soluções mobile

prof. Thyerri Mezzari

Uma pequena mudança no expo...

```
npx create-expo-app StickerSmash --template blank

cd StickerSmash
```

Create your first app

O expo atualizou o projeto inicial que era criado ao usar o comando npx create-expo-app. Agora o projeto inicial traz algumas pastas e arquivos a mais que não são interessantes para o nosso aprendizado neste momento.

Portanto para criar um novo projeto com um template “menos complexo” usem:

- npx create-expo-app MeuApp --template blank

Gerenciamento de Estado e "Componentização" em React Native

Gerenciamento de Estado

Uma das tarefas mais básicas relacionadas ao desenvolvimento de uma interface rica e interativa é determinar em que estado ela se encontra.

O gerenciamento de estado na maioria dos casos tende a ser local, sendo que cada componente pode ter o seu estado.

O exemplo mais básico para imaginarmos essa "tarefa" é pensar em um botão de opção, do tipo *Switch*. Este botão em sua configuração mais básica terá apenas dois possíveis estados...

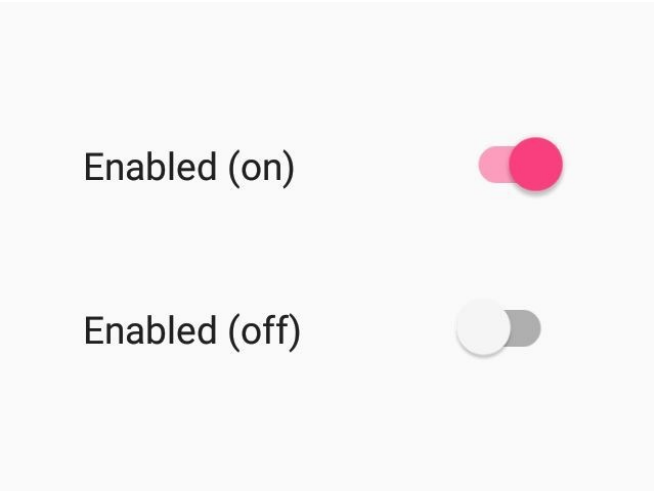
Gerenciamento de Estado

Estados possíveis de um botão *SWITCH*:

LIGADO

ou

DESLIGADO



Gerenciamento de Estado

Na representação simples do botão de SWITCH nosso componente terá um estado do tipo Booleano, em que *true* representa ligado e *false* representa desligado.

Em um mundo mais simples da programação estes valores relacionados ao estado deste botão poderiam ser apenas uma variável simples, em que iríamos escrever o valor de *false* ou *true*.

Porém o **React** em seu *estado mais puro da arte* irá sempre buscar "reagir" *(vem daí o nome React.js)* a mudanças nos elementos que envolvem a interface disposta ao usuário, seja através de uma prop/atributo *(vimos na aula passada)* ou até através de uma variável de estado local.

Gerenciamento de Estado

Sendo que em relação a estas variáveis de estado local, para que o React possa monitorar e "reagir" de maneira adequada a mudanças de valores na mesma, precisamos seguir um padrão já amplamente divulgado na indústria que envolve essa biblioteca, o padrão ditado pelos **HOOKs**.

Mas antes de conceitualizar os **HOOKs**, gostaria de deixar bem claro que podemos ter qualquer tipo de variável em estado de memória local, não apenas true ou false *(exemplo do Switch)*. Podemos "reagir" a valores relacionados a *strings*, *numbers*, *array*, *objects*, *booleans* e etc...

Engenharia da Computação

Introdução aos HOOKs

Mas, o que é um Hook?

Resumidamente, hooks são a última proposta do *core-team* React que nos permite utilizar estado, ciclo de vida, entre outras funcionalidades sem a necessidade de escrevermos componentes com classe. A funcionalidade encontra-se disponível desde a versão 16.8 do React.

Sendo ainda mais básico são funções que permitem ao dev. “ligar-se” aos recursos de state e ciclo de vida do React a partir de componentes funcionais.

State Hook

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 (
    <View style={styles.contador}>
      <Text style={styles.contadorLabel}>Você clicou {count}</Text>
      <View style={{ padding: 21 }}>
        <Button mode="contained" onPress={() => setCount(count + 1)}>
          Clique aqui
        </Button>
      </View>
    </View>
  );
}
```

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 *(vou demonstrar)*.

O ***useState*** *(e a maioria dos hooks)* retorna um par de elementos: **o valor do state atual e uma função que permite atualizá-lo**.

O único argumento para ***useState*** é o estado inicial. No exemplo anterior é 0, porque nosso contador começa do zero. Por fim, o argumento de state inicial é utilizado apenas durante a primeira renderização.

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: "Thyerri",
  sobrenome:
  "Fernandes", idade:
  "32",
  email: "thyerri.mezzari@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: "F. Mezzari" })
```

✌️ Regras dos Hooks

Hooks são funções JavaScript, mas eles impõe duas regras adicionais:

- Apenas chame Hooks **no nível mais alto**. Não chame Hooks dentro de loops, condições ou funções aninhadas.
- Apenas chame Hooks de **componentes funcionais**. Não chame Hooks de funções JavaScript comuns *(apesar de ser válido também chamar hooks de dentro dos seus próprios Hooks customizados)*.

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(() => {
  alert('Algo mudou!');
});
```

useEffect

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

```
useEffect(() => {
  alert('Valor do contador: ' + count);
}, [count]);
```

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(() => {
  alert('Valor inicial do contador: ' + count);
}, []);
```

Outros HOOKs

Outros HOOKs disponíveis "de fábrica" e que também veremos em outros momentos do curso: **useReducer**, **useRef** e **useContext**

https://pt-br.reactjs.org/docs/hooks-reference.html

Engenharia da Computação

Os famosos Componentes

Organizando e entendendo um pouco melhor os "nossos próprios" componentes

"Componentização" em React

Acredito já ter falado sobre isso algumas vezes, quase tudo em React *(Native)* pode ser considerado um ***Component***.

Com exceção de hooks, controladores puramente lógicos e libs de terceiros, o resto tudo é ***Component***.

Desde uma tela inteira a até um pequeno botão de "ligar e desligar" que está nessa mesma tela. Ou seja, tudo é um componente, e sendo um componente maior (voltando ao exemplo da tela), será formado de diversos outros componentes menores que porventura podem ter outros componentes ainda menores dentro deles ***(isso pode tender ao infinito)***.

"Componentização" em React

A seguir iremos trabalhar um exemplo básico de componente próprio que irá se "desenvolver" até ser plenamente customizável. De tão simples esse início de trabalho recomendo que vocês trabalhem no **Snack Expo**:

https://snack.expo.dev/

Outra vantagem do Snack Expo é que já vem de fábrica com o **react-native-paper** habilitado no repositório on-line para já usar diretamente.

A minha base será essa, vocês podem continuar daqui também:
https://snack.expo.dev/@thyerrimezzari/ex-kitten-card?

Um Card de Exemplo

```
export default function KittenCard() {
  return (
    <Card>
      <Card.Cover source={{ uri: "https://placekitten.com/390/240" }} />
      <Card.Title title="Gatinho bonito" />
      <Card.Content>
        <Paragraph>Some quick example text to build on the card title.</Paragraph>
      </Card.Content>
      <Card.Actions>
        <Button mode="contained" onPress={() => console.log("Pressionou")}>
          Quero adotar!
        </Button>
      </Card.Actions>
    </Card>
  );
}
```

"Componentização" em React

O componente mostrado anteriormente não tem nada de mais, é praticamente um emaranhado de componentes básicos de **Card** do React Native *(e react-native-paper)*.

Minha ideia daqui em diante é ir "melhorando" a componentização dessa base complexa para algo um pouco mais "*Reactzado*".

Card

Nosso próximo passo com esse componente é trabalhar com propriedades "customizadas".

Em React temos algo que chamamos de **props**, diretamente ligado à melhor maneira de injetar uma informação externa ao *Component* dentro dele.

```
export default function KittenCard(props) {
  return (
    <Card style={{ marginBottom: 14 }}>
      <Card.Cover source={{ uri: "https://placekitten.com/390/240" }} />
      <Card.Title title={props.title} />
      <Card.Content>
        <Paragraph>{props.text}</Paragraph>
      </Card.Content>
      <Card.Actions>
        <Button mode="contained" onPress={...}>
          {props.buttonLabel}
        </Button>
      </Card.Actions>
    </Card>
  );
}
```

Card

No código anterior nós injetamos algumas variáveis baseadas em propriedades para podermos re-aproveitar o mesmo componente mais de uma vez.

Esse é um dos conceitos fundamentais do React.

```
function App() {
  return (
    <View style={styles.container}>
      <KittenCard
        title="Adote um Gatinho!"
        text="Eles são muito fofinhos e destroem todos os seus móveis"
        buttonLabel="Quero Adotar"
      />
      <KittenCard
        title="Gato Fofinho"
        text="Eles são muito fofinhos e destroem todos os seus móveis"
        buttonLabel="Quero Apertar"
      />
    </View>
  );
}
```

Card

Utilizando o objeto ***props*** como argumento "catch-all" de um componente apesar de ser "rapidamente implementável" é muito pouco didático e também abre uma possibilidade do componente receber atributos "indesejáveis".

Sendo assim o caminho indicado para melhorar esse cenário seria declararmos de forma evidente quais atributos nós aceitamos...

Card

Desta forma também conseguimos um jeito fácil de "definir" valores padrões para argumentos opcionais, com o que foi feito por exemplo com *`buttonLabel`*.

```
function KittenCard({ title, text, buttonLabel = "Clique Aqui" }) {
  return (
    <Card style={{ marginBottom: 14 }}>
      <Card.Cover source={{ uri: … }} />
      <Card.Title title={title} />
      <Card.Content>
        <Paragraph>{text}</Paragraph>
      </Card.Content>
      <Card.Actions>
        <Button mode="contained" onPress={...}>
          {buttonLabel}
        </Button>
      </Card.Actions>
    </Card>
  );
}
```

Card

Na alteração ao lado criamos um objeto responsável por acumular qualquer outro parâmetro que não seja title, text ou buttonLabel. Na sequência aplicamos esse objeto no elemento Card principal.

Agora podemos passar um ***style={...}*** para o component Card, por exemplo.

```
function KittenCard({ title, text, buttonLabel, ...props }) {
  return (
    <Card {...props}>
      <Card.Cover source={{ uri: "..." }} />
      <Card.Title title={title} />
      <Card.Content>
        <Paragraph>{text}</Paragraph>
      </Card.Content>
      <Card.Actions>
        <Button mode="contained" onPress={...}>
          {buttonLabel}
        </Button>
      </Card.Actions>
    </Card>
  );
}
```

Card

Para deixar nosso componente ainda mais versátil, podemos trabalhar com testes opcionais para renderizar ou não um pedaço de nosso card. Por exemplo podemos criar um Card com botão opcional assim:

```
{!!buttonLabel && (
  <Button mode="contained" onPress={() => console.log("Pressionou")}>
    {buttonLabel}
  </Button>
)}
```

A linha acima só será renderizada se a variável ***buttonLabel*** estiver validamente preenchida com algo diferente de *null* ou *undefined*.

Card - Nova Ideia

Continuando nossa exploração do componente Card, podemos tomar um rumo diferente na organização/estrutura de nosso componente e de quebra explorar uma outra propriedade bem interessante, a prop "***children***".

Lembrando que até aqui, o uso de nosso componente atualmente está vinculado a parâmetros/propriedades:

```
<Card
  title="Adote um Gatinho"
  text="Eles são muito fofinhos e destroem todos os seus móveis"
  buttonLabel="Quero Adotar!"
/>
```

Card - Nova Ideia

A propriedade "**children**" é uma *reserved keyword* do mundo React. Ela serve para abrir um "portal de acesso" ao conteúdo de um Component.

Nessa ideia estamos substituindo o "text" do KittenCard por o uso de children, abrindo assim maiores possibilidades de customização do conteúdo.

```
function KittenCard({ title, children, buttonLabel, ...props }) {
  return (
    <Card {...props}>
      <Card.Cover source={{ uri: … }} />
      <Card.Title title={title} />
      <Card.Content>{children}</Card.Content>
      <Card.Actions>
        {!!buttonLabel && (
          <Button mode="contained" onPress={...}>
            {buttonLabel}
          </Button>
        )}
      </Card.Actions>
    </Card>
  );
}
```

Card - Nova Ideia

Após trocarmos por children alguma propriedade usada, podemos abastecer essa propriedade como o conteúdo do *component* customizado.

```
function App() {
  return (
    <View style={styles.container}>
      <KittenCard title="Adote um Gatinho!" buttonLabel="Quero Adotar">
        <Paragraph>Teste teste teste</Paragraph>
        <Paragraph>Outro teste</Paragraph>
        <Button mode="outlined" onPress={() => console.log("Clicou")}>
          Até um botão perdido
        </Button>
        <Paragraph>Terceiro parágrafo</Paragraph>
      </KittenCard>
    </View>
  );
}
```

obrigado 🚀