

# React Hooks

Adicionado na versão 16.8

Atualmente o React está na versão 16.12

# Não há planos para remoção das classes

São completamente opcionais

100% retrocompatíveis

# O que será apresentado

- Motivação
- Principais hooks
- Regras gerais
- Exemplo comparativo de classe vs hook
- Uso de hooks na prática



# Por que criaram os hooks?

- React não possui uma maneira nativa de vincular um comportamento reutilizável em um componente.
  - **RenderProps:** Passar uma função via props que retorna um elemento React.
  - **HOC:** É uma função que recebe um componente e retorna outro, agora com algo a mais. (connect do Redux, withFormik do Formik, withStyles do Material-UI).



# Exemplos de HOC e RenderProps

```
const mapStateToProps = state => ({ ...      HOC
})

export default withRouter(connect(mapStateToProps)(App))
```

```
<Grid container spacing={1} className={classes.formContent}>      RenderProps
  <Parcelas
    disabled={desabilitarCampos}
    quantidadeParcelas={values.quantidadeParcelas}
    diasParcelas={values.diasParcelas}
    onChange={arrayParcelas => setFieldValue("diasParcelas", arrayParcelas)}
  />
</Grid>
```

# Por que criaram os hooks?

- Componentes de classes grandes são difíceis de entender
  - Temos que manter componentes que **começam simples** mas **depois** de um tempo **viram uma bagunça de lógica** com estado e efeitos colaterais (ciclos de vida).
  - Normalmente cada método de ciclo de vida possui **lógicas que não se relacionam**.
  - Em muitos casos **não é possível** quebrar em pedaços menores já que há lógica com os estados por todo lado.



# Por que criaram os hooks?

- Classes são confusas
  - As classes dificultam o aprendizado do React.
  - Somos obrigados a aprender o uso do **this**.
  - Temos que lembrar de fazer o **bind** dos event handlers.
  - A distinção entre componentes Stateless e Statefull acabam levando a desentendimentos.

**Hooks permitem usar mais das funcionalidades do React sem classes**



# Principais Hooks

useState

useEffect

---



# useState

- **O que o useState faz?**

- Ele declara uma variável de estado.

- **O que passamos para o useState como argumentos?**

- Um estado inicial. Não precisa ser um objeto, como nas classes.

- **O que o useState retorna?**

- Ele retorna um par de valores: o **estado atual** e uma **função** que **atualiza o state**. Estes valores são capturados via desestruturação(uma vez que o retorno é um array).



# Usando estado com classe

```
class Example extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {  
      count: 0  
    };  
  }  
  
  render() {  
    return (  
      <div>  
        <p>You clicked {this.state.count} times</p>  
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>  
          Click me  
        </button>  
      </div>  
    );  
  }  
}
```

# Usando estado com Hooks

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

function Example() {
  // Declarar uma nova variável de state, na qual chamaremos de "count"
  const [count, setCount] = useState(0);


  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

## Outros exemplos do useState

```
var fruitStateVariable = useState('banana'); // Retorna um par
var fruit = fruitStateVariable[0]; // Primeiro item do par
var setFruit = fruitStateVariable[1]; // Segundo item do par
```

```
function ExampleWithManyStates() {
  // Declarar múltiplas variáveis de state!
  const [age, setAge] = useState(42);
  const [fruit, setFruit] = useState('banana');
  const [todos, setTodos] = useState([{ text: 'Learn Hooks' }]);
```

# useEffect

- Ele permite executar efeitos colaterais em um componente, semelhante aos ciclos de vida nos componentes de classes.
  - Podemos entender o useEffect como sendo os ciclos de vida **componentDidMount**, **componentDidUpdate** e **componentWillUnmount** combinados.
  - Existem dois tipos de efeitos colaterais, os que precisam de limpeza e os que não precisam.
- 

# useEffect


- **O que o useEffect faz?**

- Executa alguma coisa depois da renderização.

- **Ele executa depois de toda renderização?**

- Por padrão **sim!** Ele executa depois da primeira renderização e depois de toda atualização.

Ao contrário do `componentDidMount` ou `componentDidUpdate`, efeitos agendados com `useEffect` **não bloqueiam** o navegador. Nos casos em que se faz necessário o bloqueio, podemos usar o `useLayoutEffect`.



# Efeitos colaterais sem limpeza usando classe

```
class Exemplo extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {  
      count: 0  
    };  
  }  
  
  componentDidMount() {  
    document.title = `Você clicou ${this.state.count} vezes`;  
  }  
  
  componentDidUpdate() {  
    document.title = `Você clicou ${this.state.count} vezes`;  
  }  
  
  render() {  
    return (  
      <div>  
        <p>Você clicou {this.state.count} vezes</p>  
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>  
          Click me  
        </button>  
      </div>  
    );  
  }  
}
```

# Efeitos colaterais sem limpeza usando hooks

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

function Example() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `Você clicou ${count} vezes`;
  });

  return (
    <div>
      <p>Você clicou {count} vezes</p>
      <button onClick={() => setCount(count + 1)}>
        Clique aqui
      </button>
    </div>
  );
}
```



# Efeitos colaterais com limpeza usando classe

Observem como o **DidMount** e **WillUnmount** se espelham.

Métodos de ciclo de vida obrigam a dividir essa lógica, mesmo quando conceitualmente o código dos dois é relacionado ao mesmo efeito

```
class FriendStatus extends React.Component {
  constructor(props) {
    super(props);
    this.state = { isOnline: null };
    this.handleChange = this.handleChange.bind(this);
  }

  componentDidMount() {
    ChatAPI.subscribeToFriendStatus(
      this.props.friend.id,
      this.handleChange
    );
  }

  componentWillUnmount() {
    ChatAPI.unsubscribeFromFriendStatus(
      this.props.friend.id,
      this.handleChange
    );
  }

  handleChange(status) {
    this.setState({
      isOnline: status.isOnline
    });
  }

  render() {
    if (this.state.isOnline === null) {
      return 'Loading...';
    }
    return this.state.isOnline ? 'Online' : 'Offline';
  }
}
```

# Efeitos colaterais com limpeza usando hooks

Se o efeito retornar  
uma função, o hook  
irá executá-lo quando  
for a hora de limpar

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

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    // Especifique como limpar depois desse efeito:
    return function cleanup() {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

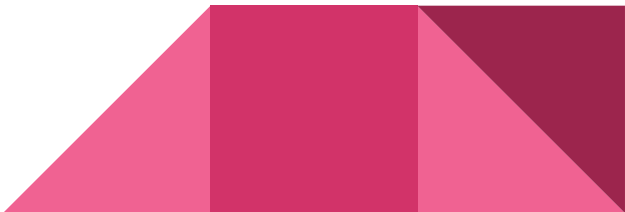
  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

# Recapitulando

**useState** nos permite adicionar variáveis ao estado do componente.

```
// Declarar uma nova variável de state, na qual chamaremos de "count"  
const [count, setCount] = useState(0);
```

**useEffect** nos deixa expressar diferentes tipos de efeitos colaterais depois que o componente renderiza. Alguns precisam de limpeza, por isso eles retornam uma função.



# Recapitulando

```
useEffect(() => {  
  function handleStatusChange(status) {  
    setIsOnline(status.isOnline);  
  }  
  
  ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);  
  return () => {  
    ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);  
  };  
});
```

outros não precisam ter a fase de limpeza, e não retornam nada.

```
useEffect(() => {  
  document.title = `Você clicou ${count} vezes`;  
});
```



# Dicas para usar o useEffect

- Use vários efeitos para separar as responsabilidades.
  - Assim como podemos usar vários useStates, podem usar vários useEffect
- Otimizando a performance
  - Em alguns casos ao aplicar o efeito em cada renderização pode causar problemas de desempenho. Em classes nós comparamos o prevProps com o this.props dentro do componentDidUpdate.
  - Para resolver esse problema, passamos um segundo argumento para o useEffect. Um array contendo o valor da variável a ser ouvida. Por exemplo:

```
useEffect(() => {  
  document.title = `Você clicou ${count} vezes`;  
}, [count]); // Apenas re-execute o efeito quando o count mudar
```

# Dicas para usar o useEffect

- Use hooks apenas em funções React
  - Se usarmos hooks dentro de funções javascript, podemos pular a execução de algum efeito, o que causará bugs
  - Isso acontece porque o **React depende da ordem em que os hooks são chamados.**



# Outros hooks

- useContext
- useReducer
- useCallback
- useMemo
- useRef
- useImperativeHandle
- useEffect
- useDebugValue

