Material de Aula: Laboratório de Desenvolvimento de Aplicações para Dispositivos Móveis

Introdução

Bem-vindo(a) ao material de aula do Laboratório de Desenvolvimento de Aplicações para Dispositivos Móveis! Este curso foi cuidadosamente elaborado para fornecer a você uma compreensão abrangente e prática sobre o desenvolvimento de aplicativos móveis, com foco em tecnologias modernas e amplamente utilizadas no mercado. Ao longo deste material, exploraremos desde os fundamentos do desenvolvimento nativo e híbrido até tópicos avançados como integração com Web Services, manipulação de multimídia, bancos de dados locais e geolocalização.

Nosso objetivo é capacitá-lo(a) a construir aplicações robustas, eficientes e com uma excelente experiência de usuário, utilizando ferramentas como Expo e React Native. Cada módulo foi desenhado para ser prático, com exemplos de código, exercícios e desafios que o(a) ajudarão a solidificar seu aprendizado e a desenvolver as habilidades necessárias para se destacar no campo do desenvolvimento mobile.

Prepare-se para uma jornada de aprendizado dinâmico, onde a teoria se une à prática para transformar suas ideias em aplicativos funcionais. Vamos começar!

Autor: Manus AI

Módulo 1: Fundamentos e Ambiente de Desenvolvimento

29/ago: Tipos de desenvolvimentos: Nativo x Híbrido. Tecnologias utilizadas em cada tipo de desenvolvimento. SDK para desenvolvimento de aplicativos. Emuladores e ferramentas de teste. Expo e React Native. Expo-dev-client. Expo-updates. EAS (Expo Application Services). Expo Go.

1. Visão Geral:

O desenvolvimento de aplicações móveis oferece diversas abordagens, sendo as mais proeminentes o desenvolvimento nativo e o híbrido. A escolha entre uma e outra impacta diretamente o desempenho, custo, tempo de desenvolvimento e a experiência do usuário. Compreender as diferenças e as tecnologias associadas a cada uma é fundamental para qualquer desenvolvedor mobile. Além disso, a configuração de um ambiente de desenvolvimento eficiente, o uso de SDKs, emuladores e ferramentas de teste, bem como a familiarização com plataformas como Expo e React Native, são passos cruciais para iniciar a jornada no desenvolvimento de aplicativos.

2. Conceitos Fundamentais:

Desenvolvimento Nativo:

O desenvolvimento nativo refere-se à criação de aplicativos especificamente para uma plataforma móvel (iOS ou Android) usando as linguagens de programação e ferramentas fornecidas pelos próprios fabricantes do sistema

operacional. Para iOS, as linguagens são Swift ou Objective-C, e o ambiente de desenvolvimento é o Xcode. Para Android, as linguagens são Java ou Kotlin, e o ambiente é o Android Studio [1].

- **Vantagens:**
    - **Desempenho Superior:** Aplicativos nativos são otimizados para a plataforma, oferecendo o melhor desempenho e responsividade [2].
    - **Acesso Total aos Recursos do Dispositivo:** Permitem acesso irrestrito a todas as funcionalidades do hardware (câmera, GPS, sensores, etc.) e APIs específicas do sistema operacional [2].
    - **Melhor Experiência do Usuário (UX):** Seguem as diretrizes de design da plataforma, proporcionando uma interface e experiência familiar e intuitiva para o usuário [2].
    - **Segurança Aprimorada:** Geralmente oferecem maior segurança, pois são construídos com as ferramentas e práticas recomendadas pela plataforma.
- **Desvantagens:**
    - **Custo e Tempo de Desenvolvimento Elevados:** Requerem equipes separadas e bases de código distintas para iOS e Android, duplicando o esforço e o custo [2].
    - **Manutenção Complexa:** Atualizações e correções de bugs precisam ser implementadas em ambas as bases de código.
    - **Curva de Aprendizagem:** Desenvolvedores precisam dominar linguagens e ecossistemas específicos de cada plataforma.

Desenvolvimento Híbrido:

O desenvolvimento híbrido envolve a criação de aplicativos que utilizam tecnologias web (HTML, CSS, JavaScript) e os encapsulam em um contêiner nativo. Isso permite que uma única base de código seja executada em múltiplas plataformas (iOS e Android) [3]. Frameworks populares para desenvolvimento híbrido incluem React Native, Flutter, Ionic e Xamarin.

- **Vantagens:**
    - **Custo e Tempo de Desenvolvimento Reduzidos:** Uma única base de código para múltiplas plataformas economiza tempo e recursos [3].
    - **Manutenção Simplificada:** Atualizações e correções são aplicadas a uma única base de código.
    - **Reutilização de Conhecimento Web:** Desenvolvedores com experiência em tecnologias web podem migrar mais facilmente para o desenvolvimento mobile.
    - **Ciclo de Desenvolvimento Rápido:** Ideal para MVPs (Minimum Viable Products) e prototipagem rápida.
- **Desvantagens:**
    - **Desempenho Potencialmente Inferior:** Embora frameworks modernos como React Native e Flutter ofereçam desempenho próximo ao nativo, ainda pode haver limitações em aplicações muito complexas ou que exigem uso intensivo de recursos gráficos [2].
    - **Acesso Limitado a Recursos Nativos:** Pode exigir o uso de plugins ou bibliotecas de terceiros para acessar certas funcionalidades nativas, o que pode introduzir dependências e complexidade [2].
    - **Experiência do Usuário (UX) Menos Nativa:** Embora frameworks tentem replicar a UI nativa, pequenas inconsistências podem surgir.

SDK (Software Development Kit):

Um SDK é um conjunto de ferramentas de desenvolvimento de software em um pacote instalável. Ele facilita a criação de aplicativos para uma plataforma específica, fornecendo bibliotecas de código, exemplos, documentação, processos e APIs [4]. Exemplos incluem o Android SDK e o iOS SDK.

Emuladores e Ferramentas de Teste:

Emuladores (ou simuladores, no caso do iOS) são softwares que replicam o ambiente de um dispositivo móvel em um computador, permitindo que os desenvolvedores testem seus aplicativos sem a necessidade de um dispositivo físico. Ferramentas de teste, como Jest e React Native Testing Library (que veremos em módulos futuros), são essenciais para garantir a qualidade e a funcionalidade dos aplicativos.

3. Tecnologias e Ferramentas:

Expo e React Native:

React Native é um framework de código aberto criado pelo Facebook para construir aplicativos móveis nativos usando JavaScript e React. Ele permite que os desenvolvedores escrevam código uma vez e o executem em iOS e Android, compilando para componentes de UI nativos [5].

Expo é um conjunto de ferramentas e serviços construído sobre o React Native que simplifica e acelera o processo de desenvolvimento. Ele abstrai muitas das complexidades do desenvolvimento nativo, como a configuração de ambientes e a compilação de código nativo, permitindo que os desenvolvedores se concentrem na lógica do aplicativo [5].

- **Expo Go:** É um aplicativo cliente que permite executar e testar projetos Expo diretamente em dispositivos físicos ou emuladores sem a necessidade de compilar o código nativo. É ideal para prototipagem rápida e testes iniciais [6].
- **Expo-dev-client:** É uma biblioteca que adiciona várias ferramentas de desenvolvimento úteis às suas compilações de depuração. Ele permite que você lance atualizações (como de prévias de PRs) e fornece um launcher UI configurável. Uma compilação de desenvolvimento é uma compilação de depuração de um aplicativo que inclui a biblioteca `expo-dev-client` [7].
- **Expo-updates:** É uma biblioteca que permite que seu aplicativo gerencie atualizações remotas para o código do seu aplicativo. Ele se comunica com o serviço de atualização remoto configurado, permitindo atualizações over-the-air (OTA) [8].
- **EAS (Expo Application Services):** São serviços de nuvem profundamente integrados para aplicativos Expo e React Native, fornecidos pela equipe por trás do Expo. O EAS simplifica o processo de construção, envio e atualização de aplicativos, oferecendo serviços como EAS Build (para compilações nativas na nuvem), EAS Submit (para envio às lojas de aplicativos) e EAS Update (para atualizações OTA) [9].

4. Exemplos Práticos (Código):

Para começar com Expo e React Native, você pode seguir os passos abaixo:

1. **Instalar Node.js e npm (ou yarn):** Certifique-se de ter o Node.js instalado em sua máquina. Você pode baixá-lo do site oficial (nodejs.org).
2. **Instalar a CLI do Expo:** Abra seu terminal e execute o comando: `bash npm install -g expo-cli` ou `bash yarn global add expo-cli`
3. **Criar um novo projeto Expo:** `bash expo init MeuPrimeiroApp cd MeuPrimeiroApp` Escolha um template (por exemplo, `blank` para um projeto vazio).

4. **Iniciar o servidor de desenvolvimento:** `bash npm start` ou `bash expo start` Isso abrirá uma página no seu navegador com um QR code. Você pode escanear este QR code com o aplicativo Expo Go no seu celular (disponível na App Store e Google Play) para ver o aplicativo em tempo real, ou usar um emulador/simulador.

Exemplo de um componente React Native simples (App.js):

```
 import React from 'react';
import { StyleSheet, Text, View } from 'react-native';

export default function App() {
  return (
    <View style={styles.container}>
      <Text style={styles.title}>Olá, Mundo Mobile!</Text>
      <Text>Este é o meu primeiro aplicativo Expo React Native.</Text>
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
  title: {
    fontSize: 24,
    fontWeight: 'bold',
    marginBottom: 10,
  },
});
```

5. Exercícios e Desafios:

1. Crie um novo projeto Expo e modifique o texto exibido na tela para incluir seu nome.
2. Adicione um novo componente `Button` ao `App.js` que, ao ser pressionado, exiba um alerta com a mensagem

'Olá do Botão!'.

1. Pesquise e explique com suas palavras a diferença entre `expo start` e `expo start --dev-client`.
2. Explique a importância do EAS (Expo Application Services) no ciclo de vida de um aplicativo Expo.

6. Recursos Adicionais:

- **Documentação Oficial do Expo:** https://docs.expo.dev/
- **Documentação Oficial do React Native:** https://reactnative.dev/
- **Tutorial: Using React Native and Expo:** https://docs.expo.dev/tutorial/introduction/ [5]
- **Expo Dev Client:** https://docs.expo.dev/versions/latest/sdk/dev-client/ [7]
- **Expo Updates:** https://docs.expo.dev/versions/latest/sdk/updates/ [8]
- **EAS (Expo Application Services):** https://expo.dev/eas [9]

Referências:

[1] https://www.dio.me/articles/nativo-vs-hibrido [2] https://www.seidor.com/pt-br/blog/tipologias-desenvolvimento-movel [3] https://blog.leanwork.com.br/aplicativo-nativo-ou-hibrido-qual-escolher/ [4] https://clevertap.com/blog/what-is-an-sdk/ [5] https://docs.expo.dev/tutorial/introduction/ [6] https://docs.expo.dev/versions/latest/sdk/dev-client/ [7] https://docs.expo.dev/versions/latest/sdk/updates/ [8] https://expo.dev/eas [9] https://docs.expo.dev/eas/

Módulo 2: Construção de Interfaces e Navegação

01/set: Construção da interface do aplicativo.

1. Visão Geral:

A interface do usuário (UI) é o que o usuário vê e interage em um aplicativo. Em React Native, a construção da UI é feita através de componentes, que são blocos de construção reutilizáveis. Compreender como organizar esses componentes, estilizá-los e criar layouts responsivos é fundamental para desenvolver aplicativos visualmente atraentes e funcionais. Este tópico abordará os componentes básicos de UI, estilização com `StyleSheet` e a criação de layouts utilizando Flexbox.

2. Conceitos Fundamentais:

Componentes de UI:

React Native fornece um conjunto de componentes básicos que mapeiam diretamente para os componentes nativos da plataforma, garantindo uma experiência de usuário consistente e performática. Alguns dos componentes mais comuns incluem:

- **`View`:** O contêiner mais fundamental para construir UI. É similar a uma `div` no HTML e suporta layout com Flexbox, estilo, toque e controles de acessibilidade. É o bloco de construção para a maioria dos outros componentes [10].
- **`Text`:** Usado para exibir texto. Suporta aninhamento para aplicar diferentes estilos a partes do texto e herda estilos de texto de seus pais [11].
- **`Image`:** Usado para exibir diferentes tipos de imagens, incluindo imagens estáticas de recursos do aplicativo, imagens temporárias de armazenamento local ou imagens de rede [12].
- **`TextInput`:** Um componente para entrada de texto. Suporta uma variedade de configurações, como teclado numérico, senha, multilinhas, etc. [13].
- **`Button`:** Um componente básico para renderizar um botão. É simples e personalizável [14].
- **`ScrollView`:** Um componente de rolagem genérico que pode conter vários componentes e visualizações. É útil quando o conteúdo excede as dimensões da tela [15].
- **`FlatList`:** Um componente de lista de alto desempenho para exibir grandes quantidades de dados roláveis de forma eficiente. Renderiza apenas os itens que estão visíveis na tela, otimizando o desempenho [16].

Estilização com `StyleSheet`:

Em React Native, a estilização é feita usando JavaScript, de forma semelhante ao CSS, mas com algumas diferenças. O `StyleSheet.create` é uma API que otimiza a criação de estilos, garantindo que eles sejam criados apenas uma vez e referenciados por ID, o que melhora o desempenho [17].

Flexbox:

Flexbox é um modelo de layout unidimensional que permite organizar itens em uma linha ou coluna. É a principal forma de layout em React Native e funciona de forma muito semelhante ao Flexbox da web. Ele permite distribuir espaço entre os itens de um contêiner e controlar seu alinhamento [18].

- **`flexDirection`:** Define a direção principal em que os itens são dispostos (row, column, row-reverse, column-reverse).

- `justifyContent`: Alinha os itens ao longo do eixo principal do contêiner (flex-start, flex-end, center, space-between, space-around, space-evenly).
- `alignItems`: Alinha os itens ao longo do eixo transversal do contêiner (flex-start, flex-end, center, stretch, baseline).
- `flex`: Define como um item flexível deve crescer ou encolher para preencher o espaço disponível.

3. Tecnologias e Ferramentas:

- **React Native Core Components:** Os componentes básicos fornecidos pelo React Native são a base para a construção de qualquer interface.
- `StyleSheet` **API:** Para a criação e otimização de estilos.
- **Flexbox:** Para a criação de layouts responsivos e flexíveis.

4. Exemplos Práticos (Código):

Exemplo de uso de componentes básicos e estilização:

```
 import React from 'react';
import { StyleSheet, Text, View, Image, TextInput, Button, ScrollView } from 'react-native';

export default function App() {
  return (
    <ScrollView style={styles.scrollView}>
      <View style={styles.container}>
        <Text style={styles.title}>Minha Primeira Interface</Text>

        <Image
          source={{ uri: 'https://reactnative.dev/img/tiny_logo.png' }}
          style={styles.logo}
        />

        <TextInput
          style={styles.input}
          placeholder="Digite seu nome"
        />

        <Button
          title="Enviar"
          onPress={() => alert('Nome enviado!')}
          color="#841584"
        />

        <View style={styles.card}>
          <Text style={styles.cardTitle}>Cartão de Exemplo</Text>
          <Text>Este é um exemplo de um cartão com algum texto de demonstração.</Text>
        </View>

        <View style={styles.flexContainer}>
          <View style={styles.box1} />
          <View style={styles.box2} />
          <View style={styles.box3} />
        </View>

      </View>
    </ScrollView>
  );
}

const styles = StyleSheet.create({
  scrollView: {
    flex: 1,
    backgroundColor: '#f0f0f0',
  },
  container: {
    flex: 1,
    padding: 20,
    alignItems: 'center',
    justifyContent: 'center',
  },
  title: {
    fontSize: 28,
    fontWeight: 'bold',
    marginBottom: 20,
    color: '#333',
  },
  logo: {
    width: 100,
    height: 100,
    marginBottom: 20,
  },
  input: {
    height: 40,
    borderColor: 'gray',
    borderWidth: 1,
    width: '80%',
    paddingHorizontal: 10,
    marginBottom: 20,
    borderRadius: 5,
  },
  card: {
    backgroundColor: '#fff',
    padding: 15,
    borderRadius: 10,
    shadowColor: '#000',
    shadowOffset: { width: 0, height: 2 },
    shadowOpacity: 0.1,
    shadowRadius: 4,
    elevation: 5,
```

```
    marginTop: 20,
    width: '90%',
  },
  cardTitle: {
    fontSize: 20,
    fontWeight: 'bold',
    marginBottom: 10,
    color: '#555',
  },
  flexContainer: {
    flexDirection: 'row',
    justifyContent: 'space-around',
    alignItems: 'center',
    width: '100%',
    marginTop: 30,
    height: 120,
    backgroundColor: '#e0e0e0',
    borderRadius: 10,
  },
  box1: {
    width: 50,
    height: 50,
    backgroundColor: 'red',
    borderRadius: 5,
  },
  box2: {
    width: 50,
    height: 50,
    backgroundColor: 'green',
    borderRadius: 5,
  },
  box3: {
    width: 50,
    height: 50,
    backgroundColor: 'blue',
    borderRadius: 5,
  },
});
```

5. Exercícios e Desafios:

1. Modifique o exemplo acima para criar um layout de duas colunas usando Flexbox, onde cada coluna contém um `Text` e um `Image`.
2. Adicione um componente `FlatList` que exiba uma lista de 5 itens. Cada item deve ter um `Text` com o nome do item e um `Button` para

remover o item da lista (apenas simule a remoção com um `alert`). 3. Experimente diferentes valores para `justifyContent` e `alignItems` no `flexContainer` do exemplo e observe as mudanças no layout.

6. Recursos Adicionais:

- **Documentação React Native - Core Components:** https://reactnative.dev/docs/components-and-apis
- **Documentação React Native - StyleSheet:** https://reactnative.dev/docs/stylesheet
- **Documentação React Native - Flexbox:** https://reactnative.dev/docs/flexbox

Referências:

[10] https://reactnative.dev/docs/view [11] https://reactnative.dev/docs/text [12] https://reactnative.dev/docs/image [13] https://reactnative.dev/docs/textinput [14] https://reactnative.dev/docs/button [15] https://reactnative.dev/docs/scrollview [16] https://reactnative.dev/docs/flatlist [17] https://reactnative.dev/docs/stylesheet [18] https://reactnative.dev/docs/flexbox

05/set: React Navigation: Stack, Bottom Tabs e Drawer.

1. Visão Geral:

A navegação é um aspecto crucial de qualquer aplicativo móvel, permitindo que os usuários se movam entre diferentes telas e funcionalidades de forma intuitiva. O React Navigation é a solução de navegação mais popular e amplamente utilizada para aplicativos React Native, oferecendo uma variedade de navegadores pré-construídos que se adaptam a diferentes padrões de UI. Neste tópico, exploraremos os navegadores mais comuns: Stack, Bottom Tabs e Drawer, e como implementá-los em seu aplicativo.

2. Conceitos Fundamentais:

React Navigation:

React Navigation é uma biblioteca modular e extensível para navegação em aplicativos React Native. Ele fornece componentes de navegação que se integram perfeitamente com o ciclo de vida do React e permitem a criação de experiências de navegação complexas e personalizadas [19].

Tipos de Navegadores:

- **Stack Navigator (`createStackNavigator`):**
    - **Conceito:** Implementa um padrão de navegação em pilha, onde cada nova tela é colocada no topo da pilha. Ao navegar para trás, a tela atual é removida da pilha, revelando a tela anterior. É ideal para fluxos de navegação lineares, como um fluxo de checkout ou detalhes de um item [20].
    - **Uso:** Perfeito para navegação hierárquica, onde o usuário avança e retrocede entre telas relacionadas.
- **Bottom Tabs Navigator (`createBottomTabNavigator`):**
    - **Conceito:** Exibe uma barra de abas na parte inferior da tela, permitindo que os usuários alternem rapidamente entre diferentes seções principais do aplicativo. Cada aba geralmente representa uma rota de nível superior [21].
    - **Uso:** Ideal para aplicativos com várias seções principais que precisam ser acessadas frequentemente e de forma direta.
- **Drawer Navigator (`createDrawerNavigator`):**
    - **Conceito:** Apresenta um menu lateral (gaveta) que desliza para fora da borda da tela, geralmente do lado esquerdo. É comumente usado para navegação de nível superior ou para acessar configurações e outras funcionalidades menos prioritárias [22].
    - **Uso:** Adequado para aplicativos com muitas seções ou funcionalidades que não cabem em uma barra de abas inferior.

3. Tecnologias e Ferramentas:

- `@react-navigation/native`: O pacote principal do React Navigation.
- `@react-navigation/stack`: Para o Stack Navigator.
- `@react-navigation/bottom-tabs`: Para o Bottom Tabs Navigator.
- `@react-navigation/drawer`: Para o Drawer Navigator.
- `react-native-screens` e `react-native-safe-area-context`: Dependências essenciais para o funcionamento do React Navigation.

4. Exemplos Práticos (Código):

Para usar o React Navigation, primeiro você precisa instalá-lo e suas dependências:

```
npm install @react-navigation/native
npx expo install react-native-screens react-native-safe-area-context
```

Em seguida, instale os navegadores que você pretende usar:

```
npm install @react-navigation/stack
npm install @react-navigation/bottom-tabs
npm install @react-navigation/drawer
```

Exemplo de Stack Navigator:

```
import * as React from 'react';
import { View, Text, Button, StyleSheet } from 'react-native';
import { NavigationContainer } from '@react-navigation/native';
import { createStackNavigator } from '@react-navigation/stack';

function HomeScreen({ navigation }) {
  return (
    <View style={styles.container}>
      <Text style={styles.title}>Tela Inicial</Text>
      <Button
        title="Ir para Detalhes"
        onPress={() => navigation.navigate('Details')}
      />
    </View>
  );
}

function DetailsScreen({ navigation }) {
  return (
    <View style={styles.container}>
      <Text style={styles.title}>Tela de Detalhes</Text>
      <Button
        title="Voltar para Início"
        onPress={() => navigation.goBack()}
      />
    </View>
  );
}

const Stack = createStackNavigator();

function App() {
  return (
    <NavigationContainer>
      <Stack.Navigator initialRouteName="Home">
        <Stack.Screen name="Home" component={HomeScreen} />
        <Stack.Screen name="Details" component={DetailsScreen} />
      </Stack.Navigator>
    </NavigationContainer>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    alignItems: 'center',
    justifyContent: 'center',
  },
  title: {
    fontSize: 24,
    marginBottom: 20,
  },
});

export default App;
```

Exemplo de Bottom Tabs Navigator:

```
 import * as React from 'react';
import { Text, View, StyleSheet } from 'react-native';
import { NavigationContainer } from '@react-navigation/native';
import { createBottomTabNavigator } from '@react-navigation/bottom-tabs';
import Ionicons from 'react-native-vector-icons/Ionicons';

function SettingsScreen() {
  return (
    <View style={styles.container}>
      <Text style={styles.title}>Configurações!</Text>
    </View>
  );
}

const Tab = createBottomTabNavigator();

function App() {
  return (
    <NavigationContainer>
      <Tab.Navigator
        screenOptions={({ route }) => ({
          tabBarIcon: ({ focused, color, size }) => {
            let iconName;

            if (route.name === 'Home') {
              iconName = focused
                ? 'home'
                : 'home-outline';
            } else if (route.name === 'Settings') {
              iconName = focused ? 'settings' : 'settings-outline';
            }

            return <Ionicons name={iconName} size={size} color={color} />;
          },
          tabBarActiveTintColor: 'tomato',
          tabBarInactiveTintColor: 'gray',
        })}
      >
        <Tab.Screen name="Home" component={HomeScreen} />
        <Tab.Screen name="Settings" component={SettingsScreen} />
      </Tab.Navigator>
    </NavigationContainer>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
  },
  title: {
    fontSize: 24,
  },
});

export default App;
```

Exemplo de Drawer Navigator:

```
 import * as React from 'react';
import { Button, View, StyleSheet, Text } from 'react-native';
import { createDrawerNavigator } from '@react-navigation/drawer';
import { NavigationContainer } from '@react-navigation/native';

function HomeScreen({ navigation }) {
  return (
    <View style={styles.container}>
      <Text style={styles.title}>Tela Inicial</Text>
      <Button
        onPress={() => navigation.openDrawer()}
        title="Abrir Gaveta"
      />
    </View>
  );
}

function NotificationsScreen({ navigation }) {
  return (
    <View style={styles.container}>
      <Text style={styles.title}>Tela de Notificações</Text>
      <Button
        onPress={() => navigation.goBack()}
        title="Voltar para Início"
      />
    </View>
  );
}

const Drawer = createDrawerNavigator();

function App() {
  return (
    <NavigationContainer>
      <Drawer.Navigator initialRouteName="Home">
        <Drawer.Screen name="Home" component={HomeScreen} />
        <Drawer.Screen name="Notifications" component={NotificationsScreen} />
      </Drawer.Navigator>
    </NavigationContainer>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    alignItems: 'center',
    justifyContent: 'center',
  },
  title: {
    fontSize: 24,
    marginBottom: 20,
  },
});

export default App;
```

5. Exercícios e Desafios:

1. Combine os três navegadores (Stack, Bottom Tabs e Drawer) em um único aplicativo. Por exemplo, tenha um `BottomTabNavigator` como navegador principal, e dentro de uma das abas, use um `StackNavigator`. Adicione um `DrawerNavigator` que possa ser acessado de qualquer tela.
2. Personalize o cabeçalho do `StackNavigator` (título, botões, cores).
3. Adicione ícones personalizados às abas do `BottomTabNavigator`.
4. Altere a posição da gaveta do `DrawerNavigator` para o lado direito da tela.

6. Recursos Adicionais:

- **Documentação Oficial do React Navigation:** https://reactnavigation.org/
- **React Navigation - Stack Navigator:** https://reactnavigation.org/docs/stack-navigator/ [20]
- **React Navigation - Bottom Tab Navigator:** https://reactnavigation.org/docs/bottom-tab-navigator/ [21]

- **React Navigation - Drawer Navigator:** https://reactnavigation.org/docs/drawer-navigator/ [22]

Referências:

[19] https://reactnavigation.org/ [20] https://reactnavigation.org/docs/stack-navigator/ [21] https://reactnavigation.org/docs/bottom-tab-navigator/ [22] https://reactnavigation.org/docs/drawer-navigator/

Módulo 3: Testes e Arquitetura de Projeto

19/set: Configuração de um ambiente de testes com jest e projetando o core do aplicativo.

1. Visão Geral:

Testar aplicações é uma prática fundamental no desenvolvimento de software para garantir a qualidade, confiabilidade e o comportamento esperado do sistema. Em aplicações móveis, onde a experiência do usuário é crucial, a automação de testes se torna ainda mais importante. Este tópico abordará a configuração de um ambiente de testes utilizando Jest, um popular framework de testes JavaScript, e introduzirá conceitos de arquitetura de software para projetar o 'core' do aplicativo, visando a testabilidade e a manutenibilidade.

2. Conceitos Fundamentais:

Testes de Software:

Testes de software são processos que verificam se um sistema de software atende aos requisitos especificados e se comporta conforme o esperado. Existem diferentes tipos de testes, cada um com um propósito específico:

- **Testes Unitários:** Focam em testar as menores unidades de código isoladamente (funções, métodos, classes). O objetivo é verificar se cada unidade funciona corretamente em si mesma [23].
- **Testes de Integração:** Verificam a interação entre diferentes unidades ou módulos do sistema. O objetivo é garantir que as partes do sistema funcionem bem juntas [23].
- **Testes de Componente:** Em React Native, testam componentes de UI isoladamente, verificando seu comportamento e renderização.
- **Testes de Ponta a Ponta (E2E):** Simulam o fluxo completo do usuário através do aplicativo, testando a aplicação como um todo, desde a interface até o backend.

Jest:

Jest é um framework de testes JavaScript desenvolvido pelo Facebook, amplamente utilizado para testar aplicações React e React Native. Ele é conhecido por sua simplicidade, velocidade e por vir com muitas funcionalidades "prontas para uso", como um runner de testes, asserções e mocks [24].

- **Características Principais do Jest:**
  - **Zero Configuration:** Para muitos projetos, o Jest funciona com pouca ou nenhuma configuração.
  - **Fast:** Executa testes em paralelo e otimiza a execução de testes para projetos grandes.
  - **Snapshot Testing:** Permite capturar um "snapshot" de um componente renderizado e compará-lo com um snapshot anterior para detectar mudanças inesperadas na UI.
  - **Mocking:** Facilita a criação de mocks para dependências, permitindo testar unidades de código isoladamente.

TDD (Test-Driven Development):

TDD é uma metodologia de desenvolvimento de software onde os testes são escritos antes do código de produção. O ciclo de TDD consiste em:

1. **Red (Vermelho):** Escrever um teste que falha (porque a funcionalidade ainda não existe).
2. **Green (Verde):** Escrever o código mínimo necessário para fazer o teste passar.
3. **Refactor (Refatorar):** Refatorar o código para melhorar sua qualidade, mantendo os testes passando.

Essa abordagem garante que o código seja testável e que os requisitos sejam compreendidos antes da implementação.

Arquitetura de Software e o 'Core' do Aplicativo:

Projetar o 'core' do aplicativo refere-se à definição da estrutura central da lógica de negócios, independente da interface do usuário ou de detalhes de infraestrutura (como banco de dados ou APIs externas). Uma arquitetura bem definida, como a Clean Architecture ou Domain-Driven Design (DDD), visa:

- **Separação de Preocupações:** Dividir o sistema em camadas distintas, onde cada camada tem uma responsabilidade única. Isso torna o código mais fácil de entender, manter e testar.
- **Independência:** O 'core' do aplicativo deve ser independente de frameworks, UI e bancos de dados. Isso significa que a lógica de negócios não deve ser afetada por mudanças nessas camadas externas.
- **Testabilidade:** Ao isolar a lógica de negócios, torna-se muito mais fácil escrever testes unitários para o 'core' sem a necessidade de simular toda a aplicação.

3. Tecnologias e Ferramentas:

- **Jest:** Framework de testes JavaScript.
- **Node.js:** Ambiente de execução para Jest.
- **npm/yarn:** Gerenciadores de pacotes para instalar Jest e outras dependências.

4. Exemplos Práticos (Código):

Configurando Jest em um projeto React Native (Expo):

1. **Instale Jest:** `bash npm install --save-dev jest` ou `bash yarn add --dev jest`
2. **Crie um arquivo de configuração `jest.config.js` (opcional, mas recomendado para personalização):** `javascript module.exports = { preset: 'react-native', setupFiles: ['<rootDir>/jest-setup.js'], // Opcional: para configurações adicionais transformIgnorePatterns: ['node_modules/(?!(jest-)?react-native|@react-native|@expo|expo(nent)?|@expo-google-fonts|react-navigation|@react-navigation/.*|@unimodules/.*|unimodules|sentry-expo|native-base|react-native-svg)'], };`
3. **Crie um script no `package.json` para executar os testes:** `json { "name": "MeuApp", "version": "1.0.0", "main": "index.js", "scripts": { "test": "jest" }, ... }`

Exemplo de Teste Unitário com Jest:

Vamos criar uma função simples e testá-la. Crie um arquivo `utils.js`:

```
// utils.js
export function sum(a, b) {
  return a + b;
}

export function multiply(a, b) {
  return a * b;
}
```

Agora, crie um arquivo de teste `utils.test.js` na mesma pasta ou em uma pasta `__tests__`:

```
// utils.test.js
import { sum, multiply } from './utils';

describe('Funções Utilitárias', () => {
  test('sum deve retornar a soma de dois números', () => {
    expect(sum(1, 2)).toBe(3);
    expect(sum(0, 0)).toBe(0);
    expect(sum(-1, 1)).toBe(0);
  });

  test('multiply deve retornar o produto de dois números', () => {
    expect(multiply(2, 3)).toBe(6);
    expect(multiply(5, 0)).toBe(0);
    expect(multiply(-2, 4)).toBe(-8);
  });
});
```

Para executar os testes, basta rodar no terminal:

```
npm test
```

5. Exercícios e Desafios:

1. Crie uma nova função em `utils.js` chamada `subtract(a, b)` que retorne a diferença entre `a` e `b`. Escreva testes unitários para essa função em `utils.test.js`.
2. Pesquise sobre "mocking" com Jest e crie um exemplo simples onde você "mocka" uma função para controlar seu comportamento em um teste.
3. Discuta a importância da separação de preocupações em uma arquitetura de software e como isso contribui para a testabilidade do código.

6. Recursos Adicionais:

- **Documentação Oficial do Jest:** https://jestjs.io/
- **Guia de Testes do React Native:** https://reactnative.dev/docs/testing
- **Introdução ao TDD:** https://martinfowler.com/bliki/TestDrivenDevelopment.html

Referências:

[23] https://www.browserstack.com/guide/unit-testing-vs-integration-testing [24] https://jestjs.io/

08/set: Passagem de parâmetros e estrutura do projeto.

1. Visão Geral:

Em aplicações complexas, é comum a necessidade de compartilhar dados entre diferentes telas e componentes. A passagem de parâmetros é um mecanismo essencial para que as telas possam receber informações e exibir conteúdo dinâmico. Além disso, uma estrutura de projeto bem organizada é crucial para a manutenibilidade,

escalabilidade e colaboração em equipes de desenvolvimento. Este tópico abordará como passar parâmetros entre telas no React Navigation e as melhores práticas para estruturar um projeto React Native.

2. Conceitos Fundamentais:

Passagem de Parâmetros entre Telas:

No React Navigation, a passagem de parâmetros é realizada através do objeto `route` que é passado para os componentes da tela. Os parâmetros podem ser enviados ao navegar para uma nova tela e acessados na tela de destino [25].

- **Enviando Parâmetros:** Ao chamar `navigation.navigate()`, você pode passar um segundo argumento que é um objeto contendo os parâmetros. Por exemplo: `navigation.navigate("Detalhes", { itemId: 86, otherParam: "qualquer coisa" })`.
- **Acessando Parâmetros:** Na tela de destino, os parâmetros podem ser acessados via `route.params`. Por exemplo: `const { itemId, otherParam } = route.params;`.

Estrutura do Projeto React Native:

Uma estrutura de projeto consistente e lógica é vital para a organização do código, especialmente em projetos maiores. Embora não exista uma única estrutura "certa", algumas práticas são amplamente adotadas para promover a clareza e a manutenibilidade:

- **Organização por Recurso/Feature:** Agrupar arquivos relacionados a uma funcionalidade específica em uma única pasta (e.g., `src/features/auth`, `src/features/products`). Isso facilita a localização de código e a remoção de funcionalidades.
- **Organização por Tipo:** Agrupar arquivos por seu tipo (e.g., `src/components`, `src/screens`, `src/services`, `src/utils`). Esta abordagem é mais comum em projetos menores.
- **Camadas de Abstração:** Separar a lógica de negócios da interface do usuário e da camada de dados. Isso pode incluir pastas para:
  - `assets/`: Imagens, fontes, ícones.
  - `components/`: Componentes de UI reutilizáveis (botões, cards, etc.).
  - `screens/` ou `pages/`: Componentes que representam telas completas do aplicativo.
  - `navigation/`: Configurações de navegação.
  - `services/` ou `api/`: Funções para interagir com APIs externas.
  - `utils/`: Funções utilitárias e helpers.
  - `hooks/`: Custom Hooks do React.
  - `context/` ou `store/`: Gerenciamento de estado global (Context API, Redux, Zustand).

3. Tecnologias e Ferramentas:

- **React Navigation:** Para a passagem de parâmetros entre telas.
- **Sistema de arquivos:** Para organizar a estrutura do projeto.

4. Exemplos Práticos (Código):

Exemplo de Passagem de Parâmetros:

```
 import * as React from 'react';
import { View, Text, Button, StyleSheet } from 'react-native';
import { NavigationContainer } from '@react-navigation/native';
import { createStackNavigator } from '@react-navigation/stack';

function HomeScreen({ navigation }) {
  return (
    <View style={styles.container}>
      <Text style={styles.title}>Tela Inicial</Text>
      <Button
        title="Ir para Detalhes com Parâmetros"
        onPress={() =>
          navigation.navigate('Details', {
            itemId: 42,
            otherParam: 'algo que você passou',
          })
        }
      />
    </View>
  );
}

function DetailsScreen({ route, navigation }) {
  const { itemId, otherParam } = route.params;

  return (
    <View style={styles.container}>
      <Text style={styles.title}>Tela de Detalhes</Text>
      <Text>ID do Item: {JSON.stringify(itemId)}</Text>
      <Text>Outro Parâmetro: {JSON.stringify(otherParam)}</Text>
      <Button
        title="Voltar"
        onPress={() => navigation.goBack()}
      />
    </View>
  );
}

const Stack = createStackNavigator();

function App() {
  return (
    <NavigationContainer>
      <Stack.Navigator initialRouteName="Home">
        <Stack.Screen name="Home" component={HomeScreen} />
        <Stack.Screen name="Details" component={DetailsScreen} />
      </Stack.Navigator>
    </NavigationContainer>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    alignItems: 'center',
    justifyContent: 'center',
  },
  title: {
    fontSize: 24,
    marginBottom: 20,
  },
});

export default App;
```

Exemplo de Estrutura de Projeto (Organização por Recurso):

```
MeuProjeto/
├── App.js
├── app.json
├── package.json
├── babel.config.js
├── node_modules/
└── src/
    ├── assets/
    │   ├── images/
    │   └── fonts/
    ├── components/
    │   ├── ButtonComponent.js
    │   └── CardComponent.js
    ├── features/
    │   ├── auth/
    │   │   ├── screens/
    │   │   │   ├── LoginScreen.js
    │   │   │   └── RegisterScreen.js
    │   │   ├── components/
    │   │   │   └── AuthForm.js
    │   │   └── services/
    │   │       └── authService.js
    │   └── products/
    │       ├── screens/
    │       │   ├── ProductListScreen.js
    │       │   └── ProductDetailScreen.js
    │       ├── components/
    │       │   └── ProductCard.js
    │       └── services/
    │           └── productService.js
    ├── navigation/
    │   ├── AppNavigator.js
    │   ├── AuthNavigator.js
    │   └── MainNavigator.js
    ├── services/
    │   └── api.js
    ├── utils/
    │   └── helpers.js
    └── hooks/
        └── useAuth.js
```

5. Exercícios e Desafios:

1. Modifique o exemplo de passagem de parâmetros para que a tela de detalhes exiba uma mensagem diferente dependendo de um parâmetro booleano (`isAdmin`) passado da tela inicial.
2. Crie uma nova tela chamada `ProfileScreen` e navegue para ela a partir da `HomeScreen` , passando o nome do usuário como parâmetro. Exiba o nome do usuário na `ProfileScreen` .
3. Proponha uma estrutura de pastas para um aplicativo de lista de tarefas (`Todo App`) que inclua funcionalidades de adicionar, editar e excluir tarefas, além de uma tela de configurações. Justifique suas escolhas.

6. Recursos Adicionais:

- **React Navigation - Passing parameters to routes:** https://reactnavigation.org/docs/passing-params/
- **React Native Folder Structure Best Practices:** https://medium.com/react-native-training/react-native-folder-structure-best-practices-c20d70513d7e

Referências:

[25] https://reactnavigation.org/docs/passing-params/

12/set: Implementação das telas.

1. Visão Geral:

Após aprender sobre a construção de interfaces com componentes básicos, estilização e navegação, o próximo passo é integrar esses conhecimentos para implementar telas completas em um aplicativo React Native. A implementação de telas envolve a combinação de múltiplos componentes, a aplicação de estilos consistentes, a integração com a lógica de navegação e, eventualmente, a conexão com dados. Este tópico focará nas melhores práticas para desenvolver telas funcionais e reutilizáveis, garantindo uma experiência de usuário coesa.

2. Conceitos Fundamentais:

Reutilização de Componentes:

Um dos pilares do React (e, consequentemente, do React Native) é a reutilização de componentes. Em vez de reescrever o mesmo código de UI várias vezes, você pode criar componentes genéricos (como `Button`, `Card`, `Header`) e reutilizá-los em diferentes telas. Isso não só acelera o desenvolvimento, mas também garante consistência visual e facilita a manutenção [26].

Boas Práticas de Desenvolvimento de UI:

- **Componentização:** Dividir a UI em componentes pequenos e focados, cada um com sua própria responsabilidade.
- **Separação de Preocupações:** Manter a lógica de apresentação (como o componente se parece) separada da lógica de negócios (o que o componente faz).
- **Props para Personalização:** Usar `props` para tornar os componentes reutilizáveis e configuráveis, permitindo que eles se adaptem a diferentes contextos sem precisar ser reescritos.
- **Gerenciamento de Estado:** Gerenciar o estado local da tela de forma eficiente, utilizando `useState` e `useEffect` para lidar com interações do usuário e efeitos colaterais.
- **Responsividade:** Projetar telas que se adaptem a diferentes tamanhos de tela e orientações (retrato/paisagem) usando Flexbox e unidades relativas.
- **Acessibilidade:** Considerar a acessibilidade desde o início, garantindo que o aplicativo seja utilizável por pessoas com deficiência (e.g., uso de `accessibilityLabel`, `role`).

Integração de Lógica de Negócios com a Interface:

As telas não são apenas estáticas; elas precisam interagir com a lógica de negócios do aplicativo. Isso pode envolver:

- **Chamadas de API:** Buscar dados de um servidor remoto para exibir na tela.
- **Manipulação de Dados:** Processar e formatar dados antes de exibi-los.
- **Eventos do Usuário:** Responder a cliques de botões, entradas de texto e outros gestos do usuário, acionando funções da lógica de negócios.
- **Gerenciamento de Estado Global:** Utilizar Context API (ou bibliotecas como Redux) para compartilhar estado entre múltiplas telas e componentes, evitando o "prop drilling" (passar props por muitos níveis de componentes).

3. Tecnologias e Ferramentas:

- **React Native Core Components:** `View`, `Text`, `Image`, `TextInput`, `Button`, `FlatList`, `ScrollView`.
- **`StyleSheet`:** Para estilização.
- **React Navigation:** Para navegação entre telas.
- **React Hooks:** `useState`, `useEffect`, `useContext` para gerenciamento de estado e efeitos colaterais.

4. Exemplos Práticos (Código):

Vamos criar um exemplo de uma tela de "Lista de Produtos" que exibe uma lista de itens e permite navegar para uma tela de "Detalhes do Produto".

`screens/ProductListScreen.js`:

```
 import React, { useState } from 'react';
import { View, Text, FlatList, TouchableOpacity, StyleSheet } from 'react-native';

const productsData = [
  { id: '1', name: 'Smartphone X', price: 1200, description: 'Um smartphone de última geração.' },
  { id: '2', name: 'Laptop Pro', price: 2500, description: 'Laptop potente para profissionais.' },
  { id: '3', name: 'Fone de Ouvido Bluetooth', price: 150, description: 'Áudio de alta qualidade sem
fio.' },
  { id: '4', name: 'Smartwatch Fit', price: 300, description: 'Monitora sua saúde e atividades.' },
];

function ProductListItem({ product, onPress }) {
  return (
    <TouchableOpacity style={styles.itemContainer} onPress={onPress}>
      <Text style={styles.itemName}>{product.name}</Text>
      <Text style={styles.itemPrice}>R$ {product.price.toFixed(2)}</Text>
    </TouchableOpacity>
  );
}

export default function ProductListScreen({ navigation }) {
  const renderItem = ({ item }) => (
    <ProductListItem
      product={item}
      onPress={() => navigation.navigate('ProductDetail', { product: item })}
    />
  );

  return (
    <View style={styles.container}>
      <Text style={styles.header}>Lista de Produtos</Text>
      <FlatList
        data={productsData}
        renderItem={renderItem}
        keyExtractor={item => item.id}
        contentContainerStyle={styles.listContent}
      />
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    padding: 16,
    backgroundColor: '#f8f8f8',
  },
  header: {
    fontSize: 28,
    fontWeight: 'bold',
    marginBottom: 20,
    textAlign: 'center',
    color: '#333',
  },
  listContent: {
    paddingBottom: 20,
  },
  itemContainer: {
    backgroundColor: '#fff',
    padding: 15,
    borderRadius: 8,
    marginBottom: 10,
    flexDirection: 'row',
    justifyContent: 'space-between',
    alignItems: 'center',
    shadowColor: '#000',
    shadowOffset: { width: 0, height: 1 },
    shadowOpacity: 0.2,
    shadowRadius: 1.41,
    elevation: 2,
  },
  itemName: {
    fontSize: 18,
    fontWeight: '600',
    color: '#555',
  },
  itemPrice: {
    fontSize: 16,
    fontWeight: 'bold',
    color: '#007bff',
```

```
  },
});
```

`screens/ProductDetailScreen.js`:

```
 import React from 'react';
import { View, Text, StyleSheet, Button } from 'react-native';

export default function ProductDetailScreen({ route, navigation }) {
  const { product } = route.params;

  return (
    <View style={styles.container}>
      <Text style={styles.productName}>{product.name}</Text>
      <Text style={styles.productPrice}>R$ {product.price.toFixed(2)}</Text>
      <Text style={styles.productDescription}>{product.description}</Text>
      <Button
        title="Voltar para Lista"
        onPress={() => navigation.goBack()}
      />
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    padding: 20,
    alignItems: 'center',
    justifyContent: 'center',
    backgroundColor: '#f8f8f8',
  },
  productName: {
    fontSize: 32,
    fontWeight: 'bold',
    marginBottom: 10,
    color: '#333',
    textAlign: 'center',
  },
  productPrice: {
    fontSize: 24,
    fontWeight: 'bold',
    color: '#007bff',
    marginBottom: 20,
  },
  productDescription: {
    fontSize: 16,
    textAlign: 'center',
    color: '#666',
    marginBottom: 30,
    lineHeight: 24,
  },
});
```

`navigation/AppNavigator.js` (para integrar as telas):

```
import * as React from 'react';
import { NavigationContainer } from '@react-navigation/native';
import { createStackNavigator } from '@react-navigation/stack';

import ProductListScreen from '../screens/ProductListScreen';
import ProductDetailScreen from '../screens/ProductDetailScreen';

const Stack = createStackNavigator();

function AppNavigator() {
  return (
    <NavigationContainer>
      <Stack.Navigator initialRouteName="ProductList">
        <Stack.Screen
          name="ProductList"
          component={ProductListScreen}
          options={{ title: 'Produtos' }}
        />
        <Stack.Screen
          name="ProductDetail"
          component={ProductDetailScreen}
          options={({ route }) => ({ title: route.params.product.name })}
        />
      </Stack.Navigator>
    </NavigationContainer>
  );
}

export default AppNavigator;
```

E no seu `App.js` principal, você importaria e renderizaria o `AppNavigator`:

```
import AppNavigator from './navigation/AppNavigator';

export default function App() {
  return <AppNavigator />;
}
```

5. Exercícios e Desafios:

1. Adicione uma funcionalidade de busca na `ProductListScreen` que filtre os produtos exibidos com base no texto digitado pelo usuário.
2. Na `ProductDetailScreen`, adicione um botão "Adicionar ao Carrinho" que, ao ser clicado, exiba um alerta com o nome do produto e uma mensagem "Adicionado ao carrinho!".
3. Crie um componente reutilizável `CustomHeader` que possa ser usado em ambas as telas (`ProductListScreen` e `ProductDetailScreen`) para exibir um título e, opcionalmente, um botão de volta.
4. Pesquise sobre `FlatList` e suas propriedades de otimização (`initialNumToRender`, `windowSize`, `removeClippedSubviews`). Explique como elas podem melhorar o desempenho de listas grandes.

6. Recursos Adicionais:

- **React Native - Componentes e APIs:** https://reactnative.dev/docs/components-and-apis
- **React Native - Hooks:** https://reactnative.dev/docs/hooks
- **React Navigation - Passando parâmetros:** https://reactnavigation.org/docs/passing-params/
- **React Native - Performance:** https://reactnative.dev/docs/performance

Referências:

[26] https://reactnative.dev/docs/components-and-apis

22/set: Implementação do core do projeto: value-objects e entities

1. Visão Geral:

No desenvolvimento de software, especialmente em sistemas complexos, a forma como modelamos o domínio do negócio é crucial para a clareza, manutenibilidade e escalabilidade do código. O Domain-Driven Design (DDD) oferece um conjunto de princípios e padrões para lidar com essa complexidade, colocando o domínio no centro do desenvolvimento. Dentro do DDD, dois conceitos fundamentais para a modelagem são os *Value Objects* e as *Entities*. Este tópico explorará esses conceitos e como implementá-los para construir o core do seu aplicativo de forma robusta e expressiva.

2. Conceitos Fundamentais:

Domain-Driven Design (DDD):

DDD é uma abordagem para o desenvolvimento de software que se concentra na modelagem do domínio do negócio. O objetivo é criar um modelo que reflita a realidade do negócio, utilizando uma linguagem ubíqua (Ubiquitous Language) que é compartilhada por desenvolvedores e especialistas de domínio. Isso garante que o código seja uma representação fiel do negócio e que todos na equipe falem a mesma língua [27].

Entities (Entidades):

Uma *Entity* é um objeto de domínio que possui uma identidade única e contínua ao longo do tempo, independentemente de seus atributos. Mesmo que os atributos de uma entidade mudem, ela ainda é a mesma entidade se sua identidade permanecer a mesma. Entidades são mutáveis e representam algo que pode ser rastreado ou referenciado. Exemplos incluem um `Usuário`, um `Produto` ou um `Pedido` [28].

- **Características Principais:**
  - **Identidade:** Possui um identificador único (ID) que o distingue de outras entidades, mesmo que seus outros atributos sejam idênticos.
  - **Mutabilidade:** Seus atributos podem mudar ao longo do tempo.
  - **Ciclo de Vida:** Geralmente têm um ciclo de vida longo e são persistidas em um banco de dados.

Value Objects (Objetos de Valor):

Um *Value Object* é um objeto de domínio que não possui uma identidade conceitual única. Ele é definido por seus atributos e é imutável. Dois objetos de valor são considerados iguais se todos os seus atributos forem iguais. Eles representam características ou descritores de algo. Exemplos incluem um `Endereço`, uma `Moeda` ou um `Intervalo de Datas` [29].

- **Características Principais:**
  - **Imutabilidade:** Uma vez criado, seus atributos não podem ser alterados. Se um atributo precisar ser modificado, um novo Value Object é criado.
  - **Igualdade por Valor:** Dois Value Objects são iguais se seus valores forem iguais, não se forem a mesma instância.
  - **Sem Identidade:** Não possuem um ID único.
  - **Comportamento:** Podem conter lógica de negócio relacionada aos seus valores.

Diferenças Chave entre Entities e Value Objects:

Característica	Entity	Value Object
Identidade	Possui identidade única (ID)	Não possui identidade única
Mutabilidade	Mutável	Imutável
Igualdade	Por referência (mesma instância/ID)	Por valor (todos os atributos iguais)
Ciclo de Vida	Longo, persistente	Curto, geralmente efêmero
Exemplos	Usuário, Produto, Pedido	Endereço, Moeda, Intervalo de Datas

3. Tecnologias e Ferramentas:

- **JavaScript/TypeScript:** Para implementar as classes e objetos.
- **Conceitos de Programação Orientada a Objetos (POO):** Classes, construtores, métodos.

4. Exemplos Práticos (Código):

Exemplo de Value Object: `Email.js`

```
// src/core/value-objects/Email.js

class Email {
  constructor(value) {
    if (!Email.isValid(value)) {
      throw new Error("Endereço de e-mail inválido.");
    }
    this._value = value.toLowerCase();
  }

  get value() {
    return this._value;
  }

  equals(other) {
    if (!(other instanceof Email)) {
      return false;
    }
    return this.value === other.value;
  }

  static isValid(email) {
    // Regex simples para validação de e-mail
    const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
    return emailRegex.test(email);
  }
}

export default Email;
```

Exemplo de Entity: `User.js`

```javascript
// src/core/entities/User.js

import Email from "../value-objects/Email";

class User {
  constructor(id, name, email) {
    if (!id || !name || !email) {
      throw new Error("ID, nome e e-mail são obrigatórios para o usuário.");
    }
    if (!(email instanceof Email)) {
      throw new Error("O e-mail deve ser uma instância de Email Value Object.");
    }

    this._id = id;
    this._name = name;
    this._email = email;
  }

  get id() {
    return this._id;
  }

  get name() {
    return this._name;
  }

  get email() {
    return this._email;
  }

  changeName(newName) {
    if (!newName || newName.trim() === "") {
      throw new Error("O nome não pode ser vazio.");
    }
    this._name = newName;
  }

  changeEmail(newEmail) {
    if (!(newEmail instanceof Email)) {
      throw new Error("O novo e-mail deve ser uma instância de Email Value Object.");
    }
    this._email = newEmail;
  }

  equals(other) {
    if (!(other instanceof User)) {
      return false;
    }
    return this.id === other.id;
  }
}

export default User;
```

5. Exercícios e Desafios:

1. Crie um Value Object `Money` que represente um valor monetário com `amount` (número) e `currency` (string, e.g., "BRL", "USD"). Implemente um método `add` que some dois objetos `Money` e retorne um novo `Money` (garantindo que as moedas sejam as mesmas).
2. Crie uma Entity `Product` com `id`, `name`, `description` e um Value Object `Money` para o `price`. Implemente um método `updatePrice` que receba um novo `Money` Value Object.
3. Discuta cenários onde a imutabilidade de Value Objects é benéfica para a robustez do sistema.

6. Recursos Adicionais:

- **Domain-Driven Design (DDD) - Wikipedia:** https://en.wikipedia.org/wiki/Domain-driven_design
- **Entities vs. Value Objects in Domain-Driven Design:** https://martinfowler.com/bliki/ValueObject.html

Referências:

[27] https://en.wikipedia.org/wiki/Domain-driven_design [28] https://martinfowler.com/bliki/ValueObject.html [29] https://martinfowler.com/bliki/ValueObject.html

26/set: Implementação do core do projeto: use-cases.

1. Visão Geral:

No contexto de uma arquitetura de software bem definida, como a Clean Architecture ou a Onion Architecture, os *Use Cases* (Casos de Uso) desempenham um papel central. Eles representam a lógica de negócios específica da aplicação, orquestrando as interações entre as entidades de domínio e as interfaces externas (como bancos de dados ou APIs). A implementação de casos de uso de forma isolada e testável é fundamental para garantir que a lógica de negócios seja robusta, fácil de manter e independente de detalhes de infraestrutura. Este tópico explorará o conceito de casos de uso e como implementá-los no core do seu aplicativo.

2. Conceitos Fundamentais:

Use Cases (Casos de Uso):

Um Caso de Uso descreve uma sequência de ações que um sistema executa para produzir um resultado observável de valor para um ator (usuário ou outro sistema). No contexto de arquitetura de software, um Caso de Uso é uma classe ou um conjunto de classes que encapsula uma funcionalidade específica do negócio. Ele atua como um orquestrador, coordenando as entidades de domínio para realizar uma tarefa [30].

- **Responsabilidade:** A principal responsabilidade de um Caso de Uso é implementar as regras de negócio específicas da aplicação. Ele não se preocupa com detalhes de UI, persistência de dados ou comunicação com serviços externos, mas sim com a lógica que governa como os dados são processados e as entidades interagem.
- **Independência:** Casos de Uso devem ser independentes de frameworks, bancos de dados e interfaces de usuário. Isso significa que eles podem ser testados isoladamente e reutilizados em diferentes contextos (e.g., UI, API REST, linha de comando).
- **Entrada e Saída:** Um Caso de Uso geralmente recebe uma entrada (parâmetros) e produz uma saída (resultado ou erro). Essa interface clara facilita a integração com outras partes do sistema.

Clean Architecture (Arquitetura Limpa):

A Clean Architecture, popularizada por Robert C. Martin (Uncle Bob), propõe uma estrutura em camadas onde as dependências fluem de fora para dentro. O core do aplicativo (entidades e casos de uso) está no centro e não deve ter conhecimento das camadas externas (UI, banco de dados, frameworks). Isso garante que a lógica de negócios seja a parte mais estável e independente do sistema [31].

- **Camadas:**
  - **Entidades (Entities):** Regras de negócio corporativas, independentes de qualquer aplicação específica.
  - **Casos de Uso (Use Cases):** Regras de negócio da aplicação, orquestram o fluxo de dados para e das entidades.
  - **Adaptadores de Interface (Interface Adapters):** Convertem dados do formato mais conveniente para os casos de uso e entidades para o formato mais conveniente para as interfaces externas (e.g., Presenters, Gateways, Controllers).
  - **Frameworks e Drivers (Frameworks & Drivers):** Detalhes de implementação, como banco de dados, frameworks web, UI.

Separação de Preocupações:

A implementação de Casos de Uso reforça o princípio da separação de preocupações, onde cada parte do sistema tem uma única responsabilidade bem definida. Isso torna o código mais modular, fácil de entender, testar e manter. Um Caso de Uso se preocupa *com o que* fazer, enquanto as camadas externas se preocupam *com como* fazer (e.g., como exibir na tela, como persistir no banco de dados).

3. Tecnologias e Ferramentas:

- **JavaScript/TypeScript:** Para implementar as classes de Caso de Uso.
- **Injeção de Dependência:** Para fornecer aos Casos de Uso as dependências necessárias (e.g., repositórios para acesso a dados).

4. Exemplos Práticos (Código):

Vamos considerar um Caso de Uso para adicionar um novo usuário ao sistema. Para isso, precisaremos de um `UserRepository` (um adaptador de interface) que lide com a persistência do usuário.

`src/core/use-cases/AddUser.js`

```
// src/core/use-cases/AddUser.js

import User from "../entities/User";
import Email from "../value-objects/Email";

class AddUser {
  constructor(userRepository) {
    this.userRepository = userRepository;
  }

  async execute(userData) {
    const { id, name, email } = userData;

    // Validação de entrada (pode ser mais robusta com bibliotecas de validação)
    if (!id || !name || !email) {
      throw new Error("Dados do usuário incompletos.");
    }

    // Criação de Value Object para o e-mail
    let userEmail;
    try {
      userEmail = new Email(email);
    } catch (error) {
      throw new Error(`Erro de validação de e-mail: ${error.message}`);
    }

    // Verificar se o usuário já existe (regra de negócio)
    const existingUser = await this.userRepository.findByEmail(userEmail.value);
    if (existingUser) {
      throw new Error("Usuário com este e-mail já existe.");
    }

    // Criação da Entidade Usuário
    const newUser = new User(id, name, userEmail);

    // Persistir o novo usuário
    await this.userRepository.save(newUser);

    return newUser;
  }
}

export default AddUser;
```

`src/core/ports/UserRepository.js` (Interface/Porta para o repositório)

```
// src/core/ports/UserRepository.js

// Esta é uma interface (ou porta) que define o contrato para qualquer implementação de repositório de usuário.
// Em JavaScript, interfaces são geralmente definidas implicitamente ou com classes abstratas/interfaces de TypeScript.

class UserRepository {
  async save(user) {
    throw new Error("Método 'save' deve ser implementado.");
  }

  async findById(id) {
    throw new Error("Método 'findById' deve ser implementado.");
  }

  async findByEmail(email) {
    throw new Error("Método 'findByEmail' deve ser implementado.");
  }
}

export default UserRepository;
```

`src/infrastructure/InMemoryUserRepository.js` (Implementação do repositório - Adaptador)

```
// src/infrastructure/InMemoryUserRepository.js

import UserRepository from "../../core/ports/UserRepository";

class InMemoryUserRepository extends UserRepository {
  constructor() {
    super();
    this.users = []; // Simula um banco de dados em memória
  }

  async save(user) {
    const existingIndex = this.users.findIndex(u => u.id === user.id);
    if (existingIndex > -1) {
      this.users[existingIndex] = user; // Atualiza
    } else {
      this.users.push(user); // Adiciona
    }
    console.log(`Usuário ${user.name} salvo/atualizado.`);
    return user;
  }

  async findById(id) {
    return this.users.find(user => user.id === id);
  }

  async findByEmail(email) {
    return this.users.find(user => user.email.value === email);
  }
}

export default InMemoryUserRepository;
```

Como usar o Caso de Uso:

```
 import AddUser from "./src/core/use-cases/AddUser";
import InMemoryUserRepository from "./src/infrastructure/InMemoryUserRepository";

async function main() {
  const userRepository = new InMemoryUserRepository();
  const addUserUseCase = new AddUser(userRepository);

  try {
    const newUser1 = await addUserUseCase.execute({
      id: "1",
      name: "Alice",
      email: "alice@example.com",
    });
    console.log("Novo usuário criado:", newUser1.name);

    const newUser2 = await addUserUseCase.execute({
      id: "2",
      name: "Bob",
      email: "bob@example.com",
    });
    console.log("Novo usuário criado:", newUser2.name);

    // Tentando adicionar usuário com e-mail duplicado
    await addUserUseCase.execute({
      id: "3",
      name: "Charlie",
      email: "alice@example.com",
    });
  } catch (error) {
    console.error("Erro ao adicionar usuário:", error.message);
  }
}

main();
```

5. Exercícios e Desafios:

1. Crie um Caso de Uso `GetUserById` que receba um ID de usuário e retorne a entidade `User` correspondente, utilizando o `UserRepository`.
2. Implemente um Caso de Uso `UpdateUserName` que receba um ID de usuário e um novo nome, atualize a entidade `User` e a persista no repositório.
3. Discuta como a injeção de dependência do `userRepository` no `AddUser` Caso de Uso contribui para a testabilidade e flexibilidade do sistema.

6. Recursos Adicionais:

- **The Clean Architecture by Uncle Bob:** https://blog.cleancoder.com/uncle-bob/2012/08/13/The-Clean-Architecture.html
- **Use Cases in Clean Architecture:** https://medium.com/swlh/use-cases-in-clean-architecture-27690628795

Referências:

[30] https://medium.com/swlh/use-cases-in-clean-architecture-27690628795 [31] https://blog.cleancoder.com/uncle-bob/2012/08/13/The-Clean-Architecture.html

29/set: Integrando o core do projeto com as telas.

1. Visão Geral:

Até agora, exploramos a construção de interfaces, navegação e a criação de um core de aplicativo robusto e independente de UI, utilizando conceitos como Value Objects, Entities e Use Cases. O próximo passo crucial é conectar essa lógica de negócios (o core) com a interface do usuário (as telas). Esta integração é onde a aplicação ganha vida, permitindo que as ações do usuário na UI acionem a lógica de negócios e que os resultados dessa

lógica sejam exibidos de volta na tela. Este tópico abordará como realizar essa integração de forma eficaz, utilizando padrões de projeto e gerenciamento de estado.

2. Conceitos Fundamentais:

Injeção de Dependência (Dependency Injection - DI):

DI é um padrão de design que permite que as dependências de um objeto sejam fornecidas a ele externamente, em vez de o objeto criá-las internamente. Isso torna o código mais modular, testável e flexível. No contexto da integração do core com as telas, significa que os componentes da UI não precisam saber como instanciar os casos de uso ou repositórios; eles simplesmente os recebem como dependências [32].

Padrões de Projeto para Integração UI-Lógica:

Diversos padrões de projeto podem ser utilizados para gerenciar a comunicação entre a camada de apresentação (UI) e a camada de lógica de negócios. Alguns dos mais comuns incluem:

- **MVVM (Model-View-ViewModel):**
    - **Model:** Representa os dados e a lógica de negócios (nossas Entities, Value Objects e Use Cases).
    - **View:** A interface do usuário (os componentes React Native).
    - **ViewModel:** Atua como um intermediário entre a View e o Model. Ele expõe dados do Model de forma que a View possa consumi-los facilmente e contém a lógica de apresentação e comandos que a View pode acionar. No React Native, um ViewModel pode ser implementado usando React Hooks personalizados ou Context API [33].
    - **Benefícios:** Separação clara de preocupações, testabilidade da lógica de apresentação, reutilização de ViewModels.
- **MVP (Model-View-Presenter):**
    - **Model:** Dados e lógica de negócios.
    - **View:** Interface do usuário (passiva, apenas exibe dados e encaminha eventos).
    - **Presenter:** Atua como um intermediário entre a View e o Model. Ele contém a lógica de apresentação e manipula os eventos da View, atualizando o Model e a View conforme necessário. O Presenter não tem conhecimento da View específica, apenas de uma interface que a View implementa.
    - **Benefícios:** Forte separação, testabilidade do Presenter.

Gerenciamento de Estado:

Para que as telas exibam os dados corretos e reajam às mudanças na lógica de negócios, é essencial um gerenciamento de estado eficaz. Além do estado local de componentes (`useState`), o estado global da aplicação pode ser gerenciado por:

- **Context API:** Uma forma nativa do React de compartilhar dados entre componentes sem a necessidade de passar props manualmente em cada nível da árvore de componentes. É ideal para dados que são considerados "globais" para uma subárvore de componentes [34].
- **Bibliotecas de Gerenciamento de Estado:** Como Redux, Zustand, Recoil, que oferecem soluções mais robustas para gerenciamento de estado complexo e escalável, especialmente em aplicações grandes.

Fluxo de Dados:

O fluxo de dados típico na integração do core com as telas segue um padrão unidirecional:

1. **Ação do Usuário na UI:** O usuário interage com a interface (e.g., clica em um botão, digita em um campo).
2. **UI Aciona Lógica de Negócios:** A View (ou ViewModel/Presenter) chama um método do Caso de Uso, passando os dados necessários.
3. **Caso de Uso Executa Lógica:** O Caso de Uso orquestra as entidades e interage com os repositórios para realizar a operação de negócio.
4. **Atualização do Estado:** O resultado da operação do Caso de Uso (dados atualizados, sucesso/erro) é usado para atualizar o estado da aplicação.
5. **UI Reage à Mudança de Estado:** A View re-renderiza-se automaticamente para refletir o novo estado, exibindo os resultados ao usuário.

3. Tecnologias e Ferramentas:

- **React Hooks (`useState`, `useEffect`, `useContext`):** Para gerenciar o estado local e global, e para implementar ViewModels simples.
- **Context API:** Para compartilhar dados e funções (como instâncias de casos de uso) entre componentes.
- **Instâncias de Casos de Uso:** As classes de casos de uso criadas no tópico anterior.
- **Repositórios:** Implementações dos repositórios para acesso a dados.

4. Exemplos Práticos (Código):

Vamos integrar o `AddUser` Use Case a uma tela de formulário de registro.

`src/screens/RegisterScreen.js`

```
 import React, { useState, useContext } from 'react';
import { View, Text, TextInput, Button, StyleSheet, Alert } from 'react-native';
import { AppContext } from '../context/AppContext'; // Vamos criar este contexto

export default function RegisterScreen({ navigation }) {
  const [id, setId] = useState('');
  const [name, setName] = useState('');
  const [email, setEmail] = useState('');
  const { addUserUseCase } = useContext(AppContext); // Obtém o caso de uso do contexto

  const handleRegister = async () => {
    try {
      const newUser = await addUserUseCase.execute({
        id,
        name,
        email,
      });
      Alert.alert('Sucesso', `Usuário ${newUser.name} registrado com sucesso!`);
      navigation.goBack(); // Volta para a tela anterior após o registro
    } catch (error) {
      Alert.alert('Erro', error.message);
    }
  };

  return (
    <View style={styles.container}>
      <Text style={styles.title}>Registrar Novo Usuário</Text>
      <TextInput
        style={styles.input}
        placeholder="ID do Usuário"
        value={id}
        onChangeText={setId}
        keyboardType="numeric"
      />
      <TextInput
        style={styles.input}
        placeholder="Nome"
        value={name}
        onChangeText={setName}
      />
      <TextInput
        style={styles.input}
        placeholder="E-mail"
        value={email}
        onChangeText={setEmail}
        keyboardType="email-address"
        autoCapitalize="none"
      />
      <Button title="Registrar" onPress={handleRegister} />
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    padding: 20,
    backgroundColor: '#f5f5f5',
  },
  title: {
    fontSize: 24,
    fontWeight: 'bold',
    marginBottom: 30,
    textAlign: 'center',
  },
  input: {
    height: 50,
    borderColor: '#ccc',
    borderWidth: 1,
    borderRadius: 8,
    paddingHorizontal: 15,
    marginBottom: 15,
    backgroundColor: '#fff',
  },
});
```

`src/context/AppContext.js` (Contexto para prover os casos de uso)

```
import React, { createContext } from 'react';
import AddUser from '../core/use-cases/AddUser';
import InMemoryUserRepository from '../infrastructure/InMemoryUserRepository';

// Instancie seus repositórios e casos de uso aqui
const userRepository = new InMemoryUserRepository();
const addUserUseCase = new AddUser(userRepository);

export const AppContext = createContext({
  addUserUseCase,
  // Adicione outros casos de uso aqui
});

export const AppProvider = ({ children }) => {
  return (
    <AppContext.Provider value={{ addUserUseCase }}>
      {children}
    </AppContext.Provider>
  );
};
```

`App.js` (Para envolver a aplicação com o provedor de contexto)

```
import React from 'react';
import { NavigationContainer } from '@react-navigation/native';
import { createStackNavigator } from '@react-navigation/stack';
import { AppProvider } from './src/context/AppContext';
import RegisterScreen from './src/screens/RegisterScreen';

const Stack = createStackNavigator();

export default function App() {
  return (
    <AppProvider>
      <NavigationContainer>
        <Stack.Navigator>
          <Stack.Screen name="Register" component={RegisterScreen} options={{ title: 'Registro' }} />
          {/* Adicione outras telas aqui */}
        </Stack.Navigator>
      </NavigationContainer>
    </AppProvider>
  );
}
```

5. Exercícios e Desafios:

1. Integre o `GetUserById` Use Case (criado no desafio anterior) a uma nova tela `UserProfileScreen`. Crie um `TextInput` para o usuário digitar um ID e um botão para buscar o perfil. Exiba os dados do usuário na tela.
2. Modifique o `AppContext` para incluir o `UpdateUserName` Use Case. Crie uma tela `EditProfileScreen` que permita ao usuário atualizar o nome de um usuário existente.
3. Pesquise sobre o padrão Repository e explique como ele facilita a integração do core do aplicativo com diferentes fontes de dados (e.g., API REST, banco de dados local).

6. Recursos Adicionais:

- **React Context API:** https://react.dev/learn/passing-props-to-a-component#passing-props-to-a-component
- **MVVM Pattern:** https://martinfowler.com/eaaDev/ModelViewViewModel.html
- **Dependency Injection:** https://martinfowler.com/articles/injection.html

Referências:

[32] https://martinfowler.com/articles/injection.html [33] https://martinfowler.com/eaaDev/ModelViewViewModel.html [34] https://react.dev/learn/passing-props-to-a-component#passing-props-to-a-component

03/out: Configuração do ambiente de testes com react native testing library.

1. Visão Geral:

Enquanto o Jest é excelente para testes unitários de lógica de negócios, a React Native Testing Library (RNTL) é a ferramenta preferida para testar componentes de interface do usuário (UI) em aplicativos React Native. Ela foca em testar o comportamento do componente da perspectiva do usuário, em vez de seus detalhes de implementação interna. Isso promove testes mais robustos e menos propensos a quebrar com refatorações. Este tópico abordará a configuração e o uso da React Native Testing Library para garantir que seus componentes de UI funcionem conforme o esperado.

2. Conceitos Fundamentais:

React Native Testing Library (RNTL):

A React Native Testing Library é uma biblioteca que fornece utilitários para testar componentes React Native de uma forma que simula como os usuários interagem com eles. A filosofia principal da Testing Library é "quanto mais seus testes se assemelham à forma como seu software é usado, mais confiança eles podem lhe dar" [35].

- **Foco no Comportamento do Usuário:** Em vez de testar a estrutura interna dos componentes (como o estado interno ou a árvore de componentes), a RNTL encoraja testar o que o usuário vê e como ele interage com o componente.
- **Consultas Amigáveis ao Usuário:** A biblioteca oferece métodos de consulta (e.g., `getByText`, `getByTestId`, `getByPlaceholderText`) que permitem encontrar elementos na tela da mesma forma que um usuário os encontraria.
- **Simulação de Eventos:** Permite simular eventos de usuário, como cliques, digitação e rolagem, para testar a interatividade dos componentes.
- **Acessibilidade:** Ao focar em como os usuários interagem, a RNTL indiretamente promove a escrita de componentes mais acessíveis, pois as consultas são baseadas em atributos de acessibilidade ou texto visível.

Diferença entre Jest e React Native Testing Library:

- **Jest:** É um *framework* de testes JavaScript de propósito geral. Ele fornece o ambiente para executar testes, asserções (`expect`), mocks e ferramentas para relatórios de cobertura de código. É usado para testes unitários de funções e classes, e também como o *runner* para a RNTL.
- **React Native Testing Library:** É uma *biblioteca* de utilitários que funciona *com* o Jest (ou outro runner de testes) para facilitar a escrita de testes de UI que se concentram na experiência do usuário. Ela não substitui o Jest, mas o complementa para testes de componentes.

3. Tecnologias e Ferramentas:

- **Jest:** Como o runner de testes.
- **`@testing-library/react-native`:** A biblioteca principal da React Native Testing Library.
- **`react-test-renderer`:** Uma dependência da RNTL que permite renderizar componentes React em um ambiente Node.js para testes.

4. Exemplos Práticos (Código):

Configurando React Native Testing Library:

1. **Instale as dependências:** `bash npm install --save-dev @testing-library/react-native react-test-renderer` ou `bash yarn add --dev @testing-library/react-native react-test-renderer`

2. **Certifique-se de que seu `jest.config.js` está configurado para React Native (geralmente já está se você usa Expo ou `react-native init`):** `javascript // jest.config.js module.exports = { preset: 'react-native', setupFilesAfterEnv: ['@testing-library/jest-native/extend-expect'], // Adiciona matchers personalizados // ... outras configurações };` Você também pode precisar instalar `@testing-library/jest-native` para ter matchers como `toBeOnTheScreen`: `bash npm install --save-dev @testing-library/jest-native`

Exemplo de Teste de Componente com RNTL:

Vamos testar um componente simples de botão que exibe um texto e chama uma função `onPress`.

`src/components/CustomButton.js`:

```
import React from 'react';
import { TouchableOpacity, Text, StyleSheet } from 'react-native';

export default function CustomButton({ title, onPress }) {
  return (
    <TouchableOpacity style={styles.button} onPress={onPress}>
      <Text style={styles.buttonText}>{title}</Text>
    </TouchableOpacity>
  );
}

const styles = StyleSheet.create({
  button: {
    backgroundColor: '#007bff',
    padding: 10,
    borderRadius: 5,
    alignItems: 'center',
    justifyContent: 'center',
  },
  buttonText: {
    color: '#fff',
    fontSize: 16,
    fontWeight: 'bold',
  },
});
```

`__tests__/CustomButton.test.js`:

```
import React from 'react';
import { render, fireEvent } from '@testing-library/react-native';
import CustomButton from '../src/components/CustomButton';

describe('CustomButton', () => {
  test('renderiza o título corretamente', () => {
    const { getByText } = render(<CustomButton title="Clique Aqui" onPress={() => {}} />);
    expect(getByText('Clique Aqui')).toBeTruthy();
  });

  test('chama a função onPress quando clicado', () => {
    const mockOnPress = jest.fn(); // Cria uma função mock
    const { getByText } = render(<CustomButton title="Testar Clique" onPress={mockOnPress} />);

    fireEvent.press(getByText('Testar Clique')); // Simula o clique

    expect(mockOnPress).toHaveBeenCalledTimes(1); // Verifica se a função foi chamada
  });

  test('renderiza o botão desabilitado quando a prop disabled é true', () => {
    const { getByText } = render(<CustomButton title="Desabilitado" onPress={() => {}} disabled />);
    const button = getByText('Desabilitado');
    expect(button.parent.props.accessibilityState.disabled).toBe(true);
  });
});
```

Para executar os testes, basta rodar no terminal:

```
npm test
```

5. Exercícios e Desafios:

1. Crie um componente `TextInput` simples que receba um `label` e um `placeholder`. Escreva um teste com RNTL para verificar se o `label` e o `placeholder` são renderizados corretamente e se o `onChangeText` é chamado quando o usuário digita.
2. Modifique o `CustomButton` para que ele mude a cor de fundo quando a prop `isLoading` for `true`. Escreva um teste que verifique essa mudança de estilo.
3. Pesquise sobre `queryBy` e `findBy` na React Native Testing Library. Explique a diferença entre eles e quando usar cada um.

6. Recursos Adicionais:

- **Documentação Oficial da React Native Testing Library:** https://testing-library.com/docs/react-native-testing-library/intro/
- **Guia de Testes do React Native (oficial):** https://reactnative.dev/docs/testing
- **Jest Native Matchers:** https://github.com/testing-library/jest-native

Referências:

[35] https://testing-library.com/docs/react-native-testing-library/intro/

Módulo 4: Integração com WebServices e Armazenamento

06/out: Acesso à WebServices com axios e consumindo uma API para autenticação.

1. Visão Geral:

A maioria dos aplicativos móveis modernos precisa interagir com serviços externos para buscar ou enviar dados, como informações de perfil de usuário, listas de produtos, ou para realizar operações como autenticação e pagamentos. Web Services e APIs (Application Programming Interfaces) são a espinha dorsal dessa comunicação. Este tópico abordará como consumir APIs RESTful em um aplicativo React Native usando a biblioteca Axios, com foco especial na autenticação de usuários.

2. Conceitos Fundamentais:

Web Services e APIs RESTful:

- **Web Service:** Um sistema de software projetado para suportar a interação máquina a máquina em uma rede. Eles permitem que diferentes aplicações troquem dados e funcionalidades [36].
- **API (Application Programming Interface):** Um conjunto de definições e protocolos que permite que diferentes softwares se comuniquem entre si. Uma API define como os componentes de software devem interagir [37].
- **RESTful API:** Um tipo de API que segue os princípios arquiteturais do REST (Representational State Transfer). APIs RESTful são stateless, usam operações HTTP padrão (GET, POST, PUT, DELETE) para manipular recursos e geralmente retornam dados em formatos como JSON ou XML [38].

Métodos HTTP:

- **GET:** Usado para solicitar dados de um recurso especificado. Não deve ter efeitos colaterais no servidor (idempotente e seguro) [39].
- **POST:** Usado para enviar dados a um recurso especificado, geralmente para criar um novo recurso [39].
- **PUT:** Usado para atualizar um recurso existente ou criar um novo se ele não existir. É idempotente (múltiplas requisições PUT idênticas terão o mesmo efeito que uma única requisição) [39].
- **DELETE:** Usado para remover um recurso especificado [39].

Axios:

Axios é um cliente HTTP baseado em Promises para o navegador e Node.js. Ele é amplamente utilizado em aplicações React Native devido à sua simplicidade, recursos robustos e facilidade de uso. Axios facilita a realização de requisições HTTP (GET, POST, PUT, DELETE, etc.), o tratamento de respostas e erros, e a configuração de interceptors para requisições e respostas [40].

- **Características Principais do Axios:**
  - Fazer requisições HTTP assíncronas.
  - Suporte a Promises.
  - Interceptar requisições e respostas.
  - Transformar dados de requisição e resposta.
  - Cancelamento de requisições.
  - Proteção contra XSRF.

Autenticação:

A autenticação é o processo de verificar a identidade de um usuário ou sistema. Em APIs, a autenticação garante que apenas usuários autorizados possam acessar recursos protegidos. Métodos comuns de autenticação incluem:

- **Token-based Authentication (Autenticação Baseada em Token):** Após o login, o servidor emite um token (e.g., JWT - JSON Web Token) para o cliente. O cliente armazena esse token e o envia em cada requisição subsequente para acessar recursos protegidos. O servidor valida o token para autorizar a requisição [41].
- **OAuth 2.0:** Um framework de autorização que permite que aplicativos de terceiros obtenham acesso limitado a contas de usuário em um serviço HTTP, como Facebook, Google, GitHub, etc. [42].

3. Tecnologias e Ferramentas:

- **Axios:** Biblioteca para requisições HTTP.
- **API RESTful:** O serviço backend com o qual o aplicativo irá interagir.

4. Exemplos Práticos (Código):

Instalação do Axios:

```
npm install axios
```

Exemplo de Consumo de API e Autenticação (Login Simples):

Vamos simular uma API de autenticação. Em um cenário real, você teria um backend rodando que forneceria esses endpoints.

`src/services/authService.js`:

```
import axios from 'axios';

const API_URL = 'https://api.example.com'; // Substitua pela URL da sua API

const authService = {
  login: async (username, password) => {
    try {
      const response = await axios.post(`${API_URL}/auth/login`, {
        username,
        password,
      });
      // Supondo que a API retorna um token no campo 'token'
      const { token } = response.data;
      // Armazenar o token de forma segura (veremos isso no próximo tópico com expo-secure-store)
      console.log('Token recebido:', token);
      return token;
    } catch (error) {
      console.error('Erro no login:', error.response ? error.response.data : error.message);
      throw new Error(error.response?.data?.message || 'Erro ao fazer login.');
    }
  },

  getProtectedData: async (token) => {
    try {
      const response = await axios.get(`${API_URL}/data/protected`, {
        headers: {
          Authorization: `Bearer ${token}`,
        },
      });
      return response.data;
    } catch (error) {
      console.error('Erro ao buscar dados protegidos:', error.response ? error.response.data :
error.message);
      throw new Error(error.response?.data?.message || 'Erro ao buscar dados protegidos.');
    }
  },
};

export default authService;
```

`src/screens/LoginScreen.js`:

```
 import React, { useState } from 'react';
import { View, Text, TextInput, Button, StyleSheet, Alert } from 'react-native';
import authService from '../services/authService';

export default function LoginScreen({ navigation }) {
  const [username, setUsername] = useState('');
  const [password, setPassword] = useState('');
  const [isLoading, setIsLoading] = useState(false);

  const handleLogin = async () => {
    setIsLoading(true);
    try {
      const token = await authService.login(username, password);
      Alert.alert('Sucesso', 'Login realizado com sucesso!');
      // Navegar para a tela principal ou armazenar o token
      // navigation.navigate('Home');
    } catch (error) {
      Alert.alert('Erro', error.message);
    } finally {
      setIsLoading(false);
    }
  };

  const handleFetchProtectedData = async () => {
    // Em um cenário real, o token viria de um armazenamento seguro
    const dummyToken =
'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MD
    try {
      const data = await authService.getProtectedData(dummyToken);
      Alert.alert('Dados Protegidos', JSON.stringify(data));
    } catch (error) {
      Alert.alert('Erro', error.message);
    }
  };

  return (
    <View style={styles.container}>
      <Text style={styles.title}>Login</Text>
      <TextInput
        style={styles.input}
        placeholder="Nome de Usuário"
        value={username}
        onChangeText={setUsername}
        autoCapitalize="none"
      />
      <TextInput
        style={styles.input}
        placeholder="Senha"
        value={password}
        onChangeText={setPassword}
        secureTextEntry
      />
      <Button title={isLoading ? 'Entrando...' : 'Entrar'} onPress={handleLogin} disabled={isLoading} />
      <View style={{ marginTop: 20 }}>
        <Button title="Buscar Dados Protegidos (Exemplo)" onPress={handleFetchProtectedData} />
      </View>
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    padding: 20,
    backgroundColor: '#f5f5f5',
  },
  title: {
    fontSize: 24,
    fontWeight: 'bold',
    marginBottom: 30,
    textAlign: 'center',
  },
  input: {
    height: 50,
    borderColor: '#ccc',
    borderWidth: 1,
    borderRadius: 8,
    paddingHorizontal: 15,
    marginBottom: 15,
    backgroundColor: '#fff',
```

```
  },
});
```

5. Exercícios e Desafios:

1. Modifique o `authService.js` para incluir uma função de registro (`register`) que envie dados de um novo usuário para um endpoint `/auth/register` .
2. Crie uma tela de registro (`RegisterScreen`) que utilize essa nova função para permitir que novos usuários se cadastrem.
3. Implemente um interceptor no Axios para adicionar automaticamente o token de autenticação (se disponível) ao cabeçalho de todas as requisições, evitando a necessidade de passá-lo manualmente em cada chamada.
4. Pesquise sobre o tratamento de erros em requisições HTTP com Axios e implemente uma lógica mais robusta para lidar com diferentes códigos de status HTTP (e.g., 401 Unauthorized, 404 Not Found).

6. Recursos Adicionais:

- **Documentação Oficial do Axios:** https://axios-http.com/docs/intro
- **MDN Web Docs - HTTP methods:** https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods
- **JSON Web Tokens (JWT):** https://jwt.io/
- **OAuth 2.0 Simplified:** https://oauth.net/2/

Referências:

[36] https://www.ibm.com/docs/en/cics/5.3/com.ibm.cics.ts.applicationprogramming.doc/topics/dfhp4_websvcs.html [37] https://aws.amazon.com/what-is/api/ [38] https://restfulapi.net/ [39] https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods [40] https://axios-http.com/docs/intro [41] https://jwt.io/ [42] https://oauth.net/2/

10/out: Utilização da context api e armazenamento no expo-secure-store.

1. Visão Geral:

Gerenciar o estado em aplicações React Native pode se tornar complexo à medida que o aplicativo cresce e mais componentes precisam acessar ou compartilhar os mesmos dados. A Context API do React oferece uma solução nativa para gerenciar o estado global de forma eficiente, evitando o "prop drilling". Além disso, para dados sensíveis como tokens de autenticação ou informações de usuário, é crucial utilizar um armazenamento seguro. O `expo-secure-store` fornece uma maneira segura de armazenar esses dados no dispositivo. Este tópico abordará como utilizar a Context API para gerenciamento de estado e o `expo-secure-store` para armazenamento seguro.

2. Conceitos Fundamentais:

Context API do React:

A Context API é uma funcionalidade do React que permite compartilhar dados entre componentes sem a necessidade de passar props manualmente em cada nível da árvore de componentes. É ideal para dados que são considerados "globais" para uma subárvore de componentes, como informações de usuário autenticado, tema da aplicação ou configurações de idioma [43].

- **`createContext` :** Cria um objeto Context. Quando o React renderiza um componente que assina este objeto Context, ele lerá o valor atual do Context do `Provider` mais próximo na árvore de componentes.

- **`Provider`:** Um componente `Provider` do Context é usado para envolver a parte da árvore de componentes que precisa acessar o valor do Context. Ele aceita uma prop `value` que será o valor atual do Context para todos os componentes filhos que o consomem.
- **`useContext` Hook:** Um hook do React que permite que componentes funcionais consumam o valor de um Context. Ele recebe o objeto Context como argumento e retorna o valor atual do Context.

`expo-secure-store`:

`expo-secure-store` é um módulo do Expo que fornece uma maneira segura de armazenar dados sensíveis no dispositivo do usuário. Ele utiliza as APIs de armazenamento seguro nativas de cada plataforma (Keychain no iOS e Keystore no Android), garantindo que os dados sejam criptografados e protegidos contra acesso não autorizado [44].

- **Uso:** Ideal para armazenar tokens de autenticação, chaves de API, senhas (embora senhas nunca devam ser armazenadas em texto puro) e outras informações confidenciais que precisam persistir entre as sessões do aplicativo.
- **Segurança:** Ao contrário do `AsyncStorage` (que armazena dados em texto simples), o `expo-secure-store` garante que os dados sejam criptografados e acessíveis apenas pelo aplicativo que os armazenou.

3. Tecnologias e Ferramentas:

- **React Context API:** Para gerenciamento de estado global.
- **`expo-secure-store`:** Para armazenamento seguro de dados sensíveis.

4. Exemplos Práticos (Código):

Instalação do `expo-secure-store`:

```
npx expo install expo-secure-store
```

Exemplo de Autenticação com Context API e `expo-secure-store`:

Vamos criar um contexto de autenticação que gerencie o estado de login do usuário e armazene o token de forma segura.

`src/context/AuthContext.js`:

```jsx
 import React, { createContext, useState, useEffect, useContext } from 'react';
import * as SecureStore from 'expo-secure-store';
import authService from '../services/authService'; // Nosso serviço de autenticação do tópico anterior

const AuthContext = createContext(null);

export const AuthProvider = ({ children }) => {
  const [userToken, setUserToken] = useState(null);
  const [isLoading, setIsLoading] = useState(true);

  useEffect(() => {
    const loadToken = async () => {
      try {
        const token = await SecureStore.getItemAsync('userToken');
        if (token) {
          setUserToken(token);
        }
      } catch (error) {
        console.error('Erro ao carregar token do SecureStore:', error);
      } finally {
        setIsLoading(false);
      }
    };
    loadToken();
  }, []);

  const signIn = async (username, password) => {
    setIsLoading(true);
    try {
      const token = await authService.login(username, password);
      await SecureStore.setItemAsync('userToken', token);
      setUserToken(token);
    } catch (error) {
      console.error('Erro no signIn:', error);
      throw error; // Re-lança o erro para ser tratado na UI
    } finally {
      setIsLoading(false);
    }
  };

  const signOut = async () => {
    setIsLoading(true);
    try {
      await SecureStore.deleteItemAsync('userToken');
      setUserToken(null);
    } catch (error) {
      console.error('Erro no signOut:', error);
    } finally {
      setIsLoading(false);
    }
  };

  return (
    <AuthContext.Provider value={{ userToken, isLoading, signIn, signOut }}>
      {children}
    </AuthContext.Provider>
  );
};

export const useAuth = () => {
  return useContext(AuthContext);
};
```

`App.js` (Para envolver a aplicação com o provedor de contexto de autenticação):

```
 import React from 'react';
import { NavigationContainer } from '@react-navigation/native';
import { createStackNavigator } from '@react-navigation/stack';
import { AuthProvider, useAuth } from './src/context/AuthContext';
import LoginScreen from './src/screens/LoginScreen';
import HomeScreen from './src/screens/HomeScreen'; // Uma tela que exige autenticação
import { ActivityIndicator, View, StyleSheet } from 'react-native';

const Stack = createStackNavigator();

function AuthStack() {
  return (
    <Stack.Navigator>
      <Stack.Screen name="Login" component={LoginScreen} options={{ headerShown: false }} />
    </Stack.Navigator>
  );
}

function AppStack() {
  return (
    <Stack.Navigator>
      <Stack.Screen name="Home" component={HomeScreen} />
      {/* Outras telas autenticadas */}
    </Stack.Navigator>
  );
}

function RootNavigator() {
  const { userToken, isLoading } = useAuth();

  if (isLoading) {
    return (
      <View style={styles.loadingContainer}>
        <ActivityIndicator size="large" color="#0000ff" />
      </View>
    );
  }

  return (
    <NavigationContainer>
      {userToken ? <AppStack /> : <AuthStack />}
    </NavigationContainer>
  );
}

export default function App() {
  return (
    <AuthProvider>
      <RootNavigator />
    </AuthProvider>
  );
}

const styles = StyleSheet.create({
  loadingContainer: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
  },
});
```

`src/screens/HomeScreen.js` **(Exemplo de tela autenticada):**

```
 import React from 'react';
import { View, Text, Button, StyleSheet } from 'react-native';
import { useAuth } from '../context/AuthContext';

export default function HomeScreen() {
  const { signOut } = useAuth();

  return (
    <View style={styles.container}>
      <Text style={styles.title}>Bem-vindo!</Text>
      <Text>Você está autenticado.</Text>
      <Button title="Sair" onPress={signOut} />
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
  },
  title: {
    fontSize: 24,
    marginBottom: 20,
  },
});
```

5. Exercícios e Desafios:

1. Modifique a `LoginScreen` para utilizar o `signIn` do `AuthContext` para autenticar o usuário. Após o login bem-sucedido, o usuário deve ser redirecionado para a `HomeScreen`.
2. Adicione uma funcionalidade de registro ao `AuthContext` e crie uma `RegisterScreen` que utilize essa funcionalidade para cadastrar novos usuários e fazer login automaticamente após o registro.
3. Implemente uma lógica no `AuthContext` para verificar a validade do token (e.g., se ele expirou) ao carregar do `SecureStore` e, se inválido, deslogar o usuário automaticamente.
4. Pesquise sobre o `AsyncStorage` do React Native e compare-o com o `expo-secure-store`, destacando as situações em que cada um é mais apropriado.

6. Recursos Adicionais:

- **React Context API (Documentação Oficial):** https://react.dev/learn/passing-props-to-a-component#passing-props-to-a-component
- **`expo-secure-store` (Documentação Oficial):** https://docs.expo.dev/versions/latest/sdk/securestore/
- **Gerenciamento de Estado em React Native:** https://reactnative.dev/docs/state-management

Referências:

[43] https://react.dev/learn/passing-props-to-a-component#passing-props-to-a-component [44] https://docs.expo.dev/versions/latest/sdk/securestore/

Módulo 5: Recursos Nativos e Multimídia

17/out: Instalação e configuração do expo-camera, captura, exibição prévia e cortes nas imagens.

1. Visão Geral:

Os aplicativos móveis frequentemente precisam interagir com os recursos de hardware do dispositivo, como a câmera, para capturar fotos e vídeos. O `expo-camera` é um módulo poderoso que fornece acesso programático à câmera do dispositivo, permitindo que os desenvolvedores criem funcionalidades como captura de fotos, gravação de vídeo, leitura de códigos QR e muito mais. Este tópico abordará a instalação, configuração e uso básico do `expo-camera` para capturar, exibir prévias e manipular imagens.

2. Conceitos Fundamentais:

Acesso à Câmera do Dispositivo:

Para que um aplicativo possa acessar a câmera do dispositivo, ele precisa solicitar permissões ao usuário. No iOS e Android, essas permissões são gerenciadas pelo sistema operacional e devem ser declaradas no arquivo de configuração do aplicativo (`app.json` no Expo). O `expo-camera` simplifica esse processo, fornecendo métodos para verificar e solicitar permissões [45].

`expo-camera`:

O `expo-camera` é um componente React Native que renderiza uma visualização da câmera ao vivo. Ele oferece métodos para:

- **Captura de Fotos:** O método `takePictureAsync()` permite capturar uma foto. Ele retorna um objeto com informações sobre a foto, incluindo a URI (Uniform Resource Identifier) local da imagem [46].
- **Gravação de Vídeos:** O método `recordAsync()` permite iniciar a gravação de um vídeo, e `stopRecording()` para parar [47].
- **Controle da Câmera:** Permite controlar aspectos como o tipo de câmera (frontal/traseira), flash, zoom, proporção da imagem, etc.

Permissões de Câmera:

Antes de usar a câmera, é fundamental verificar se o aplicativo tem permissão para acessá-la. O `expo-camera` oferece o método `Camera.requestCameraPermissionsAsync()` para solicitar a permissão de câmera e `Camera.getCameraPermissionsAsync()` para verificar o status atual da permissão [45].

Manipulação de Imagens (Exibição Prévia e Cortes):

Após a captura de uma imagem, é comum exibi-la para o usuário para uma prévia ou permitir que ele realize edições, como cortes. Para exibição, o componente `Image` do React Native é utilizado. Para cortes e outras manipulações, módulos adicionais ou bibliotecas de terceiros podem ser necessários (como o `expo-image-manipulator` para cortes, embora não seja o foco principal deste tópico, é uma ferramenta útil para mencionar).

3. Tecnologias e Ferramentas:

- **`expo-camera`:** Módulo para acesso à câmera.
- **`Image` (React Native):** Componente para exibir imagens.
- **`Permissions` (Expo):** Para gerenciar permissões do dispositivo.

4. Exemplos Práticos (Código):

Instalação do `expo-camera`:

```
npx expo install expo-camera
```

Configuração no `app.json`:

Para que o aplicativo solicite as permissões corretas, adicione as seguintes entradas na seção `plugins` ou `ios` e `android` do seu `app.json`:

```
{
  "expo": {
    // ... outras configurações
    "plugins": [
      [
        "expo-camera",
        {
          "cameraPermission": "Permita que $(PRODUCT_NAME) acesse sua câmera.",
          "microphonePermission": "Permita que $(PRODUCT_NAME) acesse seu microfone."
        }
      ]
    ],
    "ios": {
      "infoPlist": {
        "NSCameraUsageDescription": "Este aplicativo precisa de acesso à sua câmera para tirar fotos.",
        "NSMicrophoneUsageDescription": "Este aplicativo precisa de acesso ao seu microfone para gravar
vídeos."
      }
    },
    "android": {
      "permissions": [
        "android.permission.CAMERA",
        "android.permission.RECORD_AUDIO"
      ]
    }
  }
}
```

Exemplo de Componente de Câmera para Captura de Foto:

```
 import React, { useState, useEffect, useRef } from 'react';
import { StyleSheet, Text, View, TouchableOpacity, Image, Alert } from 'react-native';
import { Camera } from 'expo-camera';

export default function CameraScreen() {
  const [hasPermission, setHasPermission] = useState(null);
  const [type, setType] = useState(Camera.Constants.Type.back); // Câmera traseira por padrão
  const [capturedImage, setCapturedImage] = useState(null);
  const cameraRef = useRef(null);

  useEffect(() => {
    (async () => {
      const { status } = await Camera.requestCameraPermissionsAsync();
      setHasPermission(status === 'granted');
    })();
  }, []);

  const takePicture = async () => {
    if (cameraRef.current) {
      try {
        const photo = await cameraRef.current.takePictureAsync();
        setCapturedImage(photo.uri);
      } catch (error) {
        console.error('Erro ao tirar foto:', error);
        Alert.alert('Erro', 'Não foi possível tirar a foto.');
      }
    }
  };

  const retakePicture = () => {
    setCapturedImage(null);
  };

  if (hasPermission === null) {
    return <View />;
  }
  if (hasPermission === false) {
    return <Text>Acesso à câmera negado!</Text>;
  }

  return (
    <View style={styles.container}>
      {capturedImage ? (
        <View style={styles.previewContainer}>
          <Image source={{ uri: capturedImage }} style={styles.previewImage} />
          <View style={styles.buttonContainer}>
            <TouchableOpacity style={styles.button} onPress={retakePicture}>
              <Text style={styles.text}>Tirar Outra Foto</Text>
            </TouchableOpacity>
          </View>
        </View>
      ) : (
        <Camera style={styles.camera} type={type} ref={cameraRef}>
          <View style={styles.buttonContainer}>
            <TouchableOpacity
              style={styles.button}
              onPress={() => {
                setType(
                  type === Camera.Constants.Type.back
                    ? Camera.Constants.Type.front
                    : Camera.Constants.Type.back
                );
              }}>
              <Text style={styles.text}>Virar Câmera</Text>
            </TouchableOpacity>
            <TouchableOpacity style={styles.button} onPress={takePicture}>
              <Text style={styles.text}>Tirar Foto</Text>
            </TouchableOpacity>
          </View>
        </Camera>
      )}
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
  },
  camera: {
```

```
    flex: 1,
  },
  buttonContainer: {
    flex: 1,
    backgroundColor: 'transparent',
    flexDirection: 'row',
    margin: 20,
    justifyContent: 'space-around',
    alignItems: 'flex-end',
  },
  button: {
    flex: 0.3,
    alignSelf: 'flex-end',
    alignItems: 'center',
    backgroundColor: '#88888880',
    padding: 10,
    borderRadius: 5,
  },
  text: {
    fontSize: 18,
    color: 'white',
  },
  previewContainer: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
  },
  previewImage: {
    width: '100%',
    height: '100%',
    resizeMode: 'contain',
  },
});
```

5. Exercícios e Desafios:

1. Modifique o exemplo para incluir a funcionalidade de gravação de vídeo. Adicione botões para iniciar e parar a gravação, e exiba uma prévia do vídeo gravado.
2. Implemente um controle de zoom na câmera usando um `Slider` do React Native.
3. Pesquise sobre o `expo-image-manipulator` e integre-o ao exemplo para permitir que o usuário corte a imagem capturada antes de exibi-la na prévia.

6. Recursos Adicionais:

- **Documentação Oficial do `expo-camera`:** https://docs.expo.dev/versions/latest/sdk/camera/
- **Guia de Permissões do Expo:** https://docs.expo.dev/versions/latest/sdk/permissions/
- **`expo-image-manipulator`:** https://docs.expo.dev/versions/latest/sdk/image-manipulator/

Referências:

[45] https://docs.expo.dev/versions/latest/sdk/camera/#permissions [46] https://docs.expo.dev/versions/latest/sdk/camera/#takepictureasyncoptions [47] https://docs.expo.dev/versions/latest/sdk/camera/#recordasyncoptions

20/out: Exploração dos recursos da câmera como fotos, vídeos, microfone e leitor de qrcode.

1. Visão Geral:

Além da captura básica de fotos, a câmera de um dispositivo móvel é uma ferramenta multifuncional que pode ser explorada para diversas finalidades, como gravação de vídeos, captura de áudio (via microfone) e leitura de códigos QR. O `expo-camera` e outros módulos relacionados do Expo oferecem as capacidades para integrar essas funcionalidades avançadas em seu aplicativo. Este tópico aprofundará na exploração desses recursos, permitindo que você crie aplicações mais interativas e ricas em mídia.

2. Conceitos Fundamentais:

Gravação de Vídeo com `expo-camera`:

O `expo-camera` não se limita apenas a fotos; ele também permite a gravação de vídeos. O processo é similar ao de tirar fotos, utilizando os métodos `recordAsync()` para iniciar a gravação e `stopRecording()` para finalizá-la. É possível configurar opções como qualidade do vídeo, duração máxima e proporção [48].

Acesso ao Microfone:

Para gravar vídeos com áudio ou apenas áudio, o aplicativo precisa de permissão para acessar o microfone do dispositivo. Assim como a câmera, essa permissão deve ser solicitada ao usuário e declarada no `app.json` [49].

Leitura de QR Code e Código de Barras:

O `expo-camera` possui uma funcionalidade embutida para detectar códigos de barras e QR codes diretamente do feed da câmera. Isso é extremamente útil para aplicações que precisam escanear produtos, ingressos, ou informações de contato. A propriedade `onBarCodeScanned` do componente `Camera` é um callback que é acionado quando um código é detectado, fornecendo o tipo e os dados do código [50].

3. Tecnologias e Ferramentas:

- **`expo-camera`:** Para gravação de vídeo e leitura de QR codes.
- **`expo-permissions`:** Para gerenciar permissões de microfone.

4. Exemplos Práticos (Código):

Exemplo de Gravação de Vídeo e Leitura de QR Code:

```jsx
 import React, { useState, useEffect, useRef } from 'react';
import { StyleSheet, Text, View, TouchableOpacity, Alert, Button } from 'react-native';
import { Camera } from 'expo-camera';
import { Audio } from 'expo-av'; // Para permissão de microfone

export default function CameraAdvancedScreen() {
  const [hasCameraPermission, setHasCameraPermission] = useState(null);
  const [hasMicrophonePermission, setHasMicrophonePermission] = useState(null);
  const [type, setType] = useState(Camera.Constants.Type.back);
  const [isRecording, setIsRecording] = useState(false);
  const [scanned, setScanned] = useState(false);
  const cameraRef = useRef(null);

  useEffect(() => {
    (async () => {
      const cameraStatus = await Camera.requestCameraPermissionsAsync();
      setHasCameraPermission(cameraStatus.status === 'granted');

      const microphoneStatus = await Audio.requestPermissionsAsync();
      setHasMicrophonePermission(microphoneStatus.status === 'granted');
    })();
  }, []);

  const startRecording = async () => {
    if (cameraRef.current && hasMicrophonePermission) {
      setIsRecording(true);
      try {
        const video = await cameraRef.current.recordAsync();
        Alert.alert('Vídeo Gravado', `Vídeo salvo em: ${video.uri}`);
      } catch (error) {
        console.error('Erro ao gravar vídeo:', error);
        Alert.alert('Erro', 'Não foi possível gravar o vídeo.');
      } finally {
        setIsRecording(false);
      }
    }
  };

  const stopRecording = () => {
    if (cameraRef.current) {
      cameraRef.current.stopRecording();
    }
  };

  const handleBarCodeScanned = ({ type, data }) => {
    setScanned(true);
    Alert.alert(
      `Código ${type} escaneado!`,
      `Dados: ${data}`,
      [
        { text: 'OK', onPress: () => setScanned(false) } // Permite escanear novamente
      ]
    );
  };

  if (hasCameraPermission === null || hasMicrophonePermission === null) {
    return <View />;
  }
  if (hasCameraPermission === false) {
    return <Text>Acesso à câmera negado!</Text>;
  }
  if (hasMicrophonePermission === false) {
    return <Text>Acesso ao microfone negado! Não será possível gravar vídeos com áudio.</Text>;
  }

  return (
    <View style={styles.container}>
      <Camera
        style={styles.camera}
        type={type}
        ref={cameraRef}
        onBarCodeScanned={scanned ? undefined : handleBarCodeScanned} // Desabilita o scanner após a
primeira detecção
      >
        <View style={styles.buttonContainer}>
          <TouchableOpacity
            style={styles.button}
            onPress={() => {
              setType(
                type === Camera.Constants.Type.back
                  ? Camera.Constants.Type.front
```

```
                    : Camera.Constants.Type.back
                );
              }}>
              <Text style={styles.text}>Virar Câmera</Text>
            </TouchableOpacity>

            <TouchableOpacity
              style={styles.button}
              onPress={isRecording ? stopRecording : startRecording}>
              <Text style={styles.text}>{isRecording ? 'Parar Gravação' : 'Gravar Vídeo'}</Text>
            </TouchableOpacity>
          </View>
        </Camera>
        {scanned && (
          <Button title={'Toque para Escanear Novamente'} onPress={() => setScanned(false)} />
        )}
      </View>
    );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
  },
  camera: {
    flex: 1,
  },
  buttonContainer: {
    flex: 1,
    backgroundColor: 'transparent',
    flexDirection: 'row',
    margin: 20,
    justifyContent: 'space-around',
    alignItems: 'flex-end',
  },
  button: {
    flex: 0.4,
    alignSelf: 'flex-end',
    alignItems: 'center',
    backgroundColor: '#88888880',
    padding: 10,
    borderRadius: 5,
  },
  text: {
    fontSize: 18,
    color: 'white',
  },
});
```

5. Exercícios e Desafios:

1. Adicione uma funcionalidade para tirar fotos enquanto o vídeo está sendo gravado (se a API permitir, ou como uma funcionalidade separada).
2. Implemente um botão para alternar o flash da câmera (ligado, desligado, automático).
3. Crie uma tela separada para exibir o vídeo gravado, utilizando o componente `Video` do `expo-av`.
4. Pesquise sobre a detecção de faces com `expo-face-detector` e integre-o ao exemplo da câmera para desenhar um retângulo ao redor de faces detectadas.

6. Recursos Adicionais:

- **Documentação Oficial do `expo-camera` - Gravação de Vídeo:** https://docs.expo.dev/versions/latest/sdk/camera/#recordasyncoptions
- **Documentação Oficial do `expo-av` (Audio/Video):** https://docs.expo.dev/versions/latest/sdk/av/
- **Documentação Oficial do `expo-camera` - Leitura de Código de Barras:** https://docs.expo.dev/versions/latest/sdk/camera/#onbarcodescanned

Referências:

[48] https://docs.expo.dev/versions/latest/sdk/camera/#recordasyncoptions [49] https://docs.expo.dev/versions/latest/sdk/av/#audiorequestpermissionsasync [50] https://docs.expo.dev/versions/latest/sdk/camera/#onbarcodescanned

31/out: Acesso à biblioteca de imagens expo-image-picker.

1. Visão Geral:

Além de capturar novas fotos e vídeos com a câmera, muitos aplicativos móveis precisam permitir que os usuários selecionem mídias (fotos e vídeos) existentes de suas galerias ou bibliotecas de fotos. O `expo-image-picker` é um módulo do Expo que fornece uma interface simples e unificada para acessar a biblioteca de imagens do dispositivo, tanto para selecionar uma única imagem quanto para múltiplas. Este tópico abordará como integrar o `expo-image-picker` em seu aplicativo para permitir que os usuários escolham mídias de suas galerias.

2. Conceitos Fundamentais:

`expo-image-picker`:

O `expo-image-picker` é uma API que permite que os usuários selecionem imagens ou vídeos da biblioteca de mídia do dispositivo ou tirem uma nova foto/vídeo usando a câmera. Ele lida com as permissões necessárias e fornece a URI da mídia selecionada, que pode então ser exibida ou processada [51].

- **`launchImageLibraryAsync(options)`:** Abre a biblioteca de imagens do dispositivo para que o usuário possa selecionar uma imagem ou vídeo. As opções permitem configurar aspectos como qualidade, proporção, e se deve permitir edição [52].
- **`launchCameraAsync(options)`:** Abre a câmera do dispositivo para que o usuário possa tirar uma nova foto ou gravar um vídeo. É uma alternativa ao `expo-camera` para casos de uso mais simples de captura [53].
- **Permissões:** Assim como o `expo-camera`, o `expo-image-picker` requer permissões para acessar a biblioteca de mídia (e a câmera, se `launchCameraAsync` for usado). Ele fornece métodos para solicitar e verificar essas permissões (`ImagePicker.requestMediaLibraryPermissionsAsync()`, `ImagePicker.requestCameraPermissionsAsync()`) [54].

Manipulação de Mídia Selecionada:

Após a seleção, o `expo-image-picker` retorna um objeto com informações sobre a mídia, incluindo sua URI local. Essa URI pode ser usada para:

- **Exibir a Mídia:** Usando o componente `Image` ou `Video` do React Native.
- **Upload para um Servidor:** A URI pode ser usada para ler o arquivo e enviá-lo para um serviço de armazenamento em nuvem ou API.
- **Processamento Local:** Manipular a imagem (redimensionar, cortar) usando outras bibliotecas como `expo-image-manipulator`.

3. Tecnologias e Ferramentas:

- **`expo-image-picker`:** Módulo para acesso à biblioteca de imagens.
- **`Image` (React Native):** Para exibir imagens selecionadas.
- **`Video` (expo-av):** Para exibir vídeos selecionados.

4. Exemplos Práticos (Código):

Instalação do `expo-image-picker`:

```
npx expo install expo-image-picker
```

Configuração no `app.json`:

Para que o aplicativo solicite as permissões corretas, adicione as seguintes entradas na seção `plugins` ou `ios` e `android` do seu `app.json`:

```
{
  "expo": {
    // ... outras configurações
    "plugins": [
      [
        "expo-image-picker",
        {
          "photosPermission": "Permita que $(PRODUCT_NAME) acesse suas fotos para que você possa escolher
uma imagem de perfil.",
          "cameraPermission": "Permita que $(PRODUCT_NAME) acesse sua câmera para tirar fotos."
        }
      ]
    ],
    "ios": {
      "infoPlist": {
        "NSPhotoLibraryUsageDescription": "Este aplicativo precisa de acesso à sua biblioteca de fotos
para selecionar imagens.",
        "NSCameraUsageDescription": "Este aplicativo precisa de acesso à sua câmera para tirar fotos."
      }
    },
    "android": {
      "permissions": [
        "android.permission.READ_EXTERNAL_STORAGE",
        "android.permission.WRITE_EXTERNAL_STORAGE",
        "android.permission.CAMERA"
      ]
    }
  }
}
```

Exemplo de Seleção de Imagem da Galeria:

```
 import React, { useState, useEffect } from 'react';
import { Button, Image, View, Platform, StyleSheet, Alert } from 'react-native';
import * as ImagePicker from 'expo-image-picker';

export default function ImagePickerScreen() {
  const [image, setImage] = useState(null);

  useEffect(() => {
    (async () => {
      if (Platform.OS !== 'web') {
        const { status } = await ImagePicker.requestMediaLibraryPermissionsAsync();
        if (status !== 'granted') {
          Alert.alert('Permissão Negada', 'Desculpe, precisamos de permissões da galeria para isso
funcionar!');
        }
      }
    })();
  }, []);

  const pickImage = async () => {
    let result = await ImagePicker.launchImageLibraryAsync({
      mediaTypes: ImagePicker.MediaTypeOptions.Images, // Apenas imagens
      allowsEditing: true, // Permite edição (corte)
      aspect: [4, 3], // Proporção para corte
      quality: 1, // Qualidade da imagem (0 a 1)
    });

    if (!result.canceled) {
      setImage(result.assets[0].uri); // Use result.assets[0].uri para Expo SDK 48+
    }
  };

  const takePhoto = async () => {
    const { status } = await ImagePicker.requestCameraPermissionsAsync();
    if (status !== 'granted') {
      Alert.alert('Permissão Negada', 'Desculpe, precisamos de permissões da câmera para isso
funcionar!');
      return;
    }

    let result = await ImagePicker.launchCameraAsync({
      mediaTypes: ImagePicker.MediaTypeOptions.Images,
      allowsEditing: true,
      aspect: [4, 3],
      quality: 1,
    });

    if (!result.canceled) {
      setImage(result.assets[0].uri);
    }
  };

  return (
    <View style={styles.container}>
      <View style={styles.buttonContainer}>
        <Button title="Escolher Imagem da Galeria" onPress={pickImage} />
        <Button title="Tirar Foto com a Câmera" onPress={takePhoto} />
      </View>
      {image && <Image source={{ uri: image }} style={styles.image} />}
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    alignItems: 'center',
    justifyContent: 'center',
    padding: 20,
  },
  buttonContainer: {
    flexDirection: 'row',
    justifyContent: 'space-around',
    width: '100%',
    marginBottom: 20,
  },
  image: {
    width: 200,
    height: 200,
    borderRadius: 10,
    marginTop: 20,
```

```
  },
});
```

5. Exercícios e Desafios:

1. Modifique o exemplo para permitir que o usuário selecione múltiplos itens da galeria (fotos e/ou vídeos). Exiba todas as mídias selecionadas em uma `FlatList`.
2. Implemente uma funcionalidade para fazer o upload da imagem selecionada para um serviço de armazenamento em nuvem (e.g., Cloudinary, Firebase Storage). Você precisará de um backend ou de credenciais de API para isso.
3. Pesquise sobre as opções de `mediaTypes` no `expo-image-picker` e crie um exemplo que permita ao usuário selecionar apenas vídeos.

6. Recursos Adicionais:

- **Documentação Oficial do `expo-image-picker`:** https://docs.expo.dev/versions/latest/sdk/image-picker/
- **Guia de Permissões do Expo:** https://docs.expo.dev/versions/latest/sdk/permissions/

Referências:

[51] https://docs.expo.dev/versions/latest/sdk/image-picker/ [52] https://docs.expo.dev/versions/latest/sdk/image-picker/#launchimagelibraryasyncoptions [53] https://docs.expo.dev/versions/latest/sdk/image-picker/#launchcameraasyncoptions [54] https://docs.expo.dev/versions/latest/sdk/image-picker/#requestmedialibrarypermissionsasync

Módulo 6: Banco de Dados Local

03/nov: Banco de dados local com SQLite. API Transacional com expo-sqlite.

1. Visão Geral:

Para muitos aplicativos móveis, a capacidade de armazenar e gerenciar dados localmente no dispositivo é fundamental, especialmente quando a conectividade com a internet é intermitente ou inexistente. O SQLite é um sistema de gerenciamento de banco de dados relacional leve e embutido, amplamente utilizado em dispositivos móveis. O `expo-sqlite` é um módulo do Expo que fornece uma interface para interagir com bancos de dados SQLite em aplicativos React Native. Este tópico abordará como configurar e utilizar o `expo-sqlite` para operações básicas de banco de dados, incluindo o conceito de transações.

2. Conceitos Fundamentais:

SQLite:

SQLite é uma biblioteca de software que implementa um sistema de gerenciamento de banco de dados SQL transacional, autônomo, sem servidor, sem configuração. É o sistema de banco de dados mais amplamente implantado no mundo, presente em bilhões de dispositivos. Sua popularidade em dispositivos móveis se deve à sua leveza, eficiência e por não exigir um processo de servidor separado [55].

- **Características:**
  - **Embutido:** O banco de dados é armazenado em um único arquivo no dispositivo.
  - **Sem Servidor:** Não há um processo de servidor separado; o aplicativo interage diretamente com o arquivo do banco de dados.

- **Transacional:** Suporta transações ACID (Atomicidade, Consistência, Isolamento, Durabilidade), garantindo a integridade dos dados.

`expo-sqlite`:

O `expo-sqlite` é um módulo do Expo que permite que aplicativos React Native interajam com bancos de dados SQLite. Ele fornece uma API JavaScript para abrir bancos de dados, executar comandos SQL e processar resultados. É uma ponte entre o JavaScript do seu aplicativo e a funcionalidade SQLite nativa do dispositivo [56].

- **`SQLite.openDatabase(name, version, description, size)`:** Abre ou cria um banco de dados SQLite. Retorna um objeto de banco de dados que pode ser usado para executar transações [57].
- **`db.transaction(callback)`:** Inicia uma transação de banco de dados. Todas as operações SQL dentro do callback são executadas como uma única unidade atômica. Se qualquer operação falhar, a transação inteira é revertida (rollback); caso contrário, todas as operações são confirmadas (commit) [58].
- **`tx.executeSql(sqlStatement, arguments, successCallback, errorCallback)`:** Executa uma única instrução SQL dentro de uma transação. É o método principal para interagir com o banco de dados [59].

Transações de Banco de Dados:

Uma transação é uma sequência de operações de banco de dados executadas como uma única unidade lógica de trabalho. As transações garantem a integridade dos dados, aderindo às propriedades ACID:

- **Atomicidade:** Todas as operações dentro de uma transação são concluídas com sucesso, ou nenhuma delas é. Se uma parte da transação falhar, toda a transação é desfeita.
- **Consistência:** Uma transação leva o banco de dados de um estado válido para outro estado válido.
- **Isolamento:** As transações são executadas de forma isolada umas das outras, de modo que o resultado de uma transação não é afetado por outras transações concorrentes.
- **Durabilidade:** Uma vez que uma transação é confirmada, suas alterações são permanentes e sobrevivem a falhas do sistema.

O uso de transações é crucial para operações que envolvem múltiplas etapas e onde a falha de uma etapa pode deixar o banco de dados em um estado inconsistente (e.g., transferências bancárias, onde o débito e o crédito devem ocorrer juntos).

3. Tecnologias e Ferramentas:

- **SQLite:** O sistema de banco de dados embutido.
- **`expo-sqlite`:** Módulo Expo para interação com SQLite.

4. Exemplos Práticos (Código):

Instalação do `expo-sqlite`:

```
npx expo install expo-sqlite
```

Exemplo de Operações Básicas com `expo-sqlite` e Transações:

```
 import * as SQLite from 'expo-sqlite';
import { Alert } from 'react-native';

const db = SQLite.openDatabase('mydatabase.db');

const initDatabase = () => {
  db.transaction(tx => {
    tx.executeSql(
      'CREATE TABLE IF NOT EXISTS items (id INTEGER PRIMARY KEY AUTOINCREMENT, name TEXT, quantity
INTEGER);',
      [],
      () => console.log('Tabela items criada ou já existe.'),
      (_, error) => console.error('Erro ao criar tabela:', error)
    );
  });
};

const insertItem = (name, quantity) => {
  db.transaction(tx => {
    tx.executeSql(
      'INSERT INTO items (name, quantity) VALUES (?, ?);',
      [name, quantity],
      () => Alert.alert('Sucesso', `Item ${name} inserido com sucesso!`),
      (_, error) => Alert.alert('Erro', `Erro ao inserir item: ${error.message}`)
    );
  });
};

const getAllItems = (callback) => {
  db.transaction(tx => {
    tx.executeSql(
      'SELECT * FROM items;',
      [],
      (_, { rows }) => {
        console.log('Itens:', rows._array);
        callback(rows._array);
      },
      (_, error) => console.error('Erro ao buscar itens:', error)
    );
  });
};

const updateItemQuantity = (id, newQuantity) => {
  db.transaction(tx => {
    tx.executeSql(
      'UPDATE items SET quantity = ? WHERE id = ?;',
      [newQuantity, id],
      () => Alert.alert('Sucesso', `Quantidade do item $`{id} atualizada para `${newQuantity}.`),
      (_, error) => Alert.alert('Erro', `Erro ao atualizar item: ${error.message}`)
    );
  });
};

const deleteItem = (id) => {
  db.transaction(tx => {
    tx.executeSql(
      'DELETE FROM items WHERE id = ?;',
      [id],
      () => Alert.alert('Sucesso', `Item ${id} excluído com sucesso!`),
      (_, error) => Alert.alert('Erro', `Erro ao excluir item: ${error.message}`)
    );
  });
};

// Exemplo de uso em um componente React Native
import React, { useEffect, useState } from 'react';
import { View, Text, Button, FlatList, StyleSheet } from 'react-native';

export default function SQLiteScreen() {
  const [items, setItems] = useState([]);

  useEffect(() => {
    initDatabase();
    fetchItems();
  }, []);

  const fetchItems = () => {
    getAllItems(setItems);
  };

  const handleAddItem = () => {
```

```
    const randomName = `Item ${Math.floor(Math.random() * 1000)}`;
    const randomQuantity = Math.floor(Math.random() * 10) + 1;
    insertItem(randomName, randomQuantity);
    fetchItems(); // Atualiza a lista após a inserção
  };

  const renderItem = ({ item }) => (
    <View style={styles.itemContainer}>
      <Text style={styles.itemText}>{item.name} (Qtd: {item.quantity})</Text>
      <Button title="Excluir" onPress={() => {
        deleteItem(item.id);
        fetchItems();
      }} />
    </View>
  );

  return (
    <View style={styles.container}>
      <Text style={styles.title}>Gerenciamento de Itens</Text>
      <Button title="Adicionar Novo Item" onPress={handleAddItem} />
      <FlatList
        data={items}
        renderItem={renderItem}
        keyExtractor={item => item.id.toString()}
        style={styles.list}
      />
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    padding: 20,
    backgroundColor: '#f5f5f5',
  },
  title: {
    fontSize: 24,
    fontWeight: 'bold',
    marginBottom: 20,
    textAlign: 'center',
  },
  list: {
    marginTop: 20,
  },
  itemContainer: {
    flexDirection: 'row',
    justifyContent: 'space-between',
    alignItems: 'center',
    backgroundColor: '#fff',
    padding: 15,
    marginBottom: 10,
    borderRadius: 8,
    shadowColor: '#000',
    shadowOffset: { width: 0, height: 1 },
    shadowOpacity: 0.2,
    shadowRadius: 1.41,
    elevation: 2,
  },
  itemText: {
    fontSize: 16,
  },
});
```

5. Exercícios e Desafios:

1. Adicione uma funcionalidade para editar a quantidade de um item existente. Crie um `TextInput` e um botão de atualização ao lado de cada item na lista.
2. Implemente uma transação mais complexa que envolva múltiplas operações SQL. Por exemplo, uma função `transferItems(fromItemId, toItemId, quantity)` que diminua a quantidade de um item e aumente a de outro, garantindo que ambas as operações sejam bem-sucedidas ou nenhuma delas.
3. Pesquise sobre a diferença entre `db.transaction` e `db.exec` no `expo-sqlite` e quando usar cada um.

6. Recursos Adicionais:

- **Documentação Oficial do `expo-sqlite`:** https://docs.expo.dev/versions/latest/sdk/sqlite/
- **SQLite Home Page:** https://www.sqlite.org/
- **Transações ACID:** https://pt.wikipedia.org/wiki/ACID

Referências:

[55] https://www.sqlite.org/about.html [56] https://docs.expo.dev/versions/latest/sdk/sqlite/ [57] https://docs.expo.dev/versions/latest/sdk/sqlite/#sqliteopendatabasename-version-description-size-callback [58] https://docs.expo.dev/versions/latest/sdk/sqlite/#dbtransactioncallback-errorcallback-successcallback [59] https://docs.expo.dev/versions/latest/sdk/sqlite/#txexecutesqlsqlstatement-arguments-successcallback-errorcallback

07/nov: Implementação de um helper para acesso ao banco de dados.

1. Visão Geral:

À medida que a complexidade das interações com o banco de dados aumenta, torna-se essencial criar uma camada de abstração para gerenciar essas operações. Um "helper" ou "service" de banco de dados centraliza a lógica de acesso a dados, tornando o código mais organizado, reutilizável e fácil de manter. Essa abordagem segue o princípio da separação de preocupações, isolando a lógica de persistência do restante da aplicação. Este tópico abordará a criação de um helper para interagir com o `expo-sqlite`, encapsulando as operações CRUD (Create, Read, Update, Delete) e outras consultas comuns.

2. Conceitos Fundamentais:

Padrão de Projeto Repository:

O padrão Repository atua como um intermediário entre a camada de domínio e a camada de persistência de dados. Ele abstrai os detalhes de como os dados são armazenados e recuperados, permitindo que a lógica de negócios interaja com objetos de domínio de forma agnóstica ao banco de dados. Em vez de chamar diretamente as APIs do `expo-sqlite`, a aplicação interage com o repositório, que por sua vez, lida com as operações de baixo nível [60].

- **Benefícios:**
  - **Separação de Preocupações:** A lógica de acesso a dados é isolada da lógica de negócios.
  - **Testabilidade:** Facilita a escrita de testes unitários para a lógica de negócios, pois o repositório pode ser "mockado" ou "stubado".
  - **Manutenibilidade:** Alterações no banco de dados (e.g., migração para outro tipo de banco de dados) afetam apenas a implementação do repositório, não a lógica de negócios.
  - **Reutilização:** As operações de banco de dados podem ser reutilizadas em diferentes partes da aplicação.

Encapsulamento de Operações SQL:

Um helper de banco de dados encapsula as instruções SQL e as chamadas à API do `expo-sqlite` dentro de métodos JavaScript. Isso significa que, em vez de escrever `db.transaction(tx => tx.executeSql(...))` repetidamente, você terá métodos como `dbHelper.insertUser(user)` ou `dbHelper.getProducts()`. Essa abstração torna o código mais limpo e menos propenso a erros.

Tratamento de Erros Centralizado:

Ao centralizar as operações de banco de dados em um helper, o tratamento de erros também pode ser centralizado. Isso permite implementar uma estratégia consistente para lidar com exceções, logar erros e fornecer feedback adequado ao usuário.

3. Tecnologias e Ferramentas:

- **`expo-sqlite`:** Para a interação com o banco de dados.
- **JavaScript/TypeScript:** Para implementar o helper/service.
- **Promises/Async/Await:** Para lidar com operações assíncronas de banco de dados de forma mais legível.

4. Exemplos Práticos (Código):

Vamos criar um `DatabaseHelper.js` que encapsula as operações para uma tabela de `users`.

`src/database/DatabaseHelper.js`:

```
import * as SQLite from 'expo-sqlite';

const databaseName = 'app.db';
const db = SQLite.openDatabase(databaseName);

const DatabaseHelper = {
  init: () => {
    return new Promise((resolve, reject) => {
      db.transaction(
        tx => {
          tx.executeSql(
            'CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMARY KEY AUTOINCREMENT, name TEXT NOT NULL,
email TEXT UNIQUE NOT NULL);',
            [],
            () => {
              console.log('Tabela users criada ou já existe.');
              resolve();
            },
            (_, error) => {
              console.error('Erro ao criar tabela users:', error);
              reject(error);
            }
          );
        },
        error => {
          console.error('Erro na transação de inicialização:', error);
          reject(error);
        },
        () => console.log('Transação de inicialização concluída.')
      );
    });
  },

  insertUser: (name, email) => {
    return new Promise((resolve, reject) => {
      db.transaction(
        tx => {
          tx.executeSql(
            'INSERT INTO users (name, email) VALUES (?, ?);',
            [name, email],
            (_, result) => {
              console.log(`Usuário ${`{name} inserido com ID: `${result.insertId}`);
              resolve(result.insertId);
            },
            (_, error) => {
              console.error('Erro ao inserir usuário:', error);
              reject(error);
            }
          );
        },
        error => {
          console.error('Erro na transação de inserção:', error);
          reject(error);
        }
      );
    });
  },

  getAllUsers: () => {
    return new Promise((resolve, reject) => {
      db.transaction(
        tx => {
          tx.executeSql(
            'SELECT * FROM users;',
            [],
            (_, { rows }) => {
              console.log('Usuários encontrados:', rows._array);
              resolve(rows._array);
            },
            (_, error) => {
              console.error('Erro ao buscar usuários:', error);
              reject(error);
            }
          );
        },
        error => {
          console.error('Erro na transação de busca:', error);
          reject(error);
        }
      );
    });
```

```
  },

  getUserById: (id) => {
    return new Promise((resolve, reject) => {
      db.transaction(
        tx => {
          tx.executeSql(
            'SELECT * FROM users WHERE id = ?;',
            [id],
            (_, { rows }) => {
              console.log(`Usuário com ID ${id}:`, rows._array[0]);
              resolve(rows._array[0]);
            },
            (_, error) => {
              console.error(`Erro ao buscar usuário por ID ${id}:`, error);
              reject(error);
            }
          );
        },
        error => {
          console.error('Erro na transação de busca por ID:', error);
          reject(error);
        }
      );
    });
  },

  updateUserName: (id, newName) => {
    return new Promise((resolve, reject) => {
      db.transaction(
        tx => {
          tx.executeSql(
            'UPDATE users SET name = ? WHERE id = ?;',
            [newName, id],
            (_, result) => {
              console.log(`Usuário $`{id} atualizado. Linhas afetadas: `${result.rowsAffected}`);
              resolve(result.rowsAffected);
            },
            (_, error) => {
              console.error(`Erro ao atualizar usuário ${id}:`, error);
              reject(error);
            }
          );
        },
        error => {
          console.error('Erro na transação de atualização:', error);
          reject(error);
        }
      );
    });
  },

  deleteUser: (id) => {
    return new Promise((resolve, reject) => {
      db.transaction(
        tx => {
          tx.executeSql(
            'DELETE FROM users WHERE id = ?;',
            [id],
            (_, result) => {
              console.log(`Usuário $`{id} excluído. Linhas afetadas: `${result.rowsAffected}`);
              resolve(result.rowsAffected);
            },
            (_, error) => {
              console.error(`Erro ao excluir usuário ${id}:`, error);
              reject(error);
            }
          );
        },
        error => {
          console.error('Erro na transação de exclusão:', error);
          reject(error);
        }
      );
    });
  },
};

export default DatabaseHelper;
```

Como usar o `DatabaseHelper` em um componente React Native:

```
 import React, { useEffect, useState } from 'react';
import { View, Text, Button, FlatList, StyleSheet, Alert, TextInput } from 'react-native';
import DatabaseHelper from '../database/DatabaseHelper';

export default function UserManagementScreen() {
  const [users, setUsers] = useState([]);
  const [userName, setUserName] = useState('');
  const [userEmail, setUserEmail] = useState('');

  useEffect(() => {
    const setupDatabase = async () => {
      try {
        await DatabaseHelper.init();
        fetchUsers();
      } catch (error) {
        Alert.alert('Erro', 'Falha ao inicializar o banco de dados.');
      }
    };
    setupDatabase();
  }, []);

  const fetchUsers = async () => {
    try {
      const fetchedUsers = await DatabaseHelper.getAllUsers();
      setUsers(fetchedUsers);
    } catch (error) {
      Alert.alert('Erro', 'Falha ao buscar usuários.');
    }
  };

  const handleAddUser = async () => {
    if (!userName || !userEmail) {
      Alert.alert('Erro', 'Nome e E-mail são obrigatórios.');
      return;
    }
    try {
      await DatabaseHelper.insertUser(userName, userEmail);
      setUserName('');
      setUserEmail('');
      fetchUsers();
    } catch (error) {
      Alert.alert('Erro', `Falha ao adicionar usuário: ${error.message}`);
    }
  };

  const handleDeleteUser = async (id) => {
    try {
      await DatabaseHelper.deleteUser(id);
      fetchUsers();
    } catch (error) {
      Alert.alert('Erro', 'Falha ao excluir usuário.');
    }
  };

  const renderUserItem = ({ item }) => (
    <View style={styles.userItem}>
      <Text style={styles.userName}>{item.name}</Text>
      <Text style={styles.userEmail}>{item.email}</Text>
      <Button title="Excluir" onPress={() => handleDeleteUser(item.id)} color="red" />
    </View>
  );

  return (
    <View style={styles.container}>
      <Text style={styles.title}>Gerenciamento de Usuários</Text>
      <TextInput
        style={styles.input}
        placeholder="Nome do Usuário"
        value={userName}
        onChangeText={setUserName}
      />
      <TextInput
        style={styles.input}
        placeholder="E-mail do Usuário"
        value={userEmail}
        onChangeText={setUserEmail}
        keyboardType="email-address"
        autoCapitalize="none"
      />
      <Button title="Adicionar Usuário" onPress={handleAddUser} />
```

```
      <FlatList
        data={users}
        renderItem={renderUserItem}
        keyExtractor={item => item.id.toString()}
        style={styles.userList}
      />
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    padding: 20,
    backgroundColor: '#f5f5f5',
  },
  title: {
    fontSize: 24,
    fontWeight: 'bold',
    marginBottom: 20,
    textAlign: 'center',
  },
  input: {
    height: 40,
    borderColor: '#ccc',
    borderWidth: 1,
    borderRadius: 5,
    paddingHorizontal: 10,
    marginBottom: 10,
  },
  userList: {
    marginTop: 20,
  },
  userItem: {
    backgroundColor: '#fff',
    padding: 15,
    borderRadius: 8,
    marginBottom: 10,
    flexDirection: 'row',
    justifyContent: 'space-between',
    alignItems: 'center',
    shadowColor: '#000',
    shadowOffset: { width: 0, height: 1 },
    shadowOpacity: 0.2,
    shadowRadius: 1.41,
    elevation: 2,
  },
  userName: {
    fontSize: 16,
    fontWeight: 'bold',
  },
  userEmail: {
    fontSize: 14,
    color: '#666',
  },
});
```

5. Exercícios e Desafios:

1. Modifique o `DatabaseHelper` para incluir operações CRUD para uma nova tabela, por exemplo, `products` (id, name, price, description).
2. Crie uma tela `ProductManagementScreen` que utilize o `DatabaseHelper` para adicionar, listar, atualizar e excluir produtos.
3. Implemente uma função no `DatabaseHelper` que execute uma transação complexa, como a transferência de um item de um usuário para outro, garantindo a atomicidade da operação.
4. Pesquise sobre o uso de `async/await` com `expo-sqlite` para simplificar o código das transações e remover os callbacks aninhados.

6. Recursos Adicionais:

- **Padrão Repository:** https://martinfowler.com/eaaDev/Repository.html

- **Promises em JavaScript:** https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Promise
- **Async/Await:** https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Statements/async_function

Referências:

[60] https://martinfowler.com/eaaDev/Repository.html

10/nov: Integração do banco de dados com as telas implementação de um CRUD.

1. Visão Geral:

Após configurar um banco de dados local com SQLite e criar um helper para encapsular as operações de acesso a dados, o próximo passo é integrar essas funcionalidades com a interface do usuário. A implementação de operações CRUD (Create, Read, Update, Delete) diretamente nas telas do aplicativo é fundamental para permitir que os usuários interajam com os dados persistidos. Este tópico abordará como conectar a lógica de banco de dados (via `DatabaseHelper`) com os componentes React Native para criar uma experiência de usuário completa para gerenciamento de dados.

2. Conceitos Fundamentais:

Operações CRUD:

CRUD é um acrônimo para as quatro operações básicas de persistência de dados que a maioria dos aplicativos baseados em dados implementa [61]:

- **Create (Criar):** Adicionar novos registros ao banco de dados. No contexto de UI, isso geralmente envolve um formulário onde o usuário insere dados e um botão para salvar.
- **Read (Ler):** Recuperar dados do banco de dados para exibição. Isso pode ser uma lista de itens, detalhes de um único item, ou resultados de uma busca.
- **Update (Atualizar):** Modificar registros existentes no banco de dados. Na UI, isso pode ser um formulário pré-preenchido com dados existentes que o usuário pode editar e salvar.
- **Delete (Excluir):** Remover registros do banco de dados. Geralmente implementado com um botão de exclusão ao lado de cada item ou em uma tela de detalhes.

Integração UI-Banco de Dados:

A integração eficaz entre a UI e o banco de dados local envolve:

- **Fluxo de Dados Unidirecional:** A UI dispara eventos (e.g., clique de botão), que chamam funções no `DatabaseHelper`. O `DatabaseHelper` interage com o SQLite e, após a conclusão da operação, a UI é atualizada para refletir as mudanças nos dados.
- **Gerenciamento de Estado:** Utilizar o estado local (`useState`) e efeitos (`useEffect`) nos componentes React Native para:
  - Armazenar os dados recuperados do banco de dados para exibição.
  - Gerenciar o estado dos formulários (valores dos inputs).
  - Disparar a busca de dados quando a tela é carregada ou quando uma operação de CRUD é concluída.
- **Feedback ao Usuário:** Fornecer feedback visual ao usuário sobre o sucesso ou falha das operações (e.g., mensagens de alerta, indicadores de carregamento).

3. Tecnologias e Ferramentas:

- **`expo-sqlite`:** Para a interação com o banco de dados.
- **`DatabaseHelper`:** O helper customizado para encapsular as operações SQL.
- **React Native Components:** `TextInput`, `Button`, `FlatList`, `Alert` para construir a UI.
- **React Hooks:** `useState`, `useEffect` para gerenciamento de estado e ciclo de vida.

4. Exemplos Práticos (Código):

Vamos expandir o exemplo do `UserManagementScreen` do tópico anterior para incluir todas as operações CRUD de forma mais completa.

`src/screens/UserManagementScreen.js` (Versão Completa com CRUD):

```
 import React, { useEffect, useState } from 'react';
import { View, Text, Button, FlatList, StyleSheet, Alert, TextInput, TouchableOpacity } from 'react-
native';
import DatabaseHelper from '../database/DatabaseHelper'; // Nosso helper de banco de dados

export default function UserManagementScreen() {
  const [users, setUsers] = useState([]);
  const [userName, setUserName] = useState('');
  const [userEmail, setUserEmail] = useState('');
  const [editingUser, setEditingUser] = useState(null); // Para armazenar o usuário sendo editado

  useEffect(() => {
    const setupDatabase = async () => {
      try {
        await DatabaseHelper.init();
        fetchUsers();
      } catch (error) {
        Alert.alert('Erro', 'Falha ao inicializar o banco de dados.');
      }
    };
    setupDatabase();
  }, []);

  const fetchUsers = async () => {
    try {
      const fetchedUsers = await DatabaseHelper.getAllUsers();
      setUsers(fetchedUsers);
    } catch (error) {
      Alert.alert('Erro', 'Falha ao buscar usuários.');
    }
  };

  const handleAddOrUpdateUser = async () => {
    if (!userName || !userEmail) {
      Alert.alert('Erro', 'Nome e E-mail são obrigatórios.');
      return;
    }

    try {
      if (editingUser) {
        // Atualizar usuário existente
        await DatabaseHelper.updateUserName(editingUser.id, userName); // Assumindo que só atualizamos o
nome por simplicidade
        // Para atualizar o email, você precisaria de um método update completo no helper
        Alert.alert('Sucesso', `Usuário ${userName} atualizado com sucesso!`);
      } else {
        // Adicionar novo usuário
        await DatabaseHelper.insertUser(userName, userEmail);
        Alert.alert('Sucesso', `Usuário ${userName} adicionado com sucesso!`);
      }
      setUserName('');
      setUserEmail('');
      setEditingUser(null);
      fetchUsers(); // Atualiza a lista após a operação
    } catch (error) {
      Alert.alert('Erro', `Falha ao $`{editingUser ? 'atualizar' : 'adicionar'} usuário:
`${error.message}`);
    }
  };

  const handleDeleteUser = async (id) => {
    Alert.alert(
      'Confirmar Exclusão',
      'Tem certeza que deseja excluir este usuário?',
      [
        { text: 'Cancelar', style: 'cancel' },
        {
          text: 'Excluir',
          onPress: async () => {
            try {
              await DatabaseHelper.deleteUser(id);
              Alert.alert('Sucesso', 'Usuário excluído com sucesso!');
              fetchUsers();
            } catch (error) {
              Alert.alert('Erro', 'Falha ao excluir usuário.');
            }
          },
        },
      ],
      { cancelable: true }
    );
```

```
  };

  const handleEditUser = (user) => {
    setEditingUser(user);
    setUserName(user.name);
    setUserEmail(user.email); // Preenche o formulário com os dados do usuário
  };

  const renderUserItem = ({ item }) => (
    <View style={styles.userItem}>
      <View>
        <Text style={styles.userName}>{item.name}</Text>
        <Text style={styles.userEmail}>{item.email}</Text>
      </View>
      <View style={styles.itemActions}>
        <TouchableOpacity onPress={() => handleEditUser(item)} style={styles.actionButton}>
          <Text style={styles.actionButtonText}>Editar</Text>
        </TouchableOpacity>
        <TouchableOpacity onPress={() => handleDeleteUser(item.id)} style={[styles.actionButton,
styles.deleteButton]}>
          <Text style={styles.actionButtonText}>Excluir</Text>
        </TouchableOpacity>
      </View>
    </View>
  );

  return (
    <View style={styles.container}>
      <Text style={styles.title}>{editingUser ? 'Editar Usuário' : 'Adicionar Novo Usuário'}</Text>
      <TextInput
        style={styles.input}
        placeholder="Nome do Usuário"
        value={userName}
        onChangeText={setUserName}
      />
      <TextInput
        style={styles.input}
        placeholder="E-mail do Usuário"
        value={userEmail}
        onChangeText={setUserEmail}
        keyboardType="email-address"
        autoCapitalize="none"
        editable={!editingUser} // Não permite editar o e-mail ao editar (ID único)
      />
      <Button title={editingUser ? 'Atualizar Usuário' : 'Adicionar Usuário'} onPress=
{handleAddOrUpdateUser} />
      {editingUser && (
        <Button title="Cancelar Edição" onPress={() => {
          setEditingUser(null);
          setUserName('');
          setUserEmail('');
        }} color="gray" style={{ marginTop: 10 }} />
      )}

      <Text style={styles.listTitle}>Lista de Usuários</Text>
      <FlatList
        data={users}
        renderItem={renderUserItem}
        keyExtractor={item => item.id.toString()}
        style={styles.userList}
        ListEmptyComponent={<Text style={styles.emptyListText}>Nenhum usuário cadastrado.</Text>}
      />
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    padding: 20,
    backgroundColor: '#f5f5f5',
  },
  title: {
    fontSize: 24,
    fontWeight: 'bold',
    marginBottom: 20,
    textAlign: 'center',
  },
  input: {
    height: 40,
    borderColor: '#ccc',
    borderWidth: 1,
```

```
    borderRadius: 5,
    paddingHorizontal: 10,
    marginBottom: 10,
  },
  listTitle: {
    fontSize: 20,
    fontWeight: 'bold',
    marginTop: 30,
    marginBottom: 15,
    textAlign: 'center',
  },
  userList: {
    marginTop: 10,
  },
  userItem: {
    backgroundColor: '#fff',
    padding: 15,
    borderRadius: 8,
    marginBottom: 10,
    flexDirection: 'row',
    justifyContent: 'space-between',
    alignItems: 'center',
    shadowColor: '#000',
    shadowOffset: { width: 0, height: 1 },
    shadowOpacity: 0.2,
    shadowRadius: 1.41,
    elevation: 2,
  },
  userName: {
    fontSize: 16,
    fontWeight: 'bold',
  },
  userEmail: {
    fontSize: 14,
    color: '#666',
  },
  itemActions: {
    flexDirection: 'row',
  },
  actionButton: {
    backgroundColor: '#007bff',
    paddingVertical: 5,
    paddingHorizontal: 10,
    borderRadius: 5,
    marginLeft: 10,
  },
  deleteButton: {
    backgroundColor: '#dc3545',
  },
  actionButtonText: {
    color: '#fff',
    fontSize: 12,
  },
  emptyListText: {
    textAlign: 'center',
    marginTop: 20,
    fontSize: 16,
    color: '#666',
  },
});
```

5. Exercícios e Desafios:

1. Modifique o `DatabaseHelper` para que o método `updateUserName` possa também atualizar o e-mail do usuário. Em seguida, ajuste a `UserManagementScreen` para permitir a edição de ambos os campos.
2. Adicione uma funcionalidade de busca à `UserManagementScreen` que filtre a lista de usuários exibidos com base no nome ou e-mail digitado em um `TextInput` de busca.
3. Implemente uma validação mais robusta no formulário de adição/edição de usuário, por exemplo, verificando o formato do e-mail e se o nome não está vazio.
4. Crie uma tela de detalhes para o usuário. Ao clicar em um item da lista, navegue para essa tela, passando o ID do usuário. Na tela de detalhes, busque as informações completas do usuário e as exiba.

6. Recursos Adicionais:

- **CRUD Operations Explained:** https://www.codecademy.com/articles/what-is-crud
- **React Native Forms:** https://reactnative.dev/docs/textinput
- **React Native FlatList:** https://reactnative.dev/docs/flatlist

Referências:

[61] https://www.codecademy.com/articles/what-is-crud

Módulo 7: Geolocalização e Mapas

17/nov: Geolocalização (GPS) módulo expo-location.

1. Visão Geral:

A capacidade de determinar a localização de um dispositivo móvel é uma funcionalidade essencial para uma vasta gama de aplicativos, desde serviços de mapas e navegação até redes sociais baseadas em localização e aplicativos de entrega. O módulo `expo-location` fornece uma API unificada para acessar os serviços de geolocalização do dispositivo, como GPS, Wi-Fi e redes celulares. Este tópico abordará como obter a localização atual do usuário, monitorar mudanças de localização e gerenciar as permissões necessárias para acessar esses serviços.

2. Conceitos Fundamentais:

Geolocalização (GPS):

Geolocalização refere-se à identificação da localização geográfica de um objeto, como um dispositivo móvel. Em dispositivos móveis, isso é geralmente feito usando uma combinação de tecnologias:

- **GPS (Global Positioning System):** Utiliza sinais de satélite para determinar a posição exata do dispositivo. É o método mais preciso, mas pode consumir mais bateria e requer uma visão clara do céu.
- **Wi-Fi:** Utiliza a localização de redes Wi-Fi conhecidas para estimar a posição, útil em ambientes internos ou urbanos densos.
- **Redes Celulares (Triangulação de Torres):** Estima a localização com base na proximidade das torres de celular. Menos preciso, mas funciona em áreas com cobertura de rede.

`expo-location`:

O `expo-location` é um módulo do Expo que fornece acesso à localização do dispositivo. Ele abstrai as diferenças entre as APIs de localização do iOS e Android, oferecendo uma interface consistente para os desenvolvedores. Com ele, é possível obter a localização única, monitorar a localização em tempo real e acessar informações sobre geocodificação (converter coordenadas em endereços e vice-versa) [62].

- **`requestForegroundPermissionsAsync()`:** Solicita permissão para acessar a localização em primeiro plano (quando o aplicativo está em uso ativo) [63].
- **`requestBackgroundPermissionsAsync()`:** Solicita permissão para acessar a localização em segundo plano (quando o aplicativo não está em uso ativo) [64].
- **`getCurrentPositionAsync(options)`:** Obtém a localização atual do dispositivo. As opções permitem configurar a precisão, o tempo limite e se deve usar o cache [65].

- **`watchPositionAsync(options, callback)`:** Inicia o monitoramento contínuo da localização do dispositivo, chamando um callback sempre que a localização muda. É útil para aplicativos de navegação ou rastreamento [66].

Permissões de Localização:

O acesso à localização é uma funcionalidade sensível à privacidade, e os sistemas operacionais móveis exigem que os aplicativos solicitem permissão explícita do usuário. Existem dois tipos principais de permissões de localização:

- **Permissão em Primeiro Plano (When In Use):** Permite que o aplicativo acesse a localização apenas quando está em uso ativo (visível na tela). É a permissão mais comum e geralmente suficiente para a maioria dos aplicativos [63].
- **Permissão em Segundo Plano (Always):** Permite que o aplicativo acesse a localização mesmo quando não está em uso ativo (em segundo plano). Essa permissão é mais restritiva e deve ser solicitada apenas quando estritamente necessário, pois consome mais bateria e levanta preocupações com a privacidade [64].

É crucial explicar ao usuário por que a permissão de localização é necessária e como os dados serão utilizados para garantir a transparência e a confiança.

3. Tecnologias e Ferramentas:

- **`expo-location`:** Módulo para acesso à geolocalização.
- **GPS, Wi-Fi, Redes Celulares:** Tecnologias subjacentes para determinação da localização.

4. Exemplos Práticos (Código):

Instalação do `expo-location`:

```
npx expo install expo-location
```

Configuração no `app.json`:

Para que o aplicativo solicite as permissões corretas, adicione as seguintes entradas na seção `plugins` ou `ios` e `android` do seu `app.json`:

```
{
  "expo": {
    // ... outras configurações
    "plugins": [
      [
        "expo-location",
        {
          "locationAlwaysAndWhenInUsePermission": "Permita que $(PRODUCT_NAME) acesse sua localização em primeiro e segundo plano para fornecer serviços baseados em localização."
        }
      ]
    ],
    "ios": {
      "infoPlist": {
        "NSLocationWhenInUseUsageDescription": "Este aplicativo precisa de acesso à sua localização para mostrar sua posição no mapa.",
        "NSLocationAlwaysAndWhenInUseUsageDescription": "Este aplicativo precisa de acesso à sua localização em segundo plano para rastrear sua rota mesmo quando o aplicativo não está em uso."
      }
    },
    "android": {
      "permissions": [
        "android.permission.ACCESS_COARSE_LOCATION",
        "android.permission.ACCESS_FINE_LOCATION",
        "android.permission.ACCESS_BACKGROUND_LOCATION" // Para localização em segundo plano
      ]
    }
  }
}
```

Exemplo de Obtenção da Localização Atual:

```
 import React, { useState, useEffect } from 'react';
import { View, Text, StyleSheet, Button, Alert } from 'react-native';
import * as Location from 'expo-location';

export default function LocationScreen() {
  const [location, setLocation] = useState(null);
  const [errorMsg, setErrorMsg] = useState(null);

  useEffect(() => {
    (async () => {
      let { status } = await Location.requestForegroundPermissionsAsync();
      if (status !== 'granted') {
        setErrorMsg('Permissão para acessar a localização foi negada');
        Alert.alert('Permissão Necessária', 'Para usar este recurso, por favor, conceda permissão de
localização nas configurações do seu dispositivo.');
        return;
      }

      let currentLocation = await Location.getCurrentPositionAsync({});
      setLocation(currentLocation);
    })();
  }, []);

  const getLocation = async () => {
    try {
      let { status } = await Location.requestForegroundPermissionsAsync();
      if (status !== 'granted') {
        setErrorMsg('Permissão para acessar a localização foi negada');
        Alert.alert('Permissão Necessária', 'Para usar este recurso, por favor, conceda permissão de
localização nas configurações do seu dispositivo.');
        return;
      }

      let currentLocation = await Location.getCurrentPositionAsync({});
      setLocation(currentLocation);
      setErrorMsg(null);
    } catch (error) {
      setErrorMsg('Erro ao obter a localização: ' + error.message);
      console.error(error);
    }
  };

  let text = 'Aguardando localização...';
  if (errorMsg) {
    text = errorMsg;
  } else if (location) {
    text = `Latitude: $`{location.coords.latitude}\nLongitude: `${location.coords.longitude}\nPrecisão:
${location.coords.accuracy}m`;
  }

  return (
    <View style={styles.container}>
      <Text style={styles.paragraph}>{text}</Text>
      <Button title="Obter Minha Localização" onPress={getLocation} />
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    alignItems: 'center',
    justifyContent: 'center',
    padding: 20,
  },
  paragraph: {
    fontSize: 18,
    textAlign: 'center',
    marginBottom: 20,
  },
});
```

5. Exercícios e Desafios:

1. Modifique o exemplo para usar `watchPositionAsync` e exibir a localização do usuário em tempo real. Adicione botões para iniciar e parar o monitoramento.

2. Implemente a funcionalidade de geocodificação reversa: dada a latitude e longitude, use `Location.reverseGeocodeAsync()` para obter o endereço e exibi-lo na tela.
3. Pesquise sobre `Location.geocodeAsync()` e crie um `TextInput` onde o usuário possa digitar um endereço e, ao clicar em um botão, o aplicativo exiba as coordenadas (latitude e longitude) desse endereço.

6. Recursos Adicionais:

- **Documentação Oficial do `expo-location`:** https://docs.expo.dev/versions/latest/sdk/location/
- **Guia de Permissões do Expo:** https://docs.expo.dev/versions/latest/sdk/permissions/

Referências:

[62] https://docs.expo.dev/versions/latest/sdk/location/ [63] https://docs.expo.dev/versions/latest/sdk/location/#locationrequestforegroundpermissionsasync [64] https://docs.expo.dev/versions/latest/sdk/location/#locationrequestbackgroundpermissionsasync [65] https://docs.expo.dev/versions/latest/sdk/location/#locationgetcurrentpositionasyncoptions [66] https://docs.expo.dev/versions/latest/sdk/location/#locationwatchpositionasyncoptions-callback

21/nov: Permissões em primeiro e segundo plano.

1. Visão Geral:

O acesso à localização do usuário é uma funcionalidade poderosa, mas que exige grande responsabilidade devido às implicações de privacidade. Os sistemas operacionais móveis modernos, como iOS e Android, implementaram mecanismos rigorosos de permissão para garantir que os usuários tenham controle sobre como e quando seus dados de localização são acessados. Compreender a diferença entre permissões de localização em primeiro plano (foreground) e em segundo plano (background) é crucial para desenvolver aplicativos que respeitem a privacidade do usuário e funcionem corretamente em diferentes cenários de uso. Este tópico aprofundará nesses dois tipos de permissões e suas implicações.

2. Conceitos Fundamentais:

Permissões de Localização:

As permissões de localização são autorizações que um aplicativo deve obter do usuário para acessar os serviços de localização do dispositivo. Essas permissões são categorizadas principalmente em dois tipos:

- **Permissão em Primeiro Plano (When In Use / Foreground Location):**
    - **Conceito:** Permite que o aplicativo acesse a localização do dispositivo *apenas quando o aplicativo está em uso ativo*, ou seja, quando está visível na tela ou quando o usuário está interagindo diretamente com ele. Se o aplicativo for para o segundo plano (e.g., o usuário muda para outro aplicativo ou bloqueia a tela), o acesso à localização é interrompido [67].
    - **Casos de Uso Comuns:** Aplicativos de mapas que mostram a posição atual do usuário enquanto ele está usando o aplicativo, aplicativos de transporte que rastreiam a localização do usuário durante uma viagem ativa, aplicativos de previsão do tempo que mostram a previsão para a localização atual.
    - **Solicitação:** Geralmente, é a permissão mais fácil de obter, pois o usuário entende o contexto imediato do uso da localização.
- **Permissão em Segundo Plano (Always / Background Location):**
    - **Conceito:** Permite que o aplicativo acesse a localização do dispositivo *mesmo quando o aplicativo não está em uso ativo*, ou seja, quando está em segundo plano, a tela está bloqueada ou o usuário está

usando outro aplicativo. Essa permissão é mais abrangente e, por isso, mais restritiva [68].

- **Casos de Uso Comuns:** Aplicativos de rastreamento de fitness que registram a rota de corrida mesmo com a tela bloqueada, aplicativos de segurança familiar que permitem o rastreamento de membros da família, aplicativos de geofencing que disparam ações quando o usuário entra ou sai de uma área definida.
- **Solicitação:** É mais difícil de obter, pois os usuários são mais cautelosos em conceder acesso contínuo à sua localização. Os sistemas operacionais exigem uma justificativa clara e um caso de uso legítimo para essa permissão.

Privacidade do Usuário:

A privacidade é uma preocupação crescente para os usuários de dispositivos móveis. Ao solicitar permissões de localização, é fundamental:

- **Transparência:** Explicar claramente ao usuário por que a permissão é necessária e como os dados de localização serão usados. Isso pode ser feito através de mensagens informativas antes da solicitação da permissão.
- **Minimizar o Acesso:** Solicitar apenas o nível de permissão necessário para a funcionalidade do aplicativo. Se o aplicativo só precisa da localização quando está em uso, não solicite a permissão em segundo plano.
- **Segurança dos Dados:** Garantir que os dados de localização coletados sejam armazenados e transmitidos de forma segura.

3. Tecnologias e Ferramentas:

- `expo-location`**:** Módulo para solicitar e verificar permissões de localização.
- `app.json`**:** Para declarar as descrições de uso das permissões no iOS e Android.

4. Exemplos Práticos (Código):

Solicitando Permissões de Localização:

O `expo-location` oferece métodos assíncronos para solicitar as permissões. É uma boa prática verificar o status da permissão antes de tentar obter a localização.

```
 import React, { useState, useEffect } from 'react';
import { View, Text, Button, StyleSheet, Alert } from 'react-native';
import * as Location from 'expo-location';

export default function LocationPermissionsScreen() {
  const [foregroundPermission, setForegroundPermission] = useState(null);
  const [backgroundPermission, setBackgroundPermission] = useState(null);

  useEffect(() => {
    (async () => {
      // Verifica e solicita permissão em primeiro plano
      let { status: foregroundStatus } = await Location.requestForegroundPermissionsAsync();
      setForegroundPermission(foregroundStatus === 'granted');

      // Verifica e solicita permissão em segundo plano (se necessário)
      // A permissão em segundo plano geralmente só é solicitada após a permissão em primeiro plano ser
concedida
      // e se houver um caso de uso claro para ela.
      let { status: backgroundStatus } = await Location.requestBackgroundPermissionsAsync();
      setBackgroundPermission(backgroundStatus === 'granted');
    })();
  }, []);

  const checkPermissions = async () => {
    let foreground = await Location.getForegroundPermissionsAsync();
    let background = await Location.getBackgroundPermissionsAsync();

    Alert.alert(
      'Status das Permissões',
      `Primeiro Plano: $`{foreground.status}\nSegundo Plano: `${background.status}`
    );
  };

  return (
    <View style={styles.container}>
      <Text style={styles.title}>Gerenciamento de Permissões de Localização</Text>
      <Text style={styles.statusText}>
        Permissão em Primeiro Plano: {foregroundPermission === true ? 'Concedida' : 'Negada'}
      </Text>
      <Text style={styles.statusText}>
        Permissão em Segundo Plano: {backgroundPermission === true ? 'Concedida' : 'Negada'}
      </Text>
      <Button title="Verificar Status das Permissões" onPress={checkPermissions} />

      {/* Botões para solicitar novamente, caso o usuário tenha negado */}
      {!foregroundPermission && (
        <Button
          title="Solicitar Permissão em Primeiro Plano"
          onPress={async () => {
            let { status } = await Location.requestForegroundPermissionsAsync();
            setForegroundPermission(status === 'granted');
            if (status !== 'granted') {
              Alert.alert('Permissão Negada', 'Por favor, conceda a permissão de localização em primeiro
plano nas configurações do aplicativo.');
            }
          }}
        />
      )}
      {!backgroundPermission && foregroundPermission && (
        <Button
          title="Solicitar Permissão em Segundo Plano"
          onPress={async () => {
            let { status } = await Location.requestBackgroundPermissionsAsync();
            setBackgroundPermission(status === 'granted');
            if (status !== 'granted') {
              Alert.alert('Permissão Negada', 'Por favor, conceda a permissão de localização em segundo
plano nas configurações do aplicativo.');
            }
          }}
        />
      )}
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    alignItems: 'center',
    justifyContent: 'center',
    padding: 20,
```

```
  },
  title: {
    fontSize: 22,
    fontWeight: 'bold',
    marginBottom: 20,
    textAlign: 'center',
  },
  statusText: {
    fontSize: 16,
    marginBottom: 10,
  },
});
```

5. Exercícios e Desafios:

1. Crie um cenário onde o aplicativo tenta obter a localização em segundo plano. Se a permissão não for concedida, exiba uma mensagem informativa ao usuário explicando por que essa permissão é importante para a funcionalidade específica.
2. Implemente uma lógica para que, se o usuário negar a permissão de localização (qualquer uma delas), o aplicativo o direcione para as configurações do dispositivo para que ele possa conceder a permissão manualmente. (Dica: use `Linking.openSettings()` do React Native).
3. Pesquise sobre as diretrizes de uso de localização em segundo plano para iOS e Android e discuta as melhores práticas para garantir a aprovação do aplicativo nas lojas.

6. Recursos Adicionais:

- **Documentação Oficial do `expo-location` - Permissões:** https://docs.expo.dev/versions/latest/sdk/location/#permissions
- **Apple Human Interface Guidelines - Location Usage:** https://developer.apple.com/design/human-interface-guidelines/technologies/location-services/
- **Android Developers - Location Permissions:** https://developer.android.com/training/location/permissions

Referências:

[67] https://docs.expo.dev/versions/latest/sdk/location/#locationrequestforegroundpermissionsasync [68] https://docs.expo.dev/versions/latest/sdk/location/#locationrequestbackgroundpermissionsasync

24/nov: Módulo react-native-maps, posicionamento no mapa markers e polyline.

1. Visão Geral:

Exibir mapas interativos e visualizar dados geográficos são funcionalidades essenciais para muitos aplicativos móveis modernos. O `react-native-maps` é uma biblioteca popular e poderosa que fornece componentes de mapa nativos para iOS (MapKit) e Android (Google Maps), permitindo uma integração profunda e performática. Com ele, é possível exibir mapas, posicionar marcadores (markers) em locais específicos e desenhar polylines para representar rotas ou áreas. Este tópico abordará a instalação, configuração e uso básico do `react-native-maps` para criar experiências de mapa ricas em seu aplicativo.

2. Conceitos Fundamentais:

`react-native-maps`:

O `react-native-maps` é um componente de mapa que abstrai as APIs nativas de mapa (MapKit no iOS e Google Maps no Android) para o React Native. Ele oferece uma API JavaScript unificada para controlar o mapa, adicionar marcadores, polylines, polígonos e muito mais. Isso permite que os desenvolvedores criem funcionalidades de mapa complexas com uma única base de código [69].

- **`MapView`:** O componente principal que renderiza o mapa. Ele aceita propriedades para configurar a região inicial, tipo de mapa (padrão, satélite, híbrido), zoom, etc. [70].
- **`Marker`:** Um componente usado para exibir um marcador em uma coordenada específica no mapa. Pode ser personalizado com ícones, títulos e descrições [71].
- **`Polyline`:** Um componente usado para desenhar uma linha conectando uma série de coordenadas no mapa. É ideal para representar rotas, caminhos ou limites geográficos [72].

Coordenadas Geográficas:

As localizações no mapa são representadas por coordenadas geográficas: latitude e longitude. A latitude mede a distância de um ponto ao norte ou sul do Equador, enquanto a longitude mede a distância a leste ou oeste do Meridiano de Greenwich.

3. Tecnologias e Ferramentas:

- **`react-native-maps`:** Biblioteca para exibir mapas.
- **Google Maps API Key (Android):** Para usar o Google Maps no Android, é necessário obter uma chave de API do Google Cloud Platform e configurá-la no projeto.
- **MapKit (iOS):** No iOS, o MapKit é usado por padrão e geralmente não requer uma chave de API, mas pode precisar de configurações no Xcode.

4. Exemplos Práticos (Código):

Instalação do `react-native-maps`:

```
npm install react-native-maps
```

Configuração (Resumo - detalhes podem variar):

- **Android:**
  1. Obtenha uma chave de API do Google Maps Platform.
  2. Adicione a chave ao seu `AndroidManifest.xml`: `xml <application> <meta-data android:name="com.google.android.geo.API_KEY" android:value="YOUR_GOOGLE_MAPS_API_KEY"/> </application>`
- **iOS:**
  1. No seu `Podfile`, adicione `pod 'GoogleMaps'` se quiser usar o Google Maps no iOS (caso contrário, MapKit é o padrão).
  2. Execute `pod install` na pasta `ios/`.

Exemplo de Mapa com Marcador e Polyline:

```
 import React, { useState, useEffect } from 'react';
import { View, StyleSheet, Alert } from 'react-native';
import MapView, { Marker, Polyline } from 'react-native-maps';
import * as Location from 'expo-location';

export default function MapScreen() {
  const [currentLocation, setCurrentLocation] = useState(null);
  const [initialRegion, setInitialRegion] = useState(null);

  useEffect(() => {
    (async () => {
      let { status } = await Location.requestForegroundPermissionsAsync();
      if (status !== 'granted') {
        Alert.alert('Permissão Negada', 'Para usar o mapa, por favor, conceda permissão de
localização.');
        return;
      }

      let location = await Location.getCurrentPositionAsync({});
      setCurrentLocation(location.coords);
      setInitialRegion({
        latitude: location.coords.latitude,
        longitude: location.coords.longitude,
        latitudeDelta: 0.0922,
        longitudeDelta: 0.0421,
      });
    })();
  }, []);

  const polylineCoordinates = [
    { latitude: -19.9167, longitude: -43.9345 }, // Belo Horizonte
    { latitude: -22.9068, longitude: -43.1729 }, // Rio de Janeiro
    { latitude: -23.5505, longitude: -46.6333 }  // São Paulo
  ];

  if (!initialRegion) {
    return <View style={styles.loadingContainer}><Text>Carregando mapa...</Text></View>;
  }

  return (
    <View style={styles.container}>
      <MapView
        style={styles.map}
        initialRegion={initialRegion}
        showsUserLocation={true} // Mostra a localização do usuário
        followsUserLocation={true} // Centraliza o mapa na localização do usuário
      >
        {currentLocation && (
          <Marker
            coordinate={{
              latitude: currentLocation.latitude,
              longitude: currentLocation.longitude,
            }}
            title="Minha Localização"
            description="Você está aqui!"
          />
        )}

        <Marker
          coordinate={{
            latitude: -19.9167,
            longitude: -43.9345,
          }}
          title="Belo Horizonte"
          description="Capital de Minas Gerais"
          pinColor="blue"
        />

        <Polyline
          coordinates={polylineCoordinates}
          strokeColor="#FF0000" // red
          fillColor="rgba(255,0,0,0.5)" // red with opacity
          strokeWidth={3}
        />
      </MapView>
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
```

```
    flex: 1,
  },
  map: {
    width: '100%',
    height: '100%',
  },
  loadingContainer: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
  },
});
```

5. Exercícios e Desafios:

1. Adicione múltiplos marcadores ao mapa, representando pontos de interesse (e.g., restaurantes, parques). Personalize os ícones dos marcadores.
2. Implemente uma funcionalidade para que, ao clicar em um marcador, um `Callout` (balão de informações) seja exibido com detalhes sobre o local.
3. Crie uma `Polyline` que conecte a localização atual do usuário a um destino fixo no mapa.
4. Pesquise sobre o componente `Polygon` do `react-native-maps` e crie um polígono para destacar uma área específica no mapa.

6. Recursos Adicionais:

- **Documentação Oficial do `react-native-maps`:** https://github.com/react-native-maps/react-native-maps
- **Google Maps Platform - Get API Key:** https://developers.google.com/maps/documentation/javascript/get-api-key

Referências:

[69] https://github.com/react-native-maps/react-native-maps [70] https://github.com/react-native-maps/react-native-maps/blob/master/docs/mapview.md [71] https://github.com/react-native-maps/react-native-maps/blob/master/docs/marker.md [72] https://github.com/react-native-maps/react-native-maps/blob/master/docs/polyline.md

28/nov: Ferramentas de rotas em mapas: GooglePlacesAutoComplete.

1. Visão Geral:

Em aplicativos que dependem de localização, permitir que os usuários busquem e selecionem endereços de forma eficiente é crucial. A funcionalidade de autocompletar, como a oferecida pelo Google Places Autocomplete, melhora significativamente a experiência do usuário ao sugerir endereços à medida que o usuário digita. O `react-native-google-places-autocomplete` é um componente React Native que integra facilmente essa funcionalidade, permitindo que você adicione um campo de busca de endereços poderoso ao seu aplicativo. Este tópico abordará como configurar e utilizar essa ferramenta para aprimorar a busca de locais no mapa.

2. Conceitos Fundamentais:

Google Places API:

A Google Places API é um serviço que fornece informações sobre lugares, incluindo endereços, coordenadas geográficas, tipos de lugar, avaliações e muito mais. A funcionalidade de Autocomplete é uma parte dessa API que retorna previsões de lugares em resposta a consultas de texto. Essas previsões podem ser nomes de lugares, endereços ou categorias de lugares [73].

`react-native-google-places-autocomplete`:

Este é um componente React Native que envolve a Google Places Autocomplete API, fornecendo uma interface de usuário pronta para uso para busca de endereços. Ele lida com a comunicação com a API, a exibição das sugestões e o retorno dos detalhes do lugar selecionado. É altamente personalizável e pode ser integrado facilmente com `react-native-maps` [74].

- **Chave de API:** Para usar a Google Places API, é necessário obter uma chave de API do Google Cloud Platform e habilitar a API "Places API" para o seu projeto.
- **Eventos:** O componente fornece callbacks como `onPress` que são acionados quando o usuário seleciona uma sugestão, retornando os detalhes do lugar, incluindo as coordenadas.

3. Tecnologias e Ferramentas:

- **`react-native-google-places-autocomplete`:** Componente para autocompletar endereços.
- **Google Places API:** Serviço de backend para busca de lugares.
- **Google Maps API Key:** Necessária para autenticar as requisições à Google Places API.

4. Exemplos Práticos (Código):

Instalação do `react-native-google-places-autocomplete`:

```
npm install react-native-google-places-autocomplete
```

Exemplo de Busca de Endereços com Autocomplete:

```
 import React, { useState } from 'react';
import { View, StyleSheet, Alert, Text } from 'react-native';
import { GooglePlacesAutocomplete } from 'react-native-google-places-autocomplete';
import MapView, { Marker } from 'react-native-maps';

// Substitua pela sua chave de API do Google Maps/Places
const GOOGLE_MAPS_APIKEY = 'YOUR_GOOGLE_MAPS_API_KEY';

export default function PlacesAutocompleteScreen() {
  const [selectedPlace, setSelectedPlace] = useState(null);
  const [mapRegion, setMapRegion] = useState({
    latitude: -23.5505,
    longitude: -46.6333,
    latitudeDelta: 0.0922,
    longitudeDelta: 0.0421,
  });

  return (
    <View style={styles.container}>
      <GooglePlacesAutocomplete
        placeholder=\'Buscar endereço\'
        onPress={(data, details = null) => {
          // 'details' é um objeto que contém informações mais detalhadas sobre o lugar
          setSelectedPlace({
            name: data.description,
            latitude: details.geometry.location.lat,
            longitude: details.geometry.location.lng,
          });
          setMapRegion({
            latitude: details.geometry.location.lat,
            longitude: details.geometry.location.lng,
            latitudeDelta: 0.005,
            longitudeDelta: 0.005,
          });
        }}
        query={{
          key: GOOGLE_MAPS_APIKEY,
          language: 'pt-BR',
        }}
        styles={{
          container: {
            position: 'absolute',
            width: '100%',
            zIndex: 1,
            paddingHorizontal: 10,
            paddingTop: 10,
          },
          textInput: {
            height: 50,
            color: '#5d5d5d',
            fontSize: 16,
            borderRadius: 8,
            borderWidth: 1,
            borderColor: '#ccc',
            paddingHorizontal: 15,
          },
          predefinedPlacesDescription: {
            color: '#1faadb',
          },
        }}
        fetchDetails={true} // Importante para obter latitude e longitude
        enablePoweredByContainer={false} // Opcional: remove o logo


do Google
        keyboardShouldPersistTaps="handled" // Para evitar que o teclado feche ao selecionar um item
      />

      <MapView
        style={styles.map}
        region={mapRegion}
        onRegionChangeComplete={setMapRegion} // Atualiza a região do mapa quando o usuário move
      >
        {selectedPlace && (
          <Marker
            coordinate={{
              latitude: selectedPlace.latitude,
              longitude: selectedPlace.longitude,
            }}
            title={selectedPlace.name}
          />
```

```
        )}
      </MapView>
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    paddingTop: 60, // Espaço para o autocomplete
  },
  map: {
    flex: 1,
  },
});
```

5. Exercícios e Desafios:

1. Integre o `GooglePlacesAutocomplete` com o `expo-location` para que o mapa seja inicializado na localização atual do usuário e a busca de lugares priorize resultados próximos.
2. Adicione um botão "Limpar Busca" que resete o campo de busca e remova o marcador do mapa.
3. Personalize a aparência do `GooglePlacesAutocomplete` (cores, fontes, bordas) para que ele se ajuste ao design do seu aplicativo.
4. Pesquise sobre os diferentes tipos de `query` que podem ser passados para o `GooglePlacesAutocomplete` (e.g., `types`, `components`) para restringir os resultados da busca.

6. Recursos Adicionais:

- **Documentação Oficial do `react-native-google-places-autocomplete`:** https://github.com/FaridSafi/react-native-google-places-autocomplete
- **Google Places API Documentation:** https://developers.google.com/maps/documentation/places/web-service/overview

Referências:

[73] https://developers.google.com/maps/documentation/places/web-service/overview [74] https://github.com/FaridSafi/react-native-google-places-autocomplete

01/dez: Ferramentas de rotas em mapas: MapViewDirections.

1. Visão Geral:

Além de exibir mapas e marcar locais, a capacidade de traçar rotas entre dois ou mais pontos é uma funcionalidade crucial para aplicativos de navegação, transporte e logística. O `react-native-maps-directions` (ou `MapViewDirections` como componente) é uma biblioteca que se integra perfeitamente com o `react-native-maps` para exibir rotas no mapa, utilizando a Google Directions API. Este tópico abordará como utilizar essa ferramenta para calcular e visualizar rotas, fornecendo uma experiência de navegação completa para o usuário.

2. Conceitos Fundamentais:

Google Directions API:

A Google Directions API é um serviço que calcula rotas entre locais usando uma matriz de origens e destinos. Ela retorna informações detalhadas sobre a rota, incluindo coordenadas dos pontos de passagem, distância, duração e instruções passo a passo. Essa API é a base para o cálculo e exibição de rotas em aplicativos de mapa [75].

`react-native-maps-directions` (Componente `MapViewDirections`):

Este componente é um wrapper para a Google Directions API, projetado para ser usado como um filho do `MapView` do `react-native-maps`. Ele simplifica o processo de traçar rotas no mapa, cuidando da comunicação com a API e da renderização da polyline da rota. Ele requer uma chave de API do Google Maps para funcionar [76].

- **`origin` e `destination`:** Propriedades que definem os pontos de partida e chegada da rota, respectivamente. Podem ser coordenadas (latitude, longitude) ou endereços.
- **`waypoints`:** Propriedade opcional para adicionar pontos intermediários à rota.
- **`apikey`:** Sua chave de API do Google Maps.
- **`strokeWidth` e `strokeColor`:** Propriedades para estilizar a linha da rota no mapa.
- **`onReady`:** Um callback que é acionado quando a rota é calculada com sucesso, fornecendo detalhes como distância e duração.

3. Tecnologias e Ferramentas:

- **`react-native-maps`:** Para exibir o mapa.
- **`react-native-maps-directions`:** Para calcular e traçar rotas.
- **Google Directions API:** Serviço de backend para cálculo de rotas.
- **Google Maps API Key:** Necessária para autenticar as requisições à Google Directions API.

4. Exemplos Práticos (Código):

Instalação do `react-native-maps-directions`:

```
npm install react-native-maps-directions
```

Exemplo de Traçado de Rota entre Dois Pontos:

```
 import React, { useState, useEffect } from 'react';
import { View, StyleSheet, Alert, Text } from 'react-native';
import MapView, { Marker } from 'react-native-maps';
import MapViewDirections from 'react-native-maps-directions';
import * as Location from 'expo-location';

// Substitua pela sua chave de API do Google Maps
const GOOGLE_MAPS_APIKEY = 'YOUR_GOOGLE_MAPS_API_KEY';

export default function MapDirectionsScreen() {
  const [origin, setOrigin] = useState(null);
  const [destination, setDestination] = useState({
    latitude: -23.561356,
    longitude: -46.656000, // Exemplo: Avenida Paulista, São Paulo
  });
  const [initialRegion, setInitialRegion] = useState(null);

  useEffect(() => {
    (async () => {
      let { status } = await Location.requestForegroundPermissionsAsync();
      if (status !== 'granted') {
        Alert.alert('Permissão Negada', 'Para usar o mapa, por favor, conceda permissão de
localização.');
        return;
      }

      let location = await Location.getCurrentPositionAsync({});
      setOrigin({
        latitude: location.coords.latitude,
        longitude: location.coords.longitude,
      });
      setInitialRegion({
        latitude: location.coords.latitude,
        longitude: location.coords.longitude,
        latitudeDelta: 0.0922,
        longitudeDelta: 0.0421,
      });
    })();
  }, []);

  if (!initialRegion || !origin) {
    return <View style={styles.loadingContainer}><Text>Carregando mapa e localização...</Text></View>;
  }

  return (
    <View style={styles.container}>
      <MapView
        style={styles.map}
        initialRegion={initialRegion}
        showsUserLocation={true}
        followsUserLocation={true}
      >
        {origin && (
          <Marker
            coordinate={origin}
            title="Origem"
            description="Sua localização atual"
            pinColor="green"
          />
        )}
        <Marker
          coordinate={destination}
          title="Destino"
          description="Avenida Paulista"
          pinColor="red"
        />

        {origin && destination && (
          <MapViewDirections
            origin={origin}
            destination={destination}
            apikey={GOOGLE_MAPS_APIKEY}
            strokeWidth={4}
            strokeColor="blue"
            optimizeWaypoints={true}
            onReady={result => {
              console.log(`Distância: ${result.distance} km`);
              console.log(`Duração: ${result.duration} minutos`);
              // Você pode ajustar o zoom do mapa para mostrar a rota completa
              // mapRef.current.fitToCoordinates(result.coordinates, { edgePadding: { top: 50, right: 50,
bottom: 50, left: 50 }, animated: true });
```

```
            }}
            onError={(errorMessage) => {
              console.error('Erro ao traçar rota:', errorMessage);
              Alert.alert('Erro', 'Não foi possível traçar a rota. Verifique sua chave de API ou os
pontos de origem/destino.');
            }}
          />
        )}
      </MapView>
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
  },
  map: {
    width: '100%',
    height: '100%',
  },
  loadingContainer: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
  },
});
```

5. Exercícios e Desafios:

1. Adicione um `TextInput` para que o usuário possa digitar o endereço de destino. Use o `GooglePlacesAutocomplete` (do tópico anterior) para ajudar na seleção do destino.
2. Implemente a funcionalidade de adicionar múltiplos `waypoints` (pontos de passagem) à rota. Permita que o usuário adicione e remova esses pontos dinamicamente.
3. Exiba a distância e a duração da rota calculada em um componente `Text` na tela.
4. Pesquise sobre as diferentes opções de `mode` (e.g., `DRIVING`, `WALKING`, `BICYCLING`) no `MapViewDirections` e adicione um seletor para o usuário escolher o modo de transporte.

6. Recursos Adicionais:

- **Documentação Oficial do `react-native-maps-directions`:** https://github.com/brentvatne/react-native-maps-directions
- **Google Directions API Documentation:** https://developers.google.com/maps/documentation/directions/overview

Referências:

[75] https://developers.google.com/maps/documentation/directions/overview [76] https://github.com/brentvatne/react-native-maps-directions

05/dez: Instalação e configuração de ferramentas de rotas junto às telas.

1. Visão Geral:

Nos tópicos anteriores, exploramos como exibir mapas, marcar locais e traçar rotas usando `react-native-maps` e `react-native-maps-directions`. O próximo passo é integrar essas funcionalidades de forma coesa nas telas do aplicativo, permitindo que o usuário interaja com a busca de endereços e a visualização de rotas de maneira intuitiva. Este tópico focará na combinação dessas ferramentas para criar uma experiência de usuário fluida, onde a seleção de um local ou o cálculo de uma rota se reflete dinamicamente no mapa.

2. Conceitos Fundamentais:

Integração de Componentes:

A chave para uma experiência de usuário integrada é a comunicação eficiente entre os diferentes componentes. No React Native, isso é geralmente alcançado através de:

- **Estado Compartilhado:** Utilizar o estado do componente pai (ou um contexto global) para armazenar informações que precisam ser acessadas por múltiplos componentes filhos. Por exemplo, a localização selecionada pelo `GooglePlacesAutocomplete` pode ser armazenada no estado e passada como `prop` para o `MapView` e `MapViewDirections`.
- **Callbacks:** Componentes filhos podem notificar o componente pai sobre eventos (e.g., seleção de um lugar) através de funções de callback passadas como `props`.
- **Refs:** Em alguns casos, pode ser necessário acessar diretamente métodos de componentes filhos (e.g., para ajustar o zoom do mapa após o cálculo de uma rota). Isso pode ser feito usando `refs`.

Fluxo de Interação Típico:

Um fluxo comum para um aplicativo de rotas seria:

1. **Entrada do Usuário:** O usuário digita um endereço em um `TextInput` (integrado com `GooglePlacesAutocomplete`).
2. **Seleção de Local:** O usuário seleciona uma sugestão do autocomplete. Os detalhes do local (latitude, longitude) são obtidos.
3. **Atualização do Mapa:** O mapa é centralizado no local selecionado, e um marcador é adicionado.
4. **Cálculo de Rota:** Se um ponto de origem e destino estiverem definidos, a `MapViewDirections` calcula e exibe a rota.
5. **Exibição de Detalhes da Rota:** A distância e a duração da rota são exibidas ao usuário.

3. Tecnologias e Ferramentas:

- **`react-native-maps`:** Para exibir o mapa.
- **`react-native-google-places-autocomplete`:** Para a busca de endereços com autocompletar.
- **`react-native-maps-directions`:** Para o cálculo e exibição de rotas.
- **React Hooks (`useState`, `useEffect`, `useRef`):** Para gerenciar o estado e interações entre componentes.

4. Exemplos Práticos (Código):

Vamos criar uma tela que combina a busca de lugares com o traçado de rotas.

`src/screens/RoutePlannerScreen.js`:

```
 import React, { useState, useEffect, useRef } from 'react';
import { View, StyleSheet, Alert, Text, ActivityIndicator } from 'react-native';
import MapView, { Marker } from 'react-native-maps';
import MapViewDirections from 'react-native-maps-directions';
import { GooglePlacesAutocomplete } from 'react-native-google-places-autocomplete';
import * as Location from 'expo-location';

// Substitua pela sua chave de API do Google Maps/Places
const GOOGLE_MAPS_APIKEY = 'YOUR_GOOGLE_MAPS_API_KEY';

export default function RoutePlannerScreen() {
  const [origin, setOrigin] = useState(null);
  const [destination, setDestination] = useState(null);
  const [initialRegion, setInitialRegion] = useState(null);
  const [routeInfo, setRouteInfo] = useState(null);
  const [isLoadingLocation, setIsLoadingLocation] = useState(true);
  const mapRef = useRef(null);

  useEffect(() => {
    (async () => {
      let { status } = await Location.requestForegroundPermissionsAsync();
      if (status !== 'granted') {
        Alert.alert('Permissão Negada', 'Para usar o mapa, por favor, conceda permissão de
localização.');
        setIsLoadingLocation(false);
        return;
      }

      let location = await Location.getCurrentPositionAsync({});
      setOrigin({
        latitude: location.coords.latitude,
        longitude: location.coords.longitude,
      });
      setInitialRegion({
        latitude: location.coords.latitude,
        longitude: location.coords.longitude,
        latitudeDelta: 0.0922,
        longitudeDelta: 0.0421,
      });
      setIsLoadingLocation(false);
    })();
  }, []);

  const handlePlaceSelected = (data, details, type) => {
    if (details) {
      const newLocation = {
        latitude: details.geometry.location.lat,
        longitude: details.geometry.location.lng,
      };
      if (type === 'origin') {
        setOrigin(newLocation);
      } else {
        setDestination(newLocation);
      }
      // Ajusta o mapa para mostrar o novo ponto
      mapRef.current.animateToRegion({
        latitude: newLocation.latitude,
        longitude: newLocation.longitude,
        latitudeDelta: 0.005,
        longitudeDelta: 0.005,
      }, 1000);
    }
  };

  if (isLoadingLocation || !initialRegion) {
    return (
      <View style={styles.loadingContainer}>
        <ActivityIndicator size="large" color="#0000ff" />
        <Text>Carregando localização...</Text>
      </View>
    );
  }

  return (
    <View style={styles.container}>
      <View style={styles.autocompleteContainer}>
        <GooglePlacesAutocomplete
          placeholder='Origem (sua localização atual)'
          onPress={(data, details) => handlePlaceSelected(data, details, 'origin')}
          query={{
            key: GOOGLE_MAPS_APIKEY,
```

```
            language: 'pt-BR',
          }}
          fetchDetails={true}
          enablePoweredByContainer={false}
          styles={autocompleteStyles}
          textInputProps={{
            value: origin ? 'Minha Localização Atual' : '', // Exibe texto fixo se a origem for a
localização atual
            editable: false, // Impede edição direta se for a localização atual
          }}
        />
        <GooglePlacesAutocomplete
          placeholder='Destino'
          onPress={(data, details) => handlePlaceSelected(data, details, 'destination')}
          query={{
            key: GOOGLE_MAPS_APIKEY,
            language: 'pt-BR',
          }}
          fetchDetails={true}
          enablePoweredByContainer={false}
          styles={autocompleteStyles}
        />
      </View>

      <MapView
        ref={mapRef}
        style={styles.map}
        initialRegion={initialRegion}
        showsUserLocation={true}
        followsUserLocation={true}
      >
        {origin && (
          <Marker
            coordinate={origin}
            title="Origem"
            description="Ponto de partida"
            pinColor="green"
          />
        )}
        {destination && (
          <Marker
            coordinate={destination}
            title="Destino"
            description="Ponto de chegada"
            pinColor="red"
          />
        )}

        {origin && destination && (
          <MapViewDirections
            origin={origin}
            destination={destination}
            apikey={GOOGLE_MAPS_APIKEY}
            strokeWidth={4}
            strokeColor="blue"
            optimizeWaypoints={true}
            onReady={result => {
              setRouteInfo(result);
              mapRef.current.fitToCoordinates(result.coordinates, {
                edgePadding: { top: 50, right: 50, bottom: 50, left: 50 },
                animated: true,
              });
            }}
            onError={(errorMessage) => {
              console.error('Erro ao traçar rota:', errorMessage);
              Alert.alert('Erro', 'Não foi possível traçar a rota. Verifique sua chave de API ou os
pontos de origem/destino.');
              setRouteInfo(null);
            }}
          />
        )}
      </MapView>

      {routeInfo && (
        <View style={styles.routeInfoContainer}>
          <Text style={styles.routeInfoText}>Distância: {routeInfo.distance.toFixed(2)} km</Text>
          <Text style={styles.routeInfoText}>Duração: {routeInfo.duration.toFixed(2)} min</Text>
        </View>
      )}
    </View>
  );
}
```

```
const autocompleteStyles = StyleSheet.create({
  container: {
    flex: 0,
    position: 'relative',
    width: '100%',
    zIndex: 1,
    marginBottom: 10,
  },
  textInput: {
    height: 40,
    color: '#5d5d5d',
    fontSize: 14,
    borderRadius: 5,
    borderWidth: 1,
    borderColor: '#ccc',
    paddingHorizontal: 10,
  },
  predefinedPlacesDescription: {
    color: '#1faadb',
  },
});

const styles = StyleSheet.create({
  container: {
    flex: 1,
    paddingTop: 50, // Ajuste para o status bar
  },
  autocompleteContainer: {
    paddingHorizontal: 10,
    marginBottom: 10,
  },
  map: {
    flex: 1,
  },
  loadingContainer: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
  },
  routeInfoContainer: {
    position: 'absolute',
    bottom: 20,
    left: 20,
    right: 20,
    backgroundColor: 'rgba(255,255,255,0.9)',
    padding: 15,
    borderRadius: 10,
    alignItems: 'center',
    shadowColor: '#000',
    shadowOffset: { width: 0, height: 2 },
    shadowOpacity: 0.25,
    shadowRadius: 3.84,
    elevation: 5,
  },
  routeInfoText: {
    fontSize: 16,
    fontWeight: 'bold',
    color: '#333',
  },
});
```

5. Exercícios e Desafios:

1. Adicione um botão "Limpar Rota" que resete os estados de `origin`, `destination` e `routeInfo`, limpando o mapa.
2. Implemente a funcionalidade de permitir que o usuário selecione a origem e o destino clicando diretamente no mapa, além de usar o autocomplete.
3. Adicione um seletor para o usuário escolher o modo de transporte (`DRIVING`, `WALKING`, `BICYCLING`) e observe como a rota e as informações de distância/duração mudam.
4. Pesquise sobre como exibir as instruções passo a passo da rota (`result.legs[0].steps`) em uma lista abaixo do mapa.

6. Recursos Adicionais:

- **React Native Maps Directions - Example:** https://github.com/brentvatne/react-native-maps-directions#example
- **Google Places Autocomplete - Customization:** https://github.com/FaridSafi/react-native-google-places-autocomplete#customization

Referências:

[77] https://github.com/brentvatne/react-native-maps-directions#example [78] https://github.com/FaridSafi/react-native-google-places-autocomplete#customization

08/dez: Integração dos dados de Geolocalização com o banco de dados SQLite.

1. Visão Geral:

A capacidade de coletar dados de geolocalização é poderosa, mas para que esses dados sejam úteis a longo prazo, eles precisam ser armazenados e gerenciados de forma eficiente. A integração dos dados de geolocalização (obtidos via `expo-location`) com um banco de dados local (SQLite, via `expo-sqlite`) permite que o aplicativo persista informações de localização, como histórico de rotas, pontos de interesse salvos ou registros de visitas. Este tópico abordará como armazenar e recuperar dados de geolocalização no SQLite, combinando os conhecimentos adquiridos nos módulos anteriores.

2. Conceitos Fundamentais:

Modelagem de Dados de Geolocalização:

Ao armazenar dados de geolocalização em um banco de dados relacional como o SQLite, é importante considerar a estrutura da tabela. Uma abordagem comum é criar uma tabela para armazenar os pontos de localização, com colunas para latitude, longitude, timestamp e, opcionalmente, informações adicionais como precisão, altitude ou um ID de sessão para agrupar pontos de uma mesma rota [79].

- **Exemplo de Estrutura de Tabela:** `sql CREATE TABLE locations ( id INTEGER PRIMARY KEY AUTOINCREMENT, latitude REAL NOT NULL, longitude REAL NOT NULL, timestamp INTEGER NOT NULL, -- Unix timestamp accuracy REAL, session_id TEXT -- Para agrupar pontos de uma mesma sessão de rastreamento );`

Persistência de Dados de Localização:

O processo de persistir dados de localização envolve:

1. **Obtenção da Localização:** Utilizar `expo-location` para obter a localização atual ou monitorar mudanças de localização.
2. **Preparação dos Dados:** Extrair a latitude, longitude, timestamp e outras informações relevantes do objeto de localização.
3. **Inserção no Banco de Dados:** Utilizar o `DatabaseHelper` (ou funções diretas do `expo-sqlite`) para inserir esses dados na tabela `locations`.

Recuperação e Exibição de Dados de Localização:

Uma vez que os dados de localização estão armazenados, eles podem ser recuperados para diversas finalidades:

- **Exibição em Mapa:** Recuperar uma série de pontos de localização e usá-los para desenhar uma `Polyline` no `react-native-maps`, visualizando uma rota percorrida.
- **Histórico:** Exibir uma lista de locais visitados ou rotas salvas.
- **Análise:** Realizar consultas para analisar padrões de movimento ou tempo gasto em determinadas áreas.

3. Tecnologias e Ferramentas:

- **`expo-location`:** Para obter os dados de geolocalização.
- **`expo-sqlite`:** Para interagir com o banco de dados SQLite.
- **`DatabaseHelper`:** O helper customizado para encapsular as operações de banco de dados.
- **`react-native-maps`:** Para exibir os dados de localização no mapa.

4. Exemplos Práticos (Código):

Vamos estender nosso `DatabaseHelper` para incluir operações para a tabela `locations` e, em seguida, criar uma tela que rastreie e exiba a rota no mapa.

`src/database/DatabaseHelper.js` (Adicionar funções para locations):

```
// ... (código existente do DatabaseHelper)

  init: () => {
    return new Promise((resolve, reject) => {
      db.transaction(
        tx => {
          tx.executeSql(
            // Tabela de usuários (já existente)
            'CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMARY KEY AUTOINCREMENT, name TEXT NOT NULL,
email TEXT UNIQUE NOT NULL);',
            [],
            () => console.log('Tabela users criada ou já existe.')
          );
          tx.executeSql(
            // Nova tabela para localizações
            'CREATE TABLE IF NOT EXISTS locations (id INTEGER PRIMARY KEY AUTOINCREMENT, latitude REAL
NOT NULL, longitude REAL NOT NULL, timestamp INTEGER NOT NULL, accuracy REAL, session_id TEXT);',
            [],
            () => console.log('Tabela locations criada ou já existe.'),
            (_, error) => {
              console.error('Erro ao criar tabela locations:', error);
              reject(error);
            }
          );
        },
        error => {
          console.error('Erro na transação de inicialização:', error);
          reject(error);
        },
        () => resolve('Transação de inicialização concluída.')
      );
    });
  },

  insertLocation: (latitude, longitude, timestamp, accuracy, sessionId) => {
    return new Promise((resolve, reject) => {
      db.transaction(
        tx => {
          tx.executeSql(
            'INSERT INTO locations (latitude, longitude, timestamp, accuracy, session_id) VALUES (?, ?,
?, ?, ?);',
            [latitude, longitude, timestamp, accuracy, sessionId],
            (_, result) => {
              console.log(`Localização inserida com ID: ${result.insertId}`);
              resolve(result.insertId);
            },
            (_, error) => {
              console.error('Erro ao inserir localização:', error);
              reject(error);
            }
          );
        },
        error => {
          console.error('Erro na transação de inserção de localização:', error);
          reject(error);
        }
      );
    });
  },

  getLocationsBySessionId: (sessionId) => {
    return new Promise((resolve, reject) => {
      db.transaction(
        tx => {
          tx.executeSql(
            'SELECT * FROM locations WHERE session_id = ? ORDER BY timestamp ASC;',
            [sessionId],
            (_, { rows }) => {
              console.log(`Localizações para sessão ${sessionId}:`, rows._array);
              resolve(rows._array);
            },
            (_, error) => {
              console.error(`Erro ao buscar localizações para sessão ${sessionId}:`, error);
              reject(error);
            }
          );
        },
        error => {
          console.error('Erro na transação de busca de localização por sessão:', error);
          reject(error);
        }
```

```
      );
    });
  },

  getAllLocations: () => {
    return new Promise((resolve, reject) => {
      db.transaction(
        tx => {
          tx.executeSql(
            'SELECT * FROM locations ORDER BY timestamp ASC;',
            [],
            (_, { rows }) => {
              console.log('Todas as localizações:', rows._array);
              resolve(rows._array);
            },
            (_, error) => {
              console.error('Erro ao buscar todas as localizações:', error);
              reject(error);
            }
          );
        },
        error => {
          console.error('Erro na transação de busca de todas as localizações:', error);
          reject(error);
        }
      );
    });
  },

  // ... (outras funções existentes)
};

export default DatabaseHelper;
```

src/screens/LocationTrackerScreen.js:

```
 import React, { useState, useEffect, useRef } from 'react';
import { View, Text, StyleSheet, Button, Alert, ActivityIndicator } from 'react-native';
import MapView, { Polyline, Marker } from 'react-native-maps';
import * as Location from 'expo-location';
import DatabaseHelper from '../database/DatabaseHelper';
import { v4 as uuidv4 } from 'uuid'; // Para gerar IDs de sessão únicos

// Instale uuid: npm install uuid
// Para usar uuidv4, importe como: import { v4 as uuidv4 } from 'uuid';
// E para React Native, pode precisar de um polyfill: import 'react-native-get-random-values';
// npm install react-native-get-random-values

export default function LocationTrackerScreen() {
  const [isTracking, setIsTracking] = useState(false);
  const [currentLocation, setCurrentLocation] = useState(null);
  const [trackedLocations, setTrackedLocations] = useState([]);
  const [sessionId, setSessionId] = useState(null);
  const [mapRegion, setMapRegion] = useState(null);
  const locationSubscription = useRef(null);

  useEffect(() => {
    const setup = async () => {
      try {
        await DatabaseHelper.init();
      } catch (error) {
        Alert.alert('Erro', 'Falha ao inicializar o banco de dados.');
      }
    };
    setup();

    return () => {
      if (locationSubscription.current) {
        locationSubscription.current.remove();
      }
    };
  }, []);

  const startTracking = async () => {
    let { status } = await Location.requestForegroundPermissionsAsync();
    if (status !== 'granted') {
      Alert.alert('Permissão Negada', 'Para rastrear a localização, por favor, conceda permissão.');
      return;
    }

    // Opcional: solicitar permissão em segundo plano se o rastreamento for contínuo
    // let { status: backgroundStatus } = await Location.requestBackgroundPermissionsAsync();
    // if (backgroundStatus !== 'granted') {
    //   Alert.alert('Permissão Necessária', 'Para rastreamento em segundo plano, conceda a permissão



// de localização em segundo plano.
    //   return;
    // }

    const newSessionId = uuidv4();
    setSessionId(newSessionId);
    setTrackedLocations([]);
    setIsTracking(true);

    locationSubscription.current = await Location.watchPositionAsync(
      {
        accuracy: Location.Accuracy.High,
        timeInterval: 1000, // Atualiza a cada 1 segundo
        distanceInterval: 1, // Atualiza a cada 1 metro
      },
      async (loc) => {
        const newLocation = {
          latitude: loc.coords.latitude,
          longitude: loc.coords.longitude,
          timestamp: loc.timestamp,
          accuracy: loc.coords.accuracy,
          sessionId: newSessionId,
        };
        setTrackedLocations((prevLocations) => [...prevLocations, newLocation]);
        setCurrentLocation(newLocation);
        setMapRegion({
          latitude: newLocation.latitude,
          longitude: newLocation.longitude,
          latitudeDelta: 0.005,
          longitudeDelta: 0.005,
```

```
        });
        try {
          await DatabaseHelper.insertLocation(
            newLocation.latitude,
            newLocation.longitude,
            newLocation.timestamp,
            newLocation.accuracy,
            newLocation.sessionId
          );
        } catch (error) {
          console.error("Erro ao salvar localização no DB:", error);
        }
      }
    );
  };

  const stopTracking = () => {
    if (locationSubscription.current) {
      locationSubscription.current.remove();
      locationSubscription.current = null;
    }
    setIsTracking(false);
    Alert.alert("Rastreamento Parado", `Sessão de rastreamento ${sessionId} finalizada.`);
  };

  const loadLastSession = async () => {
    try {
      const allLocations = await DatabaseHelper.getAllLocations();
      if (allLocations.length > 0) {
        const lastSession = allLocations[allLocations.length - 1].session_id;
        const loadedLocations = await DatabaseHelper.getLocationsBySessionId(lastSession);
        setTrackedLocations(loadedLocations);
        if (loadedLocations.length > 0) {
          const firstLoc = loadedLocations[0];
          setMapRegion({
            latitude: firstLoc.latitude,
            longitude: firstLoc.longitude,
            latitudeDelta: 0.005,
            longitudeDelta: 0.005,
          });
        }
        Alert.alert("Sessão Carregada", `Última sessão ($`{lastSession}) carregada com
`${loadedLocations.length} pontos.`);
      } else {
        Alert.alert("Nenhuma Sessão", "Nenhuma sessão de rastreamento encontrada no banco de dados.");
      }
    } catch (error) {
      Alert.alert("Erro", "Falha ao carregar a última sessão.");
      console.error(error);
    }
  };

  return (
    <View style={styles.container}>
      <Text style={styles.title}>Rastreamento de Localização</Text>
      <View style={styles.buttonGroup}>
        <Button
          title={isTracking ? "Parar Rastreamento" : "Iniciar Rastreamento"}
          onPress={isTracking ? stopTracking : startTracking}
          color={isTracking ? "red" : "green"}
        />
        <Button title="Carregar Última Sessão" onPress={loadLastSession} disabled={isTracking} />
      </View>

      {mapRegion ? (
        <MapView
          style={styles.map}
          region={mapRegion}
          showsUserLocation={true}
          followsUserLocation={isTracking} // Segue o usuário apenas se estiver rastreando
        >
          {trackedLocations.length > 0 && (
            <Polyline
              coordinates={trackedLocations.map((loc) => ({
                latitude: loc.latitude,
                longitude: loc.longitude,
              }))}
              strokeWidth={5}
              strokeColor="blue"
            />
          )}
          {currentLocation && !isTracking && (
```

```
            <Marker
              coordinate={{
                latitude: currentLocation.latitude,
                longitude: currentLocation.longitude,
              }}
              title="Última Posição"
              pinColor="purple"
            />
          )}
        </MapView>
      ) : (
        <View style={styles.loadingMapContainer}>
          <ActivityIndicator size="large" color="#0000ff" />
          <Text>Preparando mapa...</Text>
        </View>
      )}

      {currentLocation && (
        <View style={styles.locationInfo}>
          <Text>Lat: {currentLocation.latitude.toFixed(6)}</Text>
          <Text>Lon: {currentLocation.longitude.toFixed(6)}</Text>
          <Text>Precisão: {currentLocation.accuracy?.toFixed(2) || 0}m</Text>
          <Text>Pontos Rastreados: {trackedLocations.length}</Text>
        </View>
      )}
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    padding: 10,
    backgroundColor: '#f5f5f5',
  },
  title: {
    fontSize: 24,
    fontWeight: 'bold',
    marginBottom: 15,
    textAlign: 'center',
  },
  buttonGroup: {
    flexDirection: 'row',
    justifyContent: 'space-around',
    marginBottom: 15,
  },
  map: {
    flex: 1,
    borderRadius: 10,
  },
  loadingMapContainer: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#e0e0e0',
    borderRadius: 10,
  },
  locationInfo: {
    marginTop: 15,
    padding: 10,
    backgroundColor: '#fff',
    borderRadius: 8,
    borderWidth: 1,
    borderColor: '#ddd',
  },
});
```

5. Exercícios e Desafios:

1. Implemente uma funcionalidade para visualizar todas as sessões de rastreamento salvas no banco de dados. Crie uma lista de sessões e permita que o usuário selecione uma para carregar e exibir no mapa.
2. Adicione um botão para exportar os dados de uma sessão de rastreamento para um arquivo CSV ou JSON, que possa ser compartilhado ou analisado externamente.
3. Pesquise sobre `Location.startLocationUpdatesAsync` e `Location.stopLocationUpdatesAsync` para rastreamento em segundo plano e integre-os ao exemplo, garantindo que as permissões adequadas sejam

solicitadas.

4. Desenvolva uma funcionalidade de geofencing: permita que o usuário defina uma área no mapa e receba uma notificação quando o dispositivo entrar ou sair dessa área.

6. Recursos Adicionais:

- **Documentação Oficial do `expo-location`:** https://docs.expo.dev/versions/latest/sdk/location/
- **Documentação Oficial do `expo-sqlite`:** https://docs.expo.dev/versions/latest/sdk/sqlite/
- **`react-native-maps`:** https://github.com/react-native-maps/react-native-maps
- **`uuid` (para IDs únicos):** https://www.npmjs.com/package/uuid

Referências:

[79] https://www.sqlite.org/datatype3.html

Conclusão e Projeto Final

12/dez e 15/dez: Finalização do projeto da disciplina.

1. Visão Geral:

Ao longo deste curso, você explorou os fundamentos e as ferramentas avançadas para o desenvolvimento de aplicações móveis com React Native e Expo. Desde a configuração do ambiente e a construção de interfaces de usuário até a integração com Web Services, gerenciamento de dados locais e utilização de recursos nativos como câmera e geolocalização, você adquiriu um conjunto robusto de habilidades. A fase final da disciplina é dedicada à consolidação desse conhecimento através da finalização de um projeto prático. Este projeto é a sua oportunidade de aplicar tudo o que aprendeu, integrar diferentes funcionalidades e criar um aplicativo completo e funcional.

2. Conceitos Fundamentais:

Ciclo de Desenvolvimento de Software:

O projeto final simula um ciclo de desenvolvimento de software real, que geralmente inclui as seguintes etapas:

- **Planejamento e Requisitos:** Definir claramente o escopo, as funcionalidades e os requisitos do aplicativo.
- **Design:** Projetar a interface do usuário (UI) e a experiência do usuário (UX), incluindo wireframes e mockups.
- **Desenvolvimento:** Implementar as funcionalidades usando as tecnologias e padrões aprendidos.
- **Testes:** Garantir a qualidade e o funcionamento correto do aplicativo através de testes unitários, de integração e de UI.
- **Depuração:** Identificar e corrigir erros no código.
- **Implantação:** Preparar o aplicativo para distribuição (e.g., via EAS Build para as lojas de aplicativos).

Integração de Funcionalidades:

O desafio do projeto final reside na integração harmoniosa das diversas funcionalidades abordadas. Por exemplo, um aplicativo de lista de tarefas pode:

- Usar `expo-camera` para adicionar fotos às tarefas.
- Utilizar `expo-location` para marcar a localização onde a tarefa foi criada.

- Armazenar as tarefas em um banco de dados SQLite local.
- Sincronizar tarefas com um backend via Axios.
- Permitir que o usuário visualize tarefas em um mapa (`react-native-maps`).

Boas Práticas de Engenharia de Software:

Durante o desenvolvimento do projeto, é fundamental aplicar as boas práticas de engenharia de software:

- **Código Limpo e Legível:** Escrever código bem organizado, com nomes de variáveis e funções claros, e comentários quando necessário.
- **Modularização:** Dividir o aplicativo em módulos menores e independentes para facilitar a manutenção e o reuso.
- **Controle de Versão:** Utilizar Git para gerenciar o histórico do código e colaborar com outros desenvolvedores (se for um projeto em equipe).
- **Tratamento de Erros:** Implementar um tratamento de erros robusto para lidar com exceções e falhas de forma graciosa.
- **Otimização de Performance:** Considerar o desempenho do aplicativo, especialmente em dispositivos móveis com recursos limitados.

3. Tecnologias e Ferramentas:

Todas as tecnologias e ferramentas abordadas nos módulos anteriores serão relevantes para o projeto final, incluindo:

- **Expo e React Native:** A base do desenvolvimento.
- **React Navigation:** Para navegação entre telas.
- **Jest e React Native Testing Library:** Para testes.
- **Axios:** Para comunicação com APIs.
- **Context API / Gerenciamento de Estado:** Para gerenciar o estado da aplicação.
- **`expo-secure-store`:** Para armazenamento seguro.
- **`expo-camera`, `expo-image-picker`:** Para recursos de multimídia.
- **`expo-sqlite`** e **`DatabaseHelper`:** Para banco de dados local.
- **`expo-location`, `react-native-maps`, `react-native-google-places-autocomplete`, `react-native-maps-directions`:** Para geolocalização e mapas.

4. Exemplos Práticos (Código):

O projeto final não terá um único exemplo de código, mas sim a integração de todos os exemplos e conceitos aprendidos. Considere o seguinte como um ponto de partida para a estrutura do seu projeto:

```
MeuProjetoFinal/
├── App.js
├── app.json
├── package.json
├── babel.config.js
├── node_modules/
└── src/
    ├── assets/
    ├── components/
    │   ├── common/
    │   │   └── CustomButton.js
    │   └── ui/
    │       └── ProductCard.js
    ├── context/
    │   └── AuthContext.js
    ├── core/
    │   ├── entities/
    │   │   └── User.js
    │   ├── ports/
    │   │   └── UserRepository.js
    │   ├── use-cases/
    │   │   └── AddUser.js
    │   └── value-objects/
    │       └── Email.js
    ├── database/
    │   └── DatabaseHelper.js
    ├── navigation/
    │   └── AppNavigator.js
    ├── screens/
    │   ├── Auth/
    │   │   └── LoginScreen.js
    │   │   └── RegisterScreen.js
    │   ├── Main/
    │   │   ├── HomeScreen.js
    │   │   ├── ProductListScreen.js
    │   │   ├── ProductDetailScreen.js
    │   │   ├── CameraScreen.js
    │   │   ├── ImagePickerScreen.js
    │   │   ├── UserManagementScreen.js
    │   │   ├── LocationScreen.js
    │   │   ├── RoutePlannerScreen.js
    │   │   └── LocationTrackerScreen.js
    ├── services/
    │   └── authService.js
    └── utils/
        └── helpers.js
```

5. Exercícios e Desafios (Projeto Final):

Escolha um dos temas abaixo ou proponha um novo, e desenvolva um aplicativo completo, aplicando o máximo de conceitos e tecnologias aprendidas:

1. **Aplicativo de Gerenciamento de Tarefas (Todo App) Avançado:**
    - Permitir adicionar, editar, excluir e marcar tarefas como concluídas.
    - Adicionar fotos às tarefas (usando `expo-image-picker` ou `expo-camera`).
    - Marcar a localização onde a tarefa foi criada (`expo-location`).
    - Armazenar tarefas no SQLite local.
    - Opcional: Sincronizar tarefas com um backend (simulado ou real) via Axios.
    - Opcional: Exibir tarefas em um mapa se tiverem localização.
2. **Aplicativo de Diário de Viagem:**
    - Permitir que o usuário crie entradas de diário para diferentes viagens.
    - Cada entrada pode incluir texto, fotos (`expo-image-picker`), e a localização atual (`expo-location`).
    - Exibir as viagens e os pontos visitados em um mapa (`react-native-maps`).

    - Armazenar os dados no SQLite.
    - Opcional: Funcionalidade de busca de lugares (`GooglePlacesAutocomplete`).
3. **Aplicativo de Catálogo de Produtos Local:**
    - Gerenciar um catálogo de produtos (nome, descrição, preço, foto).
    - Armazenar os produtos no SQLite.
    - Permitir adicionar fotos aos produtos (`expo-image-picker`).
    - Implementar funcionalidades CRUD completas para os produtos.
    - Opcional: Tela de detalhes do produto com visualização em mapa se o produto tiver uma localização associada.

Requisitos para o Projeto Final:

- O aplicativo deve ser funcional e demonstrar a integração de pelo menos 3-4 módulos diferentes abordados no curso.
- O código deve ser bem organizado, legível e seguir as boas práticas de desenvolvimento.
- Incluir testes unitários para a lógica de negócios (core) e testes de componente para a UI (com Jest e React Native Testing Library).
- Fornecer um README.md detalhado explicando como configurar e executar o projeto, as funcionalidades implementadas e as tecnologias utilizadas.

6. Recursos Adicionais:

- **Documentação do Expo:** https://docs.expo.dev/
- **Documentação do React Native:** https://reactnative.dev/
- **Artigos sobre Clean Architecture e DDD:** Revise os recursos dos módulos 3 e 4.
- **GitHub:** Explore projetos de código aberto para inspiração e melhores práticas.

Este projeto é a culminação do seu aprendizado. Dedique-se, explore e divirta-se construindo algo significativo!