1. Consumindo uma API RESTful com React Hooks

Nesta atividade prática, vamos trabalhar com os conceitos de sobre como consumir uma API RESTful com React Hooks.

1.1 Introdução

Com o advento do React Hooks, nós podemos ter uma lógica gerenciamento de estado em componentes funcionais - o que nos permite ter componentes funcionais como "contêineres". Desta forma, se quisermos que um componente carregue dados em sua montagem (*mouting*), não precisamos mais de um componente de classe. Pelo menos, não apenas porque precisamos de gerenciamento de estado ou de eventos de ciclo de vida (*lifecycle events*).

Existem hooks que nos auxiliam nestas tarefas:

- `useState` e `useReducer`: nos permite adicionar funcionalidade de gerenciamento de estado ao nosso componente.
- `useEffect`: nos dá a possibilidade de executar os chamados efeitos colaterais (*side effects*), tais como: buscar dados de uma API de forma assíncrona.

Em seguida vamos desenvolver um exemplo básico de um componente React que carrega e exibe uma lista de usuários obtidos de uma API RESTful. Assim, buscaremos usuários aleatórios do JSONPlaceholder, uma API RESTful online que oferece dados fake para teste. Como resultado, temos uma resposta JSON similar a esta:


```
[
  {
    "id": 1,
    "name": "Leanne Graham",
    "username": "Bret",
    "email": "Sincere@april.biz",
    "address": {
      "street": "Kulas Light",
      "suite": "Apt. 556",
      "city": "Gwenborough",
      "zipcode": "92998-3874",
      "geo": {
        "lat": "-37.3159",
        "lng": "81.1496"
      }
    },
    "phone": "1-770-736-8031 x56442",
    "website": "hildegard.org",
    "company": {
```

```
      "name": "Romaguera-Crona",
      "catchPhrase": "Multi-layered client-server neural-net",
      "bs": "harness real-time e-markets"
    }
  },
 …
]
```

1.2 Criando um projeto de teste e instalando a API Axios

Vamos criar um projeto chamado my-api-jsonplaceholder, digitando este comando no terminal:

```
expo init my-api-jsonplaceholder
```

Em seguida, algumas opções serão apresentadas. Selecione a opção “blank”. O Expo será executado no diretório que você especificar. Este processo leva alguns segundos e, logo após terminar, digite o seguinte comando para entrar no diretório recém criado do nosso projeto:

```
cd my-api-jsonplaceholder
```

Você agora está dentro da raiz do seu projeto. Existem várias opções disponíveis para adicionar o Axios ao seu projeto, dependendo do seu gerenciador de pacotes. Estamos usando o Yarn, assim, digite a seguinte linha de comando:

```
yarn add axios
```

Você pode abrir a aplicação no editor Visual Studio Code. Para isso, dentro deste diretório digite o seguinte comando:

```
code .
```

O Visual Studio Code deverá abrir com a pasta raiz do seu projeto sendo acessada. Em seguida, você inicializará um servidor de desenvolvimento local. Assim, execute o comando:

```
expo start
```

Ao executar esse script, o Expo CLI inicia o Metro Bundler, que é um servidor HTTP que compila o código JavaScript de nosso aplicativo (usando o Babel) e o serve ao aplicativo Expo.

1.3 Consumindo uma API RESTful com React Hooks

Vamos ver um exemplo prático de como consumir dados vindos de uma API RESTful com React Hooks.

No arquivo App.js, insira o seguinte código:

```
import { StatusBar } from "expo-status-bar";
import React, { useEffect, useState } from "react";
import { StyleSheet, Text, View, FlatList } from "react-native";
import axios from "axios";

export default function App() {
 const [data, setData] = useState([]);
 const Item = ({ data }) => (
   <View style={styles.item}>
     <Text style={styles.title}>{data}</Text>
   </View>
 );
 async function listUsers() {
   try {
     const response = await axios.get(
       "https://jsonplaceholder.typicode.com/users"
     );
     setData(response.data);
   } catch (err) {
     console.log(err);
   }
 }
 // A cada renderização do componente, o hook useEffect
 //executa a função listUsers().
 useEffect(() => {
   listUsers();
 }, []);
```

```
 const renderItem = ({ item }) => (
   <Item data={item.username + " : " + item.name} />
 );

 return (
  <View style={styles.container}>
              <Text     style={styles.header}>Consulta     à     API
JSONPlaceholder</Text>
    <FlatList
       data={data}
       renderItem={renderItem}
       keyExtractor={(item) => item.id.toString()}
     />
     <StatusBar style="auto" />
   </View>
 );
}

const styles = StyleSheet.create({
 container: {
   flex: 1,
 },
 item: {
   backgroundColor: "#77AAFF",
   padding: 10,
   marginVertical: 8,
   marginHorizontal: 16,
 },
 title: {
   fontSize: 14,
 },
```

```
  header: {
    backgroundColor: "#89BBDD",
    textAlign: "center",
    padding: 20,
  },
  error: {
    color: "#FF0000",
    fontSize: 14,
    textAlign: "center",
  },
});
```

Vejamos a seguir como criamos este código.

1.3.1 Hook useState

O hook `useState` usa a atribuição via desestruturação (*destructuring assignment*) para fornecer o valor atual de um estado (que chamamos de `data`) e uma função para definir um novo valor de estado (que chamamos de `setData`). O argumento que passamos para `useState` é o valor inicial para `data` no estado do componente.

```
const [data, setData] = useState([]);
```

Logo, você pode imaginar que agora o estado do componente está assim:

```
{
  data: []
}
```

Então, agora temos um estado em nosso componente. Mas, precisamos definir como obter os dados da API RESTful nesse campo. Para esta tarefa, precisamos usar o hook `useEffect`.

1.3.2 Hook useEffect

A cada renderização do componente, o hook `useEffect` executa a função que passamos como o primeiro argumento. Por questão de performance, não queremos carregar os usuários vindos da API cada vez que o componente é renderizado novamente. Por isso, passamos um array vazio como segundo argumento.

O segundo argumento de `useEffect` nos permite passar dependências para o Hook. Por exemplo, um valor no estado ou uma string usada para filtrar os dados que serão retornados da API. Cada vez que uma dessas dependências mudar, o hook `useEffect` será executado novamente. Neste exemplo simples, como queremos buscar os dados apenas uma vez, passamos um array vazio.

```
async function listUsers() {
  try {
    const response = await axios.get(
      "https://jsonplaceholder.typicode.com/users"
    );
    setData(response.data);
  } catch (err) {
    console.log(err);
  }
}

// A cada renderização do componente, o hook useEffect
//executa a função listUsers().
useEffect(() => {
  listUsers();
}, []);
```

Uma vez que o `Axios` é uma API baseada em Promises, podemos usar a sintaxe assíncrona do ES2017: a funcionalidade `async/await`. Uma função assíncrona pode conter uma expressão `await`, que pausa a execução da função assíncrona e espera pela resolução da Promise passada, e depois retoma a execução da função assíncrona e retorna o valor resolvido. Veja que dentro de `listUsers()` nós atualizamos o estado de nosso componente (usando `setData`) com os dados recém buscados da API (`response.data`).

1.3.3 Carregando os dados na tela

Uma vez que buscamos os dados da API, vamos carregá-los na tela em formato de uma lista. Lembre-se que o componente FlatList exibe uma lista de rolagem de dados variáveis, mas estruturados de forma semelhante. O componente FlatList requer dois props: `data` e `renderItem`. O prop `data` é a fonte de informação da lista. Já o `renderItem` pega um

item da fonte e retorna um componente formatado para renderizar. Ainda, a prop `keyExtractor` diz à lista para usar os ids como sendo a chave identificadora de cada item.

```
<FlatList data={data} renderItem={renderItem}
              keyExtractor={item => item.id.toString()} />
```

Veja o resultado na tela:

Funcionou! Em seguida, vamos criar nosso próprio hook.

1.3.4 Escrevendo seu próprio hook

Começaremos escrevendo nosso hook para buscar os usuários da API JSONPlaceholder. Assim, crie o diretório hooks. Dentro dele crie o arquivo useFindUsers.js com o seguinte código:

```
import { useEffect, useState } from "react";
import axios from "axios";
export function useFindUsers() {
```

```
 // Definimos os estados necessários para o nosso hook
 const [data, setData] = useState([]);
 // Busca os usuários da API JSONPlaceholder
 async function listUsers() {
   try {
     const response = await axios.get(
       "https://jsonplaceholder.typicode.com/users"
     );
     // Atualiza o estado de 'data' com o resultado da API
     setData(response.data);
   } catch (err) {
     console.log(err);
   }
 }
 // A cada renderização do componente, o hook useEffect
 //executa a função listUsers().
 useEffect(() => {
   listUsers();
 }, []);

 return data;
}
```

Observe que o hook `useFindUsers()` é apenas uma simples função que podemos exportar para outros componentes, tal como o App.js. Em seguida, nós definimos os estados necessários para o nosso hook com o código

```
const [data, setData] = useState([]);
```

Em seguida, com a função assíncrona `listUsers()`, nós buscamos os usuários vindos da API JSONPlaceholder usando a API Axios. E, finalmente, atualizamos o estado de '`data`' com o resultado da API

```
try {
     const response = await axios.get(
```

```
      "https://jsonplaceholder.typicode.com/users"
    );
    // Atualiza o estado de 'data' com o resultado da API
    setData(response.data);
  }
```

Cabe lembrar que o hook useEffect pode ser comparado aos *lifecycles events* componentDidMount, componentDidUpdate e componentDidUnmount dos componentes de classe.

Agora que temos o hook `useFindUsers()`, podemos usá-lo em um componente da nossa aplicação, em nosso caso, vamos usar em App.js:

```
import { StatusBar } from "expo-status-bar";
import React from "react";
import { useFindUsers } from './hooks/useFindUsers';
import { StyleSheet, Text, View, FlatList } from "react-native";

export default function App() {
  // use seu próprio hook para carregar os dados da API
 const data = useFindUsers();

 const Item = ({ data }) => (
   <View style={styles.item}>
     <Text style={styles.title}>{data}</Text>
   </View>
 );
 const renderItem = ({ item }) => (
   <Item data={item.username + " : " + item.name} />
 );
 return (
  <View style={styles.container}>
```

```
            <Text     style={styles.header}>Consulta     à     API
JSONPlaceholder</Text>
     <FlatList
       data={data}
       renderItem={renderItem}
       keyExtractor={(item) => item.id.toString()}
     />
     <StatusBar style="auto" />
   </View>
 );
}

const styles = StyleSheet.create({
 container: {
   flex: 1,
 },
 item: {
   backgroundColor: "#77AAFF",
   padding: 10,
   marginVertical: 8,
   marginHorizontal: 16,
 },
 title: {
   fontSize: 14,
 },

 header: {
   backgroundColor: "#89BBDD",
   textAlign: "center",
   padding: 20,
 },
 error: {
```

```
    color: "#FF0000",
    fontSize: 14,
    textAlign: "center",
  },
});
```

Veja que o resultado na tela não mudou. Está tudo funcionando.

1.3.5 Melhorando seu código

Uma vez que nosso hook `useFindUsers()` está funcionando, podemos implementar algumas melhorias. Por exemplo, podemos tratar o momento em que a API JSONPlaceholder foi chamada e ainda não retornou um dado. Também podemos melhorar o tratamento de erros vindos esta API.

```
const [loading, setLoading] = useState(false);
const [error, setError] = useState(null);
```

Assim, altere o arquivo useFindUsers.js com o seguinte código:

```
import { useEffect, useState } from "react";
import axios from "axios";

export function useFindUsers() {
 // Definimos os estados necessários para o nosso hook, isso
inclui:
 // usuários (data), estado de carregamento (loading) e erros
(error)
 const [data, setData] = useState([]);
 const [loading, setLoading] = useState(false);
 const [error, setError] = useState(null);
 // Busca os usuários da API JSONPlaceholder
 async function listUsers() {
   try {
     const response = await axios.get(
       "https://jsonplaceholder.typicode.com/users"
     );
     // Atualiza o estado de 'loading'
     setLoading(false);
     // Atualiza o estado de 'data' com o resultado da API
     if (response.data) {
       setData(response.data);
     } else {
       setData([]);
     }
   } catch (err) {
     // Atualiza o estado de 'error', 'data' e de 'loading'
     setError("Lamento, ocorreu um erro!");
     setData([]);
```

```
    setLoading(false);
  }
}
// A cada renderização do componente, o hook useEffect
//executa a função listUsers().
useEffect(() => {
  // Primeiro, definimos os estados de loading e error
  setLoading(true);
  setError(null);

  // executa a função listUsers().
  listUsers();
}, []);

return { data, loading, error };
}
```

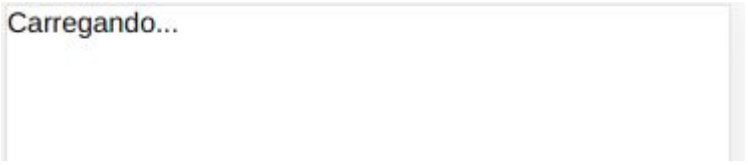
Veja que nós definimos os estados necessários para o nosso hook, isso inclui: usuários (data), estado de carregamento (loading) e erros (error). Ainda, dentro do hook `useEffect()`, primeiro, nós definimos os estados de `loading` e `error` e, depois, executamos a função `listUsers()`.

Em seguida, altere o arquivo App.js da seguinte forma:

```
...
export default function App() {
 // use seu próprio hook para carregar os dados da API
const { data, loading, error } = useFindUsers()
if (loading) return (
  <View style={styles.container}>
    <Text style={styles.title}>Carregando...</Text>
  </View>
);
if (error) return (
  <View style={styles.container}>
```

```
      <Text style={styles.error}>{error}</Text>
    </View>
  );
...
```

Veja que o resultado na telamudou, um pouco. Ao carregar o componente pela primeira vez, recebemos a mensagem de “Carregando…”.



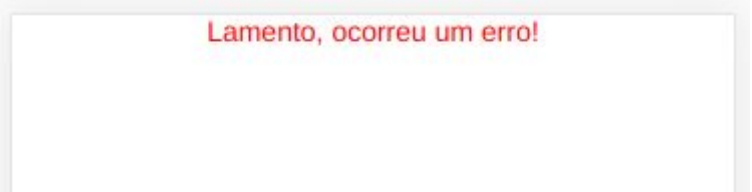
Em seguida, ao receber os resultados da API JSONPlaceholder, a tela é atualizada.

Agora, vamos testar chamar a API com o URL errado:

```
const response = await axios.get(
      "https://jsonplaceholderERRADO.typicode.com/users"
```

```
    );
```

Veja o resultado na tela:



Ou seja, ele entrou no tratamento de erros de nosso código, o método `catch()`.

Volte a URL para a correta, ou seja:

```
const response = await axios.get(
      "https://jsonplaceholder.typicode.com/users"
    );
```

1.3.6 Criando o componente Users

Vamos mover o código que trata sobre a listagem de usuários vindos da API para um novo componente. Assim, crie o diretório components e, em seguida, crie o arquivo Users.js neste novo diretório.

```
import React from "react";
import { useFindUsers } from '../hooks/useFindUsers';
import { StyleSheet, Text, View, FlatList } from "react-native";


function Users() {
 // use seu próprio hook para carregar os dados da API
 const { data, loading, error }  =  useFindUsers();


 const Item = ({ data }) => (
   <View style={styles.item}>
     <Text style={styles.title}>{data}</Text>
   </View>
 );
 const renderItem = ({ item }) => (
   <Item data={item.username + " : " + item.name} />
```

```
 );

 return (
   <View style={styles.container}>
     { loading ?
       <Text style={styles.title}>Carregando...</Text> :
       <FlatList
       data={data}
       renderItem={renderItem}
       keyExtractor={(item) => item.id.toString()}
     />
   }
   {error ? <Text style={styles.error}>{error}</Text> : null}
   </View>
 );
}
export default Users;

const styles = StyleSheet.create({
 container: {
   flex: 1,
 },
 item: {
   backgroundColor: "#77AAFF",
   padding: 10,
   marginVertical: 8,
   marginHorizontal: 16,
 },
 title: {
   fontSize: 14,
 },
 error: {
```

```
    color: "#FF0000",
    fontSize: 14,
    textAlign: "center",
  },
});
```

Agora, vamos alterar o arquivo App.js para instanciar o novo componente Users.js.

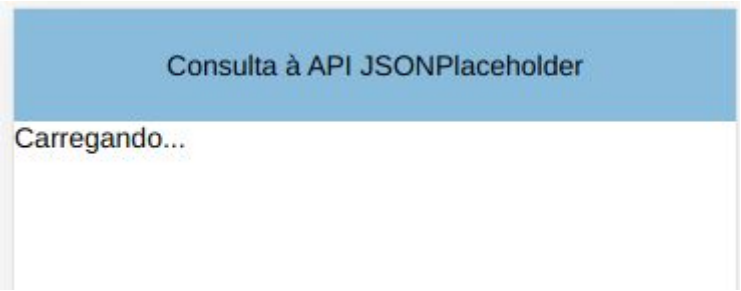
```
import { StatusBar } from "expo-status-bar";
import React from "react";
import Users from './components/Users';
import { StyleSheet, Text, View } from "react-native";

export default function App() {
 return (
  <View style={styles.container}>
             <Text     style={styles.header}>Consulta     à     API
JSONPlaceholder</Text>
    <Users />
    <StatusBar style="auto" />
  </View>
 );
}

const styles = StyleSheet.create({
 container: {
    flex: 1,
 },
 header: {
    backgroundColor: "#89BBDD",
    textAlign: "center",
    padding: 20,
 },
```

```
});
```

Veja que o resultado na tela não mudou muito, ou seja, ao carregar o componente pela primeira vez, recebemos a mensagem de “Carregando…”



Em seguida, ao receber os resultados da API JSONPlaceholder, a tela é atualizada.

1.3.7 Melhorando o State

Observe que em nosso hook `useFindUsers()` estamos fazendo o gerenciamento de estados para tratar das informações de usuários, estado de carregamento e erros da seguinte forma:

```
// Definimos os estados necessários para o nosso hook, isso
inclui:
// usuários (data), estado de carregamento (loading) e erros
(error)
 const [data, setData] = useState([]);
 const [loading, setLoading] = useState(false);
 const [error, setError] = useState(null);
```

Podemos agrupar esta informação em um objeto único, uma vez que todos estão tratando sobre as chamadas a uma API externa.

```
// Definimos os estados necessários para o nosso hook, isso
inclui:
// usuários (data), estado de carregamento (loading) e erros
(error)
 const [state, setDataState] =
     useState({loading: true, data: [], error: null});
```

Logo, altere o arquivo useFindUsers.js para o seguinte código:

```
import { useEffect, useState } from "react";
import axios from "axios";

export function useFindUsers() {
  // Definimos os estados necessários para o nosso hook, isso
inclui:
// usuários (data), estado de carregamento (loading) e erros
(error)
 const [state, setDataState] = useState({
   loading: true,
   data: [],
   error: null,
 });
```

```
// Busca os usuários da API JSONPlaceholder
async function listUsers() {
  try {
    const response = await axios.get(
      "https://jsonplaceholder.typicode.com/users"
    );
    // Atualiza o estado de 'data' com o resultado da API
    if (response.data) {
      //Atualiza o estado de 'data' e 'loading'
      setDataState({ data: response.data, loading: false });
    } else {
      //Atualiza o estado de 'data' e 'loading'
      setDataState({ data: [], loading: false });
    }
  } catch (err) {
    // Atualiza o estado de 'error', 'data' e de 'loading'
    setDataState({
      data: [],
      loading: false,
      error: "Lamento, ocorreu um erro!",
    });
  }
}
// A cada renderização do componente, o hook useEffect
//executa a função listUsers().
useEffect(() => {
  // Primeiro, definimos os estados de loading e error
  setDataState({loading:true, error: null});

  // executa a função listUsers().
  listUsers();
}, []);
```

```
 return { state };
}
```

Ainda, veja que estamos retornando o este objeto mais complexo, o `state`.

```
return { state };
```

Em seguida, altere o arquivo User.js:

```
...
function Users() {
 // use seu próprio hook para carregar os dados da API
 const { state } = useFindUsers();
 // Desestruture isLoading, data e error de state
 const { data, loading, error } = state;
...
```

Veja que estamos desestrurutando `isLoading, data` e `error` de `state`.

```
const { data, loading, error } = state;
```

Assim, não precisamos usar `state.data` ou `state.loading` ou `state.error` no código. Veja que não houve alteração do resultado na tela.

Funcionou!!!

1.3.8 Utilizando o hook useReducer

Após aprendermos e experimentarmos os conceitos básicos do hook `useState`, vamos analisar como o hook `useReducer` poder ser usado na execução de ações assíncronas, tais como fazer solicitações para uma API externa.

O hook `useReducer` também pode ser usado para atualizar o estado, tal como o useState, mas ele o faz de uma maneira mais sofisticada: ele aceita uma função redutora (*reducer function*) e um estado inicial e, depois, retorna o estado real e uma função de despacho (dispatch function). A função `dispatch` altera o estado de maneira implícita, mapeando ações para suas respectivas transições de estado.

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

Assim, crie em hooks o arquivo useFindUsersReducer.js com o seguinte código:

```
import { useEffect, useReducer } from "react";

import axios from "axios";
```

```
const initialState = {
 loading: true,
 data: [],
 error: null,
};

const reduce = (state, action) => {
 // Atualiza o estado de 'error', 'data' e de 'loading'
 // De acordo com o tipo de ação
 switch (action.type) {
   case "OnFetching":
     return {
       loading: true,
       data: [],
       error: null,
     };
   case "OnSuccess":
     return {
       loading: false,
       data: action.payload,
       error: null,
     };
   case "OnFailure":
     return {
       loading: false,
       data: [],
       error: "Lamento, ocorreu um erro!",
     };
   default:
     return state;
 }
};
```

```
// O hook é apenas uma simples função que podemos exportar
export function useFindUsersReducer(search) {
  //  Definimos  os  estados  necessários  para  o  nosso  hook,  isso
inclui:
  //  usuários  (data),  estado  de  carregamento  (loading)  e  erros
(error)
 const [state, dispatch] = useReducer(reduce, initialState);
 // A cada renderização do componente, o hook useEffect
 // executa a função listUsers().
 useEffect(() => {
   // Ao mover a função listUsers para dentro do useEffect,
   // podemos ver claramente os valores que ela usa.
   async function listUsers() {
     try {
       const response = await axios.get(
         "https://jsonplaceholder.typicode.com/users"
       );

       // Atualiza o estado de 'data' com o resultado da API
       if (response.data) {
          //Senão, atualiza o estado em caso de sucesso com o array
vindo da API
         dispatch({ type: "OnSuccess", payload: response.data });
       } else {
         //Atualiza o estado em caso de erro
         dispatch({ type: "OnFailure" });
       }
     } catch (err) {
       //Atualiza o estado em caso de erro
       dispatch({ type: "OnFailure" });
     }
```

```
    }

    // Primeiro, definimos os estados em caso de pesquisa
    dispatch({ type: "OnFetching" });

    // executa a função listUsers().
    listUsers();
  }, []);

  return { state };
}
```

Observe que, inicialmente, nós estruturamos nosso estado.

```
const initialState = {
  loading: true,
  data: [],
  error: null,
};
```

Essa estrutura nos permite obter facilmente os dados de status e resposta de uma requisição a uma API externa. Agora que temos nossa estrutura de dados do estado, nós precisávamos implementar três ações.

1. OnFetching - a ação a ser despachada quando uma solicitação de API está em andamento
2. OnSuccess - a ação a ser despachada quando uma solicitação da API é concluída com êxito
3. OnFailure - a ação a ser despachada e solicitações de uma API resulta em erro

Assim, a função `dispatch` altera o estado de maneira implícita, mapeando ações para suas respectivas transições de estado.

```
const reduce = (state, action) => {
  // Atualiza o estado de 'error', 'data' e de 'loading'
  // De acordo com o tipo de ação
  switch (action.type) {
    case "OnFetching":
      return {
```

```
        loading: true,
        data: [],
        error: null,
      };
    case "OnSuccess":
      return {
        loading: false,
        data: action.payload,
        error: null,
      };
    case "OnFailure":
      return {
        loading: false,
        data: [],
        error: "Lamento, ocorreu um erro!",
      };
    default:
      return state;
  }
};
```

O objeto que está sendo enviado com a função dispatch possui uma propriedade obrigatória `type` e uma propriedade opcional chamada `payload`. O `type` informa à função `reduce` qual transição de estado precisa ser aplicada e o `payload` pode ser usado adicionalmente pela função redutora para definir o novo estado.

```
dispatch({ type: "OnSuccess", payload: response.data });
```

Após analisarmos o código do hook `useFindUsersReducer`, vamos instanciar este novo hook no componente Users.js:

```
import React from "react";
import { useFindUsersReducer } from
'../hooks/useFindUsersReducer';
import { StyleSheet, Text, View, FlatList } from "react-native";
```

```
function Users() {
 // use seu próprio hook para carregar os dados da API
 const { state } = useFindUsersReducer();
 // Desestruture isLoading, data e error de state
 const { data, loading, error } = state;
...
```

Finalmente, veja que o comportamento na tela não alterou.

1.3.9 Quando usar o hook useState e o useReducer?

Abaixo, veremos alguns cenários onde podemos usar estes hooks:

Cenário	useState()	useReducer()
Tipo de estado	Use ao trabalhar com Number, Boolean e String	Use ao trabalhar com objeto ou array
Número de transições de estado	Deve ser usado quando você tiver apenas 1 ou 2 chamadas setState	Deve ser usado quando tivermos muitas transições de estado
Lógica de negócios	O hook useState não tem lógica de negócios	Quando sua aplicação envolve transformação ou manipulação de dados

		complexos
Local vs Global	Deve ser usado ao lidar com um único componente e precisar executar operações localmente	Quando quiser lidar com vários componentes, ou seja, para passar dados de um componente para outro.

Referências

- Site Oficial do React. Using the Effect Hook. Disponível em: https://reactjs.org/docs/hooks-effect.html
- Site Oficial do React. Hooks FAQ. Disponível em: https://reactjs.org/docs/hooks-faq.html#is-it-safe-to-omit-functions-from-the-list-of-dependencies
- Site MDN. Funções assíncronas. Disponível em: https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Statements/funcoes_assincronas