Objetivos:

i. React Native;
ii. Conversão de RN para código nativo;
iii. Diferenças entre o RN e React para web;
iv. Plataforma Expo;
v. Criar projeto Expo;
vi. Bundlers;
vii. Estrutura de pastas e arquivos de um projeto RN criado com Expo;
viii. Estrutura de uma tela RN criado com Expo;
ix. Definição de estilos com StyleSheet;
x. Layout com Flexbox;
xi. Campos de entrada.

i. React Native

O React Native (RN) é um framework de desenvolvimento de aplicativos móveis criado pelo Facebook. Ele permite que os desenvolvedores criem aplicativos móveis nativos para iOS e Android usando JavaScript (JS) e React.

Principais características do RN:

- Código único para múltiplas plataformas: o RN nos permite escrever a maior parte do código em JS, que é compartilhado entre as plataformas iOS e Android. Isso reduz significativamente o esforço de desenvolvimento e manutenção;
- Componentes nativos: os componentes do RN são mapeados para widgets nativos, o que significa que os aplicativos têm a aparência e o desempenho de aplicativos nativos. Isso inclui componentes de UI (User Interface), como botões, textos e listas;
- Desenvolvimento rápido: com recursos como recarga em tempo real (hot reloading) e ferramentas de depuração integradas, o RN acelera o ciclo de desenvolvimento. Os desenvolvedores podem ver as mudanças instantaneamente no emulador ou no dispositivo real;
- Comunidade e ecossistema: o RN tem uma grande comunidade de desenvolvedores e uma ampla gama de bibliotecas e ferramentas de terceiros, o que facilita a adição de funcionalidades como navegação, gestão de estado, e integração com APIs externas;
- Compatibilidade com código nativo: o RN permite integrar código nativo escrito em Java (para Android) e Swift/Objective-C (para iOS), o que é útil quando precisamos de funcionalidades específicas ou desejamos otimizar partes do nosso aplicativo.

Desvantagens do RN:

- Desempenho: embora seja muito próximo ao nativo, ainda pode haver casos em que o desempenho não é tão bom quanto os aplicativos nativos puros, especialmente para aplicativos muito complexos ou que requerem uso intensivo de recursos;
- Dependência de bibliotecas de terceiros: muitas funcionalidades podem depender de bibliotecas de terceiros, que podem não ser tão bem mantidas quanto o código nativo.

Embora o RN ofereça muitas vantagens, existem alguns casos em que um aplicativo RN pode não ser tão bom quanto um aplicativo nativo. Aqui estão alguns exemplos desses casos:

1. Desempenho e uso intensivo de recursos:
    - Aplicativos de jogos complexos: jogos de alta performance que exigem gráficos complexos e uso intensivo de GPU geralmente se beneficiam mais de código nativo otimizado para cada plataforma;
    - Aplicativos de realidade aumentada/virtual: aplicativos que utilizam AR ou VR exigem um alto nível de desempenho e integração com hardware que é mais facilmente alcançado com código nativo.
2. Integração profunda com hardware:
    - Funcionalidades específicas de hardware: recursos que exigem integração profunda com hardware específico, como processamento de imagens em tempo real, manipulação avançada de câmera, ou acesso a sensores específicos, são geralmente mais eficazes quando implementados em código nativo;
    - Bluetooth e conectividade avançada: aplicativos que necessitam de conectividade avançada, como comunicação com dispositivos via Bluetooth Low Energy (BLE), podem ter limitações ao usar RN devido à dependência de bibliotecas de terceiros.
3. Acesso a APIs e funcionalidades recentes:
    - APIs nativas recém-lançadas: quando novas APIs ou funcionalidades são lançadas para Android ou iOS, pode levar algum tempo para que essas APIs estejam disponíveis em RN. O desenvolvimento nativo permite acesso imediato a essas funcionalidades;
    - Funcionalidades exclusivas de plataforma: alguns recursos exclusivos de uma plataforma (como certos serviços de Google Play no Android ou algumas funcionalidades específicas do iOS) podem não ser facilmente acessíveis ou totalmente suportados no RN.
4. Desempenho de animações complexas:
    - Animações e interações complexas: animações muito complexas ou interações que exigem uma renderização suave e altamente responsiva podem não ter o mesmo desempenho em RN, especialmente em dispositivos mais antigos ou menos poderosos.
5. Tamanho do aplicativo:

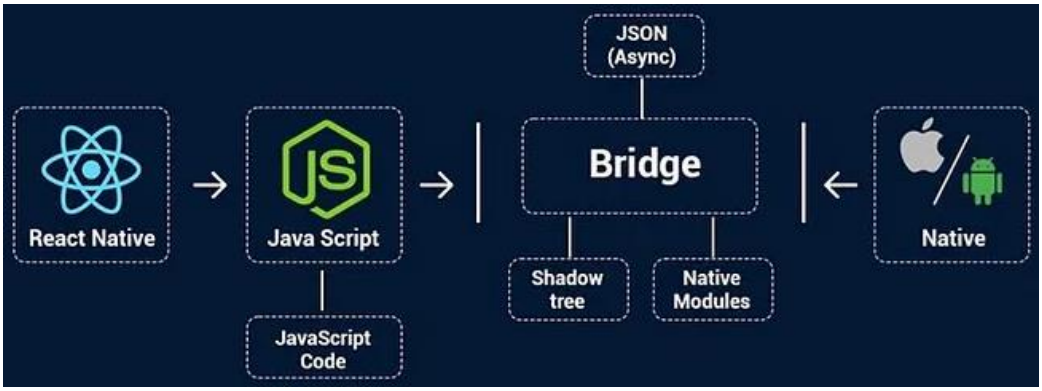
- Aplicativos com muitas dependências: o uso de muitas bibliotecas de terceiros pode aumentar o tamanho do aplicativo. Embora isso possa acontecer com aplicativos nativos também, as dependências de bibliotecas de terceiros em RN podem exacerbar o problema.

6. Consistência de UI:
   - UI altamente personalizada: aplicativos que requerem uma interface de usuário altamente personalizada e que seguem estritamente as diretrizes de design de cada plataforma podem ser mais facilmente desenvolvidas em código nativo, garantindo uma aparência e comportamento que atendam precisamente às expectativas dos usuários de cada plataforma.
7. Suporte e manutenção de bibliotecas:
   - Manutenção de bibliotecas de terceiros: algumas bibliotecas de terceiros em RN podem não ser tão bem mantidas quanto o código nativo. Problemas de compatibilidade ou bugs podem surgir, exigindo que os desenvolvedores invistam tempo para resolver ou encontrar alternativas.

ii. Conversão de RN para código nativo

A conversão de código RN para código nativo ocorre através de uma combinação de técnicas que permitem que o JS interaja com APIs nativas de cada plataforma (iOS e Android). Aqui estão os principais componentes e o processo envolvido:

1. React Native Bridge: o coração do RN é a Bridge (ponte), que atua como um intermediário entre o código JS e o código nativo. O processo funciona da seguinte maneira:



Fonte: https://diwakersingh31.medium.com/react-native-bridge-android-0bdd17d9dd44

   a) Execução do JS: o código JS é executado em uma máquina virtual JS (como JavaScriptCore no iOS ou Hermes no Android);
   b) Comunicação via Ponte (Bridge): a ponte permite a comunicação assíncrona entre o JS e o código nativo. Quando uma operação precisa ser executada no lado nativo (como exibir um componente de UI), o JS envia uma mensagem através da ponte;
   c) Execução de código nativo: a mensagem é recebida pelo código nativo, que então executa a operação correspondente (por exemplo, renderizar um componente de UI nativo);

d) Retorno de resultados: qualquer resultado ou resposta do código nativo é enviado de volta ao JS através da mesma ponte.

2. Componentes e views: o RN utiliza componentes para definir a UI. Esses componentes são mapeados para os componentes nativos correspondentes de cada plataforma:
   - Componentes Básicos: componentes básicos como <View>, <Text>, <Image> são mapeados diretamente para as views nativas correspondentes em iOS (UIView, UILabel, UIImageView) e Android (View, TextView, ImageView);
   - Custom components: podemos criar componentes personalizados que encapsulam lógica e visualizações específicas. Esses componentes podem ser escritos em JS e, se necessário, integrar-se a componentes nativos através da bridge.
3. Estilo e layout:
   - StyleSheet: define estilos de maneira semelhante ao CSS, que são então convertidos para os atributos de layout nativos correspondentes;
   - Flexbox: o RN implementa o Flexbox para o layout, que é traduzido para as propriedades de layout nativas de cada plataforma.
4. Eventos e interatividade: eventos de usuário como toques, gestos e rolagem são gerenciados pelo RN e traduzidos para eventos nativos:
   - Event handlers: manipuladores de eventos como onPress, onChangeText são vinculados a eventos nativos correspondentes e comunicados de volta ao JS através da bridge;
   - Gestures: bibliotecas como react-native-gesture-handler facilitam a integração de gestos complexos e interações com a interface do usuário.
5. Bibliotecas de terceiros: o RN possui uma ampla gama de bibliotecas que fornecem acesso a funcionalidades nativas:
   - Bibliotecas externas: muitas bibliotecas oferecem integração nativa pronta, o que facilita a utilização de funcionalidades nativas sem precisar escrever muito código nativo;
   - Linking: ferramentas como react-native link (ou a configuração automática do Expo) ajudam a integrar essas bibliotecas ao seu projeto.

O RN está em transição do modelo de bridge para o novo sistema chamado Fabric, que oferece uma comunicação mais direta, desempenho aprimorado e maior flexibilidade. Fabric está sendo adotado progressivamente, e os desenvolvedores são incentivados a seguir as orientações oficiais para tirar proveito das novas funcionalidades e melhorias de desempenho que ele oferece.

Shadow Tree: é uma estrutura de dados interna que o RN usa para gerenciar a interface do usuário (UI). O código em RN é definido através de componentes e suas propriedades no JS. Esses componentes são convertidos em uma árvore de componentes nativos, conhecida como "shadow tree".

Funções e benefícios do Shadow Tree:

1. Gerenciamento de layout: o shadow tree é responsável por calcular o layout de todos os componentes da UI. Ele usa o algoritmo de layout Flexbox para determinar a posição e o tamanho dos componentes;
2. Intermediação entre JS e nativo: o shadow tree serve como uma representação intermediária dos componentes JS antes de serem convertidos em componentes nativos. Ele facilita a comunicação eficiente entre o código JS e a UI nativa;
3. Performance: ao gerenciar o layout e outras operações de UI no shadow tree, o RN pode reduzir a quantidade de operações que precisam ser executadas diretamente na UI thread (thread principal), melhorando o desempenho e a suavidade da aplicação;
4. Isolamento: o shadow tree permite que o layout e outras operações sejam calculados de forma independente do thread de UI principal, reduzindo a latência e melhorando a responsividade da aplicação.

Native Modules: são componentes que permitem que o código JS interaja com APIs nativas do sistema operacional (iOS e Android). Eles são essenciais para acessar funcionalidades específicas do dispositivo que não estão disponíveis diretamente no JS.

Funções e Benefícios dos Native Modules:

1. Acesso a funcionalidades nativas: permitem que o código JS chame funcionalidades específicas do sistema operacional, como acesso ao GPS, câmera, armazenamento etc.;
2. Interoperabilidade: os native modules permitem que desenvolvedores escrevam código nativo em Java (para Android) ou Objective-C/Swift (para iOS) e exponham essas funcionalidades para o JS através de uma interface consistente;
3. Performance: alguns cálculos ou operações podem ser mais eficientes quando executados no código nativo, e os native modules permitem essa otimização;
4. Extensibilidade: permitem que os desenvolvedores estendam as capacidades do RN, integrando bibliotecas e SDKs nativos que não têm uma contraparte JS.

iii. Diferenças entre o RN e React para web

O RN e React para web compartilham a mesma filosofia básica de desenvolvimento de interfaces de usuário com componentes reutilizáveis, mas diferem em muitos aspectos devido às suas diferentes plataformas-alvo. Aqui estão as principais diferenças entre os dois:

1. Plataforma-alvo:

- React:
  - Destinado ao desenvolvimento de aplicações web;
  - O DOM (Document Object Model) é usado para representar e manipular a interface do usuário.
- React Native:
  - Destinado ao desenvolvimento de aplicações móveis para iOS e Android;
  - Utiliza APIs nativas do SO para renderizar componentes, em vez do DOM.

2. Componentes de interface de usuário:
   - React:
     - Usa elementos HTML e componentes web;
     - Exemplo: , , <button> etc.
   - React Native:
     - Usa componentes específicos do RN que são mapeados para componentes nativos de cada plataforma;
     - Exemplo: <View> (mapeado para UIView no iOS e View no Android), <Text>, <Image>, <TouchableOpacity> etc.
3. Estilização:
   - React:
     - Utiliza CSS para estilizar componentes;
     - Pode usar frameworks e pré-processadores CSS como SASS, LESS ou Styled Components.
   - React Native:
     - Utiliza um objeto de estilo semelhante ao CSS, mas as propriedades e valores são ligeiramente diferentes e específicos para mobile;
     - Usa a API StyleSheet para criar estilos.
4. Navegação:
   - React:
     - Utiliza bibliotecas de roteamento como react-router para navegação entre páginas.
   - React Native:
     - Utiliza bibliotecas específicas para navegação em aplicativos móveis, como react-navigation ou React Native Navigation.
5. Manipulação de Layout:
   - React:
     - Usa o modelo de layout do CSS, incluindo Flexbox, Grid etc.
   - React Native:
     - Usa Flexbox para layout, mas não possui suporte para CSS Grid;
     - O layout é gerenciado usando o sistema de estilo baseado em Flexbox específico do RN.

6. Acesso a funcionalidades nativas:
   - React:
     - Acesso limitado a funcionalidades nativas do dispositivo, principalmente através de APIs web padrão (por exemplo, geolocalização, câmera, armazenamento local).
   - React Native:
     - Acesso completo às funcionalidades nativas do dispositivo através de módulos nativos (por exemplo, GPS, câmera, contatos etc.);
     - Permite escrever módulos nativos em Swift/Objective-C para iOS e Java/Kotlin para Android.
7. Ferramentas e ambiente de desenvolvimento:
   - React:
     - Desenvolvedores geralmente usam navegadores e ferramentas de desenvolvimento web como Chrome DevTools;
     - Bundlers como Webpack são comumente usados.
   - React Native:
     - Usa o Metro bundler para empacotamento e desenvolvimento;
     - Ferramentas específicas como Expo podem ser usadas para simplificar o desenvolvimento e a implantação;
     - Teste e depuração são realizados em emuladores ou dispositivos reais.
8. Desempenho:
   - React:
     - O desempenho pode ser limitado pela natureza interpretada do JS no navegador e pelo modelo de renderização do DOM.
   - React Native:
     - Melhor desempenho para interfaces de usuário complexas e animações, pois os componentes são renderizados usando APIs nativas;
     - O código JS é executado em uma thread separada e se comunica com o código nativo através de uma bridge.
9. Ecossistema e bibliotecas:
   - React:
     - Possui um vasto ecossistema de bibliotecas e componentes para praticamente qualquer necessidade.
   - React Native:
     - Também possui um ecossistema rico, mas com bibliotecas específicas para mobile, algumas das quais precisam de módulos nativos adicionais.
10. Ciclo de vida de componentes:

- React e React Native:
  - Compartilham o mesmo ciclo de vida de componentes, hooks (como useState, useEffect), e o sistema de gerenciamento de estado (como Redux ou Context API).

iv. Plataforma Expo

O Expo é uma plataforma que simplifica o desenvolvimento e a implantação de aplicativos RN. Ele oferece um conjunto de ferramentas, bibliotecas e serviços que facilitam a criação, construção e distribuição de aplicativos móveis. O ecossistema Expo abrange várias áreas-chave do desenvolvimento de aplicativos, incluindo configuração, desenvolvimento, teste, construção e implantação.

Principais partes do ecossistema Expo:

1. Expo CLI:
   - Ferramenta de linha de comando para criar e gerenciar projetos Expo.
2. Expo SDK:
   - Conjunto de bibliotecas e APIs que fornecem acesso a funcionalidades nativas do dispositivo, como câmera, localização, notificações push, sensores etc.;
   - É integrado diretamente ao Expo e facilita o uso dessas funcionalidades sem necessidade de escrever código nativo.
3. Expo Go:
   - Aplicativo disponível na App Store e Google Play que permite aos desenvolvedores e testadores executem aplicativos Expo em dispositivos reais;
   - Facilita a visualização de mudanças de código em tempo real sem a necessidade de recompilar ou reinstalar o aplicativo.
4. Expo Snack (https://snack.expo.dev):
   - Editor de código online que permite experimentar e compartilhar rapidamente código RN/Expo diretamente no navegador;
   - Integração com Expo Go para testar o código em dispositivos reais.

v. Criar projeto Expo

1. Instalação do pacote `create-expo-app`: a ferramenta create-expo-app é uma CLI (Command Line Interface) projetada para facilitar a criação e configuração de novos projetos Expo. Ela oferece uma maneira rápida e simplificada de começar a desenvolver aplicativos RN com Expo, substituindo o antigo expo-cli para a criação de projetos:

```
npm i create-expo-app -g
npm list -g
```

2. Criação do projeto: o comando a seguir criar um projeto Expo com uma configuração.

   `expo-template-blank-typescript` é um template que configura o projeto com suporte a TS e um ambiente inicial vazio:

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

   Será criada uma pasta de nome `app`. Abra essa pasta no VS Code.

3. Assim como definido na propriedade `scripts` do `package.json`, podemos rodar o projeto em android, ios e navegador (web):

   ```
   "scripts": {
       "start": "expo start",
       "android": "expo start --android",
       "ios": "expo start --ios",
       "web": "expo start --web"
     }
   ```

   Inicialmente testaremos o aplicativo no navegador. Ao digitar o comando `npm run web` será mostrada a mensagem dizendo que precisaremos instalar alguns pacotes. A seguir tem-se um exemplo do comando sugerido, porém, recomenda-se usar o comando sugerido por ele, ou seja, não copie esse aqui:

   ```
   npx expo install react-native-web react-dom @expo/metro-runtime
   ```

   Execute o comando `npm run web` novamente após instalar os pacotes sugeridos. O aplicativo pode ser testado no navegado usando a URL sugerida (http://localhost:8081) ou abra o aplicativo no Expo Go escaneando o QR Code.

   Para executar o aplicativo para Android o Expo CLI precisa usar a instalação do Android SDK (Software Development Kit) no seu computador. O Android SDK inclui ferramentas para compilar, construir e depurar aplicativos Android e fornece emuladores que permitem testar aplicativos Android em um ambiente virtual antes de implantar em um dispositivo físico. O Expo CLI usa essas ferramentas para construir, testar e depurar aplicativos Android. Porém, não usaremos Android SDK para emular dispositivos.

   Use o comando `npm run web` ou `npm start` para subir a aplicação e escaneie o QR code usando a câmera do celular para testar o aplicativo no Expo Go do seu dispositivo.

vi. Bundlers

Na construção de aplicativos usando frameworks como RN e Expo, os bundlers (empacotadores) desempenham papéis distintos em relação ao processamento e à preparação do código para as diferentes plataformas.

Metro bundler:

- Papel: é o empacotador de módulos usado pelo RN para aplicativos móveis. Ele é responsável por compilar e agrupar todos os módulos JS do aplicativo em um único arquivo JS que pode ser carregado pelo aplicativo móvel;
- Funções:
  - Transformação de código: o Metro converte o código JS moderno (incluindo ES6+) e JSX (sintaxe usada pelo React) em um formato que pode ser compreendido pelos ambientes de execução JS em dispositivos móveis;
  - Empacotamento de módulos: agrupa todos os módulos JS e dependências em um único arquivo para otimizar o carregamento do aplicativo;
  - Atualizações e hot reloading: suporta atualização rápida e hot reloading para que possamos ver as mudanças no código quase instantaneamente sem precisar recompilar o aplicativo inteiro;
  - Resolução de dependências: gerencia e resolve dependências de módulos JS, garantindo que todas as dependências sejam incluídas no pacote final.
- Uso: o Metro Bundler é automaticamente utilizado quando executamos o comando `expo start` para iniciar o servidor de desenvolvimento.

Web bundler:

- Papel: é utilizado para criar pacotes otimizados de código JS para a web. Em um contexto de desenvolvimento com RN para Web ou projetos Expo que suportam web, o Web Bundler é responsável por preparar o código para execução em navegadores.
- Funções:
  - Transformação de código: transforma código JS moderno e JSX para um formato compatível com navegadores web;
  - Empacotamento e minificação: agrupa e minifica o código JS e os recursos estáticos (como CSS e imagens) para otimizar o desempenho no navegador;
  - Resolução de dependências: resolve e inclui todas as dependências necessárias para a execução do aplicativo no ambiente web;
  - Hot reloading: suporta hot reloading para desenvolvimento rápido na web, semelhante ao que é suportado pelo Metro Bundler para dispositivos móveis.
- Uso: o Web Bundler é usado automaticamente quando executamos comandos como expo start para a versão web do projeto, ou quando usamos ferramentas como Webpack em projetos React que são compilados para a web.

vii. Estrutura de pastas e arquivos de um projeto RN criado com Expo

A estrutura do projeto é organizada para facilitar o desenvolvimento e a gestão do código, mantendo a simplicidade e a integridade do ambiente de desenvolvimento. A seguir está uma visão geral da estrutura típica de um projeto Expo, incluindo os principais diretórios e arquivos encontrados

Descrição dos arquivos e pastas:

- app.json: arquivo de configuração principal para o Expo. Define configurações do projeto, como nome, ícone e splash screen, além de outros ajustes específicos do Expo;
- app.config.js: arquivo de configuração adicional que pode ser usado para configurações dinâmicas ou que exigem lógica personalizada. É uma alternativa ao app.json e permite configuração via JS;
- babel.config.js: configuração do Babel, o compilador JS que permite a utilização de sintaxes modernas (como JSX e TS) no projeto;
- package.json: arquivo que define as dependências do projeto, scripts de execução e metadados sobre o projeto;
- App.tsx: o componente principal do aplicativo, onde a estrutura de navegação e os componentes principais são definidos. Este é o ponto de entrada para a aplicação;
- node_modules/: diretório onde as dependências do projeto são instaladas. Não deve ser modificado manualmente;
- assets/: contém arquivos estáticos como imagens, fontes e outros recursos que são usados no aplicativo;
- components/: diretório para componentes React reutilizáveis e customizados. Componentes podem ser organizados em subdiretórios conforme necessário;
- constants/: contém constantes que são usadas em todo o projeto, como cores e valores fixos;
- hooks/: contém hooks personalizados que encapsulam lógica reutilizável. Usado para compartilhar lógica entre componentes de forma limpa e modular;
- navigation/: diretório para a configuração de navegação do aplicativo, geralmente usando bibliotecas como React Navigation. Inclui definidores de navegação, como stacks, tabs e drawers;
- screens/: contém os componentes de tela ou páginas principais do aplicativo. Cada tela geralmente corresponde a uma rota na navegação do aplicativo;
- services/: contém serviços que interagem com APIs externas, BD ou outras funcionalidades backend;
- utils/: contém funções utilitárias e helpers que são usados em várias partes do projeto.

viii. Estrutura de uma tela RN criado com Expo

A propriedade `"main": "expo/AppEntry.js"` no arquivo `package.json` de um projeto Expo especifica o ponto de entrada do aplicativo. O arquivo `AppEntry.js` é responsável por inicializar o ambiente Expo e registrar o componente principal do aplicativo, permitindo que o Expo gerencie e execute o aplicativo corretamente. Essa

configuração é parte fundamental do fluxo de inicialização do Expo e garante que a aplicação seja carregada e executada de maneira adequada.

Exemplo simplificado do que pode estar dentro do arquivo `expo/AppEntry.js`:

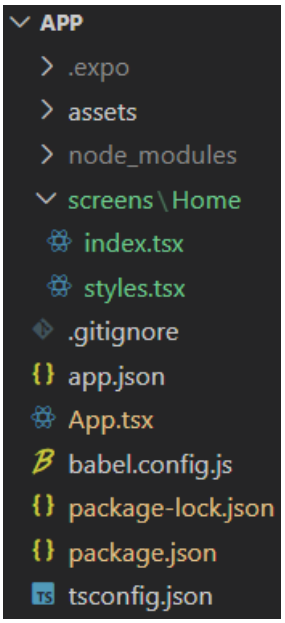
```
import { registerRootComponent } from 'expo';
import App from './App';
registerRootComponent(App); // Registro do componente principal
```

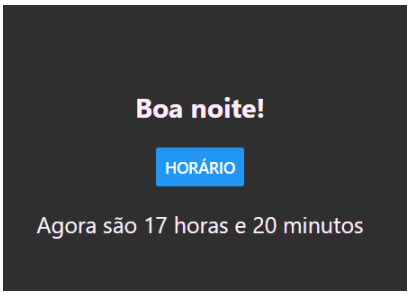
No contexto de um projeto Expo, o arquivo `AppEntry.js` mencionado não é um arquivo que normalmente veremos ou precisamos gerenciar diretamente. Em vez disso, ele é parte da configuração interna do Expo e faz parte do processo de construção e empacotamento que o Expo realiza.

Quando criamos um projeto com Expo, o Expo cria uma estrutura que inclui seu ponto de entrada, mas de uma forma que é abstraída do desenvolvedor. A configuração do projeto e a entrada são gerenciadas por Expo internamente e não são expostas diretamente em sua estrutura de arquivos padrão.

Na prática os arquivos `App.js` ou `App.tsx` servem como o ponto de entrada para o nosso aplicativo.

A estrutura de uma tela (screen) em um projeto RN pode variar dependendo das preferências do desenvolvedor e das práticas do projeto. No entanto, uma estrutura típica segue o padrão dos componentes React. Como exemplo, a figura ao lado mostra a estrutura do projeto usado para criar a seguinte tela.





Na estrutura proposta o componente `App` faz a chamada do componente `Home` (tela).

Arquivo: App.tsx

```
import Home from './screens/Home';

const App: React.FC = () => {
  return <Home />;
};

export default App;
```

O componente Home, definido na pasta Home, separa os estilos (CSS) do código JSX. Porém, essa separação é uma questão de gosto.

Arquivo: Home/index.tsx

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

const Home: React.FC = () => {
  const [time,setTime] = useState("");

  useEffect( () => {
    showTime();
  });

  const showTime = () => {
     setTime(`Agora são ${new Date().getHours()} horas e ${new Date().getMinutes()}
minutos`);
  };

  return (
    <View style={styles.container}>
      <Text style={styles.title}>Boa noite!</Text>
      <Button title="Horário" onPress={showTime} />
      <Text style={styles.msg}>{time}</Text>
    </View>
  );
};

export default Home;
```

Arquivo: Home/styles.tsx

```
import { StyleSheet } from 'react-native';

const styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'column',
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: 'rgb(48, 47, 47)',
  },
  title: {
    display: 'flex',
    fontSize: 24,
    fontWeight: 'bold',
    marginBottom: 20,
    color: '#fef'
  },
```

```
  msg: {
    fontSize: 20,
    marginTop: 20,
    color: '#fef'
  },
});

export default styles;
```

ix. Definição de estilos com StyleSheet

Embora o StyleSheet seja frequentemente descrito como uma classe na documentação, tecnicamente é uma API que oferece um método para criar e gerenciar estilos.

A principal função da API StyleSheet é fornecer o método `create`. O `create` recebe um objeto JSON com os estilos, que são transformados em um formato que o motor do RN pode processar mais rapidamente. Em vez de criar objetos de estilo dinamicamente em cada renderização, o `StyleSheet.create` permite que os estilos sejam definidos uma vez e referenciados depois.

O `StyleSheet` suporta diversas propriedades de estilo que são semelhantes às usadas no CSS, adaptadas para o RN:

- Layout: flex, justifyContent, alignItems, margin, padding, position, top, left etc.;
- Tipografia: fontSize, fontWeight, color, textAlign, lineHeight etc.;
- Estilo de fundo: backgroundColor, borderRadius, borderWidth, borderColor etc.;
- Dimensões e posicionamento: width, height, minWidth, minHeight etc.

No RN, os estilos são definidos usando a notação *camel case* em vez da notação com hífen (-) usada no CSS para web. Essa escolha de notação é usada para garantir consistência com a sintaxe JS, simplificar a acessibilidade das propriedades e evitar ambiguidades, visto que o traço/hífen é a representação do operador de subtração.

Os estilos são aplicados nos componentes usando a propriedade `style`:

```
<Text style={styles.title}>Boa noite!</Text>
```

> Estilos em RN são definidos usando objetos JS e a API StyleSheet, ou seja, não há uma estrutura de classes como no CSS.

x. Layout com Flexbox

O RN adota o modelo de layout flexbox, que é uma técnica amplamente utilizada para construir layouts responsivos e flexíveis no CSS.

- Propriedade `display: flex`: é a configuração padrão para todos os componentes de container, ou seja, não precisamos explicitamente definir `display: flex` nos estilos;
- Principais propriedades do Flexbox:

- flexDirection: define a direção dos itens no container (row, column);
- justifyContent: alinha os itens ao longo do eixo principal (flex-start, center, flex-end, space-between, space-around);
- alignItems: alinha os itens ao longo do eixo transversal (flex-start, center, flex-end, stretch);
- alignSelf: permite que um item específico se alinhe de forma diferente em relação aos outros itens no container;
- flex: define como um item deve crescer em relação aos outros itens dentro do container.

O componente `View` é um dos componentes fundamentais para construir a estrutura de uma tela. Ele atua como um container que pode agrupar e organizar outros componentes, servindo como uma base para o layout e a estrutura da interface do usuário.

xi. Campos de entrada

O componente `TextInput` é usado para ler a entrada de texto do usuário (https://reactnative.dev/docs/textinput).

```
<TextInput
    onChangeText={setAge}
    value={age}
    placeholder="Idade"
    keyboardType="numeric"
    style={styles.input}
    maxLength = {2}
  />
```

Algumas propriedades do componente TextInput:

- value: é usada para indicar o valor a ser exibido no campo de entrada;
- placeholder: é um texto exibido no campo de entrada quando a propriedade value está vazia;
- keyboardType: determina o teclado a ser aberto, só funciona no dispositivo. Os valores possíveis são: default, number-pad, decimal-pad, numeric, email-address e phone-pad;
- maxLength: número máximo de caracteres que podem ser fornecidos. O valor precisa ser um número.

O componente `Button` cria um botão simples (https://reactnative.dev/docs/button).

O evento `onPress` é disparado ao pressionar o botão. No código a seguir será chamada a função add.

```
<Button title="Horário" onPress={showTime} />
```

O componente `Switch` cria um campo de escolha sim/não (https://reactnative.dev/docs/switch). Ele aceita as seguintes propriedades:

- value: possui o valor de início, precisa ser um valor true/false;

- onValueChange: precisa receber uma função que inverte o valor atual, ou seja, se for true para a ser false.

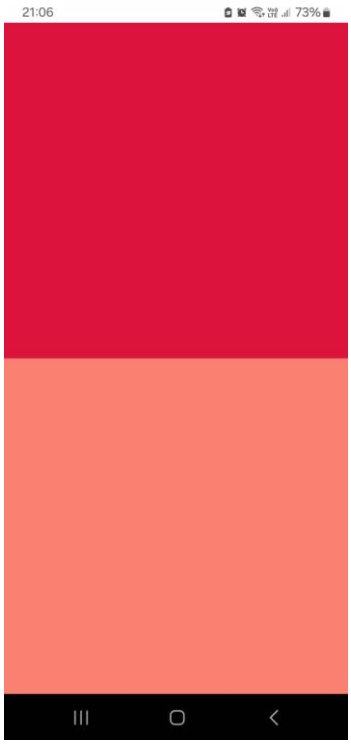
```
const [visible, setVisible] = useState(false);
const change = () => setVisible((previousState) => !previousState);
<Switch
        trackColor={{ false: "yellow", true: "cyan" }}
        thumbColor="red"
        onValueChange={change}
        value={visible}
      />
```

A variável `change` possui uma função que inverte o estado atual de `true` para `false` e o contrário.

Exercícios

Para cada exercício, adicione uma tela na pasta screens do projeto criado no exemplo. Neste exemplo o componente App chama o componente Home.

Exercício 1 – Fazer um aplicativo RN com a tela dividida em duas partes na vertical.



Requisitos:

- Crie um componente na pasta screens de nome Um;
- Altere o componente App para chamar o componente Um;
- Excluir da tela a status bar do dispositivo. Veja na figura ao lado que a StatusBar não faz parte da tela do aplicativo

Dicas:

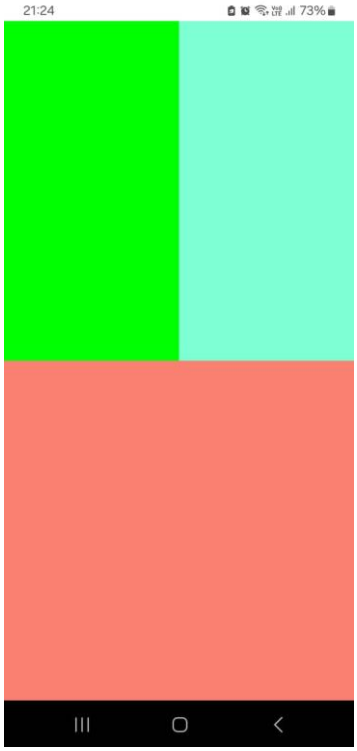
- Crie um componente container pai usando View;
- Crie dois componentes filhos usando View;
- Use a propriedade flexDirection com valor column para tornar o eixo principal na coluna. Desta forma, os elementos filhos serão posicionados um abaixo do outro;
- Use a propriedade flex com valor 0.5 (50%) em cada componente filho;
- Use a propriedade backgroundColor com os valores crimson e salmon;
- Importe Constants do módulo expo-contants para ter acesso a informações constantes do dispositivo (https://docs.expo.dev/versions/latest/sdk/constants/);

- Use a propriedade paddingTop com valor Constants.statusBarHeight na View pai para deslocar o início da parte superior do aplicativo.

Exercício 2 – Alterar o aplicativo do Exercício 1 para dividir a tela assim como é mostrado ao lado.

Dicas:

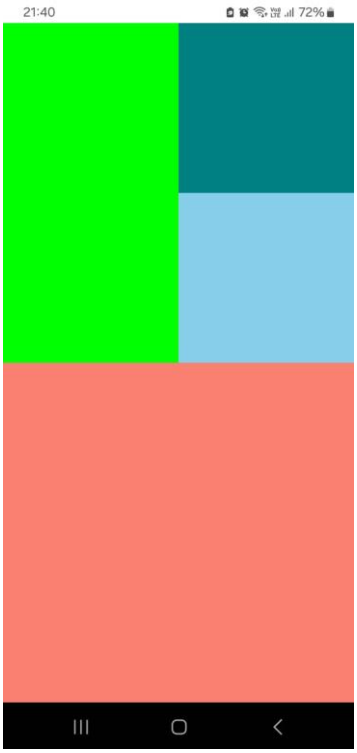
- Adicione dois componentes View filhos do componente que possui a cor crimson do Exercício 1;
- Use a propriedade flexDirection com valor row no componente que possui a cor crimson para tornar o eixo principal na linha. Desta forma, os elementos filhos serão posicionados um à direita do outro;
- Use a propriedade flex com valor 0.5 (50%) em cada componente filho;
- Use as cores lime e aquamarine.



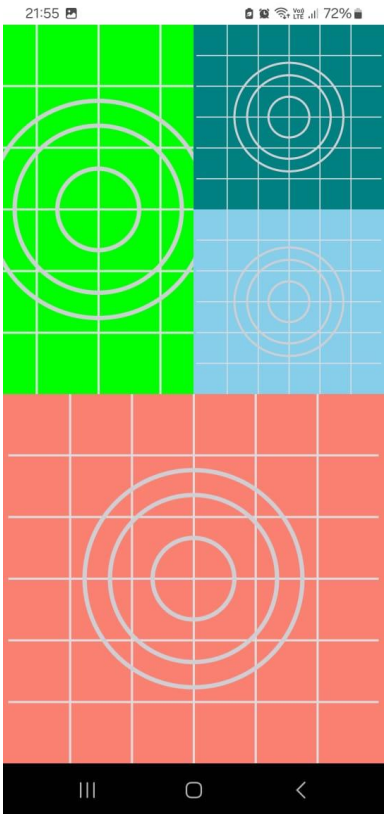
Exercício 3 – Alterar o aplicativo do Exercício 2 para dividir a tela assim como é mostrado ao lado.

Dica:

- Use as cores teal e skyblue.



Exercício 4 – Adicionar a imagem no aplicativo do Exercício 3.



Dica:

- Use o componente Image para exibir a imagem (https://reactnative.dev/docs/image);
- Use a instrução a seguir para importar o arquivo png na variável `logo`:
  `import logo from "../../assets/adaptive-icon.png";`
- Use a propriedade alignSelf:center para alinhar a imagem no centro;
- Use a propriedade flex:1 para a imagem ocupar toda a área;
- Use a propriedade resizeMode com valor contain para adequar as dimensões da imagem (https://reactnative.dev/docs/image#resizemode).

Observação:

- A importação do arquivo PNG na variável `logo` pode causar erro.
  `import logo from "../../assets/adaptive-icon.png";`
  O erro ocorre porque o TypeScript, usado em arquivos .tsx, não tem informações de tipo para módulos de imagem como PNGs, o que é necessário para que o linting e o sistema de tipos do TS reconheçam e processem corretamente esses arquivos.
  Para resolver o erro crie um arquivo chamado `declarations.d.ts` (ou qualquer nome com a extensão `.d.ts`) na pasta `types`. Adicione o seguinte código no arquivo `declarations.d.ts` para declarar módulos de imagem PNG:

```
declare module "*.png" {
  const value: any;
  export default value;
}
```

  Isso informa ao TS que qualquer importação de arquivos com a extensão .png deve ser tratada como um módulo que exporta qualquer valor.

Exercício 5 – Transformar as imagens do Exercício 4 em botões clicáveis. Ao clicar no botão será exibida uma mensagem de alerta com o texto “Boa noite!”.

Dicas:

- Use o componente TouchableOpacity para criar uma área clicável (https://reactnative.dev/docs/touchableopacity);
- Coloque as dimensões das imagens em 64x64;
- Use a propriedade justifyContent: "center" e alignItems: "center", nos componentes pais, para centralizar as imagens;
- Use o componente Alert para exibir a janela de alerta ao clicar no botão TouchableOpacity (https://reactnative.dev/docs/alert);
- Use o evento onPress do componente TouchableOpacity para exibir a janela de alerta.