front-end

prof. Lucas Ferreira

engenharia de software

engenharia de computação

"Componentização", Gerenciamento de Estado: Hooks e Context API

parte 2

Introdução aos HOOKs

Conforme vimos na aula passada, **HOOKs** são funções que permitem ao dev. “ligar-se” aos recursos de state e ciclo de vida do React a partir de componentes funcionais *(não baseados em classe)*.

Iniciamos também um uso aplicado básico de ***useState*** o hook mais simples e mais usado em apps hoje em dia.

Outros HOOKs disponíveis e que também veremos são: **useReducer, useEffect**, **useRef** e **useContext**

Mas primeiro...

State Hook

Relembrando: como já vimos é o nosso primeiro e mais básico hooks, serve como introdução ao conceito.

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

function Contador() {
  // Declara uma nova variável de state, que chamaremos de "count"
  const [count, setCount] = useState(0);

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

Outro uso para State Hook

O state hook (***useState***) foi chamado de dentro de um componente funcional para adicionar algum estado local. O React irá preservar este state entre re-renderizações. E achei que valia pena voltar nele para frisar que o uso dele é muito amplo, não só strings *(useState(""))* ou números *(useState(0))*.

Podemos literalmente "pendurar" em um state hook qualquer objeto (primitivo ou não) de JavaScript.

Um uso que acho que vale a pena demonstrar é um usar um state hook para manipular um *object*...

Outro uso para State Hook

O *useState* abaixo recebe um objeto complexo, sendo que a ideia dele é com um único hook podermos explorar uma infinidade de "campos" ao invés de criar um *useState* para cada campo de um formulário (só um exemplo:

```
const [formData, setFormData] = useState({
  nome: "Lucas",
  sobrenome: "Ferreira",
  idade: "34",
  email: "lucas.ferreira@satc.edu.br"
});
```

Outro uso para State Hook

O único óbice desta estratégia é que sempre que formos atualizar o nosso hook não podemos esquecer de preservar todos os outros dados do objeto também. Para facilitar essa tarefa podemos usar um spread operator.

Por exemplo atualizado o sobrenome do *useState* apresentado no slide anterior:

```
setFormData({ ...formData, sobrenome: "R. Ferreira" })
```

Outro uso para State Hook

Neste contexto podemos criar uma função meio "coringa" para atualizar o objeto de forma mais "leve" em termos de complexidade de reuso:

```
function updateField(field, value) {
  setFormData({ ...formData, [field]: value });
}
```

Usando o recurso acima poderíamos atualizar o campo sobrenome assim:

```
updateField("sobrenome", "R. Ferreira");
```

Outro uso para State Hook

Voltando ao nosso hook de objeto *(e nesse caso serve para qualquer tipo de useState ~ até os mais simples)* também podemos setar o valor atualizado de um hook com base em uma função lógica que retorna valores:

```
const [formData, setFormData] = useState(...)
```

Podemos usar a atualização via *setFormData*

desse jeito aqui também:

```
setFormData((estadoAtual) => {
  const novaIdade = 27;

  // podemos atualizar um hook imbuído de lógica
  // neste exemplo se a nova idade for menor de 18
anos
  // não iremos atualizar a pessoa
  if (novaIdade < 18) {
    return estadoAtual;
  }

  return { ...estadoAtual, idade: `${novaIdade}` };
});
```

useReducer

Nosso próximo hook é muito familiar para os amantes de Redux (😫) pois é o mais próximos que temos da mecânica de "redutores" dentro do mecanismo natural/nativo dos hooks.

O ***useReducer*** possibilita acoplarmos junto de um estado inicial uma função de certa complexidade que ficará responsável por limitar e "auxiliar" as possíveis alterações que um estado pode sofrer. Ou seja um "meio de campo" entre a ideia de poder mudar de qualquer forma o valor de um *useState*.

Por fim ***useReducer*** é geralmente preferível em relação ao *useState* quando se tem uma lógica de manutenção de estado complexa que envolve múltiplos sub-valores, ou quando o próximo estado depende do estado anterior

useReducer

```
const [state, dispatch] = useReducer(reducer, initialState);
```

Na linha acima podemos visualizar o padrão de declaração de um ***useReducer***. Diferente do *useState* devemos sempre usar esse hooks junto de um "reducer" uma espécie de processador de ações que irá manipular o estado (por nós) através de determinados comandos "despachados".

useReducer

Por padrão recomendado um "reducer" será uma função, com a seguinte assinatura:

```
function reducer(state, action) {
  // action.type
}
```

O primeiro parâmetro "state" irá receber sempre o estado atual para entendermos onde estamos agora.

O segundo parâmetro "action" irá receber a ação (junto dos dados) que o reducer deverá executar e modificar o futuro estado desse hook.

useReducer

Num primeiro exemplo simples podemos pensar em um contador simples, incremento e decremento. A ideia seria "chavear" para não permitir a alteração da contagem em um vetor maior que +1 ou -1.

Caso usássemos *useState* a qualquer momento poderia ser chamado setCount mudado o número totalmente. Mas poderíamos pensar em algo assim:

```
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(`this action doesn't exists`);
  }
}

const [ state, dispatch ] = useReducer(reducer, { count: 0 });
```

useReducer

Na sequência podemos "despachar" um comando ao nosso hook (*useReducer*) para executar determinada ação.

No caso do *incremento* da contagem seria assim:

```
dispatch({ type: 'INCREMENT' })
```

E o *decremento*:

```
dispatch({ type: 'DECREMENT' })
```

useReducer

Por fim existe a possibilidade de junto com o disparo (dispatch) de um comando para o ***useReducer*** também enviarmos dados.

Por exemplo um hook cujo o *reducer* adicione itens de uma lista de compras:

```
function reducer(state, action) {
  switch (action.type) {
    case 'ADD_ITEM':
      return [ ...state, action.payload ];
    default:
      throw new Error();
  }
}
```

Além do nosso tipo de ação (***type***) também podemos enviar novos valores a serem usados para dentro do reducer normalmente indicando um pacote ou carga (***payload***).

useEffect

O *Hook de Efeito* (***useEffect***) te permite executar efeitos colaterais em componentes funcionais *(não baseado em classes)*. Junto do *useState*, podemos assegurar que é um dos principais e mais usados HOOKs atualmente.

O useEffect é um hook anônimo e normalmente não é associado a uma variável (nomeado) diretamente. A ideia dele é executar alguma ação automática sempre que algo relevante acontecer com um componente. Ele pode rodar apenas na "criação" do componente ou sempre que um ou mais hooks sofrem alterações.

Pensem nele como um monitor de funcionamento do componente, sempre que algo se mexer (ou for criado) esse "carinha" poderá agir, se assim vocês programarem.

useEffect

Em sua assinatura um *useEffect* recebe até 2 parâmetros. Primeiro a função que deve ser executada sempre que for a hora do "efeito" acontecer, e o segundo é sobre qual o limite de uso desse "efeito".

Por exemplo o hook abaixo será executado sempre que o componente sofrer qualquer alteração (não importa em qual magnitude):

```
useEffect(() => {
  document.title = `Bem-vindo ${nome}`;
});
```

useEffect

Caso queiramos que o efeito aconteça estritamente apenas quando a variável nome for alterada, devemos mudar o hook para o seguinte:

```
useEffect(() => {
  document.title = `Bem-vindo ${nome}`;
}, [nome]);
```

useEffect

E se quisermos que esse hook de efeito "rode" apenas uma única vez logo que o componente for renderizado (e aparecer em tela) devemos mudar para o seguinte:

```
useEffect(() => {
  document.title = `Bem-vindo ${nome}`;
}, []);
```

useRef

O **useRef** atua como uma função que retorna um objeto ref e recebe um argumento que inicializa a propriedade .current desse objeto.

```
const refContainer = useRef(initialValue);
```

Sempre que usado o **useRef** retorna um objeto ref mutável, no qual a propriedade *.current* é inicializada para o argumento passado (initialValue). O objeto retornado persistirá durante todo o ciclo de vida do componente.

Essencialmente, **useRef** é como uma "caixa" que pode conter um valor mutável em sua propriedade .current ***(vou demonstrar)***.

useRef

Umas das aplicações para refs é para podermos acessar elementos DOM ou do React. Se passarmos um ref para um componente, o React configura propriedade .current do ref para o nó DOM correspondente sempre que esse nó for alterado. No exemplo abaixo, o ref inputEl é passado no input e ao clicarmos no botão podemos focá-lo:

```
function TextInputWithFocusButton() {
  const inputEl = useRef(null);
  const onButtonClick = () => {
    // `current` aponta para o evento de `focus` gerado pelo campo de texto
    inputEl.current.focus();
  };
  return (
    <>
      <input ref={inputEl} type="text" />
      <button onClick={onButtonClick}>Focus no input</button>
    </>
  );
}
```

useRef

Por fim, tenha em mente que o ***useRef*** não avisa quando o conteúdo é alterado. Mexer na propriedade .current não causa uma nova renderização.

Sendo assim refs podem ser uma ótima opção para guardar valores entre renderizações de componentes.

```
function Counter() {
  const [count, setCount] = useState(0);
  const prevCountRef = useRef();

  useEffect(() => {
    prevCountRef.current = count;
  }, [count]);

  return (
    <div>
      <h3>
        Valor do contador agora: {count} <br />
        Valor do contador antes: {prevCountRef.current}
      </h3>
      <button onClick={() => setCount(count + 1)}>Somar um ao
contador</button>
    </div>
  );
}
```

Context API

e useContext

Context API

Em uma aplicação típica do React, os dados são passados de cima para baixo (de pai para filho) via props, mas esse uso pode ser complicado para certos tipos de props (como preferências locais ou tema de UI), que são utilizadas por muitos componentes dentro da aplicação.

A **Context API** fornece a forma de compartilhar dados como esses, entre todos componentes da mesma árvore de componentes, sem precisar passar explicitamente props entre cada nível (de pai para filho infinitamente).

Antes de continuar vou exemplificar *(ao vivo ~ fora dos slides)* o problema que buscamos resolver...

Quando usar Context

O uso de ***Context*** é indicado para compartilhar dados que podem ser considerados “globais” para a árvore de componentes do React.

Por exemplo, usuário autenticado ou o idioma preferido, são alguns casos comuns, normalmente variáveis de estado global que devem estar disponíveis de/em "qualquer ponta" de nosso app React.

React.createContext

```
const MeuContext = React.createContext(valorPadraoInicial);
```

Serve para criar um objeto Context.

Quando o React renderiza um componente que usa este objeto Context, este terá o valor atual lido do provedor em maior top-level na árvore de componentes próximos.

Context.Provider

```
<MeuContexto.Provider value={/* novo valor que poderá ser usar como padrão */}>
```

Cada objeto Context vem com um componente **Provider** "de brinde", que permite a componentes consumidores a assinarem (para receber) mudanças no contexto.

O componente Provider aceita uma prop value que pode ser passada para ser consumida por componentes que são descendentes (filhos) deste Provider. Um Provider pode ser conectado a vários consumidores.

Todos consumidores que são descendentes (filhos) de um Provider serão renderizados novamente sempre que a prop value do Provider for alterada.

useContext

```
const value = useContext(MeuContext);
```

Este hook aceita um objeto de contexto (o valor retornado de React.createContext) e retorna o valor atual do mesmo.

O valor de contexto atual é determinado pela prop value do <MeuContext.Provider> mais próximo acima do componente pai na árvore estrutural da aplicação.

Quando o <MeuContext.Provider> mais próximo acima do componente for atualizado, este Hook acionará um novo renderizador com o value de contexto mais recente passando para o provedor MeuContext.

Por exemplo...

Vamos supor que tenhamos uma aplicação que possui uma ideia de "tema escuro" e "tema claro". Essa aplicação têm infinitos níveis de componente.

Por exemplo temos uma Pagina -> que tem um Painel -> que tem uma Lista -> que tem um Item -> que tem um Botão.

Dentro dessa "jogada" existe a necessidade de se o usuário trocar o tema da aplicação de "claro" para "escuro" o botão que antes tinha fundo cinza e letra preta, passe a ter fundo preto e letra branca.

Neste exemplo para facilitar o aviso ao Botão que está tão profundamente inserido iremos criar um ***ThemeContext*** para propagar por toda a aplicação se estamos usando um tema escuro ou claro.

Por exemplo...

```
const ThemeContext = React.createContext("light");

function App() {
  return (
    <ThemeContext.Provider value={"light"}>
      <Painel />
    </ThemeContext.Provider>
  );
}

function Painel() {
  return (
    <ul>
      <li>
        <strong>Pão d'agua</strong>
        <ButtonComprar />
      </li>
    </ul>
  );
}

function ButtonComprar() {
  const theme = useContext(ThemeContext);

  return (
    <button className={theme === "light" ? "bt-claro" : "bt-escuro"}>
      Comprar
    </button>
  );
}
```

NOSSO 4º EXERCÍCIO

MAIS UM EXERCÍCIO! 😱

Após passarmos o "carro" em componentes e também em hooks, acredito que já possamos fazer a nossa primeira "prática" sobre React.js em forma de exercício.

Para este exercício em específico vou estar aceitando que trabalhem em **DUPLAS** *(mesma da outra vez)* ou **INDIVIDUALMENTE** e "lanço" o exercício hoje mas vocês terão **até a próxima aula às 20h para entregar** naquele clássico esquema de 1 hora na faixa para me pedir ajuda caso estejam *empacados* 👊

MAIS UM EXERCÍCIO! 😱

MAIS UM EXERCÍCIO! 😱

Iremos trabalhar com a ideia de criar uma simples lista de compras. As ações serão as mais básicas, a pessoa digita o nome de um item a ser comprado no formulário, aperta o botão e esse nome será adicionado na lista acima.

Para lhes ajudar a não perder tempo criei uma base para este exercício no github, segue o link do repositório:

https://github.com/lucasferreira/front-end-exercicio-lista

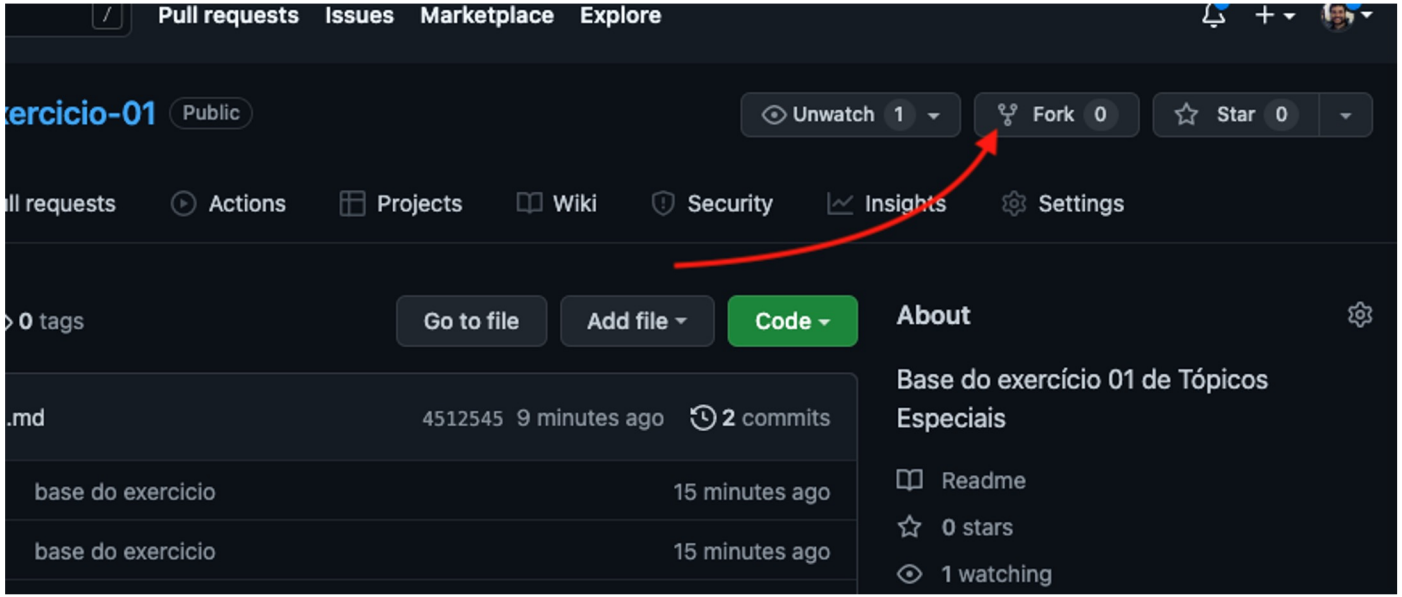
MAIS UM EXERCÍCIO! 😱

A base que estou sugerindo (github, slide anterior) é apenas uma "ajuda", serve para vcs não perderem tempo com uma estrutura básica de HTML e nem perder tempo com CSS. Caso vocês não curtam o que eu "adiantei" fiquem à vontade para criarem suas listas do zero, do jeito que acharem melhor com o CSS/visual que acharem melhor também.

Porém se forem usar o que eu sugeri, sugiro se "logar" em seus github.com, acessar meu link e **clicar em FORK**, este comando irá clonar minha base de código sobre o usuário de vcs e lá vcs poderão mexer à vontade sem "prejudicar o meu projeto original".

Na sequência poderão baixar o repositório clone de vcs para suas máquinas e trabalharem dali. Quem tiver dificuldade pode usar o **GitHub Desktop** *(visual)*.

MAIS UM EXERCÍCIO! 😱



MAIS UM EXERCÍCIO! 😱

Meu template vem de fábrica com o **vite** configurado, então logo após "checkarem" o repositório em suas máquinas de trabalho, vocês terão que primariamente rodar o comando abaixo para instalar as dependências dentro da pasta do projeto:

npm install

Depois vocês poderão ligar o servidor de desenvolvimento/testes usando o outro comando:

npm start

MAIS UM EXERCÍCIO! 😱

Como requisito técnico teremos o seguinte:

- Montar uma mini-micro-app de lista de compras
- Usar apenas **React.js** *(tudo componentizado)*
- Ter pelo menos **2 componentes** na micro-aplicação *(ou seja quebrar meu HTML em pelo menos 2)*

  Usar algum HOOK para armazenar a listagem de itens a ser comprado (array/matriz)
- **Criar um input + botão que quando preenchido (e apertado) adiciona um novo item no hook da lista de compras e por consequência irá mostrar em tela os itens pendentes de compras atualizados.**

  Poderão utilizar onClick no botão ou onSubmit no form para adicionar o novo item.

MAIS UM EXERCÍCIO! 😱

REQUISITOS ADICIONAIS PARA VOCÊ QUE JÁ MANJA DO BAGULHO:

- Marcar o item como "já comprado" *(pode alterar algo no visual do item para indicar)*
- Excluir o item da lista *(talvez um botão só para isso em cada linha?)*
- Poder adicionar além do nome do item a ser comprado também a quantidade desejada *(precisará de outro campo no form)* e adicionar na matriz/array de itens "objetos complexos" ao invés de apenas uma string com o nome do treco solta.

obrigado 🚀