

Guia Passo a Passo: Navegação Entre Telas no React Native Usando React Navigation

Objetivos:

- Configurar um projeto básico de React Native.
- Implementar navegação entre três telas usando o React Navigation.
- Adicionar estilização para centralizar botões e ajustar seu tamanho.

Pré-requisitos:

- Node.js instalado.
- Editor de código (como VSCode).

Passo 1: Criar o projeto React Native

Execute o seguinte comando para criar um projeto:

```
npx create-expo-app MeuAppDeNavegacao --template
```

Passo 2: Instalar as dependências do React Navigation

Instale o núcleo da biblioteca:

```
npm install @react-navigation/native
```

Depois, instale as dependências adicionais:

```
npm install react-native-screens react-native-safe-area-context
```

Passo 3: Instalar a biblioteca de navegação em pilha

Instale o pacote de navegação em pilha:

```
npm install @react-navigation/stack
```

Passo 4: Configurar o projeto

No arquivo App.js:

```
● ● ●  
1 import * as React from 'react';
2 import { NavigationContainer } from '@react-navigation/native';
3 import { createStackNavigator } from '@react-navigation/stack';
4 import HomeScreen from './src/screens/HomeScreen';
5 import DetailsScreen from './src/screens/DetailsScreen';
6 import ProfileScreen from './src/screens/ProfileScreen';
7
8 const Stack = createStackNavigator();
9
10 export default function App() {
11   return (
12     <NavigationContainer>
13       <Stack.Navigator initialRouteName="Home">
14         <Stack.Screen name="Home" component={HomeScreen} />
15         <Stack.Screen name="Details" component={DetailsScreen} />
16         <Stack.Screen name="Profile" component={ProfileScreen} />
17       </Stack.Navigator>
18     </NavigationContainer>
19   );
20 }
```

Passo 5: Criar as telas

Agora, crie os componentes das telas com a estilização para os botões:

HomeScreen.js

```
1 import React from 'react';
2 import { View, Text, Button, StyleSheet, Dimensions } from 'react-native';
3
4 const windowHeight = Dimensions.get('window').width;
5
6 export default function HomeScreen({ navigation }) {
7   return (
8     <View style={styles.container}>
9       <Text style={styles.title}>Home Screen</Text>
10      <View style={styles.buttonContainer}>
11        <Button
12          title="Go to Details"
13          onPress={() => navigation.navigate('Details')}
14        />
15      </View>
16      <View style={styles.buttonContainer}>
17        <Button
18          title="Go to Profile"
19          onPress={() => navigation.navigate('Profile')}
20        />
21      </View>
22    </View>
23  );
24}
```

```
1 const styles = StyleSheet.create({
2   container: {
3     flex: 1,
4     justifyContent: 'center',
5     alignItems: 'center',
6     backgroundColor: '#f0f8ff', // Cor de fundo da tela
7   },
8   title: {
9     fontSize: 24,
10    marginBottom: 20,
11  },
12  buttonContainer: {
13    backgroundColor: '#add8e6', // Cor de fundo do container do botão
14    margin: 10,
15    width: windowWidth * 0.5, // 50% da largura da tela
16    borderRadius: 5,
17  },
18 });

```


DetailsScreen.js

```
1 import React from 'react';
2 import { View, Text, Button, StyleSheet, Dimensions } from 'react-native';
3
4 const windowHeight = Dimensions.get('window').width;
5
6 export default function DetailsScreen({ navigation }) {
7   return (
8     <View style={styles.container}>
9       <Text style={styles.title}>Details Screen</Text>
10      <View style={styles.buttonContainer}>
11        <Button
12          title="Go to Home"
13          onPress={() => navigation.navigate('Home')}
14        />
15      </View>
16      <View style={styles.buttonContainer}>
17        <Button
18          title="Go to Profile"
19          onPress={() => navigation.navigate('Profile')}
20        />
21      </View>
22    </View>
23  );
24}
```

```
1 const styles = StyleSheet.create({
2   container: {
3     flex: 1,
4     justifyContent: 'center',
5     alignItems: 'center',
6     backgroundColor: '#faf0e6', // Cor de fundo da tela
7   },
8   title: {
9     fontSize: 24,
10    marginBottom: 20,
11  },
12  buttonContainer: {
13    backgroundColor: '#ffebcd', // Cor de fundo do container do botão
14    margin: 10,
15    width: windowWidth * 0.5, // 50% da largura da tela
16    borderRadius: 5,
17  },
18});
```


ProfileScreen.js

```
1 import React from 'react';
2 import { View, Text, Button, StyleSheet, Dimensions } from 'react-native';
3
4 const windowHeight = Dimensions.get('window').width;
5
6 export default function ProfileScreen({ navigation }) {
7   return (
8     <View style={styles.container}>
9       <Text style={styles.title}>Profile Screen</Text>
10      <View style={styles.buttonContainer}>
11        <Button
12          title="Go to Home"
13          onPress={() => navigation.navigate('Home')}
14        />
15      </View>
16      <View style={styles.buttonContainer}>
17        <Button
18          title="Go to Details"
19          onPress={() => navigation.navigate('Details')}
20        />
21      </View>
22    </View>
23  );
24}
```

```
1 const styles = StyleSheet.create({
2   container: {
3     flex: 1,
4     justifyContent: 'center',
5     alignItems: 'center',
6     backgroundColor: '#e6e6fa', // Cor de fundo da tela
7   },
8   title: {
9     fontSize: 24,
10    marginBottom: 20,
11  },
12  buttonContainer: {
13    backgroundColor: '#ddaa0dd', // Cor de fundo do container do botão
14    margin: 10,
15    width: windowWidth * 0.5, // 50% da largura da tela
16    borderRadius: 5,
17  },
18 });

```

Passo 6: Executar o projeto

Execute o projeto com:

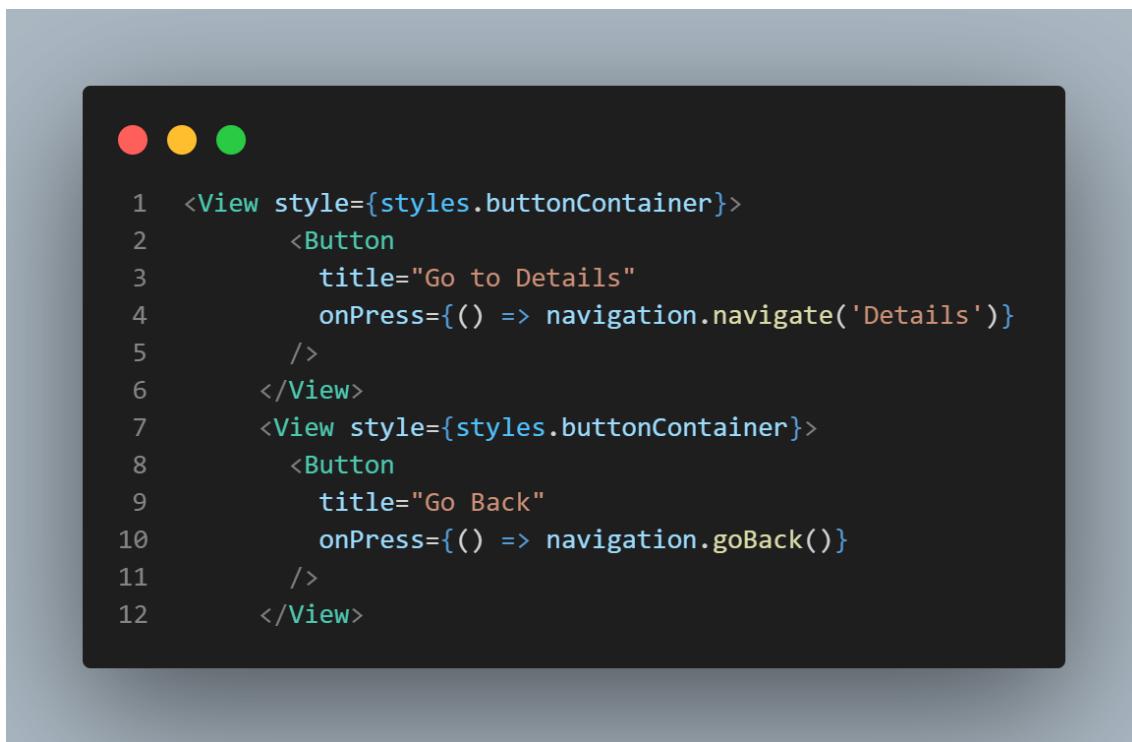
```
npx expo start
```

Explicação:

- **React Navigation:** A biblioteca usada para navegar entre as telas.
- **Stack Navigator:** Gerencia a navegação entre as telas empilhando-as, permitindo "voltar" à tela anterior.
- **navigation.navigate:** Função usada para trocar de tela programaticamente.
- **Dimensions:** Usado para obter a largura da tela.
- **container:** Centraliza os elementos na tela.
- **buttonContainer:** Define o tamanho e a cor de fundo dos botões.

Com esses passos, você terá um aplicativo básico com navegação entre três telas usando o React Navigation.

O botão Voltar será renderizado automaticamente em um navegador de pilha sempre que for possível para o usuário retornar da tela atual — em outras palavras, o botão Voltar será renderizado sempre que houver mais de uma tela na pilha. Geralmente é isso que você deseja, mas em outras circunstâncias você desejará ter um controle maior na navegação para voltar. Para isso, aqui está a atualização do código da ProfileScreen com a remoção da seta de voltar e a adição de um terceiro botão que utiliza o goBack para voltar à tela anterior:



Alterações:

- **Remoção da seta de voltar:** Isso pode ser feito adicionando a opção headerShown: false no Stack.Screen que define a ProfileScreen no App.js:

```
<Stack.Screen name="Profile" component={ProfileScreen} options={{ headerShown: false }} />
```

- **Botão "Go Back":** Agora existe um terceiro botão que chama navigation.goBack(), retornando à tela anterior.

Com essa alteração, o botão "Go Back" substitui a seta de voltar, mantendo a navegação totalmente controlada pelos botões na tela.

Guia Passo a Passo: Armazenamento Interno com AsyncStorage no React Native

Objetivo

Este guia fornece uma introdução ao uso do AsyncStorage no React Native, uma biblioteca que permite armazenar dados localmente no dispositivo. Esse recurso é útil para salvar informações de forma persistente entre sessões, como preferências de usuário, configurações ou dados simples.

Objetivos Gerais

1. Compreender o conceito de armazenamento interno em dispositivos móveis.
2. Aprender a configurar e utilizar o AsyncStorage no React Native.
3. Criar um aplicativo simples que utilize o AsyncStorage para salvar e recuperar dados.

Definição

AsyncStorage é uma biblioteca para React Native que fornece um sistema de armazenamento baseado em chave-valor de maneira assíncrona. Esse armazenamento é persistente e adequado para dados leves que precisam ser mantidos entre sessões. É ideal para salvar pequenas informações, como tokens de autenticação, preferências e estados simples de aplicativo.

Nota: A partir das versões mais recentes do React Native, o AsyncStorage foi movido para o pacote `@react-native-async-storage/async-storage`, devendo ser instalado separadamente.

Usos do AsyncStorage em Aplicativos Móveis

1. **Autenticação Persistente:**
 - Armazenar tokens de autenticação (como JWT) para manter o usuário logado entre sessões.
 - Evita que o usuário precise fazer login toda vez que abrir o aplicativo, melhorando a experiência de uso.
2. **Preferências do Usuário:**
 - Salvar configurações e preferências como temas (claro/escuro), configurações de idioma, notificações ou layout.
 - Isso permite que o usuário personalize o aplicativo, e essas escolhas sejam lembradas ao longo do uso.
3. **Dados de Formulário ou Estado Temporário:**
 - Manter dados não enviados, como respostas de formulários ou rascunhos de mensagens.

- Por exemplo, se o usuário começa a preencher um formulário e sai do aplicativo, ele pode retornar e encontrar o progresso salvo.

4. Histórico de Pesquisa ou Navegação:

- Armazenar as consultas de pesquisa recentes, produtos visualizados, ou páginas acessadas.
- Permite ao aplicativo mostrar um histórico personalizado ao usuário, como uma lista de pesquisas recentes.

5. Cache de Dados de API:

- Salvar dados de API como listas de produtos, posts de blog ou mensagens.
- Isso ajuda a reduzir chamadas de API e melhorar a velocidade de carregamento ao exibir dados recentes, mesmo quando o usuário está offline ou com internet lenta.

Esses são apenas alguns exemplos, mas o AsyncStorage pode ser útil sempre que você precisar armazenar dados leves que precisem ser persistentes entre sessões no aplicativo.

Exemplo Prático

Vamos criar um pequeno aplicativo de lista de tarefas ("To-do") que permite ao usuário adicionar e remover itens, e persiste esses dados usando AsyncStorage. Cada tarefa salva será recuperada automaticamente na próxima vez que o aplicativo for aberto.

Passo 1: Configuração Inicial

1. Crie um projeto React Native:

```
npx create-expo-app ToDoApp --template blank  
cd ToDoApp
```

2. Instale o AsyncStorage:

```
npx expo install @react-native-async-storage/async-storage
```

3. Importe o AsyncStorage no seu código:

```
import AsyncStorage from '@react-native-async-storage/async-storage';
```

Passo 2: Estrutura do Aplicativo

Vamos desenvolver o aplicativo criando uma interface simples para adicionar e exibir as tarefas.

Código Completo

```
● ○ ●

1 import React, { useState, useEffect } from 'react';
2 import { View, Text, TextInput,
3         Button, FlatList, TouchableOpacity,
4         StyleSheet, SafeAreaView } from 'react-native';
5 import AsyncStorage from '@react-native-async-storage/async-storage';
6
7 const App = () => {
8     const [task, setTask] = useState('');
9     const [tasks, setTasks] = useState([]);
10
11    // Função para salvar tarefas no AsyncStorage
12    const saveTasks = async (tasksArray) => {
13        try {
14            await AsyncStorage.setItem('tasks', JSON.stringify(tasksArray));
15        } catch (error) {
16            console.log('Erro ao salvar tarefas:', error);
17        }
18    };
19
20    // Função para carregar tarefas ao iniciar o app
21    const loadTasks = async () => {
22        try {
23            const storedTasks = await AsyncStorage.getItem('tasks');
24            if (storedTasks !== null) {
25                setTasks(JSON.parse(storedTasks));
26            }
27        } catch (error) {
28            console.log('Erro ao carregar tarefas:', error);
29        }
30    };
}
```

```
● ● ●

1 // Adiciona uma nova tarefa
2 const addTask = () => {
3   if (task.trim() !== '') {
4     const newTasks = [...tasks, task];
5     setTasks(newTasks);
6     saveTasks(newTasks); // Salva a lista atualizada
7     setTask('');
8   }
9 };
10
11 // Remove uma tarefa pelo índice
12 const removeTask = (index) => {
13   const newTasks = tasks.filter((_, i) => i !== index);
14   setTasks(newTasks);
15   saveTasks(newTasks); // Salva a lista atualizada
16 };
17
18 // Carrega as tarefas ao iniciar o aplicativo
19 useEffect(() => {
20   loadTasks();
21 }, []);
22
```

```
 1  return (
 2      <View style={styles.container}>
 3          <Text style={styles.title}>Lista de Tarefas</Text>
 4          <TextInput
 5              style={styles.input}
 6              placeholder="Digite uma nova tarefa"
 7              value={task}
 8              onChangeText={(text) => setTask(text)}
 9          />
10          <Button title="Adicionar Tarefa" onPress={addTask} />
11          <FlatList
12              data={tasks}
13              keyExtractor={(item, index) => index.toString()}
14              renderItem={({ item, index }) => (
15                  <View style={styles.taskContainer}>
16                      <Text style={styles.taskText}>{item}</Text>
17                      <TouchableOpacity onPress={() => removeTask(index)}>
18                          <Text style={styles.deleteText}>Excluir</Text>
19                          </TouchableOpacity>
20                      </View>
21                  )})
22              />
23          </View> 
24      );
25  );
26
```

```
● ● ●

1 const styles = StyleSheet.create({
2   container: {
3     flex: 1,
4     padding: 20,
5     backgroundColor: '#f5f5f5',
6   },
7   title: {
8     fontSize: 24,
9     fontWeight: 'bold',
10    marginTop: 20,
11    marginBottom: 20,
12  },
13   input: {
14     height: 40,
15     borderColor: '#cccccc',
16     borderWidth: 1,
17     marginBottom: 10,
18     paddingHorizontal: 10,
19   },
20   taskContainer: {
21     flexDirection: 'row',
22     justifyContent: 'space-between',
23     alignItems: 'center',
24     paddingVertical: 10,
25     borderBottomWidth: 1,
26     borderBottomColor: '#ddd',
27   },
28   taskText: {
29     fontSize: 18,
30   },
31   deleteText: {
32     color: 'red',
33     fontWeight: 'bold',
34   },
35 });
36
37 export default App;
```

Passo 3: Explicação do Código

1. **Carregamento de Tarefas:** No useEffect, chamamos loadTasks() para recuperar as tarefas salvas no AsyncStorage ao iniciar o aplicativo.
2. **Adicionar e Salvar Tarefas:** A função addTask() adiciona a tarefa digitada e atualiza a lista de tarefas. Em seguida, a função saveTasks() armazena a lista atualizada no AsyncStorage.
3. **Remover e Atualizar Tarefas:** A função removeTask() remove a tarefa da lista pelo índice, atualiza o estado e salva a nova lista.
4. **Interface:** Utilizamos um TextInput para digitar novas tarefas, um botão para adicionar e um FlatList para exibir cada tarefa. Cada tarefa inclui um botão "Excluir" para removê-la.

Passo 4: Teste do Aplicativo

- **Adicionar Tarefa:** Digite uma tarefa no campo de entrada e pressione "Adicionar Tarefa". A tarefa será salva automaticamente no armazenamento interno.
- **Excluir Tarefa:** Pressione "Excluir" ao lado de uma tarefa para removê-la.
- **Persistência de Dados:** Feche e abra o aplicativo. As tarefas devem permanecer salvas.

Conclusão

Com este aplicativo, você aprendeu a usar o AsyncStorage para armazenar e recuperar dados no React Native. Esse recurso é útil para manter informações leves entre sessões, garantindo que o usuário possa continuar de onde parou.

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. Inicialize 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
```

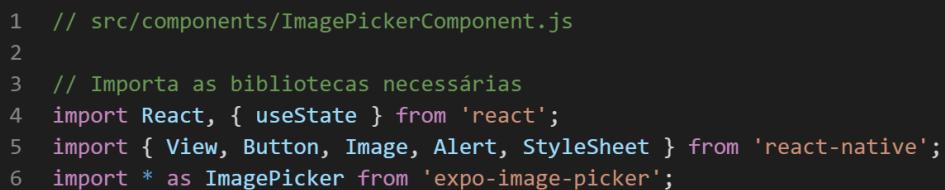
- **Explicação:**

- `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.

Criando uma REST API com JSON Server – parte 1

Nessa série de tutoriais, você vai aprender de uma forma simples e eficaz a implementar uma *REST API* totalmente falsa com zero codificação em pouquíssimo tempo.

Imagine você na seguinte situação. Você é um desenvolvedor *front-end* e precisa de um *back-end* rápido para prototipagem e simulação do projeto para a reunião com o cliente, e não tem tempo para esperar a equipe de *back-end* desenvolver e te entregar as funcionalidades necessárias.

Para resolver essa situação, você pode usar o *JSON Server* que é uma biblioteca capaz de criar uma *REST API fake* com todos os *endpoints* de um recurso, como os 4 principais verbos *HTTP*: *GET*, *POST*, *PUT* e *DELETE*. Dessa forma, seu *front-end* poderá consumir essa *API* simulada, possibilitando a criação de toda a camada *HTTP* da aplicação.

Agora vamos instalar o *JSON Server*. Para isso vou levar em conta que você já tenha um projeto *React* criado. Não que seja obrigatório ter um projeto, e ainda ser somente *React*. Apenas iremos aproveitar o projeto criado em aula. Mas você poderá instalar o *JSON Server* independentemente do local.

Instando o *JSON Server*

Em seu projeto, crie uma pasta chamada “**backend**” conforme a figura 1.

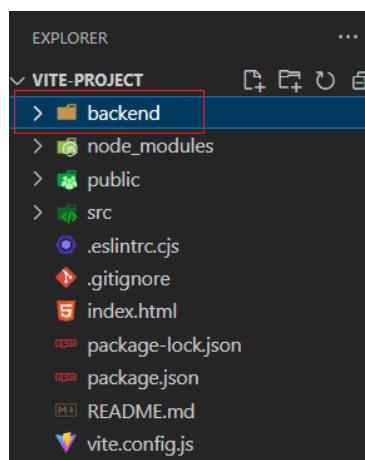


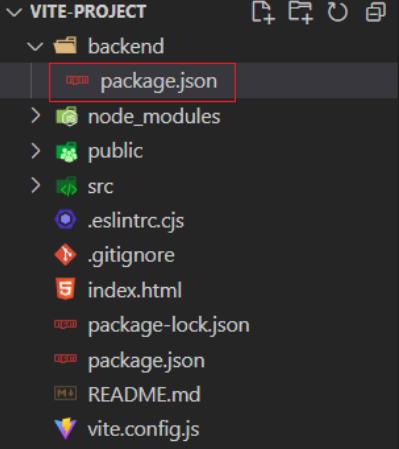
Figura 1 - criação da pasta backend no projeto.

Conforme a figura 2, abra o terminal do **VSCode** e execute o comando **cd backend** para entrar na pasta.

```
PS C:\Users\rafae\projetos\react\vite-project> cd backend
```

Figura 2 - comando para entrar na pasta backend.

O próximo passo, execute o comando **npm init -y**, ele irá criar o arquivo “**package.json**” para gerenciar nossas dependências, mostrado na figura 3.

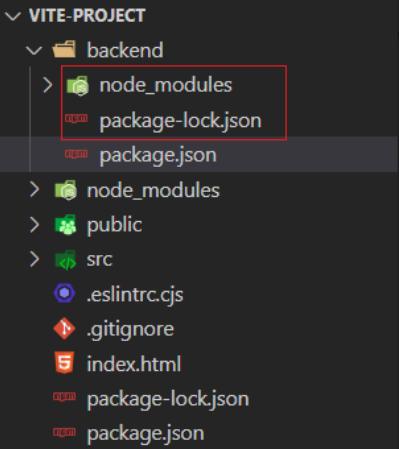


```

VITE-PROJECT
  ↘ backend > package.json > ...
    1 {
    2   "name": "backend",
    3   "version": "1.0.0",
    4   "description": "",
    5   "main": "index.js",
    6   "scripts": {
    7     "test": "echo \\\"Error: no test specified\\\" && exit 1"
    8   },
    9   "keywords": [],
   10   "author": "",
   11   "license": "ISC"
   12 }
   13
  
```

Figura 3 - resultado após a execução do comando **npm init -y**.

Para instalar o **JSON Server**, digite o comando **npm i json-server**. Com isso ele irá criar a dependência em nosso projeto conforme a figura 4.



```

VITE-PROJECT
  ↘ backend > package.json > ...
    4   "description": "",
    5   "main": "index.js",
    6   "scripts": {
    7     "test": "echo \\\"Error: no test specified\\\" && exit 1"
    8   },
    9   "keywords": [],
   10   "author": "",
   11   "license": "ISC",
   12   "dependencies": {
   13     "json-server": "^0.17.4"
   14   }
   15
  
```

Figura 4 - dependência do **JSON Server** criada.

Com o processo finalizado, agora iremos criar um arquivo com o nome “**db.json**” dentro da pasta “**backend**”. Esse arquivo servirá para que o **JSON Server** crie a *REST API* baseado nele. Nele iremos criar objeto **JSON** que terá todos os *endpoints* da nossa *API*.

```
backend > {} db.json > [ ] products
1  {
2    "products": []
3    [
4      {
5        "id": 1,
6        "name": "Caneta BIC preta",
7        "price": 5.89
8      },
9      {
10     "id": 2,
11     "name": "Notebook Mac Pro",
12     "price": 12000.00
13   },
14   {
15     "id": 3,
16     "name": "Samsung 20+",
17     "price": 5000.89
18   }
19 }
```

Figura 5 - arquivo db.json com o objeto json.

Vá para o arquivo ‘package.json’ e faça a alteração conforme demonstrado na figura 6.

```
6  "scripts": {
7    "test": "echo \"Error: no test specified\" && exit 1"
8  },
  

6  "scripts": {
7    "start": "json-server --watch db.json --port 3001"
8  },
```



Figura 6 - alteração do arquivo package.json

Basicamente estamos configurando para iniciar o *JSON Server* com o arquivo “db.json” na porta 3001 e fique monitorando as eventuais alterações com a opção –watch.

No terminal dentro da pasta “**backend**” execute o comando **npm start** para iniciar nossa API.

```

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

○ PS C:\Users\rafae\projetos\react\vite-project\backend> npm start

  > backend@1.0.0 start
  > json-server --watch db.json --port 3001

  \{^_^\}/ hi!

  Loading db.json
  Done

  Resources
  http://localhost:3001/products

  Home
  http://localhost:3001

  Type s + enter at any time to create a snapshot of the database
  Watching...

```

Figura 7 - execução da API.

Para testarmos nossa API, abra o navegador e digite a url <http://localhost:3001/products> . você obterá

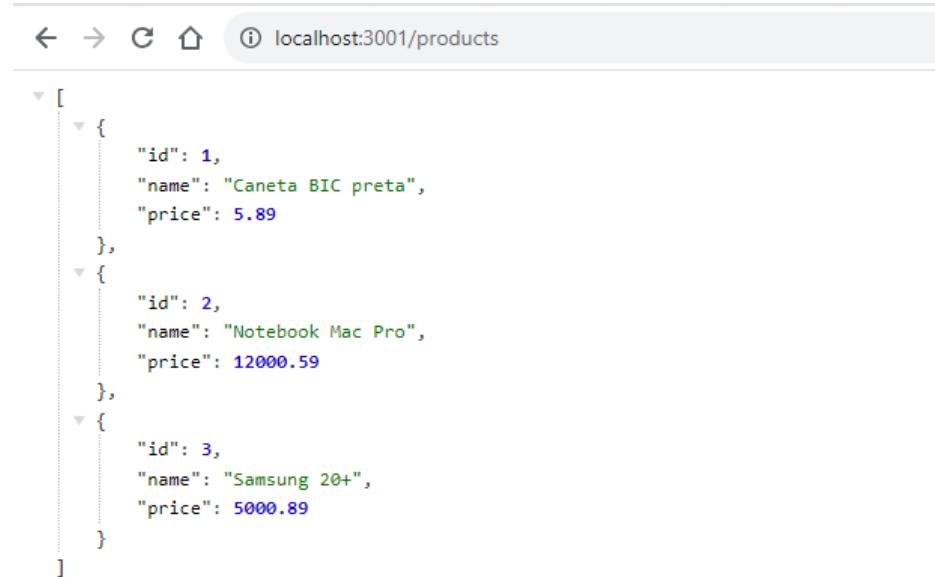


Figura 8 - resposta da API ao digitar a url.

Digitando a url <http://localhost:3001/products/1> , você obterá apenas o produto de id igual a 1

```
{
  "id": 1,
  "name": "Caneta BIC preta",
  "price": 5.89
}
```

Figura 9 - resposta da API trazendo apenas um produto.

Outra maneira de testar a sua API é usar um cliente REST como o **POSTMAN** ou **INSOMNIA**. No **VSCode** existem plugins que você pode instalar e usá-los para testar os métodos como **GET**, **POST**, **PUT** e **DELETE**.

Clique o ícone de *Extensions* e procure por *Thunder Client*, conforme a figura 10, mas fique à vontade em escolher outra extensão.

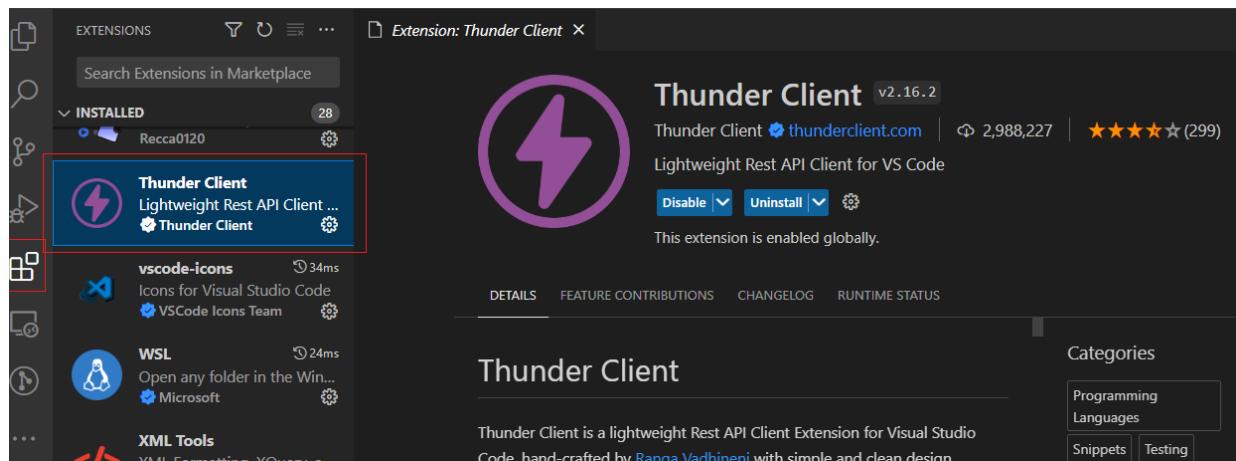


Figura 10 - extensão do VSCode para um cliente REST.

Em breve, disponibilizarei um tutorial de como utilizar o *Thunder Client*.

Rotas

Com base no arquivo '**db.json**', aqui estão todas os métodos *HTTP* que você poderá utilizar em sua aplicação:

Tabela 1 - métodos *HTTP* e rotas disponíveis em nossa API.

Método HTTP	Rotas	Ação
GET	/products	Obtém todos os produtos.
GET	/products/1	Obtém o produto com id igual a 1.
POST	/products	Salva um produto.
PUT	/products/1	Atualiza todos os dados do produto com id igual a 1.
PATCH	/products/1	Atualiza parte dos dados do produto com id igual a 1.
DELETE	/products/1	Remove o produto com o id igual a 1.

Concluindo...

Muitas vezes no desenvolvimento de uma aplicação do lado do *front-end* precisamos ter a estrutura de *REST API* para apresentação, prototipação ou até mesmo realizar testes, mas que ela ainda está sendo desenvolvida pela equipe *back-end*. Não queremos criar nosso *front-end* de uma maneira e depois ter que alterar quando o *back-end* entregar a estrutura de rotas, *endpoints* e tudo que envolve uma *REST API*.

Para resolver essa situação, vimos nesse tutorial como criar uma *REST API fake* para que possámos consumi-la de tal forma como se estivéssemos consumindo a *API* real de nossa aplicação. *JSON Server* é um aplicativo *JavaScript* que nos permite realizar tal feito.

Referências

json-server. Disponível em: <<https://www.npmjs.com/package/json-server>>. Acesso em: 29 nov. 2023.

EGGHEADIO. Expert led courses for front-end web developers. Disponível em: <<https://egghead.io/lessons/javascript-creating-demo-apis-with-json-server>>. Acesso em: 29 nov. 2023.

FRANCO, F. Simulando uma API REST com JSON Server de maneira simples. Disponível em: <<https://www.fabricadecodigo.com/json-server/#:~:text=O%20que%20%C3%A9%20o%20JSON%20Server&text=JSON%20Server%20%C3%A9%20uma%20biblioteca>>. Acesso em: 29 nov. 2023.

Criando uma REST API com JSON Server – parte 2

Agora que já sabemos como criar uma *REST API fake* utilizando o *JSON Server*, em que de forma fácil e rápida conseguimos ter um *back-end* para nos ajudar no desenvolvimento *front-end*, nesse tutorial iremos aprender como gerar dados *fake* em massa. Assim nossa simulação, prototipação e testes ficarão mais precisos.

Sabemos que gerar uma quantidade grande de dados manualmente é penoso, ainda mais tendo que muitas vezes, inventar dados para cada situação, tais como para um cliente, primeiro nome, último nome, endereço, número de telefone, e-mail etc. Haja tanta criatividade e tempo, não é mesmo?

Para resolvermos essa situação, podemos usar duas bibliotecas do *JavaScript* que de forma rápida conseguimos gerar esses dados randomicamente e com a quantidade que desejarmos. Essas bibliotecas são *Faker* e *Lodash*. A primeira gera grandes quantidades de dados falsos (mas realistas) para teste e desenvolvimento. A segunda oferece modularidade, desempenho e extras, tornando o *JavaScript* mais fácil, eliminando o incômodo de trabalhar com *arrays*, números, objetos, *string* etc.

Instalando as bibliotecas

Abra um terminal na pasta “*bakend*” do seu projeto e digite o comando **npm install faker** e em seguida **npm install lodash** .

Em seguida crie um arquivo chamado “**generate.js**” dentro da pasta “*backend*” conforme demonstrado na figura 1.

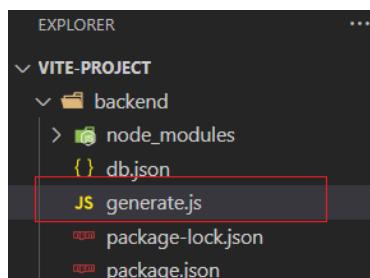


Figura 1 - Arquivo generate.js

Abra o arquivo “**generate.js**” e conforme a figura 2, digite o código que será explicado posteriormente.

```

backend > JS generate.js > ...
1 import { faker } from '@faker-js/faker/locale/pt_BR';
2 import lodash from 'lodash';
3 import fs from 'fs';
4
5 const peoples = lodash.times(50, function(n){
6   const firstName = faker.person.firstName();
7   const lastName = faker.person.lastName();
8   return {
9     id: n+1,
10    firstname: firstName,
11    lastname: lastName,
12    avatar: faker.image.avatar(),
13    address: faker.location.streetAddress(),
14    email: faker.internet.email({firstName: firstName.toLowerCase(), lastName: lastName.toLowerCase()})
15  });
16 });
17
18 const data = {};
19 data.peoples = peoples;
20 fs.writeFile('db.json', JSON.stringify(data), (err) => {
21   if(err) throw err;
22   console.log('Finalizado...');
23 });

```

Figura 2 - Código para gerar massa de dados fake

Explicando o código

Vamos à explicação, na **linha 1** é feita a importação da biblioteca *Faker* para geração de dados falsos. *Faker* oferece suporte a muitas localidades diferentes. Ao usar a instância padrão, *import {faker}from '@faker-js/faker'* você obtém dados em inglês. Para obter dados de uma localidade em português do Brasil, acrescente *'/locale/pt_BR'* na importação. Você poderá combinar várias localidades ou escolher a localidade desejada. Consulte o link <https://fakerjs.dev/guide/localization.html> para maiores informações quanto a localidades.

Na **linha 2** importamos a biblioteca *Lodash* para no ajudar na construção de nossos objetos que serão transformados em *JSON*. *Lodash* é ótimo para interagir com *arrays*, objetos e *string*, além de manipular e testar valores criando funções compostas. Para mais informações sobre *Lodash* visite o site <https://lodash.com/> .

Terminando as importações, na **linha 3** importamos o *FS*, um módulo integrado do *Node.js* que fornece uma *API* para interagir com o sistema de arquivos do computador em que o *Node.js* está sendo executado. Ele permite a leitura, gravação, exclusão e manipulação de arquivos e diretórios. Para mais detalhes consulte a documentação disponível no link <https://nodejs.org/docs/v0.3.1/api/fs.html> .

Entre as **linhas 5 e 16** é construído uma lista de objetos com dados *fake* de pessoas usando o *Lodash* juntamente com o *Faker*. Na linha 5 usamos o método *times()* do *Lodash* onde é invocado *n* tempos de interação, retornando um *array* dos resultados de cada interação. Em nosso código estamos informando que ele terá 50 (cinquenta) interações e para cada interação executará uma função anônima que retornará um objeto *fake*. Você poderá informar a quantidade de interações necessárias para a geração de dados para os seus testes.

Nas **linhas 6 e 7** usamos o objeto o módulo *person* do *Faker* para gerar os dados *firstName* (primeiro nome) e *lastName* (último nome ou sobrenome) e armazenamos em duas constantes que serão utilizadas posteriormente. *Faker* possui vários módulos (categorias) de dados para serem gerados aleatoriamente tais como *Person*, *Finance*, *Image*, *Internet*, *Commerce*, *Company* e muitos outros, cada qual com seu conjunto de dados. Para saber quais dados *fake* você pode gerar com *Faker*, consulte sua *API* pelo link <https://fakerjs.dev/api/>.

Seguindo as **linhas 9 a 14** é retornado dados *fakes* que irão compor uma pessoa.

Na **linha 18** é declarado uma constante ‘*data*’ que recebe uma lista de objetos vazia. Na sequência na **linha 19**, adicionamos um objeto ‘*peoples*’ a lista ‘*data*’ atribuindo a lista de pessoas geradas pelo *Lodash*.

Finalmente na **linha 20**, usamos o método *fs.writeFile* para escrever no arquivo ‘**db.json**’ a massa de dados gerada. Como o arquivo ‘**db.json**’ deve ser composto por uma *String Json*, precisamos converter nossos dados gerados em *String*. Para isso usamos o método *stringify* do objeto *JSON*.

Executando o código

Com o entendimento do código, vamos executá-lo para gerar os dados no arquivo ‘**db.json**’ que será usado pelo *JSON Serve*. Abra o terminal dentro da pasta ‘*backend*’ e digite o comando **node generate.js**

Pronto!!! O arquivo ‘**db.json**’ foi atualizado com a massa de dados *fakes* conforme podemos verificar na figura 3. Basta agora executar o *JSON Serve* para servir uma *REST API* para ser consumida pela sua aplicação.

```
backend > {} db.json > [ ] peoples
 1 <> {
 2 <>   "peoples": [
 3 <>     {
 4 <>       "id": 1,
 5 <>       "firstname": "Aline",
 6 <>       "lastname": "Santos",
 7 <>       "avatar": "https://avatars.githubusercontent.com/u/86743847",
 8 <>       "address": "741 Lívia Avenida",
 9 <>       "email": "aline.santos@live.com"
10 <>     },
11 <>     {
12 <>       "id": 2,
13 <>       "firstname": "Janaina",
14 <>       "lastname": "Pereira",
15 <>       "avatar": "https://cloudflare-ipfs.com/ipfs/Qmd3W5DuhgHirLHGViXi6V76LhCkZUz6pnFt5AJBiyyHye/avatar/624.jpg",
16 <>       "address": "129 Núbia Rua",
17 <>       "email": "janaina16@gmail.com"
18 <>     },
19 <>     {
20 <>       "id": 3,
21 <>       "firstname": "Samuel",
22 <>       "lastname": "Albuquerque",
23 <>       "avatar": "https://avatars.githubusercontent.com/u/78643454",
24 <>       "address": "695 Emanuel Rua",
25 <>       "email": "samuel18@hotmail.com"
26 <>     },
27 <>     {
28 <>       "id": 4,
29 <>       "firstname": "Sara",
30 <>       "lastname": "Pereira",
31 <>       "avatar": "https://avatars.githubusercontent.com/u/48739768",
32 <>       "address": "28853 Silva Avenida",
33 <>       "email": "sara.pereira91@yahoo.com"
34 <>     }
  
```

Figura 3 - Resultado contendo a massa de dados gerada pelo *Faker* e *Lodash*

Concluindo...

Quando precisarmos de uma massa de dados para realizarmos a prototipação, apresentação ou até mesmo testes de nossas aplicações, podemos obtê-la por meio de dados fakes gerados aleatoriamente.

Nesse tutorial vimos como gerar essa massa por meio da biblioteca *Faker* juntamente com *Lodash*, de forma fácil e rápido gerando grandes quantidades de dados de diversos tipos, tais como dados pessoais, comercial, internet entre outros.

Existem várias situações a serem exploradas que não foram abordadas aqui, mas com esses conhecimentos básicos e um pouco de pesquisa, você poderá resolver qualquer situação. Explore mais sobre o tema.

Referências

Faker | Faker. Disponível em: <<https://fakerjs.dev/>>.

Lodash. Disponível em: <<https://lodash.com/>>.

EGGHEADIO. Expert led courses for front-end web developers. Disponível em: <<https://egghead.io/lessons/javascript-creating-demo-apis-with-json-server>>.

Criando uma REST API com JSON Server – parte 3

Com o nosso *backend JSON Server* rodando, vamos utilizar a extensão *Thunder Client* para consumir nossa *REST API*. Lembrando que esse *REST Client* não é exclusivo para o *JSON Server*, ele pode ser usado para consumir e testar qualquer *API* que o seu aplicativo estiver usando.

O propósito desse tutorial é fornecer conhecimentos básicos sobre a ferramenta para a sua inicialização nos conceitos de *REST API*, deixando que você explore mais detalhes acessando a sua documentação pelo link <https://github.com/rangav/thunder-client-support#setenv> .

Levando em conta que você já instalou a extensão demonstrada no tutorial anterior, localize e clique no ícone do *Thunder Client* na *Action Bar* do *VScode* conforme mostrado na figura 1.

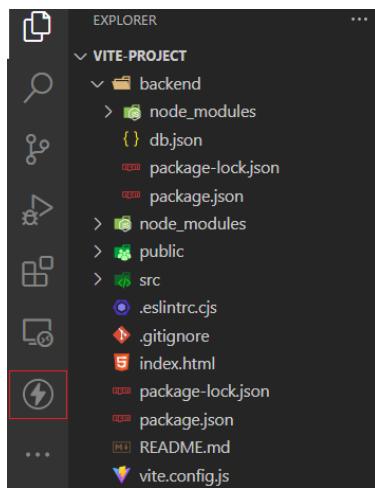


Figura 1 - Extensão do VSCode para Rest Api

Caso não apareça o ícone em sua *Action Bar*, certifique-se que tenha instalado a extensão ou clique nas reticências para mostrar mais opções.

Ao clicar no ícone do *Thunder Client*, irá aparecer no lado esquerdo do *VSCode* suas opções de tarefas para serem executadas conforme demonstrado na figura 2.

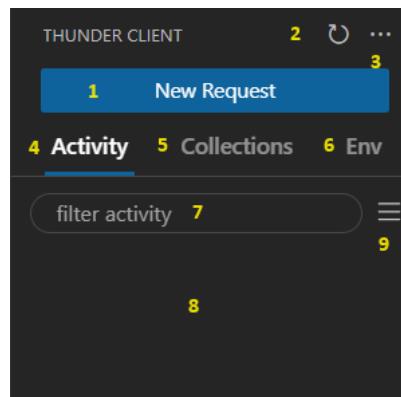


Figura 2 - Interface do Thunder Client

- 1 – Botão para adicionar uma nova requisição;
- 2 – Botão para atualizar o resultado de uma requisição;
- 3 – Menu com diversas opções para serem executadas no *Thunder Client*;
- 4 – Aba *Activity* (Atividade), onde é listada todas as requisições feitas na ferramenta;
- 5 – Aba *Collections* (Coleções), onde você poderá criar e gerenciar grupos de requisições;
- 6 – Aba *Environment Variables* (Variáveis de ambiente), onde você poderá criar e gerenciar variáveis locais e globais que poderão ser usadas nas requisições;
- 7 – Local para filtrar a lista de atividades, coleções e variáveis de ambiente;
- 8 – Local onde serão exibidas as atividades, coleções e variáveis de ambiente salvas na ferramenta;
- 9 – Menu com opções para serem executadas nas atividades, coleções e variáveis de ambiente.

Realizando a primeira requisição

Tomando como base o arquivo *JSON* criado no primeiro tutorial, usaremos os *endpoints* e os métodos *HTTP* conforme demonstrado na tabela 1.

Tabela 1- Métodos *HTTP* e os *endpoints*

Método <i>HTTP</i>	Rotas	Ação
GET	/products	Obtém todos os produtos.
GET	/products/1	Obtém o produto com id igual a 1.
POST	/products	Salva um produto.
PUT	/products/1	Atualiza todos os dados do produto com id igual a 1.
PATCH	/products/1	Atualiza parte dos dados do produto com id igual a 1.
DELETE	/products/1	Remove o produto com o id igual a 1.

Clique no botão “**New Request**”, aparecerá um formulário com campos a serem preenchidos conforme a requisição que desejamos realizar.

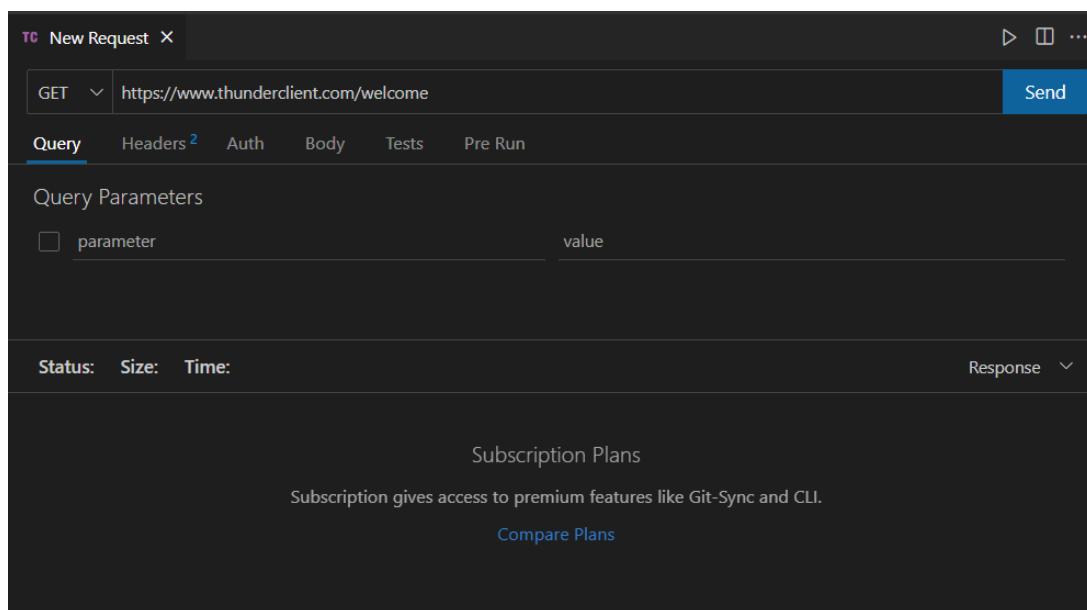


Figura 3 - Interface para nova resquisição

Seguindo a tabela 1, vamos executar uma requisição com o método *GET* para obter todos os produtos. Para tal, altere a *url* ao lado da palavra *GET* para a *url* do nosso *JSON Server* <http://localhost:3001/products> e em seguida clique no botão “*Send*”.

Perceba na figura 4 que iremos ter como resposta um erro informando que a conexão foi recusada pelo servidor (*Connection was refused by the server*).

The screenshot shows the Thunder Client interface. At the top, there's a dropdown for 'Method' set to 'GET' and a text input for 'URL' containing 'http://localhost:3001/products'. Below the URL input are tabs for 'Query', 'Headers', 'Auth', 'Body', 'Tests', and 'Pre Run'. Under the 'Query' tab, there's a section for 'Query Parameters' with a table where you can add parameters. The main area shows the request details: 'Status: ERROR', 'Size: 0 B', and 'Time: 0 ms'. To the right, there's a 'Response' tab which is currently inactive. The response body is displayed below the tabs, showing the error message: 'Connection was refused by the server.'

Figura 4 - Resposta da requisição informando o erro de conexão

Por algum motivo o *Thunder Client* não consegue acessar o *localhost*. Uma forma que podemos contornar esse problema é substituir a palavra *localhost* por *[::1]*.

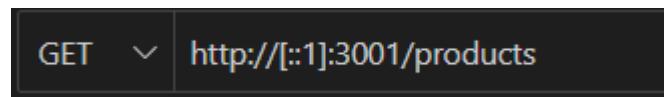


Figura 5 - Requisição para obter todos os produtos

Após a substituição, clique novamente no botão “*Send*” e obteremos como resposta um JSON com todos os produtos conforme mostrado na figura 6.

This screenshot shows the Thunder Client interface after changing the URL to 'http://[::1]:3001/products'. The 'Activity' sidebar on the left lists three previous requests: one to '[::1]:3001/products' just now, one to '127.0.0.1:3001/products' just now, and one to 'localhost:3001/products' just now. The main request details show 'Status: 200 OK', 'Size: 220 Bytes', and 'Time: 53 ms'. The 'Response' tab is active, displaying the JSON response:

```

1  [
2   {
3     "id": 1,
4     "name": "Caneta BIC preta",
5     "price": 5.89
6   },
7   {
8     "id": 2,
9   }
10 ]

```

Figura 6 - Resultado da requisição

Observe também, conforme enviamos as requisições elas são armazenadas na aba *Activity* em forma de lista, onde você poderá clicar em uma delas para enviar posteriormente ou excluir da lista.

Agora vamos enviar uma requisição para obtermos apenas o produto que tem o id igual a 1. Altere a *url* da requisição conforme a figura 7 e em seguida clique no botão “**Send**”.

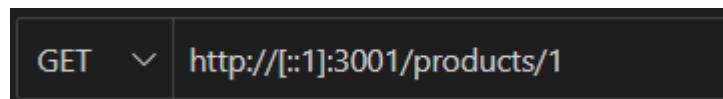


Figura 7 - Requisição para obter o produto do id igual a 1

Você verá na figura 8, a resposta trazendo apenas um produto conforme nossa requisição.

This screenshot shows the same interface as Figure 7, but after the 'Send' button has been clicked. The status bar now shows 'Status: 200 OK', 'Size: 60 Bytes', and 'Time: 23 ms'. The response body is a JSON object representing a single product:

```
1  {
2    "id": 1,
3    "name": "Caneta BIC preta",
4    "price": 5.89
5 }
```

Figura 8 - Resposta da requisição trazendo apenas um produto

Criando variáveis de ambiente

Em muitos casos quando desenvolvemos a *API*, sua *url* fica um pouco extensa como por exemplo, <https://www.meu-dominio.com/api/v1/products> até chegar no *endpoint* “**products**”. Imagine toda vez que você for enviar uma nova requisição ter que digitar esse caminho. Para resolver essa situação podemos criar uma variável de ambiente e salvar nela a *url base* e usá-la juntamente com os *endpoints*.

Clique na aba “*Env*” e em seguida no ícone de menu ao lado da palavra “**filter environment**” e depois na opção “**New Environment**” conforme demonstrado na figura 9.

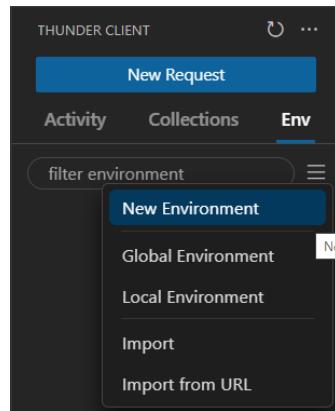


Figura 9 - Menu de opções para variáveis de ambiente

Abrirá um campo de texto acima solicitando o nome do ambiente. Escreva “**general**” e pressione **ENTER**. Irá aparecer do lado esquerdo o ambiente. Agora clique no nome do ambiente e preencha o formulário conforme a figura 10 e ao final clique no botão “**Save**”.

Variable Name	Value
url_base	http://[:1]:3001
variable	value

Link to .env file

Figura 10 - Interface de variáveis de ambiente

Dessa forma será criada uma variável de nome “**url_base**” com o valor [http://\[:1\]:3001](http://[:1]:3001) que poderemos usá-la em nossas requisições.

Para testarmos a variável, clique no botão “**New Request**” e escreva a *url* com o método *GET* conforme demonstrado na figura 11.

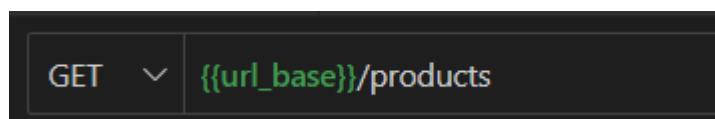


Figura 11 - Url usando uma variável

Veja que o nome da variável está envolvido por duas chaves abrindo “{{“ e duas chaves fechando “}}”. Dessa forma quando executarmos a requisição a variável será substituída pelo seu valor, nos poupando de digitarmos URLs extensas.

Esse foi apenas um exemplo da utilidade de variáveis de ambiente, para maiores detalhes e uso consulte a documentação.

Adicionando um produto

Para adicionar um produto devemos usar o método *POST* e enviar dados no corpo (*body*) da requisição por meio do *Form-encode*. Clique no botão “**New Request**” e preencha os dados da requisição conforme a figura 12 e ao final clique no botão “**Send**”.

The screenshot shows a Postman interface. At the top, there's a dropdown for 'Method' set to 'POST' and a URL field containing '{{url_base}}/products'. To the right is a blue 'Send' button. Below the URL field are tabs: 'Query', 'Headers 2', 'Auth', 'Body', 'Tests', and 'Pre Run'. The 'Body' tab is active and highlighted with a red border. Underneath are sub-tabs: 'JSON', 'XML', 'Text', 'Form', 'Form-encode' (which is also highlighted with a red border), 'GraphQL', and 'Binary'. The 'Form-encode' section is expanded, showing a table with two rows. The first row has a checked checkbox next to 'name' and the value 'Computador Desktop Dell i9'. The second row has a checked checkbox next to 'price' and the value '2755.99'. There is also an empty row with an unchecked checkbox next to 'name' and a blank 'value' field. On the right side of this table area, there is a small 'Import' link.

Figura 12 - Adicionando um produto por meio do método POST

Você terá como resposta a adição do produto conforme a figura 13 e 14.

The screenshot shows the response details for a POST request. At the top, it says 'Status: 201 Created' in green, 'Size: 75 Bytes' in green, and 'Time: 15 ms' in green. Below this, the response body is displayed as a JSON object:

```

1  {
2      "name": "Computador Desktop Dell i9",
3      "price": "2755.99",
4      "id": 4
5  }

```

Figura 13 - Resposta da requisição do método POST

The screenshot shows the content of a file named 'db.json'. The file is a JSON object with a single key 'products' which is an array of product objects. The last object in the array is highlighted with a red box:

```

backend > {} db.json > [ ] products > {} 2
1  {
2      "products": [
3          {
4              "id": 1,
5              "name": "Caneta BIC preta",
6              "price": 5.89
7          },
8          {
9              "id": 2,
10             "name": "Notebook Mac Pro",
11             "price": 12000.59
12         },
13         {
14             "id": 3,
15             "name": "Samsung 20",
16             "price": 5000.89
17         },
18         {
19             "name": "Computador Desktop Dell i9",
20             "price": "2755.99",
21             "id": 4
22     }
23

```

Figura 14 - Arquivo db.json com o produto adicionado

Alterando um dado do produto

Se quisermos alterar um dado de um produto específico, usamos o método *PATCH*, passar o dado no corpo (*body*) da requisição por meio do *Form-encode* e na *url* o id do produto conforme demonstrado na figura 15.

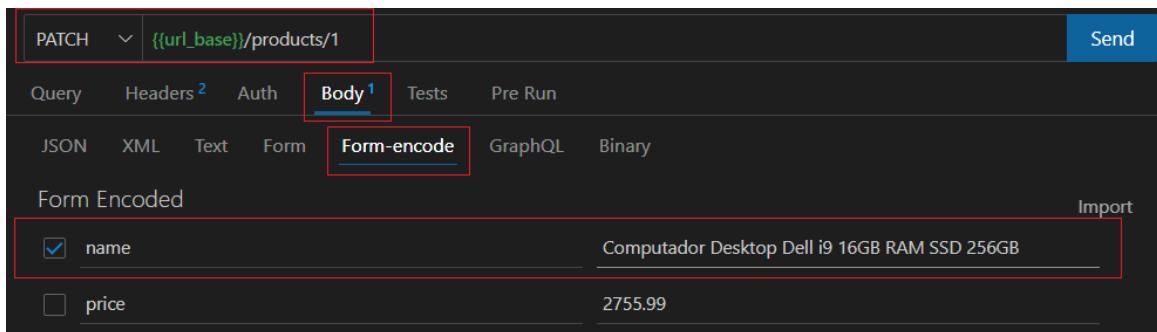


Figura 15 - Usando o método *PATCH* para alterar um dado do produto

Caso queira alterar todos os dados do produto, basta escolher o método *PUT* e passar os dados via o *Form-encode*.

Excluindo um produto

Finalizando nossa demonstração, agora iremos excluir um produto utilizando o método *DELETE* informando o id do produto. Adicione nova requisição clicando no botão “**New Request**” e deixe a *url* de acordo com a figura 16 e clique no botão “**Send**”.



Figura 16 - Método *DELETE* para excluir o produto de id igual a 4

Concluindo...

Ferramentas *REST Client* são essenciais para os desenvolvedores testarem as *APIs* em desenvolvimento ou como a situação demostrada nesse tutorial, onde utilizamos o *JSON Server* para fornecer uma *API fake* para prototipação ou apresentação do aplicativo. Existem várias ferramentas que podem ser utilizadas com essa finalidade, e aqui demonstramos de forma básica uma extensão existente no **VSCode**, o *Thunder Client*, que nos permite de forma fácil e rápida testarmos os *endpoints* sem ter que executar outro aplicativo.

Há muitas funcionalidades a serem exploradas no *Thunder Client* conforme seus conhecimentos em *REST API* vão avançando. Não pare por aqui, consulte a documentação e realize novos experimentos.

Referências

VADHINENI, R. **Thunder Client**. Disponível em:
<<https://github.com/rangav/thunder-client-support#setenv>>. Acesso em: 2 dez.
2023.



Aula 01 - Introdução a Programação de Aplicativos Mobile II

Data 01/08/2025

Versão 1.1

Etec
Bento Quirino
Campinas

PAM II – PROGRAMAÇÃO DE APLICATIVOS MOBILE II

Prof. Jackson Sá

Jackson.sa@etec.sp.gov.br

Prof. Rafael Anderson Cruz

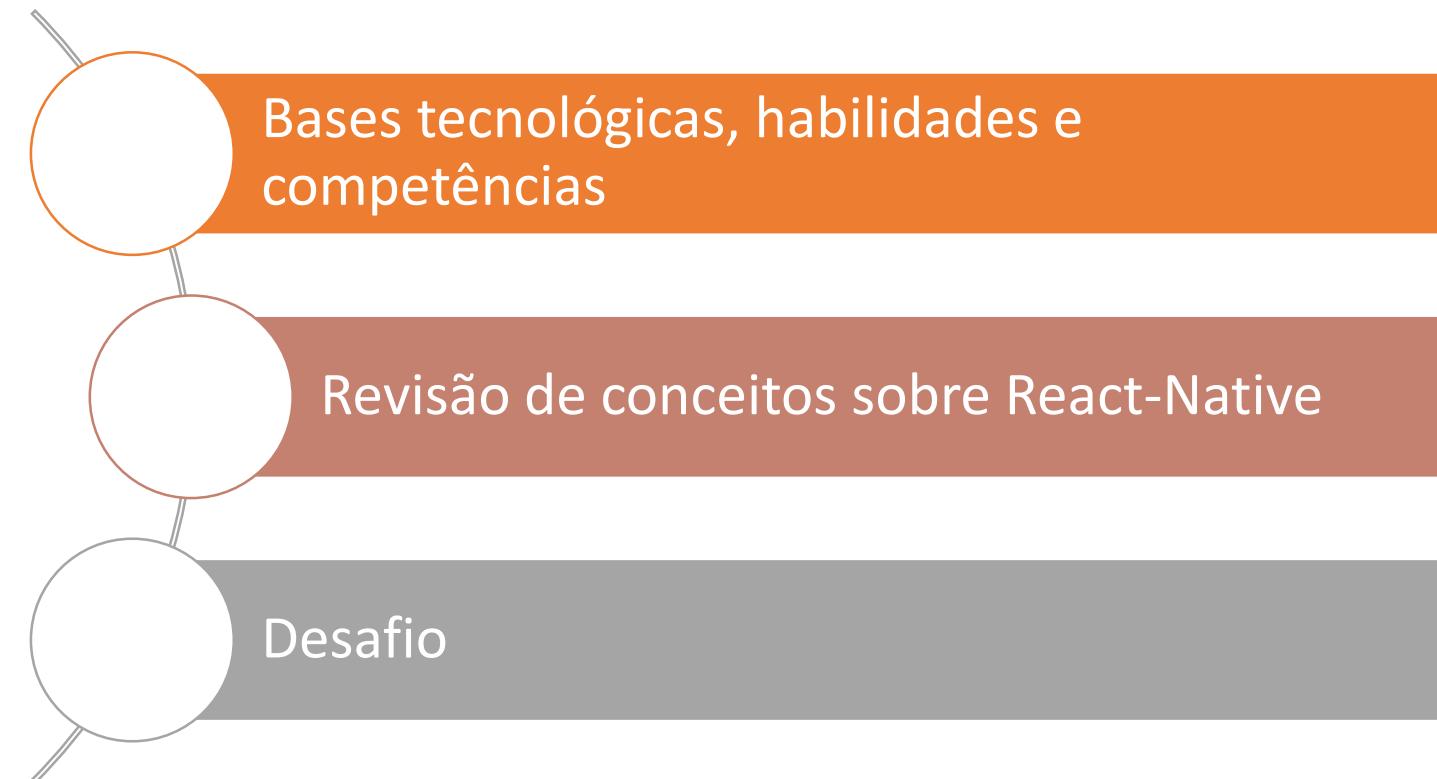
rafael.cruz43@etec.sp.gov.br

Introdução à Programação de Aplicativos Mobile II - Aula 01
(Apresentação das bases tecnológicas, habilidades e competências)

Objetivo

Nessa aula, iremos apresentar uma introdução à Programação de Aplicativos Mobile II, assim como, a sua importância dentro do curso e o que iremos aprender.

Será apresentado:



Bases tecnológicas, habilidades e competências

Bases tecnológicas

Conectividade

- Consumo de APIs REST;
- Comunicação TCP full-duplex (sockets);
- Integração com dispositivos embarcados via Bluetooth.

Autenticação

Recursos do dispositivo

- Câmera;
- Sensores;
- Localização, orientação e mapas;
- Telefonia e SMS.

Empacotamento e distribuição

Bases tecnológicas, habilidades e competências

Habilidades

- 1.1 Codificar aplicativos em tecnologia móvel.
- 1.2 Utilizar ambientes de desenvolvimento mobile.
- 1.3 Elaborar aplicativos com acesso a banco de dados.
- 1.4 Construir layout de aplicativos dispositivos móveis.
- 1.5 Utilizar recursos avançados do dispositivo (smartphones e tablets).

Competências

1. Projetar aplicativos, selecionando linguagens de programação e ambientes de desenvolvimento.

Revisão de conceitos sobre React Native

Formas de Criar Projetos React Native

React Native é um framework para desenvolvimento de aplicativos móveis utilizando JavaScript e React. Existem duas principais formas de criar um projeto React Native:

- Utilizando o **Expo** (fácil e rápido para iniciar)
- Utilizando o **React Native CLI** (mais flexível e personalizável)

Revisão de conceitos sobre React Native

Criando um Projeto com Expo

Expo é uma plataforma que facilita o desenvolvimento com React Native, eliminando a necessidade de configuração complexa do ambiente.

Passo a passo:

1. Instale o Node.js (se ainda não tiver)
2. Execute o comando no terminal:
 - **npx create-expo-app MeuProjeto –template blank**
3. Entre no diretório do projeto:
 - **cd MeuProjeto**
4. Execute o aplicativo:
 - **npx expo start (--tunnel caso esteja em redes diferentes)**
5. Escaneie o QR Code com o aplicativo Expo Go no celular para testar.

Revisão de conceitos sobre React Native

Criando um Projeto com React Native CLI

Essa abordagem é ideal para projetos que precisam de maior controle sobre os códigos nativos.

Passo a passo:

1. Instale o Node.js, Android Studio e as ferramentas nativas
2. Execute o comando no terminal:
 - **npx react-native init MeuProjeto**
3. Entre no diretório do projeto:
 - **cd MeuProjeto**
4. Execute o projeto no Android:
 - **npx react-native run-android**
5. Execute o projeto no iOS (MacOS):
 - **npx react-native run-ios**

Revisão de conceitos sobre React Native

Estrutura Básica de um Projeto React Native

Ao criar um projeto React Native, é essencial entender sua estrutura básica. Isso facilita a organização do código e a manutenção do aplicativo.

Estrutura de Pastas

Um projeto padrão do React Native criado com Expo ou React Native CLI geralmente possui a seguinte estrutura:

```
MeuProjeto/
  |-- node_modules/      # Dependências instaladas
  |-- android/           # Código nativo para Android
  |-- ios/                # Código nativo para iOS
  |-- src/
    |-- components/       # Componentes reutilizáveis
    |-- screens/           # Telas do aplicativo
    |-- assets/             # Imagens, ícones e fontes
    |-- services/           # Serviços como API requests
  |-- App.js                 # Ponto de entrada do aplicativo
  |-- package.json          # Configuração do projeto
  |-- babel.config.js       # Configuração do Babel
  |-- metro.config.js        # Configuração do Metro Bundler
```

Revisão de conceitos sobre React Native

Entendendo o App.js

O arquivo App.js é o ponto de entrada do projeto e define a interface principal. Um exemplo básico:

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

export default function App() {
  return (
    <View style={styles.container}>
      <Text>Olá, React Native!</Text>
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#f0f0f0',
  },
});
```

Revisão de conceitos sobre React Native

Importando e Exportando Componentes

Os componentes são parte essencial de um projeto React Native. Podemos criar componentes separados e importá-los no App.js:

Criando um componente em
src/components/MeuComponente.js

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

export default function MeuComponente() {
  return <Text>Este é um componente reutilizável!</Text>;
}
```

Importando no App.js

```
import MeuComponente from './src/components/MeuComponente';
```

Revisão de conceitos sobre React Native

Componentes

Existem dois tipos principais de componentes em React Native:

- Componentes funcionais (recomendados):

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

const MeuComponente = () => {
  return <Text>Componente Funcional</Text>;
};

export default MeuComponente;
```

Revisão de conceitos sobre React Native

Componentes

- Componentes de classe (menos utilizados):

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

class MeuComponente extends Component {
  render() {
    return <Text>Componente de Classe</Text>;
  }
}

export default MeuComponente;
```

Revisão de conceitos sobre React Native

Comparação Direta

Componente Funcional x Componente Classe

Característica	Funcional	Classe
Simplicidade	<input checked="" type="checkbox"/> Mais simples	<input type="checkbox"/> Mais verboso
Performance	<input checked="" type="checkbox"/> Melhor otimizado	<input type="checkbox"/> Menos eficiente
Uso de Hooks	<input checked="" type="checkbox"/> Suporta Hooks	<input type="checkbox"/> Não suporta Hooks
Código mais limpo	<input checked="" type="checkbox"/> Sim	<input type="checkbox"/> Não

Revisão de conceitos sobre React Native

Props e Hooks (useState e useEffect)

No React Native, props e Hooks são fundamentais para criar componentes reutilizáveis e gerenciar estados de maneira eficiente. Os Hooks como useState e useEffect simplificam o desenvolvimento de componentes funcionais.

Revisão de conceitos sobre React Native

O que são Props?

As props (propriedades) permitem passar dados de um componente pai para um componente filho.
Exemplo de Uso de Props

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

const Saudacao = ({ nome }) => {
  return <Text>Olá, {nome}!</Text>;
};

export default function App() {
  return <Saudacao nome="João" />;
}
```

Características das Props

- São imutáveis dentro do componente filho.
- Permitem reutilizar componentes com diferentes valores.

Revisão de conceitos sobre React Native

Hook useState

O useState permite que componentes funcionais tenham estado próprio.

Exemplo de Uso do useState

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

export default function Contador() {
  const [contador, setContador] = useState(0);

  return (
    <View>
      <Text>Contador: {contador}</Text>
      <Button title="Incrementar" onPress={() => setContador(contador + 1)} />
    </View>
  );
}
```

Características do useState

- Permite atualizar estados dentro do componente.
- Não modifica diretamente a variável de estado.

Revisão de conceitos sobre React Native

Hook useEffect

O useEffect é um Hook do React que permite executar **efeitos colaterais** em componentes funcionais. Efeitos colaterais são comportamentos que ocorrem **fora do ciclo padrão de renderização**, como:

- Buscar dados de uma API;
- Atualizar o título da telaAcessar armazenamento local;
- Adicionar ou remover event listeners;
- Atualizar o layout ou estado com base em mudanças.

Características do useState

- **Reatividade** - Executa efeitos sempre que o componente renderiza ou quando alguma dependência muda.
- **Limpeza** - Pode retornar uma função de limpeza (cleanup) usada para cancelar timers, listeners, etc.
- **Substitui o ciclo de vida** - Equivale aos métodos de classe componentDidMount, componentDidUpdate e componentWillUnmount.
- **Controle de execução** - O array de dependências permite definir quando o efeito será executado.

Revisão de conceitos sobre React Native

Hook useEffect

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

export default function App() {
  const [contador, setContador] = useState(0);

  useEffect(() => {
    console.log('O contador foi atualizado:', contador);
  }, [contador]);

  return (
    <View style={styles.container}>
      <Text style={styles.texto}>Contador: {contador}</Text>
      <Button title="Adicionar" onPress={() => setContador(contador + 1)} />
    </View>
  );
}
```

Desafio

Após a revisão dos conceitos sobre React Native, aplique o seus conhecimentos no desenvolvimento da tela abaixo.





OBRIGADO