

Guia Passo a Passo: Manipulando galeria de imagens e contatos em React Native

1. Pré-requisitos

Antes de começar, certifique-se de ter:

- **Node.js** instalado. [Baixar Node.js](#)
- **Expo CLI** instalado globalmente. Você pode instalar o Expo CLI usando o seguinte comando:

```
npm install -g expo-cli
```

- **Ambiente de desenvolvimento** configurado para Android e/ou iOS. Você pode usar emuladores ou dispositivos físicos com o aplicativo Expo Go instalado.

2. Configuração do Ambiente com Expo CLI

1. **Inicie um novo projeto Expo utilizando npx create-expo-app:**

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

- **Explicação:**
 - npx create-expo-app: Utiliza o npx para executar o comando create-expo-app sem necessidade de instalação global.
 - DeviceResourcesApp: Nome do diretório do projeto.
 - --template blank: Especifica que será usado o template blank, que fornece uma configuração limpa e mínima para começar.

2. **Navegue até o diretório do projeto:**

```
cd DeviceResourcesApp
```

3. **Inicie o aplicativo:**

```
npx expo start
```

- **Explicação:**
 - Este comando inicia o servidor de desenvolvimento do Expo.
 - Uma interface será aberta no navegador onde você pode escanear o QR code com o aplicativo Expo Go no seu dispositivo móvel ou usar emuladores para visualizar o aplicativo.

3. Gerenciamento de Permissões

Para acessar recursos como galeria e contatos, é necessário solicitar permissões ao usuário. O Expo simplifica esse processo através de suas APIs.

Configurando Permissões no Expo

1. Editar app.json OU app.config.js:

Adicione as permissões necessárias para iOS e Android.



```
1  {
2    "expo": {
3      "name": "DeviceResourceApp",
4      "slug": "DeviceResourceApp",
5      "version": "1.0.0",
6      "orientation": "portrait",
7      "icon": "./assets/icon.png",
8      "userInterfaceStyle": "light",
9      "splash": {
10        "image": "./assets/splash.png",
11        "resizeMode": "contain",
12        "backgroundColor": "#ffffff"
13      },
14      "ios": {
15        "supportsTablet": true,
16        "infoPlist": {
17          "NSPhotoLibraryUsageDescription": "Este aplicativo precisa acessar sua galeria de fotos.",
18          "NSContactsUsageDescription": "Este aplicativo precisa acessar seus contatos."
19        }
20      },
21      "android": {
22        "adaptiveIcon": {
23          "foregroundImage": "./assets/adaptive-icon.png",
24          "backgroundColor": "#ffffff"
25        },
26        "permissions": [
27          "READ_CONTACTS",
28          "WRITE_CONTACTS",
29          "READ_EXTERNAL_STORAGE",
30          "WRITE_EXTERNAL_STORAGE"
31        ]
32      },
33      "web": {
34        "favicon": "./assets/favicon.png"
35      }
36    }
37  }
```

- **Explicação:**

- **iOS (infoPlist):** Define as mensagens que serão exibidas ao solicitar permissões.
- **Android (permissions):** Lista as permissões necessárias para o aplicativo.

4. Manipulando Galeria e Imagens

Para manipular a galeria e imagens, utilizaremos a biblioteca `expo-image-picker`, que é integrada ao Expo.

4.1. Instalação e Configuração

1. **Instale a biblioteca `expo-image-picker`:**

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

- **Explicação:**

- `npx expo install`: Garante que a versão instalada seja compatível com a versão do Expo SDK que você está usando.
- `expo-image-picker`: Biblioteca para selecionar imagens da galeria ou tirar fotos com a câmera.

4.2. Uso Básico com Comentários

Vamos criar um componente que permite ao usuário selecionar uma imagem da galeria e exibi-la na tela.

1. **Crie o componente `ImagePickerComponent.js`:**

```
1 // src/components/ImagePickerComponent.js
2
3 // Importa as bibliotecas necessárias
4 import React, { useState } from 'react';
5 import { View, Button, Image, Alert, StyleSheet } from 'react-native';
6 import * as ImagePicker from 'expo-image-picker';
```

```
● ● ●  
1 // Define o componente funcional  
2 const ImagePickerComponent = () => {  
3     // Estado para armazenar a URI da imagem selecionada  
4     const [imageUri, setImageUri] = useState(null);  
5  
6     // Função para solicitar permissão e abrir a galeria  
7     const selectImage = async () => {  
8         // Solicita permissão para acessar a galeria  
9         const { status } = await ImagePicker.requestMediaLibraryPermissionsAsync();  
10  
11        // Verifica se a permissão foi concedida  
12        if (status !== 'granted') {  
13            Alert.alert('Permissão Negada', 'Permissão para acessar a galeria foi negada.');//  
14            return;  
15        }  
16  
17        // Abre a galeria para seleção de imagem  
18        const result = await ImagePicker.launchImageLibraryAsync({  
19            mediaTypes: ImagePicker.MediaTypeOptions.Images, // Apenas imagens  
20            allowsEditing: true, // Permite edição básica  
21            quality: 1, // Qualidade da imagem (1 é a melhor)  
22        });  
23  
24        // Verifica se o usuário cancelou a operação  
25        if (result.cancelled) {  
26            Alert.alert('Operação Cancelada', 'Você cancelou a seleção de imagem.');//  
27            return;  
28        }  
29  
30        // Define a URI da imagem selecionada no estado  
31        setImageUri(result.uri);  
32    };
```

```
● ● ●  
1     return (  
2         // Contêiner principal com estilo centralizado  
3         <View style={styles.container}>  
4             /* Botão para selecionar imagem */  
5             <Button title="Selecionar Imagem" onPress={selectImage} />  
6  
7             {/* Exibe a imagem selecionada, se houver */}  
8             {imageUri && (  
9                 <Image  
10                    source={{ uri: imageUri }} // Fonte da imagem  
11                    style={styles.image} // Estilo da imagem  
12                    />  
13             )}  
14         </View>  
15     );  
16 };
```



```
1 // Define os estilos utilizados no componente
2 const styles = StyleSheet.create({
3   container: {
4     flex: 1, // Ocupa todo o espaço disponível
5     justifyContent: 'center', // Centraliza verticalmente
6     alignItems: 'center', // Centraliza horizontalmente
7     padding: 20, // Espaçamento interno
8     backgroundColor: '#fff', // Cor de fundo branca
9   },
10  image: {
11    width: 200, // Largura da imagem
12    height: 200, // Altura da imagem
13    marginTop: 20, // Espaçamento acima da imagem
14    borderRadius: 10, // Bordas arredondadas
15  },
16 });
17
18 // Exporta o componente para uso externo
19 export default ImagePickerComponent;
```

2. Utilize o componente no aplicativo principal App.js:

```
1 // App.js
2
3 // Importa as bibliotecas necessárias
4 import React from 'react';
5 import { SafeAreaView, StyleSheet } from 'react-native';
6 import ImagePickerComponent from './src/components/ImagePickerComponent';
7
8 // Define o componente principal do aplicativo
9 const App = () => {
10   return (
11     // SafeAreaView para garantir que o conteúdo não ultrapasse áreas seguras do dispositivo
12     <SafeAreaView style={styles.container}>
13       {/* Renderiza o componente de seleção de imagem */}
14       <ImagePickerComponent />
15     </SafeAreaView>
16   );
17 };
18
19 // Define os estilos utilizados no aplicativo principal
20 const styles = StyleSheet.create({
21   container: {
22     flex: 1, // Ocupa todo o espaço disponível
23     backgroundColor: '#f0f0f0', // Cor de fundo cinza claro
24   },
25 });
26
27 // Exporta o componente principal
28 export default App;
```

5. Manipulando Contatos

Para manipular contatos, utilizaremos a biblioteca expo-contacts, que é compatível com o Expo e facilita o acesso aos contatos do dispositivo.

5.1. Instalação e Configuração

1. Instale a biblioteca expo-contacts:

```
npx expo install expo-contacts
```

- **Explicação:**
 - expo-contacts: Biblioteca para acessar e manipular os contatos do dispositivo.

5.2. Uso Básico com Comentários

Vamos criar um componente que solicita permissão para acessar os contatos, carrega os contatos e os exibe em uma lista.

1. Crie o componente ContactsComponent.js:

```
1 // src/components/ContactsComponent.js
2
3 // Importa as bibliotecas necessárias
4 import React, { useEffect, useState } from 'react';
5 import { View, Text, FlatList, Button, Alert, StyleSheet } from 'react-native';
6 import * as Contacts from 'expo-contacts';
```

```
1 // Define o componente funcional
2 const ContactsComponent = () => {
3     // Estado para armazenar os contatos
4     const [contacts, setContacts] = useState([]);
5
6     // Função para solicitar permissão e carregar contatos
7     const loadContacts = async () => {
8         // Solicita permissão para acessar contatos
9         const { status } = await Contacts.requestPermissionsAsync();
10
11        // Verifica se a permissão foi concedida
12        if (status !== 'granted') {
13            Alert.alert('Permissão Negada', 'Permissão para acessar contatos foi negada.');
14            return;
15        }
16
17        try {
18            // Obtém todos os contatos do dispositivo
19            const { data } = await Contacts.getContactsAsync({
20                fields: [Contacts.Fields.Emails, Contacts.Fields.PhoneNumbers],
21            });
22
23            // Verifica se há contatos
24            if (data.length > 0) {
25                setContacts(data); // Atualiza o estado com os contatos obtidos
26            } else {
27                Alert.alert('Sem Contatos', 'Nenhum contato encontrado.');
28            }
29        } catch (error) {
30            // Trata possíveis erros na obtenção dos contatos
31            Alert.alert('Erro', 'Ocorreu um erro ao carregar os contatos.');
32            console.error(error);
33        }
34    };
}
```

```
1 // Executa a função de carregar contatos quando o componente é montado
2 useEffect(() => {
3     loadContacts();
4 }, []);
5
6 // Função para renderizar cada item da lista de contatos
7 const renderItem = ({ item }) => (
8     <View style={styles.contactItem}>
9         {/* Nome completo do contato */}
10        <Text style={styles.contactName}>
11            {item.firstName} {item.lastName}
12        </Text>
13
14        {/* Lista de números de telefone do contato */}
15        {item.phoneNumbers && item.phoneNumbers.map((phone, index) => (
16            <Text key={index} style={styles.contactDetail}>
17                 {phone.number}
18            </Text>
19        ))}
20
21        {/* Lista de emails do contato */}
22        {item.emails && item.emails.map((email, index) => (
23            <Text key={index} style={styles.contactDetail}>
24                 {email.email}
25            </Text>
26        ))}
27    </View>
28 );
```

```
1     return (
2         // Contêiner principal com estilo de preenchimento
3         <View style={styles.container}>
4             {/* Botão para recarregar os contatos manualmente */}
5             <Button title="Recarregar Contatos" onPress={loadContacts} />
6
7             {/* Lista de contatos exibida usando FlatList */}
8             <FlatList
9                 data={contacts} // Dados da lista
10                keyExtractor={({item}) => item.id} // Chave única para cada item
11                renderItem={renderItem} // Função para renderizar cada item
12                contentContainerStyle={styles.list} // Estilo do conteúdo da lista
13            />
14        </View>
15    );
16};
```

```
1 // Define os estilos utilizados no componente
2 const styles = StyleSheet.create({
3   container: {
4     flex: 1, // Ocupa todo o espaço disponível
5     padding: 20, // Espaçamento interno
6     backgroundColor: '#fff', // Cor de fundo branca
7   },
8   list: {
9     marginTop: 20, // Espaçamento acima da lista
10  },
11  contactItem: {
12    padding: 15, // Espaçamento interno
13    borderBottomWidth: 1, // Linha de separação inferior
14    borderColor: '#eee', // Cor da linha de separação
15  },
16  contactName: {
17    fontSize: 18, // Tamanho da fonte
18    fontWeight: 'bold', // Peso da fonte
19  },
20  contactDetail: {
21    fontSize: 14, // Tamanho da fonte
22    color: '#555', // Cor do texto
23    marginTop: 5, // Espaçamento acima do texto
24  },
25 });
26
27 // Exporta o componente para uso externo
28 export default ContactsComponent;
```

2. Utilize o componente no aplicativo principal App.js:

```
1 // App.js
2
3 // Importa as bibliotecas necessárias
4 import React from 'react';
5 import { SafeAreaView, StyleSheet } from 'react-native';
6 import ImagePickerComponent from './src/components/ImagePickerComponent';
7 import ContactsComponent from './src/components/ContactsComponent';
8
9 // Define o componente principal do aplicativo
10 const App = () => {
11   return (
12     // SafeAreaView para garantir que o conteúdo não ultrapasse áreas seguras do dispositivo
13     <SafeAreaView style={styles.container}>
14       /* Renderiza o componente de seleção de imagem
15       <ImagePickerComponent />
16     */
17     /* ScrollView para permitir rolagem caso o conteúdo exceda a tela */
18     <ScrollView>
19       /* Renderiza o componente de contatos */
20       <ContactsComponent />
21     </ScrollView>
22   </SafeAreaView>
23 );
24 };
25
26 // Define os estilos utilizados no aplicativo principal
27 const styles = StyleSheet.create({
28   container: {
29     flex: 1, // Ocupa todo o espaço disponível
30     backgroundColor: '#f0f0f0', // Cor de fundo cinza claro
31   },
32 });
33
34 // Exporta o componente principal
35 export default App;
```

6. Considerações Finais

- **Tratamento de Erros:** Sempre trate possíveis erros ao acessar recursos do dispositivo, como permissões negadas ou falhas na obtenção de dados. Utilize try-catch e exiba mensagens amigáveis para o usuário.
- **Desempenho:** Ao lidar com grandes quantidades de dados (como muitos contatos), utilize componentes otimizados como FlatList, que implementa renderização preguiçosa e outras otimizações de desempenho.
- **Segurança e Privacidade:** Respeite a privacidade dos usuários. Solicite apenas as permissões necessárias e explique claramente o motivo para o usuário. Nunca colete ou armazene dados sensíveis sem o consentimento explícito.
- **Testes em Diferentes Dispositivos:** Teste seu aplicativo em diferentes versões de Android e iOS, bem como em dispositivos com diferentes tamanhos de tela e capacidades, para garantir compatibilidade e comportamento consistente.

- **Atualizações de Bibliotecas:** Mantenha as bibliotecas e o SDK do Expo atualizados para aproveitar melhorias, novas funcionalidades e correções de segurança.
- **Explorar Funcionalidades Avançadas:** Conforme seu aplicativo evolui, considere adicionar funcionalidades mais avançadas, como edição de contatos, upload de imagens para servidores, integração com APIs externas, e muito mais.

Este guia utiliza o **Expo CLI** com o comando `npx create-expo-app --template` para simplificar o desenvolvimento e fornece comentários detalhados em cada linha de código, facilitando a compreensão e manutenção do projeto. O Expo oferece uma ampla gama de ferramentas e APIs que tornam o desenvolvimento em React Native mais acessível e eficiente.

EXTRA

1. Inserindo Ícones Usando Emojis

Código Utilizado

No componente `ContactsComponent`, ao exibir os detalhes dos contatos, utilizei emojis para representar ícones de telefone e e-mail da seguinte maneira:

```
<Text key={index} style={styles.contactDetail}>
  📞 {phone.number}
</Text>
```

Explicação

- **Emoji Direto no Texto:**
 - O emoji de telefone 📞 é inserido diretamente dentro do componente `<Text>`. Isso é uma maneira rápida e simples de adicionar ícones sem a necessidade de bibliotecas adicionais.
 - **Vantagens:**
 - **Simplicidade:** Não requer instalação de bibliotecas externas.
 - **Rapidez:** Fácil de implementar para protótipos ou projetos pequenos.
 - **Desvantagens:**
 - **Personalização Limitada:** Emojis podem não ter o estilo ou tamanho desejado para todos os casos de uso.
 - **Consistência Visual:** Diferentes plataformas ou dispositivos podem renderizar emojis de maneira ligeiramente diferente.

2. Inserindo Ícones Usando Bibliotecas de Ícones

Para uma abordagem mais robusta e personalizável, é recomendável utilizar bibliotecas de ícones. Com o **Expo**, uma das opções mais populares é a biblioteca `@expo/vector-icons`, que já vem pré-instalada em projetos Expo.

Passo a Passo para Inserir Ícones Usando `@expo/vector-icons`

2.1. Instalação (Se necessário)

Em projetos criados com **Expo**, a biblioteca `@expo/vector-icons` já está incluída. Portanto, geralmente não é necessário instalar nada adicional. No entanto, se por algum motivo não estiver disponível, você pode instalá-la utilizando:

```
npx expo install @expo/vector-icons
```

2.2. Importação do Componente de Ícone

Primeiro, importe o conjunto de ícones que deseja utilizar. A biblioteca `@expo/vector-icons` suporta vários conjuntos de ícones, como **FontAwesome**, **MaterialIcons**, **Ionicons**, entre outros.

Por exemplo, para usar **FontAwesome**:

```
import { FontAwesome } from '@expo/vector-icons';
```

2.3. Utilizando o Ícone no Código

Substitua o emoji pelo componente de ícone correspondente. Por exemplo, para inserir um ícone de telefone usando **FontAwesome**:

```
<FontAwesome name="phone" size={20} color="#555" />
```

2.4. Exemplo Completo no Componente ContactsComponent

Vamos modificar o componente `ContactsComponent` para utilizar ícones da biblioteca `@expo/vector-icons` em vez de emojis.

Passo 1: Importar o Conjunto de Ícones

No início do arquivo `ContactsComponent.js`, importe o conjunto de ícones desejado. Neste exemplo, usaremos **FontAwesome**:

```
import { FontAwesome } from '@expo/vector-icons';
```

Passo 2: Substituir Emojis por Componentes de Ícone

Atualize o método renderItem para utilizar os ícones:

```
● ● ●  
1 // Função para renderizar cada item da lista de contatos  
2 const renderItem = ({ item }) => (  
3   <View style={styles.contactItem}>  
4     /* Nome completo do contato */  
5     <Text style={styles.contactName}>  
6       {item.firstName} {item.lastName}  
7     </Text>  
8  
9     /* Lista de números de telefone do contato */  
10    {item.phoneNumbers && item.phoneNumbers.map((phone, index) => (  
11        <View key={index} style={styles.contactDetailContainer}>  
12          <FontAwesome name="phone" size={16} color="#555" style={styles.icon} />  
13          <Text style={styles.contactDetail}>{phone.number}</Text>  
14        </View>  
15      ))}  
16  
17     /* Lista de emails do contato */  
18     {item.emails && item.emails.map((email, index) => (  
19        <View key={index} style={styles.contactDetailContainer}>  
20          <FontAwesome name="envelope" size={16} color="#555" style={styles.icon} />  
21          <Text style={styles.contactDetail}>{email.email}</Text>  
22        </View>  
23      ))}  
24    </View>  
25  );
```

Passo 3: Atualizar os Estilos

Adicione estilos para alinhar os ícones e o texto corretamente:

```
● ● ●  
1 // ... outros estilos  
2 contactDetailContainer: {  
3   flexDirection: 'row', // Alinha ícone e texto na horizontal  
4   alignItems: 'center', // Alinha verticalmente ao centro  
5   marginTop: 5, // Espaçamento acima  
6 },  
7 icon: {  
8   marginRight: 10, // Espaçamento entre o ícone e o texto  
9 },
```

Passo 4: Resultado

Com essas alterações, o componente `ContactsComponent` exibirá ícones de telefone e e-mail ao lado dos respectivos detalhes, proporcionando uma interface mais elegante e consistente.

2.5. Outros Conjuntos de Ícones

A biblioteca `@expo/vector-icons` inclui diversos conjuntos de ícones. Alguns dos mais populares incluem:

- **Ionicons**
- **MaterialIcons**
- **Entypo**
- **Feather**
- **AntDesign**

Você pode escolher o conjunto que melhor se adequa ao design do seu aplicativo.

Exemplo usando Ionicons:

```
import { Ionicons } from '@expo/vector-icons';

// Uso do ícone
<Ionicons name="call" size={16} color="#555" style={styles.icon} />
```

3. Considerações Finais

- **Escolha da Biblioteca de Ícones:**
 - **Simplicidade vs. Flexibilidade:** Enquanto emojis são rápidos e fáceis, bibliotecas de ícones oferecem maior flexibilidade em termos de personalização e consistência visual.
 - **Consistência de Design:** Usar uma biblioteca de ícones garante que todos os ícones do aplicativo tenham um estilo visual coerente.
- **Performance:**
 - **Emojis:** Leves e sem impacto significativo na performance.
 - **Bibliotecas de Ícones:** Podem aumentar o tamanho do bundle, mas oferecem recursos avançados que justificam seu uso.
- **Acessibilidade:**
 - **Emojis:** Podem ser interpretados de maneira diferente por leitores de tela.
 - **Componentes de Ícone:** Melhor integração com ferramentas de acessibilidade.
- **Customização:**
 - **Bibliotecas de Ícones:** Permitem alterar facilmente cor, tamanho e outros atributos, além de suportar animações e interações.

4. Recursos Adicionais

- **Documentação do Expo Vector Icons:** Expo Vector Icons Documentation
- **Catálogo de Ícones:** Explore os diferentes conjuntos de ícones disponíveis na biblioteca para encontrar os que melhor atendem às suas necessidades.
- **Tutorial de Uso de Ícones com React Native:** Existem diversos tutoriais que detalham o uso de ícones em React Native, oferecendo exemplos práticos e dicas de design.

Utilizar ícones de maneira eficaz pode melhorar significativamente a experiência do usuário e a estética do seu aplicativo. Enquanto emojis são uma solução rápida, bibliotecas de ícones como `@expo/vector-icons` oferecem maior flexibilidade e profissionalismo, especialmente em aplicativos mais complexos ou voltados para produção.