcionais nós não temos acesso a variável `this`, e aí, onde é que salvamos estas referências?

A resposta é: `useRef`.

Ele também é muito fácil de usar, basta o atribuirmos à uma variável, passar um valor inicial como argumento (no exemplo, o `null`) e enviar esta variável na propriedade `ref` do elemento:

```
function App() {  
  const inputRef = useRef(null);  
  
  return (  
    <input ref={inputRef} />  
  );  
}
```

Bem fácil, né?



Clique para assistir o vídeo sobre `useRef`

Agora uma outra característica é que, seguindo os exemplos à cima, diferente das refs nas classes onde dentro de `this.input` temos a referência direta para o nosso target, ou seja, se quisermos acessar a propriedade `value` do input, basta usarmos:

```
this.input.value
```

No `useRef` temos um objeto e a referência do target fica dentro de uma propriedade chamada `current`. Então para obtermos o `value` do nosso input, usamos:

```
inputRef.current.value
```

Apenas para ficar ainda mais claro, vamos ver um exemplo aplicado em um componente real:

```
function App() {
  const inputRef = useRef(null);

  function handleClick() {
    alert(inputRef.current.value);
  }

  return (
    <div>
      <input ref={inputRef} />
      <button onClick={handleClick}>Obter valor</button>
    </div>
  );
}
```




useImperativeHandle e forwardRef

Em casos de uso de código imperativo — o que é recomendável evitar mas as vezes se faz necessário — conseguimos manipular o valor da referência de um componente React usando o **useImperativeHandle**.

Antes de mais nada, este hook só funciona junto do `forwardRef`, que já existia nas versões anteriores do React e serve para encaminhamento de referências, em outras palavras, definir referências para componentes React assim como fazemos com elementos HTML.

Vamos primeiro entender a estrutura do `forwardRef`, que é, na verdade, muito simples: ele é uma função que recebe outra função (um componente) como argumento:

```
const Form = React.forwardRef(() => {  
  return (  
    <form>  
      Formulário  
    </form>  
  );  
});
```

 componente react

Diferente dos componentes convencionais que já conhecemos, quando estamos usando o `forwardRef` o nosso componente recebe, além das props, um segundo argumento: a referência que foi definida na propriedade `ref` quando o componente `Form` foi renderizado.



```
function App() {
  const formRef = useRef(null);

  return (
    <Form ref={formRef} />
  );
}

const Form = React.forwardRef((props, ref) => {
  return (
    <form>
      Formulário
    </form>
  );
});
```

Pronto. É assim que declaramos componentes usando o forwardRef. Bem tranquilo, né?

Agora, como mencionei nas primeiras linhas deste capítulo, usando o useImperativeHandle conseguimos definir quais serão os valores e métodos que estarão disponíveis dentro da referência do componente.

A estrutura deste hook é muito semelhante ao que já vimos anteriormente: o primeiro parâmetro é a ref (que veio do segundo argumento do nosso componente), o segundo é uma função que retorna um objeto (este objeto será o valor da nossa ref) e por último, e não menos importante, um array de dependências que funciona da exata mesma forma que já vimos no useEffect, useCallback e useMemo:

```
useImperativeHandle(ref, () => ({
  key: 'value',
}), []);
```

Agora aplicando isso no exemplo anterior do Form, faríamos desta forma:



```

function App() {
  const formRef = useRef(null);

  return (
    <
      <Form ref={formRef} />
      <button onClick={() => formRef.current.submit()}>
        Submit
      </button>
    </>
  );
}

const Form = React.forwardRef((props, ref) => {
  useImperativeHandle(ref, () => ({
    submit: () => {
      console.log('Submit invoked!');
    },
  })), []);

  return (
    <form>
      Formulário
    </form>
  );
});

```



useEffect vs useLayoutEffect

Tudo o que já aprendemos sobre o `useEffect` se aplica também para o `useLayoutEffect`. Literalmente tudo. Função de efeito, array de dependências e todas aquelas outras características de funcionamento que já foram abordadas anteriormente. Eles são exatamente iguais.

A única diferença de um pro outro é o tempo de execução, ou seja, em que momento do ciclo de vida do componente cada um vai ser chamado.

Enquanto o `useEffect` executa após o componente ser renderizado e “printado” no navegador, o `useLayoutEffect` é executado antes do browser receber as atualizações, ou seja, antes do usuário ver em tela o resultado de uma atualização de estado, por exemplo.

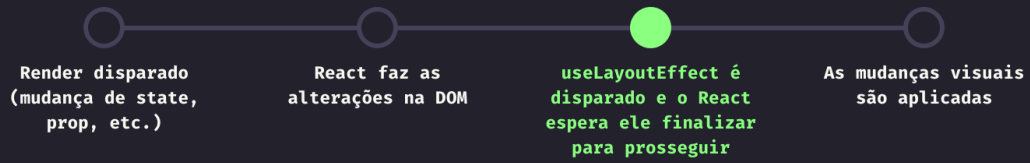
Outra característica é que diferente do `useEffect` que é assíncrono, o `useLayoutEffect` é síncrono, o que quer dizer que o React vai esperar a função de efeito do `useLayoutEffect` terminar para só então pular para o próximo stage da timeline do lifecycle do componente, que no caso é a etapa onde o browser recebe as atualizações e atualiza o componente em tela. Por isso, prefira sempre usar o `useEffect` para evitar bloqueios nas atualizações visuais dos seus componentes.

Preparei duas ilustrações para as coisas ficarem mais claras:



Clique para assistir o vídeo sobre `useLayoutEffect`

Lifecycle do `useLayoutEffect`



Clique para assistir o vídeo sobre `useLayoutEffect`

Custom Hooks

Já abordamos praticamente todos os hooks, resta apenas um, mas... antes dele vamos conversar um pouquinho sobre os Custom Hooks, uma das coisas mais legais que os hooks trouxeram.

Indo direto ao ponto, nós podemos usar qualquer hook em qualquer função JavaScript, ou seja, pra usar `useState`, `useEffect`, ou qualquer outro hook, nós não precisamos criar um componente, basta criar uma função JavaScript comum, colocar toda a regra de negócio lá dentro e sair usando onde e quantas vezes quiser. Chega de High Order Functions e Render Props...

Pra exemplificar, vamos criar um hook chamado `useLocalStorage`, que será um “`useState` turbinado”. A ideia é: sempre que atualizarmos o valor do state iremos salvar as novas informações no `localStorage`. Além disso, também inicializaremos seu valor verificando se há algo que foi previamente salvo:

```
function useLocalStorage(key, initialValue = '') {
  const [state, setState] = useState(() => {
    const storedData = localStorage.getItem(key);

    if (storedData) {
      // Se houver algo salvo, retorna ...
      return JSON.parse(storedData);
    }

    // Se não houver nada salvo, retorna initialValue ...
    return initialValue;
  });

  useEffect(() => {
    // Toda vez que state for alterado, salva no localStorage ...
    localStorage.setItem(key, JSON.stringify(state));
  }, [key, state]);

  // Deixamos o state e o setState acessíveis para
  // quem for usar este hook ...
  return [state, setState];
}
```



O código é autoexplicativo, estamos apenas usando os conceitos que já vimos nos capítulos de `useState` e `useEffect`.

E agora é só usar onde quiser:

```
function App() {  
  const [name, setName] = useLocalStorage('name');  
  
  return (  
    <input value={name} onChange={e => setName(e.target.value)} />  
  );  
}
```

É ou não é uma feature fantástica pra abstrair regra de negócio? É muito massa!

E só pra fecharmos: assim como todos os hooks nativos do React, **os custom hooks precisam sempre começar com a palavra "use"**.



useDebugValue

O último hook que temos disponível — pelo menos até a versão 16.13.1 — é o `useDebugValue`. Ele é bem simples tanto em sintaxe e assinatura quanto em funcionamento.

Ele serve apenas para mostrar um label (um texto qualquer) ao lado do nome dos Custom Hooks dentro do React DevTools.

Podemos usa-lo de duas formas: a primeira e mais simples é apenas chamá-lo enviando uma string como único argumento.

Para exemplificar, vamos usar o hook `useLocalStorage` que criamos no capítulo anterior:

```
function useLocalStorage(key, initialValue = '') {
  const [state, setState] = useState(() => {
    const storedData = localStorage.getItem(key);

    if (storedData) {
      return JSON.parse(storedData);
    }

    return initialValue;
  });

  useEffect(() => {
    localStorage.setItem(key, JSON.stringify(state));
  }, [key, state]);

  useDebugValue(`Debug: ${state}`);

  return [state, setState];
}
```

Isso vai gerar o seguinte resultado no DevTools:



The screenshot shows the React DevTools hooks panel. It lists a hook named 'hooks' with a sub-entry 'LocalState: "Debug: Mateus"'. A small icon of a monitor is visible in the top right corner of the panel.



Clique para assistir o vídeo sobre `useDebugValue`

A segunda forma de usar este hook é enviando uma função de formatação (opcional) como segundo parâmetro. Esta função recebe como único argumento o texto enviado no primeiro parâmetro do hook, no nosso exemplo seria o “Debug: \${state}”, e o valor que ela retornar é o que vai aparecer no DevTools.

```
useDebugValue(`Debug: ${state}`, message => {  
  return message.toUpperCase();  
});
```

E o resultado final é este:



```
hooks  
  ▶ LocalState: "DEBUG: MATEUS"
```

É interessante que você use esta função sempre que precisar fazer cálculos pesados, formatações, enfim, operações muito caras, pois ela será executada apenas quando um hook estiver sendo inspecionado, o que poupa recursos e melhora a performance da aplicação.



Conclusão

Em poucas páginas conseguimos abordar todos os hooks disponíveis no React (até a v16.13.1, última lançada enquanto escrevo este eBook).

Todos eles são muito simples de entender e mais ainda de usar, porém, quando combinados, entregam um poder absurdo pro desenvolvedor (principalmente quando estamos falando de Custom Hooks) e tornam o React muito mais prazeroso de se trabalhar.

Eu espero que este eBook tenha sido útil pra você e que assim como eu disse lá no começo, você use ele e a Playlist no YouTube como guias sempre que surgir alguma dúvida.

E mais uma vez: muito obrigado!

