REACT.

MATERIAL DE APOIO

Disciplina: React

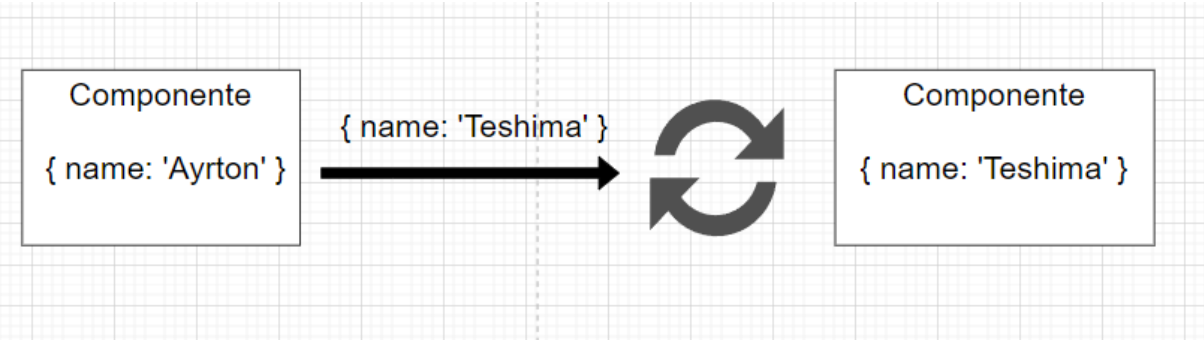
Nome da aula: React Hooks

Professor(a): Ayrton Teshima

Tutor(a): Rodrigo Vasconcelos

Introdução

O que impulsionou o React, além do desenvolvimento declarativo e baseado em componentes, foi sua reatividade. Essa definitivamente é uma das coisas mais interessantes do React, ele “automaticamente” reflete as atualizações na tela para você.



Fonte: autoral

Na imagem acima, podemos ver um componente que possui a informação *name*. Quando essa informação é alterada, neste caso para ‘Teshima’, o React reflete o novo *name* e o renderiza na tela.

É agora que começamos a trabalhar com o estado do componente.

Objetivos da aula

- Aprender a dar dinamismo aos seus projetos
- Explorar o sistema reativo do React, que é uma das suas principais propostas como biblioteca
- Compreender o React Hooks

Resumo

descomplica

Até o momento vimos todos os componentes estáticos. Agora, imagina que de forma dinâmica, o conteúdo precisa ser alterado e essa informação ser refletida na tela do usuário. Isso acontece a todo tempo em todo lugar.

Exemplos de reatividade em aplicações web:

- Se você está no Google Flight e clica para abrir as opções de um voo, a tela reage a esse clique e se transforma para mostrar as novas informações;
- Se você seleciona um filme no Netflix, a tela reage a isso e abre uma nova tela com detalhes do filme;
- Se você selecionar o assento do cinema ou do ônibus, a tela vai refletir e vai marcar o assento para você;
- Você scrollando o Instagram, quando já acabou as informações previamente carregadas, ele precisa fazer uma requisição para buscar novos posts e, no retorno dessa requisição, a tela renderiza (reage) esses novos posts.

Existem inúmeras situações que a tela vai reagir de acordo com uma ação/input.
No React, você programa a tela para como ela vai se apresentar quando a informação mudar e não o passo a passo de como a mudança acontece.

Reatividade com classes

O React dá suporte ao sistema reativo, tanto para componentes feitos em classes quanto para componentes criados com funções através dos hooks que veremos mais à frente. Vamos ver inicialmente com classes caso você ainda se depare com códigos de componentes React utilizando.

Quando você cria um componente com classes, você precisa herdar da classe *Component* do React.

```
import React from 'react';

class Assento extends React.Component {

}
```

Fonte: autoral

Fazendo isso, você ganha acesso a dois recursos que permitem você gerenciar o estado do componente. São eles:

- Propriedade ***state*** que permite você ter acesso ao estado do componente;
- Método ***setState*** que permite você atualizar o estado passando um novo objeto.

Podemos definir o estado inicial do nosso componente através do método construtor da classe.

```
import React from 'react';

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

  render() {
    <button disabled={this.state.disabled}>Botão</button>
  }

}
```

Fonte: autoral

Na imagem acima, definimos que o estado inicial do componente terá apenas uma propriedade que é a *disabled,* definida como *false*. Veja que no método *render* já utilizamos o *state* acessando como uma propriedade normal de classe *this.state.disabled*.

Toda vez que alterarmos o valor do *state* com o método *setState*, automaticamente o React vai renderizar o componente com essa nova informação.

Exemplo prático

Vamos dar continuidade a nossa classe *Assento* e criar a funcionalidade de, ao selecionar um assento de ônibus, ele ficar desabilitado, como se você estivesse comprando aquela poltrona para uma viagem.

```
import React from 'react';

class Assento extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      disabled: false
    }
  }
  handleClick() {
    this.setState({
      disabled: true
    })
  }

  render() {
    return (
      <button
        disabled={this.state.disabled}
        type="button"
        onClick={this.handleClick.bind(this)}
        className="assento"
        >
          {
            this.state.disabled
              ? <span>X</span>
              : <strong>{this.props.posicao}</strong>
          }
        </button>
    )
  }
}
```

Fonte: autoral

No código do componente acima, criamos um botão que, ao ser clicado (evento *onClick*), vai invocar o método *handleClick,* que vai atualizar o estado do componente. Com isso, antes o botão estava habilitado e agora passa a estar desabilitado, pois o valor de *disabled* agora é *true*.

Veja que em nenhum momento após atualizar o estado, mandamos renderizar na tela o botão desabilitado. O React se encarrega de refletir as mudanças na tela para gente. Observe também que nosso componente *Assento* recebe uma propriedade, que é a *posição*, que serve para indicar o número do assento.

Reatividade com Hooks

Agora vamos nos concentrar em como parar de utilizar a classe para gerenciar o estado e substituir por um hook. Vamos concentrar em alterar nosso componente feito em classe para um feito com função mantendo a reatividade utilizando hooks.

Os hooks nativos do React, ou mesmo os criados por você, tem uma nomenclatura iniciada pelo prefixo 'use', para facilmente identificar que aquilo é um hook. Logo mais vamos nos aprofundar nos hooks em si, por enquanto vamos nos ater ao gerenciamento do estado.

Exemplo prático

Vamos converter nosso componente de classe em função. Veja que no exemplo abaixo estamos renderizando nosso botão, mas sem passar o valor de *disabled* de forma dinâmica e sem implementar a função *handleClick* que deve atualizar o estado.

```
const Assento = (props) => {
  const handleClick = () => {

  }

  return (
    <button
      disabled={false}
      type="button"
      onClick={handleClick}
      className="assento"
      >
        {
          disabled
            ? <span>-</span>
            : <strong>{props.posicao}</strong>
        }
      </button>
  )
}
```

Fonte: autoral

```
import { useState } from 'react'

const Assento = (props) => {
  const [ disabled, setDisabled ] = useState(false);

  const handleClick = () => {

  }

  return (
    <button
      disabled={false}
      type="button"
      onClick={handleClick}
      className="assento"
      >
        {
          disabled
            ? <span>-</span>
            : <strong>{props.posicao}</strong>
        }
      </button>
  )
}
```

Fonte: autoral

O primeiro passo é importar o invocar o hook *useState*, que é o responsável pelo estado do componente:

Você invoca o hook *useState* e passa o valor inicial do estado, no nosso caso é *false* pois estamos trabalhando com o valor *disabled* de um botão, que só recebe valores *true* e *false.*

Diferente do *state* de classes, no geral, no hook *useState,* você trabalha com uma única informação. Ou seja, se você precisa gerenciar várias informações diferentes no componente, você provavelmente vai ter várias chamadas do *useState* dentro do seu componente.

A invocação do hook *useState* retorna um *array* com dois valores, o valor do estado e a função que modifica o estado. Veja que estamos aplicando a desestruturação de *array* para pegar os valores retornados da função *useState*.

Agora basta utilizar a informação de estado *disabled* e invocar a função que modifica ela:

```jsx
import { useState } from 'react'

const Assento = (props) => {
  const [ disabled, setDisabled ] = useState(false);

  const handleClick = () => {
   setDisabled(true);
  }

  return (
    <button
      disabled={disabled}
      type="button"
      onClick={handleClick}
      className="assento"
      >
        {
          disabled
            ? <span>-</span>
            : <strong>{props.posicao}</strong>
        }
      </button>
  )
}
```

Fonte: autoral

Veja que no evento de click, sempre invocamos a função *handleClick*, que chama a função modificadora do estado *setDisabled* alterando *disabled* para *true.* A partir desse momento, o *disabled* é *true* e o botão passa a estar desabilitado.

Como aplicar na prática o que aprendeu

Criando seu próprio hook

Criar seus próprios hooks é um recurso muito poderoso que o React nos oferece. Antes você só conseguia criar e compartilhar componentes. Porém, um componente significa ter uma estrutura de elementos HTML sendo trazidos junto e muitas vezes a gente não quer isso.

Algumas coisas interessantes que você precisa saber sobre criação de hooks:

Hooks podem e devem utilizar outros hooks (principalmente os nativos do React);

Hooks podem e devem retornar estruturas de dados e primitivos JavaScript, como um array, objeto, string, booleano, etc; Os seus hooks também, por convenção, devem iniciar com o prefixo ‘use’.

Vamos criar um hook que faz uma busca por CEP e retorna um objeto com o endereço.

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

export const useCEP = (cep) => {
  const [ endereco, setEndereco ] = useState({});

  function fetchCEP(cep) {
    return fetch(`https://viacep.com.br/ws/${cep}/json/`)
      .then(d => d.json())
  }

  useEffect(() => {
    fetchCEP(cep).then(d => {
      setEndereco(d);
    })
  }, [cep]); // Para buscar sempre que receber um novo cep

  return endereco;
}
```

Fonte: autoral

Veja que o hook é uma função simples que utiliza outros hooks do React. Os mais utilizados em geral são os hooks *useEffect* e *useState*.

No código acima criamos um hook que você invoca passando um CEP e ele retorna um endereço. Observe que a busca pelo endereço do CEP é uma operação assíncrona, ou seja, ela não retorna imediatamente, precisa esperar o tempo da requisição. Por isso, utilizamos o hook *useState* para popular com os dados do endereço sempre que um novo endereço é atualizado.

Também utilizamos o hook *useEffect* para que ele monitore o parâmetro do hook CEP e consiga realizar uma nova busca sempre que o valor passado para o hook seja um novo.

Basicamente acontece o seguinte:

O hook recebe um CEP;

O hook *useEffect* percebe esse novo valor e faz o fetch para a API de CEP;

Quando o endereço é retornado da API, ele atualiza o estado do Hook;

O valor do estado *endereco* é o valor que o hook retorna;

Agora podemos reutilizar essa lógica de busca CEP em qualquer novo componente, por exemplo:

```
import { useCEP } from './hooks/useCEP'; // importando o hook que criamos

export const ViaCEP = ({ cep }) => {
  const endereco = useCEP(cep);

  return (
    <table>
      <tbody>
        <tr>
          <td>Bairro</td>
          <td>{endereco.bairro}</td>
        </tr>
        <tr>
          <td>Complemento</td>
          <td>{endereco.complemento}</td>
        </tr>
        <tr>
          <td>UF</td>
          <td>{endereco.uf}</td>
        </tr>
        <tr>
          <td>Localidade</td>
          <td>{endereco.localidade}</td>
        </tr>
      </tbody>
    </table>
  )
}
```

Fonte: autoral

Esse é o nosso componente *ViaCEP* que utiliza o hook *useCEP*. Sempre que esse componente receber um novo *cep* em suas propriedades, o hook vai ser chamado novamente retornando o novo endereço.

O componente poderia alterar o CEP por contra própria ao invés de receber como propriedade, assim funcionaria da mesma forma. Para fazer isso, basta que seu CEP esteja no estado do componente para que quando ele atualizar, você passe para o hook novamente.

Conteúdo bônus

Tópicos avançados

Se aprofundando em hooks

O que são hooks?

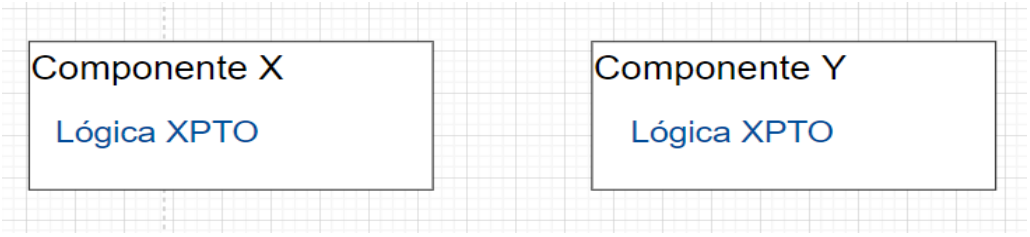
Toda forma de desenvolver vem para solucionar problemas, mas trazem novos desafios, e com React não foi diferente. *Hooks* foi algo que surgiu somente na versão 16, ou seja, depois de anos de React existir.

Conforme vamos desenvolvendo aplicações complexas e sofisticadas, nossos componentes vão ficando gigantes. Isso tem diversas consequências negativas, algumas são:

- Muito código acoplado ao componente
- Difícil de compartilhar lógica e comportamento entre componentes

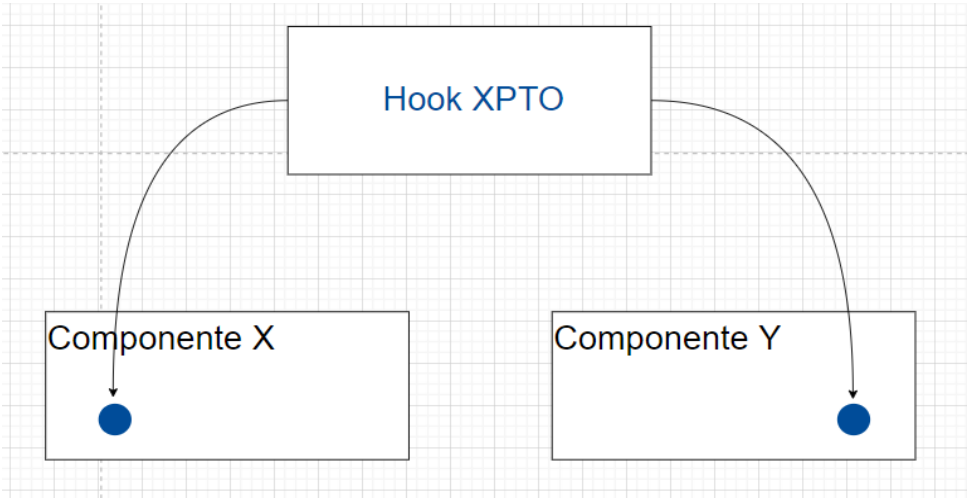
Hooks vem para resolver essas questões… Com os hooks é possível criar código que pode ser compartilhado entre componentes!

Na imagem abaixo podemos ver dois componentes (*ComponenteX* e *ComponentY*) que possuem a mesma lógica. Porém, como antes dos hooks era difícil compartilhar código entre componentes, foi duplicado o código no *ComponenteX* e no *ComponenteY*.



Fonte: autoral

Antes você só conseguia compartilhar componentes. Entretanto, a necessidade aqui é compartilhar código sem levar o JSX junto, ou seja, apenas lógica mesmo.



Fonte: autoral

Um dos grandes ganhos dos hooks é justamente esse, conseguir compartilhar o código sem levar o JSX em si. Na imagem acima, podemos ver o *ComponenteX* e *ComponenteY* utilizando o mesmo hook, ou seja, a mesma lógica está nos dois componentes, mas se precisarmos alterar a lógica em si, só precisamos alterar no hook e não nos dois componentes.

Hooks do React

Hooks é uma tecnologia que o React oferece para que você possa escrever código compartilhável entre componentes sem precisar ser um elemento em si (JSX). As possibilidades com isso são milhares. Por isso, o próprio React fornece alguns hooks do seu “core”, ou seja, nativos da biblioteca. Aqui mesmo já utilizamos um, o *useState,* para gerenciar o estado do componente.

Você pode acessar a documentação oficial do React e estar sempre se atualizando, pois, a biblioteca de tempos em tempos lança novos hooks nativos, veja acessando o link https://pt-br.reactjs.org/docs/hooks-reference.html

descomplica

Existem diversas bibliotecas de React que exploram e disponibilizam hooks para você utilizar dentro dos seus componentes. Vejam alguns exemplos para compreender o poder dos hooks:

- https://react-hook-form.com/ **(acesso em 14/10/2022)**
- https://react-dnd.github.io/react-dnd/about **(acesso em 14/10/2022)**
- https://react-spectrum.adobe.com/react-aria/ **(acesso em 14/10/2022)**

Ou seja, hook é uma forma de você reutilizar e compartilhar código de forma padronizada no React, pois a forma até então era de disponibilizar componentes inteiros, mas nem sempre era útil porque você só quer a lógica por trás e não o elemento/componente em si.

Ciclo de vida

Junto do hook *useState*, outro extremamente essencial é o hook *useEffect.* Para explorá-lo, precisamos entender um pouco sobre o ciclo de vida de um componente.

Os componentes possuem ciclo de vida, que você pode, de certa forma, interferir para executar algo durante cada estágio. Basicamente, componentes nascem, morrem e se atualizam. Cada estágio desse, você pode executar um código. Vamos dizer quando um componente “nasce”, ou seja, é adicionado na tela, você quer fazer uma requisição para uma API ou salvar algo no localStorage ou mesmo dar play em um vídeo que está na tela? Você também pode executar esses códigos quando o componente “morre” ou quando ele é renderizado novamente com nova informação, esses são estágios do ciclo de vida.

Mas como agendamos esses códigos para serem executados nos estágios da vida do componente? Com componentes feitos com classes, você tem a forma antiga de definir alguns métodos no componente como *componentDidMount* e *componentWilUnmount.* Mas com a chegada dos hooks, você usa o hook *useEffect.*

```jsx
import { useEffect, useState } from 'react'

const Assento = (props) => {
  const [ disabled, setDisabled ] = useState(false);

  const handleClick = () => {
   setDisabled(true);
  }

  useEffect(() => {
    console.log('Olá mundo')
  }, []);

  return (
    <button
      disabled={disabled}
      type="button"
      onClick={handleClick}
      className="assento"
      >
        {
          disabled
            ? <span>-</span>
            : <strong>{props.posicao}</strong>
        }
      </button>
  )
}
```

Fonte: autoral

Voltemos ao exemplo do componente *Assento*. Aqui nós importamos a função *useEffect* e invocamos ela dentro do nosso componente. O *useEffect* aceita dois argumentos:

Função a ser executada nos estágios

Dependência que o *useEffect* vai 'escutar/monitorar' para, caso o valor mude, ele invoque a função do primeiro argumento

Quando queremos escutar apenas o ciclo padrão do componente, a gente passa o array vazio. Ou seja, no código acima, a função que passamos para o *useEffect* só é executada quando o componente "nasce", quando ele é renderizado na tela.

Se você quiser sempre executar uma função quando o valor do estado *disabled* alterar, você pode passar ele ali como dependência no *useEffect* da seguinte forma:

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

const Assento = (props) => {
  const [ disabled, setDisabled ] = useState(false);

  const handleClick = () => {
   setDisabled(true);
  }

  useEffect(() => {
    console.log('Olá mundo')
  }, []);

  useEffect(() => {
    console.log('O estado do botão foi alterado para: ', disabled)
  }, [disabled]);

  return (
    <button
      disabled={disabled}
      type="button"
      onClick={handleClick}
      className="assento"
      >
        {
          disabled
            ? <span>-</span>
            : <strong>{props.posicao}</strong>
        }
      </button>
  )
}
```

Fonte: autoral

Veja no seu console do devtools que sempre que você clica no botão do assento, vai imprimir o texto que programamos para caso a informação do *disabled* altere.
Veja também que invocamos outro *useEffect* dentro do componente, isso é totalmente possível e normal, já que eles possuem necessidades diferentes. Enquanto um só vai ser invocado quando o componente é renderizado pela primeira vez, o outro vai ser executado quando *disabled* for alterado.

E como fazemos se quisermos invocar uma função sempre que o componente deixar de existir (morrer)? Basta no *useEffect* que você passa um array vazio nas dependências, na função, você retorna uma outra função, assim:

```
useEffect(() => {
    console.log('Olá mundo');
    return () => {
      console.log('Componente morreu')
    }
  }, []);
```

Fonte: autoral

Na função que passamos no *useEffect* acima, ela retorna numa outra função agora. Essa função é justamente a que vai ser executada quando o componente deixar de existir.

Referência Bibliográfica

SILVA, M. S. **React** - Aprenda Praticando: Desenvolva Aplicações web Reais com uso da Biblioteca React e de Seus Módulos Auxiliares. Novatec Editora, 2021.

STEFANOV, S. **Primeiros passos com React:** Construindo aplicações web. Novatec Editora, 2019.

Exercícios

1. Uma das funcionalidades que o React trouxe e fez com que a biblioteca/framework logo se tornasse popular foi seu sistema reativo. Qual é o grande ganho com esse sistema reativo?

a) O desacoplamento de código entre componentes
b) Torna as páginas muito mais otimizadas para os sistemas de busca
c) Renderização de componentes automática apenas com alteração do estado.
d) Melhora da sintaxe para escrever “HTML” dentro do JavaScript
e) Torna as páginas muito otimizadas desacoplando os códigos

2. Qual foi o principal objetivo do React ao introduzir hooks em sua biblioteca?

a) Trazer mais possibilidades ao sistema reativo da biblioteca

b) Compartilhamento de regra e lógica de componente desacoplada do JSX (parte da interface)

c) Unicamente focado em otimizar performance dos componentes

d) Utilizar sistema reativo em componentes feito com classes

e) Utilizar o sistema inato ao sistema reativo da biblioteca

3. O que é o ciclo de vida dos componentes?

a) Momentos ‘chave’ de renderização dos componentes, como quando nasce (renderizado), recebe novas propriedades e é removido da interface.

b) Grupo de métodos herdados da classe React para criação dos seus componentes com classe

c) É um conjunto específico de hooks dos componentes

d) São recursos adicionais para componentes feito com funções

e) É a etapa de desenvolvimento de um componente

4. Qual dos hooks é utilizado no ciclo de vida

a) useState

b) useMemo

c) useReducer

d) useEffect

e) useMecmo

descomplica

Gabarito

1. Letra C. O sistema reativo permite que o desenvolvedor não se preocupe com a atualização da interface de forma manual. Faz com que o desenvolvedor apenas foque em gerenciar o estado que o próprio React vai detectar as mudanças que precisam ser feitas na tela, e o fará.

2. Letra B. Criar lógica do lado de fora dos componentes que antes poderiam estar apenas dentro. Com isso temos o grande ganho de compartilhar comportamento entre componentes.

3. Letra A. Como o próprio nome diz, é o ciclo de vida do componente. Quando falamos de ciclo de vida, são as etapas chave de renderização de um componente onde podemos disparar funções agendadas para tal.

4. Letra D. O hook que conseguimos agendar funções a serem invocadas nas etapas do ciclo de vida de um componente React é o useEffect.