Interface gráfica com React Native

Apresentação dos componentes de interface gráfica disponíveis no framework de desenvolvimento mobile React Native.

Profª. Alexandre Paixão

Propósito

Conhecer os componentes de interface gráfica disponíveis no framework React Native para o desenvolvimento de aplicativos mobile.

Preparação

Para o acompanhamento do conteúdo e a codificação dos exemplos a serem apresentados ao longo deste conteúdo, será necessária a utilização de uma IDE (sigla de *integrated development environment*). Recomenda-se o Visual Studio Code, que é um software gratuito. Além disso, será preciso configurar o ambiente de desenvolvimento e testes no qual diferentes configurações e ferramentas poderão ser usadas. Para mais detalhes a respeito dessa etapa, o site oficial do React Native deverá ser consultado. Os códigos dos exemplos em React Native estarão disponibilizados ao longo do conteúdo em formato texto. Desse modo, eles poderão ser facilmente copiados e colados na interface do ambiente de desenvolvimento para sua execução e validação.

Objetivos

- Esquematizar os passos para a definição da interface interativa de um aplicativo mobile.
- Descrever os componentes de lista e multivalorados disponíveis no React Native.
- Descrever os três principais modelos de navegação do React Native.
- Listar os recursos de estilização e animação.

Introdução

O processo de desenvolvimento de aplicativos envolve uma série de conhecimentos. Entre eles, destacam-se a organização do processo como um todo e o conhecimento de ferramentas, bibliotecas e demais recursos a serem utilizados no projeto.

Este conteúdo tem como um de seus principais objetivos apresentar alguns dos recursos e componentes disponíveis no framework React Native que estão voltados para a construção da interface gráfica: os elementos de interatividade, navegação e estilização. Ao final deste texto, teremos visto os conceitos necessários para a construção da interface – normalmente composta por telas e componentes reutilizáveis – de um aplicativo mobile.

Primeiras palavras

Um importante passo antes de se iniciar, de fato, a codificação de um aplicativo (e o mesmo pensamento também deve valer para qualquer tipo de software) é estruturar como o aplicativo funcionará.

Para isso, é preciso ter em mente não só os requisitos de software, ou seja, as funcionalidades do aplicativo, mas também como tais funcionalidades devem estar dispostas, como são acessadas por meio da navegação pelo aplicativo e, por fim, mas não somente, como se dispõe cada elemento da interface gráfica de modo a criar a melhor experiência possível para o usuário.

Interface de aplicações.

Tendo em mente a disposição dos elementos da interface gráfica citada, destacaremos adiante algumas técnicas para a definição da interface de um aplicativo mobile.

Requisitos ou funcionalidades

Como primeiro passo para a esquematização da interface e interatividade de nosso aplicativo mobile, devemos ter em mãos as suas funcionalidades, ou seja, precisamos conhecer tudo o que o aplicativo deverá fazer e tudo o que poderá ser feito por meio dele. Essa lista de funcionalidades pode ser:

Simples

Como uma mera lista.

Elaborada

Incluindo protótipos e requisitos, como paleta de cores e restrições visuais ou estruturais, além de outros aspectos.

Como exercício, confeccionaremos juntos a **lista de funcionalidades de um aplicativo bancário**. Por meio desse app, deve ser possível, graças ao **fluxo básico**, realizar as seguintes atividades:

1. Logar com as credenciais (usuário e senha) previamente fornecidas pela instituição bancária.
2. Após o login, o usuário deverá visualizar, de imediato, um resumo de seu saldo e os últimos lançamentos realizados em sua conta.
3. Na primeira tela após o login, deverá ser possível exibir banners publicitários rotativos para o usuário.
4. As demais ações ou opções disponíveis, após o usuário se logar, são:

- Consulta de extrato detalhado.
- Pagamento de contas.
- Transferência bancária.

- Consulta de fatura e lançamentos de cartão de crédito.
- Consulta de investimento financeiro.

Já no **fluxo alternativo**, que acontece quando o usuário não se loga, tem de ser possível:

1. Exibir os canais de contato.
2. Exibir banners rotativos de publicidade.

As funcionalidades descritas, embora bastante reduzidas na comparação com um aplicativo real, nos ajudam a começar o planejamento da interface de nosso aplicativo. Graças a tais requisitos é possível, por exemplo, entender quantas telas existirão no aplicativo, assim como identificar os elementos de interação, desde botões simples até elementos visuais com função de publicidade.

Telas e elementos visuais

Continuando nosso exercício e considerando os requisitos descritos, podemos fazer um primeiro esboço dos **sete elementos** que comporão a interface de nossa aplicação:

Nº	Tela	Elementos visuais	Elementos de interação
1	Inicial / login	Logomarca da instituição; banner publicitário rotativo; canais de comunicação.	Input de dados; botão de ação.
2	Home (após login)	Logomarca da instituição; textos; títulos (*label*); banner publicitário rotativo.	Botão de ação; menu de navegação.
3	Consulta de extrato	Logomarca; textos; títulos (*label*).	Botão de ação; menu de navegação.
4	Pagamento de contas	Logomarca; texto; títulos (*label*).	Botão de ação; menu de navegação; input de dados; acesso à câmera do dispositivo móvel.
5	Transferência bancária	Logomarca; texto; títulos (*label*); listagem de texto.	Botão de ação; menu de navegação; input de dados; seleção de dados.
6	Consulta de fatura de lançamentos de cartão de crédito	Logomarca; texto; títulos (*label*); listagem de texto.	Botão de ação; menu de navegação; input de dados; seleção de dados.
7	Consulta de investimento financeiro	Logomarca; texto; títulos(*label*); listagem de texto.	Botão de ação; menu de navegação; input de dados; seleção de dados.

Quadro: Esboço dos elementos da interface.
Elaborado por: Alexandre Paixão.

Após o preenchimento inicial do quadro contendo as telas e os elementos visuais com base nos requisitos – e tendo em mente que tal exercício está bastante simplificado se comparado a exemplos reais –, identificamos a presença de, pelo menos, sete telas em nosso aplicativo.

Além disso, é possível verificar que alguns desses elementos visuais aparecerão em mais de uma tela. Esse fator é importante!

> Elementos que aparecem em mais de uma tela podem ser “componentizados”, o que permite que sejam reaproveitados ao longo da aplicação.

Definição de componentes

Escolha dos componentes

Dando seguimento à esquematização da interface e dos elementos de interatividade de nosso app, agora cumpriremos dois passos:

Confecção de protótipos navegáveis

O primeiro passo, embora seja muito indicado, não será tratado aqui, já que foge à proposta de nosso conteúdo.

Escolha dos componentes nativos ou de terceiros a serem utilizados

O segundo passo deve levar em conta alguns aspectos que vão além de preferências visuais ou estéticas.

Saiba mais

Há várias ferramentas disponíveis para a confecção de protótipos navegáveis – inclusive gratuitas.

Partiremos então do segundo passo. Como dito, é preciso tomar alguns cuidados com a organização, a fim de “conteinerizar” os elementos utilizados, e a compatibilidade ou as restrições, em que questões – como a dos elementos que não podem ser aninhados dentro de outros ou as diferenças de tamanho de tela e espaço útil disponível nos diferentes dispositivos móveis – devem ser consideradas.

Alguns fragmentos de código com diferentes organizações dos componentes na tela, assim como comentários sobre cada um deles, podem ser vistos a seguir:

```
plain-text
return (
                Hello World!
 );
```

O primeiro fragmento possui uma “View” como **container principal**, dentro da qual os demais componentes ficam aninhados. Em termos de semântica, embora o código em questão funcione sem erros, é recomendado não aninhar componentes de imagem dentro de componentes de texto como está apresentado anteriormente.

Dica

Utilize o componente View como container de outros componentes, inclusive de outras Views. Isso ajudará a organizar o código e a estilizá-lo.

Observemos outro fragmento:

```
plain-text
return (


         Lorem ipsum dolor sit amet,
         consectetur adipiscing elit, sed do
         eiusmod tempor incididunt ut labore et
         dolore magna aliqua. Ut enim ad
         minim veniam, quis nostrud exercitation ullamco
         laboris nisi ut
         aliquip ex ea commodo consequat. Duis aute irure
         dolor in reprehenderit in voluptate velit esse
         cillum dolore eu   fugiat nulla
         pariatur. Excepteur sint occaecat cupidatat non
         proident, sunt in
         culpa qui officia deserunt mollit anim id est laborum.



 );
```

Os pontos de atenção desse novo fragmento estão na utilização dos seguintes componentes:

ScrollView

O ScrollView insere um scroll próprio para o conteúdo nele inserido. Ele deve ser utilizado quando houver muito texto a ser exibido na tela. Considerando as limitações de tamanho em um dispositivo móvel, a utilização desse componente melhora a experiência do usuário, permitindo a organização das informações mais importantes que queremos exibir em uma única tela.

SafeAreaView

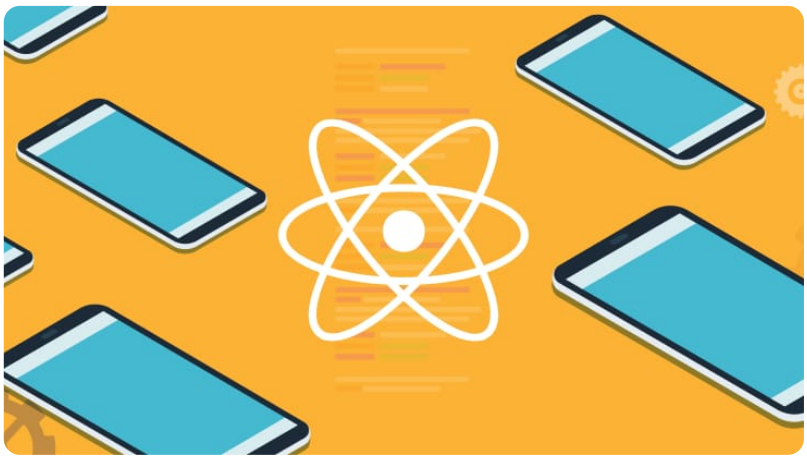
Em termos de experiência do usuário, o SafeAreaView, disponível apenas para a plataforma iOS, gera uma área segura no aplicativo e garante que o conteúdo por ele contido não seja afetado pelas características e dimensões do dispositivo e nem por outros elementos da própria aplicação, como barras de navegação, entre outros exemplos.

Você pode utilizar tal componente para a plataforma iOS e implementar, por meio de estilos condicionais, uma “área de conteúdo segura” para a plataforma Android.

Saiba mais

Leia mais sobre o SafeAreaView na documentação oficial do React Native.

Ainda na área de conteúdo seguro, cabe destacar um último componente: o **KeyboardAvoidingView**. Ele resolve problemas comuns existentes entre os componentes visuais e o teclado virtual, ajustando automaticamente a altura e a posição de acordo com a altura do teclado.



Componentes de interface e interação

Veremos agora alguns **elementos de interface**, além dos vistos anteriormente, e de **interação** que nos ajudarão a esquematizar as telas de nosso aplicativo.

Modal

Este componente é uma boa estratégia para apresentar mais conteúdo sem que seja necessário levar o usuário para outra página ou mesmo criar um scroll vertical muito grande. O acesso à janela Modal pode ser feito por meio do clique em um botão, por exemplo. Veja o fragmento a seguir:

```
plain-text

return (

      {
         setModalVisible(!modalVisible);
     }}
     >


         Texto dentro da modal
          setModalVisible(!modalVisible)}
         >
             FecharModal



      setModalVisible(true)}
     >
     Abrir Modal


 );
```

Botões

O fragmento de código anterior contém um elemento de interação, o Pressable, cujo funcionamento equivale ao de um botão. O React Native possui alguns componentes com essa mesma função, como, por exemplo:

- Button
- TouchableHighlight
- TouchableOpacity
- TouchableWithoutFeedback

Tais componentes compartilham entre si algumas semelhanças e diferenças, sendo a diferença mais importante a menor possibilidade de estilização presente no Button.

> A principal característica do Button é reagir a eventos de interação por parte do usuário, como o toque, leve ou demorado, sobre eles.

Quando estivermos montando nossas telas, deveremos definir claramente seu título a fim de indicar qual ação será disparada no toque. Outra dica é utilizar, além de um título, um ícone. Isso vale, por exemplo, quando tais elementos são usados tanto individualmente quanto em barras de navegação.

Exemplo de button.

Recomendação

Conforme a orientação constante na documentação oficial do React Native, devemos dar preferência à implementação do componente Pressable.

ActivyIndicator

Este componente exibe elementos visuais que comumente chamamos de “loading”. Embora bastante negligenciado, trata-se de um dos mais importantes elementos em termos de interatividade, pois seu uso permite informar ao usuário que uma tarefa está sendo processada.

Loading.

Recomendação

A sugestão em relação a esse componente é utilizá-lo sempre que uma requisição a um recurso externo seja realizada. Com isso, mesmo que haja demora na resposta, o usuário é informado de que algo está acontecendo. Do contrário, a espera pela conclusão de uma tarefa demorada sem nenhuma indicação visual pode dar a ele a sensação de que o aplicativo travou, por exemplo.

Outros componentes

Além dos componentes vistos até aqui, há vários outros disponíveis para a organização das telas de nosso aplicativo. Eles possuem, além de organizar visualmente os elementos ou fornecer interatividade, diferentes funções, como permitir a navegação entre telas e a apresentação otimizada de recursos específicos, como listas e cards.

Processo de definição da interface de um aplicativo mobile

Neste vídeo, falaremos sobre os pontos relevantes e de atenção ao realizarmos uma esquematização do passo a passo para definição da interface interativa de um aplicativo mobile.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Vem que eu te explico!

Os vídeos a seguir abordam os assuntos mais relevantes do conteúdo que você acabou de estudar.

Modal

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

ActivyIndicator

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Verificando o aprendizado

Questão 1

Em relação aos passos para a construção de uma interface interativa que possibilite ao usuário uma boa experiência de utilização de aplicativos, selecione a opção verdadeira.

A

As cores devem ter destaque na construção de uma interface, sendo sempre sugerido utilizar a maior gama possível de combinações a fim de agradar à maioria dos usuários.

B

As etapas para a criação de uma interface interativa são altamente subjetivas. Logo, o melhor a se fazer é seguir as preferências do próprio desenvolvedor.

C

Durante o processo de definição de uma interface, é indispensável ter na equipe vários profissionais para garantir que seja coletada a maior quantidade possível de opiniões e feedbacks.

D

O processo de esquematização da interface interativa de um aplicativo precisa levar em conta aspectos, como os requisitos de software e os recursos disponíveis na linguagem a ser utilizada em sua construção. Além disso, outros aspectos, como um design em sintonia com os demais supracitados, tendem a garantir uma boa experiência para os usuários.

E

Ao utilizar o framework React Native, é possível implementar qualquer interface interativa a partir de um design elaborado em protótipos navegáveis independentemente de sua complexidade.

A alternativa D está correta.

O React Native é um framework poderoso e flexível para a construção de aplicativos mobile. Entretanto, embora ele possua diversos componentes nativos, assim como outros produzidos por terceiros, é preciso esquematizar bem o processo de confecção das interfaces dos aplicativos, levando em consideração as características da linguagem e dos componentes disponíveis, pois nem sempre o que é pensado/idealizado pode, de fato, ser reproduzido. Desse modo, a esquematização deve levar em conta tanto os recursos disponíveis quanto as possíveis restrições, assim como comportamentos e outras características disponíveis na linguagem.

Questão 2

O React Native possui vários componentes para a confecção de interfaces interativas. Em relação aos componentes estudados, é correto afirmar que:

A

O React Native é bastante flexível, exceto em relação à conteinerização ou ao aninhamento de componentes, que deve obedecer algumas restrições bastante rígidas de conteinirização.

B

Em termos de componentes de interface, estão disponíveis as mesmas opções tanto para a plataforma Android quanto para a iOS, sem exceções.

C

Além de contar com vários componentes nativos, o React Native permite que outros sejam criados a partir de novas tags e de elementos implementados pelos próprios desenvolvedores.

D

O React Native conta com vários componentes de interface e interação: alguns, específicos para a plataforma Android; outros, para a iOS. Além disso, há vários componentes de terceiros desenvolvidos a partir dos componentes nativos. Trata-se de uma framework bastante flexível, permitindo a combinação e o aninhamento entre diferentes tipos de componentes.

E

Todos os componentes de interface do React Native têm como base o elemento View.

A alternativa D está correta.

O framework React Native permite que diferentes tipos de interface interativa sejam criados a partir dos componentes disponibilizados. Nesse sentido, além dos componentes específicos, semanticamente falando, é possível usar outros para as mesmas função. Um bom exemplo é o componente View, que pode conter outros elementos do mesmo tipo ou não, assim como textos diretamente. Outra característica importante é o fato de possuir componentes otimizados para algumas funcionalidades, com os indicadores de atividade, entre outros.

Componentes específicos para tarefas específicas

Existem componentes para a construção de interfaces interativas que, além de serem utilizados com a finalidade de montagem da interface e de interação, também podem ser empregados para outras funcionalidades.

O componente View é um ótimo exemplo: por meio dele, podemos tanto organizar nossas telas em bloco quanto exibir textos. Neste módulo, abordaremos os componentes que possuem responsabilidades específicas: os de lista e multivalorados.

Componentes de lista

O React Native possui três principais componentes nativos de listas:

- VirtualizedList
- SectionList
- FlatList

Estabeleceremos as definições e a utilidade desses três componentes, assim como suas implementações. Antes disso, no entanto, é importante entender o comportamento geral deles.

Como os próprios nomes sugerem, trata-se de componentes voltados para a exibição de dados no formato de lista (listas, *arrays*, coleções, etc.). Você verá inclusive que eles possuem algumas características semelhantes às das listas em geral, como o fato de possuir índice e a capacidade de iteração em seu conteúdo.

VirtualizedList

Este componente constitui a implementação-base para os outros dois componentes de lista, que acabam sendo mais utilizados devido às suas particularidades – o que, aliás, faz com que ambos também sejam mais bem documentados.

Exemplo de uso do VirtualizedList.

Segundo a documentação oficial, o VirtualizedList oferece mais flexibilidade que o FlatList, tendo melhor suporte a dados imutáveis contra as matrizes simples normalmente usadas com o FlatList.

Das características do VirtualizedList, destacam-se:

Otimização

Otimização do consumo de memória e de desempenho no manuseio de grandes listas.

Manutenção de janela

Manutenção de janela finita de renderização de itens ativos e substituição daqueles que estejam fora da janela de renderização.

Adaptação da janela à rolagem de tela

A adaptação permite que os itens sejam renderizados de forma incremental, o que minimiza a ocorrência de espaços em branco.

Observaremos a seguir um exemplo de implementação de lista utilizando o componente VirtualizedList:

```plain-text
import React from 'react';
 import { View, VirtualizedList, StyleSheet, Text, StatusBar } from 'react-native';

 const DATA = [];

 const getItems = (data, index) => ({
     id: Math.random().toString(12).substring(0),
     title: `Item ${index+1}`
 });

 const getItemsCount = (data) => 100;

 const Item = ({ title }) => (

     {title}

 );

 const App = () => {
     return (

          }
         keyExtractor={item => item.key}
         getItemCount={getItemsCount}
         getItem={getItems}
         />

     );
 }

 const styles = StyleSheet.create({
     container: {
     flex: 1,
     marginTop: StatusBar.currentHeight,
     },
     item: {
     backgroundColor: '#0000ff',
     height: 100,
     justifyContent: 'center',
     marginVertical: 8,
     marginHorizontal: 16,
     padding: 10,
     },
     title: {
     fontSize: 22,
     },
 });

 export default App;
```

Recomendação

Antes de continuar, copie e execute o código para conseguir ver o componente em funcionamento.

O código do exemplo imprime 100 retângulos na tela contendo o texto “Item”, seguido de um número de 1 a 100. Embora simples, ele nos permite observar o componente em funcionamento. Experimente rolar a tela, indo até o final e retornando ao início. Na prática, nossas listas são alimentadas por conteúdo proveniente de fontes externas, como APIs, por exemplo.

É normal, nessas situações, ocorrer uma pequena demora no primeiro carregamento dos dados. Entretanto, após isso, o VirtualizedList cuidará para que a visualização de todo o conteúdo seja feita de forma suave, sem quebras no layout ou interrupções. Ainda nos atendo ao código, precisamos fazer alguns comentários sobre os **atributos** desse componente:

Data

Atributo que recebe os dados a serem apresentados no formato {chave:valor}.

getItem

Função (data:any, index:number) responsável por extrair o conteúdo dos dados que serão apresentados.

getItemCount

Função (data:any) responsável por determinar a quantidade de itens a serem exibidos.

renderItem

Função (info:any) que exibe e renderiza os dados da lista. Como vimos acima, é possível utilizar diferentes tipos de componentes para a exibição dos dados, atribuindo-os diretamente na declaração do componente ou utilizando uma função. Em nosso exemplo, utilizamos o componente funcional Item para isso.

keyExtractor

Função (item:object, index:number) usada para extrair uma chave única para cada item da lista a ser exibida no componente. Esta chave é usada pelo React para o armazenamento em cache e o rastreamento dos itens da lista.

initialNumToRender

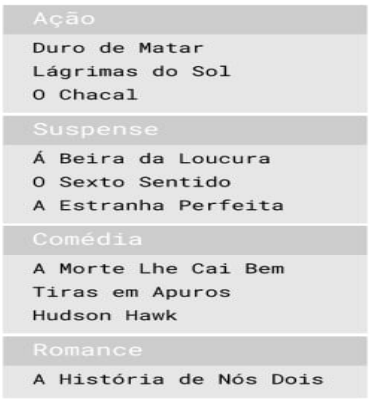
Atributo numérico que indica a quantidade inicial de itens a serem renderizados.

Saiba mais

Além dos atributos e das funções descritas, há outros disponíveis para o VirtualizedList. Consulte a documentação oficial a respeito.

SectionList

A SectionList é um componente para a exibição de dados em formato de lista que permite que eles sejam seccionados, ou seja, exibidos em seções. Para ficar mais fácil de entender o funcionamento e a utilidade desse componente, imagine que você tenha uma lista de produtos para exibir e que esses produtos pertencem a diferentes categorias de produtos.



Exemplo de SectionList.

Ao utilizar o SectionList, é possível agrupá-los por categoria, exibindo algum dado que a identifique, como seu nome, por exemplo, e aninhados a cada diferente categoria seus respectivos produtos. Como frisamos, tal componente facilita esse trabalho de seccionamento dos dados, evitando a necessidade de criar mais códigos para ter o mesmo resultado utilizando outros componentes de lista.

O código adiante mostra como a SectionList pode ser utilizada. Repare nos atributos, nos quais são definidos tanto os dados a serem listados quanto o dado a ser utilizado como “agrupador” ou definidor de cada seção.

Outro detalhe: nesse exemplo, é utilizado uma string no formato JSON como fonte de dados. Na prática, você poderá consumir os dados de fontes externas, tomando o cuidado de organizá-los para que eles tenham um elemento que identifique a seção e outro para os dados de cada seção. Vamos ao exemplo:

```plain-text
import React from 'react';
 import { StyleSheet, Text, View, SectionList, StatusBar } from 'react-native';

 const DADOS= [
     {
     titulo: 'Eletrônicos',
     data: ['TV', 'Caixa de Som', 'Toca-discos Retrô']
     },
     {
     titulo: 'Vestuário',
     data: ['Camisas', 'Camisetas', 'Casacos']
     },
     {
     titulo: 'Livros',
     data: ['Ficção', 'Suspense', 'Policiais']
     }
 ];

 const Item = ({ titulo }) => (

     {titulo}

 );

 const App = () => (

      item + index}
         renderItem={({ item }) => }
         renderSectionHeader={({ section: { titulo } }) => (
         {titulo}
         )}
     />

 );

 const styles = StyleSheet.create({
     container: {
     flex: 1,
     paddingTop: StatusBar.currentHeight,
     marginHorizontal: 16
     },
     item: {
     backgroundColor: '#fffccc',
     padding: 20,
     marginVertical: 8
     },
     header: {
     fontSize: 32,
     backgroundColor: '#fff'
     },
     titulo: {
     fontSize: 24
     }
 });

 export default App;
```

Analise o código anterior e repare que a SectionList possui alguns atributos semelhantes aos vistos na VirtualizedList. Sendo assim, devemos mencionar **outros atributos que lhe são característicos**:

sections

Propriedade obrigatória e que define os dados a serem listados.

renderSectionHeader

Define o elemento a ser renderizado no início de cada seção.

Observe novamente o código do nosso exemplo onde utilizados o **renderSectionHeader**, e perceba que exibimos um componente Text contendo a chave “título” da constante DADOS, que é usada para alimentar a lista.

FlatList

Conceitos e funcionalidades

O último componente de lista que veremos é o FlatList. Ele é otimizado, em termos de desempenho, para renderizar listas simples e básicas. Além disso, possui suporte para vários recursos muito úteis. Conheça alguns deles a seguir:

- Possui suporte a múltiplas plataformas.
- Permite exibir os itens de forma horizontal.
- Possui suporte a cabeçalho e rodapé.
- Permite a inclusão de separadores (componentes que podem ser usados para separar os itens da lista).
- Permite a atualização de seu conteúdo por meio do movimento de “puxar e soltar” ou por rolagem (*scrolling*).

Saiba mais

Alguns desses recursos também são suportados pelos demais componentes de lista, os quais, aliás, também podem possuir recursos específicos. Para mais detalhes, consulte a documentação oficial do React Native.

Vejamos um exemplo de implementação do FlatList:

```
plain-text
```

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

 const DADOS = [
     {
     id: '1',
     descricao: 'TV Led 49',
     categoria_id: 1
     },
     {
     id: '4',
     descricao: 'Camisa Trilha',
     categoria_id: 2
     },
     {
     id: '4',
     descricao: 'Qualquer semelhança é mera coincidência',
     categoria_id: 3
     },
 ];

 const Item = ({ descricao }) => (

     {descricao}

 );

 const App = () => {
     const renderItem = ({ item }) => (

     );

     return (

          item.id}
         />

     );
 }

 const styles = StyleSheet.create({
     container: {
     flex: 1,
     marginTop: StatusBar.currentHeight || 0,
     },
     item: {
     backgroundColor: 'yellow',
     padding: 20,
     marginVertical: 8,
     marginHorizontal: 16,
     },
     title: {
     fontSize: 12,
     },
 });

 export default App;
```

Recursos adicionais do FlatList

O componente FlatList, assim como os outros dois apresentados anteriormente, possui alguns recursos adicionais para tratar a exibição de dados considerando aspectos mais avançados, como, por exemplo, a **atualização dos dados**.

No que tange à atualização dos dados, ou seja, quando quisermos fazer com que o componente “escute” e monitore os dados a fim de identificar eventuais atualizações, teremos de incluir uma propriedade a mais: o “extraData”. Tal propriedade poderá receber um método ou um **state**.

Exemplo de FlatList.

Saiba mais

State é todo dado capaz de variar ao longo da aplicação, podendo tal mudança ser (ou não) proveniente da interação do usuário.

Outro recurso muito interessante presente no FlatList – e nos demais componentes de lista – é o **“onEndReached”** (e, consequentemente, o “onEndReachedThreshold). Com o uso dessa função, podemos informar ao nosso aplicativo que determinada ação deverá ser executada quando o usuário chegar ou visualizar o final dos elementos da lista. Normalmente, ela é usada para carregar, mediante demanda, novos conteúdos.

Comentário

Você, com certeza, já observou o emprego desse recurso em aplicativos de redes sociais, pois inicialmente é carregado um conjunto limitado de dados. Conforme você rola a tela para baixo e chega ao final desse conjunto inicial de dados, novas informações são carregadas.

Próximos passos

Até aqui, você viu como esquematizar a interface interativa de um aplicativo utilizando diversos componentes, alguns mais gerais e de múltiplos usos, como a View, e outros mais específicos e voltados para determinadas necessidades, como as listas.

Introduziremos novos componentes no próximo módulo. Por intermédio deles, será possível organizar a navegação entre as diversas telas que normalmente compõem um aplicativo.

Componentes de lista no React Native

Neste vídeo, abordaremos os principais componentes de lista e multivalorados disponíveis no React Native.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Vem que eu te explico!

Os vídeos a seguir abordam os assuntos mais relevantes do conteúdo que você acabou de estudar.

VirtualizedList

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

FlatList

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Verificando o aprendizado

Questão 1

Tendo complementado seu conhecimento sobre os componentes nativos disponíveis no framework React Native, é correto afirmar que:

A

O React Native é um framework tão flexível que podemos utilizar quaisquer componentes para realizar quaisquer tarefas. Em outras palavras, até mesmo os ditos “componentes com funções específicas”, como as listas, não passam de componentes codificados a partir dos nativos básicos, como a View e o Text.

B

No React Native, estão disponíveis nativamente diversos componentes que permitem a organização visual do conteúdo, como a exibição de dados, inclusive contando com componentes específicos e otimizados para algumas tarefas, como, por exemplo, os de lista.

C

Embora o React Native possua uma grande quantidade de componentes nativos, alguns deles estão limitados à plataforma Android, não existindo equivalentes na plataforma iOS.

D

Os componentes de lista não podem ser usados em conjunto em um mesmo aplicativo devido a problemas de compatibilidade e otimização de desempenho.

E

Os componentes FlatList e VirtualizedList são especializações do componente SectionList, exceto pelo fato de ambos não possibilitarem a exibição de títulos para as seções contidas nos dados que renderizam.

A alternativa B está correta.

No React Native, há diversos componentes disponíveis para diferentes funções. Embora seja possível combinar tais elementos para a criação de componentes, é recomendado utilizar os componentes nativos conforme suas funcionalidades. Nesse sentido, é possível montar uma lista utilizando View e Text. Entretanto, tal implementação não teria o mesmo desempenho que um componente de List, disponível justamente para a exibição performática desse tipo de dado.

Questão 2

Sobre o componente VirtualizedList, marque a afirmativa correta.

A

É recomendado o uso do VirtualizedList para os casos em que se precisa de menos flexibilidade que aquela fornecida pelos componentes Flat e SectionList.

B

O VirtualizedList, por ser um componente básico, não possui algumas funcionalidades presentes nos outros componentes de lista, como a função “extraData”, por exemplo.

C

No VirtualizedList, que é uma lista otimizada a partir do componente ScrollView, há um limite para a quantidade de dados possíveis de serem renderizados.

D

O VirtualizedList possui a capacidade de tratar grandes quantidades de dados. Entretanto, nesses casos, não há nenhum mecanismo que impeça a ocorrência de alguns travamentos de tela quando dados antigos são substituídos por novos.

E

O componente VirtualizedList é a implementação básica dos outros dois componentes de lista: FlatList e SectionList. Esses componentes são muito parecidos, tendo em comum uma série de recursos.

A alternativa E está correta.

Nativamente, o React Native disponibiliza alguns componentes cuja função é exibir dados em formato de listas. Embora, em tais casos, pudéssemos utilizar componentes genéricos, como o ScrollView, é recomendado usar os componentes específicos para tal função. Entre esses componentes, encontra-se o VirtualizedList. Ainda que seja uma implementação básica dos outros dois componentes de lista, ele compartilha a maioria dos recursos presentes nos demais, sendo indicado para a renderização de dados imutáveis.

Navegando em um aplicativo mobile

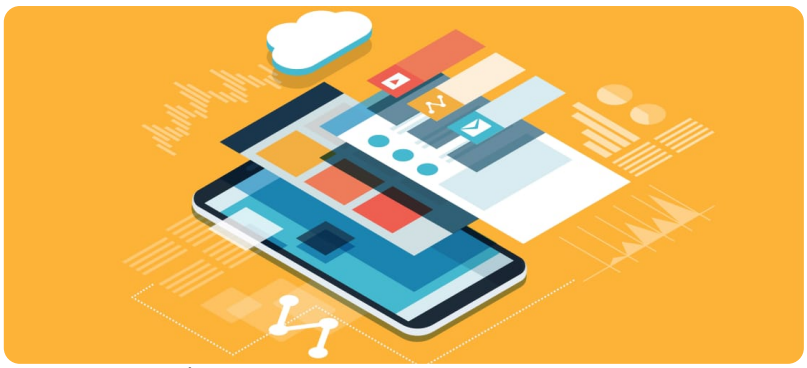
Estamos acostumados a utilizar o conceito de navegação proveniente dos sites ou sistemas web. Em tais ambientes, estão presentes termos como:

- Páginas – página inicial/home.
- Páginas secundárias.
- Links – links externos e internos.
- *Breadcrumb*.
- Menus – menu horizontal e vertical.
- Navbar.

Com isso, é normal, no desenvolvimento de aplicativos mobile, se pegar emprestado alguns desses termos a fim de fazer uma assumpção e/ou uma correlação de conceitos. Pode-se afirmar que um aplicativo é normalmente composto por várias páginas.

> ⚠
>
> Atenção
>
> Embora muitos desenvolvedores utilizem o termo “página”, é comum também se referir às seções do aplicativo como “telas”. É ainda mais usual referenciar tais termos em seus equivalentes no idioma inglês: pages ou screens.

Ao longo deste módulo, falaremos sobre os aspectos que envolvem a navegação entre as telas/páginas de um aplicativo mobile. Para isso, voltaremos a fazer as correlações mencionadas, descrevendo os **três principais modelos** disponíveis no React Native.



Navegação entre páginas de aplicativo mobile.

Preparando o nosso ambiente

Para implementar os exemplos que serão demonstrados neste módulo, é necessário instalar algumas novas dependências em nosso projeto (se você ainda não criou um projeto, recomendo que o faça agora). Essas dependências serão listadas a seguir:

- @react-navigation/native
- react-native-reanimated
- react-native-gesture-handler
- react-native-screens
- react-native-safe-area-context

Dica

Para instalar as dependências descritas, use um gerenciador de dependências (NPM ou YARN) a partir do terminal, estando na raiz da pasta que contém seu projeto. Observe a necessidade de executar o comando “react-native link” caso esteja utilizando uma versão do React Native Cli inferior a 0.60. Além disso, no ambiente iOS, será preciso rodar os pods por meio do comando “npx pod-install ios” para completar o processo de “linking” das dependências.

Stack Navigation

Apresentação

O primeiro modelo de navegação que veremos é o Stack Navigation. Como seu nome diz, esse modelo consiste no empilhamento de telas. Ou seja, cada nova tela acessada por meio dele é colocada por cima em uma pilha, sobrepondo a tela anterior.

Podemos relacionar o Stack Navigator a dois conceitos já conhecidos:

Exemplo de navegação com Stack Navigation.

Navegador

O conceito do Stack Navigator é bastante similar ao do navegador utilizado em um site (ou sistema web): por meio de links ou botões exibidos em um menu, na barra de navegação e até mesmo em links no meio do próprio conteúdo, se é levado até uma nova página. Entrando nela, pode-se continuar navegando para novas páginas ou voltar – seja por intermédio de um link ou pela opção “voltar” presente no navegador – para a anterior.

Histórico

Outro conceito interessante que pode ser verificado é o de histórico. Perceba que seu navegador guarda o histórico das páginas que você visita, permitindo que você volte ou avance diretamente pelos botões presentes no navegador ou por meio de links no site/sistema visitado ou usado.

Seguindo os dois conceitos apresentados, o Stack Navigator provê um mecanismo que possibilita:

- Transição entre telas.
- Gestão do histórico de navegação em um aplicativo.

Para se codificar a navegação utilizando esse modelo, é preciso haver um NavigationContainer e, no mínimo, um createNativeStackNavigator. Além disso, são necessárias pelo menos duas telas já codificadas em nosso aplicativo para se definir a navegação entre ambas.

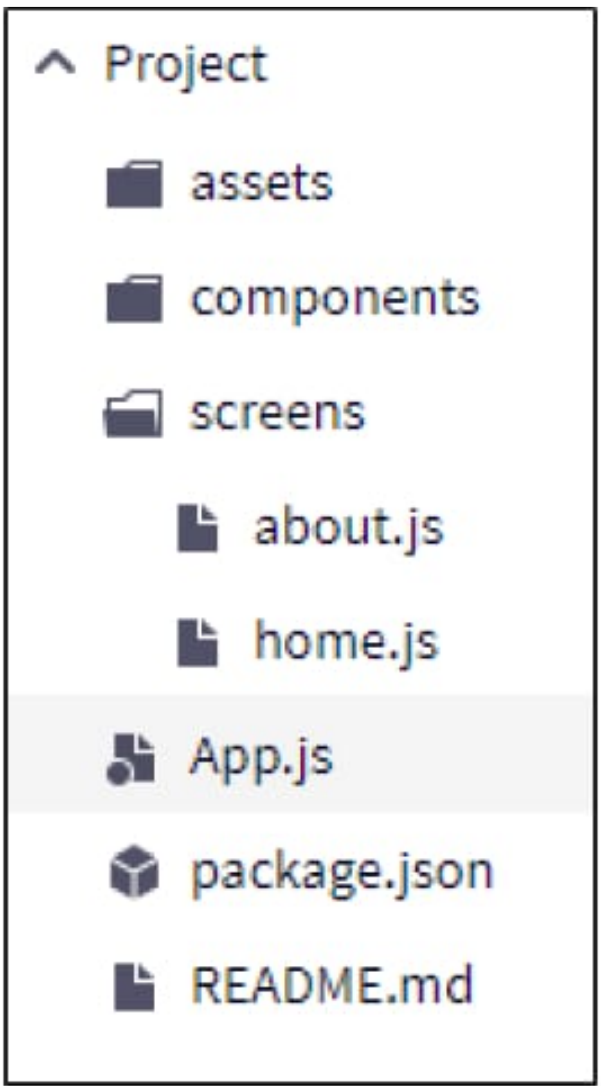
Mais adiante, veremos um código que demonstra a criação do ponto de entrada de um aplicativo (App.js) e da navegação entre duas telas (Home e Sobre).

Dica

Antes de codificarmos, precisamos instalar a seguinte dependência: @react-navigation/native-stack

Estrutura do projeto

A imagem a seguir mostra a estrutura do projeto no qual foi utilizado o Expo (por meio do snack.expo.dev):



Estrutura de um projeto.

O script App.js é o ponto de entrada do aplicativo em que o StackNavigator foi definido. A pasta screens guarda as telas About e Home. O código da tela Home pode ser visto a seguir:

```plain-text
import * as React from 'react';
 import { View, Text, Button } from 'react-native';

 function HomeScreen({navigation}) {
     return (

         Home Screen
          navigation.navigate('About')} />

                           );
                         }

             export default HomeScreen;
```

O próximo código é da tela About:

```plain-text
import * as React from 'react';
 import { View, Text, Button } from 'react-native';

 function AboutScreen({navigation}) {
     return (

         About Screen
          navigation.navigate('Home')} />

                           );
                         }

            export default AboutScreen;
```

Criação do Stack Navigator

Este código mostra a criação do Stack Navigator a partir do script App.js:

```
plain-text

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

 import HomeScreen from './screens/home';
 import AboutScreen from './screens/about';

 const Stack = createNativeStackNavigator();

 function App() {
     return (









     );
 }

 export default App;
```

Análise do código

Como de praxe, a primeira coisa a se fazer ao escrevermos nosso código em React Native é importar as dependências que utilizaremos. Em nosso caso, além dos componentes de exibição, como View e Text, importaremos:

NavigationContainer

Container responsável por conter toda a navegação de nosso aplicativo.

createNativeStackNavigator

Responsável por criar o Stack de navegação.

Repare que criamos uma constante denominada **Stack** – ela poderia ter qualquer nome, ficando à sua escolha – e a definimos como uma instância de **createNativeStackNavigator**. Na função App, usaremos essa constante para definir os parâmetros de navegação.

Em primeiro lugar, criamos o container **Stack.Navigator**. Dentro dele, informamos as opções de navegação, ou seja, quais telas farão parte de nossa pilha de navegação e serão gerenciadas pelo Stack. Isso é feito com a propriedade **Stack.Screen**. Tal propriedade recebe como parâmetros um “name”, pelo qual nos referiremos a cada item, e um “componente”, em que definimos quais telas serão apresentadas. Em nosso exemplo, passamos como “component” as telas **HomeScreen** e **AboutScreen** (ambas também importadas no início do script).

Agora, falaremos novamente das características do mecanismo provido pelo Stack Navigator:

Navegação entre telas

Até esse ponto, nosso modelo de navegação Stack está criado com duas possíveis rotas, a tela Home e a About. A navegação entre ambas via Stack está sendo feita nas próprias telas por meio dos componentes “Button” nelas inseridos. Esses componentes, por sua vez, foram inseridos graças à ação “onPress”, em que temos:

- navigation.navigate('About')
- navigation.navigate('Home')

Repare que as telas Home e About recebem como propriedade (props) o “navigation”, proveniente do Stack Navigator. Tal propriedade permite que naveguemos usando o método “navigate” – a exemplo do que faríamos em uma página web usando um link – entre as telas (screens) que fazem parte da nossa pilha de telas. Esse método recebe como parâmetro o valor do atributo “name” que definimos no App.js em Stack.Screen.

Histórico

Uma funcionalidade nativa do Stack é permitir que voltemos à tela anterior após termos navegado por intermédio da nossa pilha de telas. Para isso, o Stack inclui naturalmente uma opção de voltar logo no início da tela. Podemos usar tal botão ou então implementar tal opção por meio deste código:

- navigation.push(‘Home’)

Opções avançadas

Além dos recursos até aqui apresentados, o Stack Navigator possui outras opções de configuração e navegação. É possível ainda combinar mais de um Stack Navigator em nosso aplicativo.

Recomendação

Primeiro, implemente e busque entender o funcionamento dos códigos apresentados, para só então se aprofundar no entendimento do modelo de navegação Stack Navigator.

Apresentação

Este estilo de navegação é composto por guias ou abas que contêm botões – com título e ícone ou por suas possíveis combinações – por meio dos quais se navega ao pressionar tais elementos. Essa navegação é normalmente inserida na parte inferior do aplicativo, embora também possa se localizar no topo.

Exemplo de navegação com Tab Navigation.

Atenção

Antes de vermos os códigos de um exemplo funcional de navegação por Tab, precisamos instalar a seguinte dependência: @react-navigation/bottom-tabs

Estrutura do projeto

Em nosso exemplo, poderemos utilizar a mesma estrutura de pastas do projeto anterior (Stack Navigator), mantendo nossas **screens Home** e **About**. Entretanto, será necessário modificar nosso App.js.

Recomendação

Crie um projeto copiando do anterior apenas as duas telas já mencionadas.

Criação do Tab Navigator

Nosso App.js ficará desta forma:

```
plain-text

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

 import HomeScreen from './screens/home';
 import AboutScreen from './screens/about';

 const Tab = createBottomTabNavigator();

 function App() {
     return (









     );
 }

 export default App;
```

Análise do código

Perceba, no código acima, que a criação do Tab Navigator, em termos de sintaxe, é semelhante à do Stack Navigator. Basta, portanto, realizar as seguintes operações:

1. Importar a dependência.
2. Criar uma constante (Tab) como instância de “createBottomTabNavigator”.
3. Criar, dentro do NavigationContainer, o Tab.Navigator e, em seguida, o Tab.Screen, no qual são setadas as telas a serem acessadas por intermédio dessa navegação.

Navegação entre telas

Em termos práticos, no lugar de navegar entre as telas usando um botão com a props “navigation” e o método “navigate” (ou o botão de voltar), como fizemos no primeiro modelo de navegação, utilizaremos no Tab os botões que ficam dentro da guia, na parte inferior da tela, em nosso aplicativo.

Nesse exemplo simples, omitimos o ícone, deixando apenas um título para cada tela acessível na guia. No entanto – e como já apontamos –, é possível estilizar a Tab adicionando ícones, assim como definir outras propriedades e comportamentos, como definir qual será a primeira tela exibida quando o aplicativo for carregado, entre outros exemplos.

Dica

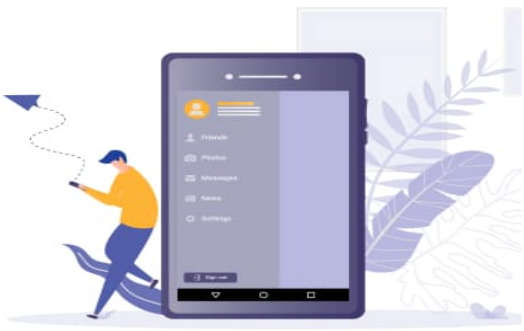
Tendo mantido o mesmo código para as screens Home e About, clique nos botões que incluímos no exemplo anterior. Perceba que é possível navegar entre as telas, no Tab Navigator, usando o props “navigation” e seu método “navigate”.

Drawer Navigation

Apresentação

O último modelo de navegação que veremos é o Drawer Navigation. Esse modelo consiste em fornecer um menu de navegação que inicialmente fica invisível na tela, sendo acessado ao se arrastar a tela da esquerda (ou, caso queiramos, da direita) em direção ao meio dela.

Ao se fazer tal movimento, as opções de navegação definidas dentro do Drawer ficam visíveis, permitindo, com isso, que se clique em uma de suas opções e navegue até a tela escolhida.



Exemplo de navegação com Drawer Navigation.

Dica

Para usar o Drawer, é necessário instalar a seguinte dependência: @react-navigation/drawer

Estrutura do projeto

Manteremos a estrutura utilizada nos modelos de navegação abordados anteriormente, criando um projeto e reaproveitando os códigos das telas Home e About. Com isso, precisamos modificar apenas o ponto de entrada de nosso aplicativo.

Criação do Drawer Navigation

Após criar um projeto, modifique o App.js para que ele fique desta forma:

```
plain-text

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

 import HomeScreen from './screens/home';
 import AboutScreen from './screens/about';

 const Drawer = createDrawerNavigator();

 function App() {
     return (









     );
 }

 export default App;
```

Análise do código

O código para criação do Drawer é semelhante aos códigos dos demais modelos de navegação, onde temos as operações listadas a seguir:

1. Importar a dependência e as telas a serem navegadas.
2. Criar a constante que vai conter a instância do componente de navegação.
3. Definir os itens de navegação dentro do NavigationContainer.

Atenção

No Drawer, em específico, foi incluído um atributo adicional, que é opcional, por meio do qual definimos a tela inicial a ser apresentada. Quando ela não é definida por meio de atributo, em todos os modelos de navegação, por padrão, é exibida a primeira tela definida dentro dos itens: screens.

Navegação entre telas

O Drawer cria um menu (estilo hambúrguer ou sanduíche) no topo/header de nosso aplicativo. Tal menu contém o link para as telas incluídas como screens no Drawer.Navigator. Além disso, esses links também ficam visíveis quando se arrasta a tela do canto esquerdo para o meio. Por fim, a aplicação de determinados métodos permite:

Navegar entre as telas

Usando os métodos disponíveis no props "navigation", como o navigate ou o goBack.

Abrir e fechar o Drawer

Empregando os métodos "navigation.openDrawer" e "navigation.closeDrawer".

Combinando os modelos de navegação

Os modelos de navegação apresentados podem ser usados individualmente, como demonstramos nos códigos apresentados no módulo, ou de forma combinada.

> Caso a opção seja pela combinação, é importante saber que só é possível ter um NavigationContainer em nosso aplicativo. Assim, para combinar os modelos de navegação, será necessário definir um(s) como item(s) do(s) outro(s) na propriedade "component".

Outro aspecto que precisa ser levado em consideração ao se combinar mais de um modelo de navegação é que tal comportamento tende a fazer mais sentido quando existem muitas telas em nosso aplicativo. Ou seja, para um aplicativo com apenas duas telas, como o que usamos de exemplo, não faz muito sentido combinar as navegações.

Modelos de navegação no React Native

Os três principais modelos de navegação utilizados no design de interface interativa com o React Native são abordados no vídeo a seguir.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Vem que eu te explico!

Os vídeos a seguir abordam os assuntos mais relevantes do conteúdo que você acabou de estudar.

Stack Navigation

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Drawer Navigation

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Verificando o aprendizado

Questão 1

Ao compararmos o modelo de navegação tradicional em um website ou sistema web com os modelos disponíveis no React Native, podemos dizer que:

A

São modelos totalmente diferentes que não compartilham nenhum tipo de semelhança entre si.

B

Em websites, há apenas um modelo de navegação disponível por meio de links contidos em menus, enquanto no React Native existem três diferentes modelos.

C

Assim como no website, em React Native só pode haver um modelo de navegação em nosso aplicativo, não sendo possível combinar os diferentes modelos existentes.

D

O StackNavigator é o único modelo de navegação em React Native a manter uma semelhança com os modelos de navegação existentes em websites.

E

Os três principais modelos de navegação disponíveis para o React Native implementam, em linhas gerais, modelos bem conhecidos do ambiente web, como a navegação por links e botões, expostos em guias ou menus, sendo possível inclusive voltar e avançar em seu histórico de navegação.

A alternativa E está correta.

Com a utilização de bibliotecas externas, é possível adicionar ao React Native alguns componentes que permitem a criação de diferentes modelos de navegação. Tais componentes podem ser utilizados

individualmente ou combinados entre si, fornecendo diferentes formas de se navegar entre as telas de um aplicativo.

Questão

Considere um aplicativo que possua duas telas: HomeScreen e AboutScreen. Qual será a primeira tela exibida a partir do código a seguir, no qual dois diferentes modelos de navegação são combinados?

```
plain-text

//...
//imports
//...

const TabNav = createBottomTabNavigator();
function TabNavScreen() {
  return (




  );
}Com a utilização de bibliotecas externas, é possível adicionar ao React Native alguns componentes que permitem a criação de diferentes modelos de navegação. Tais componentes podem ser utilizados individualmente ou combinados entre si, fornecendo diferentes formas de se navegar entre as telas de um aplicativo.

const Drawer = createDrawerNavigator();
function DrawerScreen(){
  return (




  );
}


function App() {
  return (







  );
}

export default App;
```

A

A tela AboutScreen contendo o menu de acesso ao Drawer Navigator.

B

A tela AboutScreen sem nenhum elemento que permita navegar para a tela HomeScreen.

C

A tela AboutScreen com o Tab Bar Navigator, tendo os botões de navegação para a HomeScreen e a própria AboutScreen.

D

A tela HomeScreen contendo o menu de acesso ao Drawer Navigator.

E

Nenhuma tela será exibida, sendo apresentado um erro informando que não foi possível localizar o componente TabNavScreen.

A alternativa A está correta.

Podemos combinar diferentes componentes de navegação no React Native. Para isso, precisamos definir cada modelo, seus itens (ou screens) e suas propriedades. No exemplo acima, o principal componente de navegação é o Drawer Navigator, que possui dois itens: o name= "Home", apontando para o componente "TabNavScreen"; e o name="About", voltado para o componente "AboutScreen". O primeiro componente que deveria ser exibido era o TabNavScreen (que tem como primeiro item o HomeScreen, seguido do AboutScreen), uma vez que ele foi declarado antes. Entretanto, a propriedade initialRouteName, no Drawer Navigator, aponta para "About". Com isso, a primeira tela carregada será a AboutScreen, contendo o menu de acesso ao Drawer Navigator.

Estilizando aplicativos em React Native

Estilização de aplicativo.

A estilização de aplicativos em React Native segue os mesmos princípios daquela realizada em páginas web. No entanto, enquanto, nessas páginas, utilizam-se as folhas de estilo **(CSS)**, no React Native emprega-se um código JavaScript por meio da propriedade (props) **style**.

Essa propriedade é nativa e pode ser aplicada em qualquer componente. Além dela, há a possibilidade de utilizar uma biblioteca denominada **styled**. Veremos a seguir como é possível, de forma nativa, estilizar o nosso aplicativo.

Saiba mais

CSS A estilização de aplicativos em React Native segue os mesmos princípios daquela realizada em páginas web. No entanto, enquanto nessas páginas utilizam-se as folhas de estilo (CSS - do inglês Cascading Style Sheets), no React Native emprega-se um código JavaScript por meio da propriedade (props) style.

Como funciona a propriedade Style

Como já mencionamos, os estilos são definidos usando o código JavaScript. Em relação às propriedades, temos algo bastante similar à CSS, com a diferença de que, em React Native, as propriedade são declaradas no padrão camelCase. A propriedade CSS margin-top em React seria, portanto, declarada como marginTop. Outra semelhança entre a **props Style** e a CSS é a possibilidade de definir nossos estilos de forma:

- Inline no próprio componente.
- Interna no mesmo script.
- Externa em um script separado.

Este fragmento demonstra a definição inline da cor do texto para o componente Text:

```
plain-text

//...
 Texto Azul
 //...
```

Podemos ver adiante o mesmo estilo sendo declarado de forma interna. Note que, para isso, usamos o método **StyleSheet**.create, que pertence ao core do React Native e precisa ser importado:

```
plain-text

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

 const Estilos = () => {
     return (
             Texto Azul
     );
 };

 const styles = StyleSheet.create({
     textoAzul: {
     color: 'blue'
     },
 });

 export default Estilos;
```

A utilização do Style de forma externa segue a mesma convenção que a interna. É necessário, assim, haver a importação do script (que precisa ter a extensão .js) de estilo no script em que ele será usado, como pode ser visto nos códigos a seguir, começando pelo App.js:

```
plain-text

import React from 'react';
 import { Text } from 'react-native';
 import styles from './styles'

 const Estilos = () => {
     return (
             Texto Azul
     );
 };

 export default Estilos;
```

Agora, observe o código styles.js:

```
plain-text

import { StyleSheet } from 'react-native';

 const styles = StyleSheet.create({
     textoAzul: {
     color: 'blue'
     },
 });

 export default styles;
```

Styled Components

Apresentação

A biblioteca **Styled Components** constitui uma alternativa muito utilizada para a props Style quando o assunto é estilizar um componente em React Native. Com o uso dessa lib, é possível estilizar nosso aplicativo com CSS de forma bem parecida com a que fazemos na web.

Como utilizar a Styled Components

Como toda dependência em React Native, para se usar a Style, é preciso instalá-la. Seu gerenciador de dependências usual instala a seguinte lib:

- styled-components

No exemplo prático a seguir, é possível verificar a utilização da Styled Components:

```
plain-text

import React from 'react';
 import { Text } from 'react-native';
 import styled from 'styled-components/native'

 const Estilos = () => {
     return (
             Texto Azul
     );
 };

 const StyledText = styled.Text`
     color: blue;
     margin-top: 50px;
     margin-left:50px;
 `;

 export default Estilos;
```

As propriedades de estilo são declaradas no formato tradicional da CSS. Já seu estilo é definido com a notação de componente. No exemplo, é criado um componente Text, o StyledText, para o qual são definidas algumas propriedades de estilo. Em seguida, esse componente é usado para conter o texto “Texto Azul”.

Além da possibilidade de usar as propriedades CSS de forma similar à utilizada na web, a lib Style Components tem várias outras funcionalidades disponíveis. Apontaremos três exemplos a seguir:

1. Entender (e sobrescrever) estilos.
2. Passar as propriedades de estilo entre os componentes como props.
3. Ter suporte a animações (que será visto a seguir).

Animações em React Native

Em React Native, estão disponíveis dois recursos nativos para a animação de objetos: a **Animated API** e a **LayoutAnimation API**. Com os recursos disponibilizados por ambas, é possível, entre outras, realizar as seguintes ações:

- Animar os componentes de nossa aplicação;
- Inserir efeitos de fade in ou fade out;
- Mover itens de lugar a partir da interação do usuário.

Conheceremos agora algumas **propriedades dessas duas APIs**, assim como alguns exemplos de código.

Animated API

Funcionamento

Esta API se baseia em relações declarativas entre entradas e saídas, fazendo uso de transformações configuráveis por meio de métodos, como iniciar e parar, por meio dos quais podemos controlar a sua execução. Nativamente, é possível animar os seguintes componentes React:

- View
- Text
- Image
- ScrollView
- FlatList
- SectionList

Além disso, também é possível criar componentes personalizados e utilizar o método **Animated.createAnimatedComponent()** para animá-los.

Veremos em seguida nosso primeiro exemplo de animação. No entanto, antes de vermos o código, temos de fazer algumas considerações:

Hooks

No código, são utilizados dois hooks: useRef e useEffect. Não se preocupe ainda em relação a eles. Por enquanto, saiba que o useEffect permite que ações sejam executadas quando o aplicativo for renderizado na tela.

Animated.timing

Veja a sintaxe do método Animated.timing. São definidos nele o valor até o qual o efeito de fade será aplicado e o tempo de execução da animação. Quanto maior for o tempo, mais lenta será a execução.

Animated.Text

No exemplo, o Animated.Text é utilizado como container para o texto a ser exibido a aplicação da animação. Como já mencionamos, também poderíamos ter usado o Animated.View e outros elementos.

Vamos ao código. Nele, a frase “Texto com fade in” será exibida com o efeito de fade in quando o aplicativo for iniciado.

```
plain-text

import React, { useRef, useEffect } from 'react';
 import { Animated, Text, View } from 'react-native';

 const FadeInText = (props) => {
     const fadeAnim = useRef(new Animated.Value(0)).current

     React.useEffect(() => {
     Animated.timing(
         fadeAnim,
         {
         toValue: 1,
         duration: 1000,
         }
     ).start();
     }, [fadeAnim])

     return (

         {props.children}

     );
 }

 export default () => {
     return (


         Texto com Fade In


     )
 }
```

Outros recursos da Animated API

O exemplo anterior demonstra uma animação simples. Entretanto, a Animated API possui vários outros recursos, como, por exemplo:

- Configuração de animações (tempo, duração, funções de atenuação etc.).
- Composição de animações (combinação de animações em sequência, em paralelo etc.).
- Combinação de valores animados (por meio de operações matemáticas).
- Interpolação.
- Rastreamento de valores e gestos.

Saiba mais

Cada um desses recursos pode ser explorado a fundo por meio da documentação oficial do React Native.

LayoutAnimation API

Funcionamento e análise de código

Demonstraremos o funcionamento da LayoutAnimation API. Em seguida, faremos comentários sobre o código utilizado. Vamos lá!

```
plain-text

import React from 'react';
 import { NativeModules, LayoutAnimation, Text, TouchableOpacity,
     StyleSheet, View, } from 'react-native';

 const { UIManager } = NativeModules;

 UIManager.setLayoutAnimationEnabledExperimental &&
     UIManager.setLayoutAnimationEnabledExperimental(true);

 export default class App extends React.Component {
     state = {w: 200, h: 200, };

     _onPress = () => {
     LayoutAnimation.spring();
     this.setState({w: this.state.w - 5, h: this.state.h - 5})
     }

     render() {
     return (




             Pressione para diminuir o quadrado



     );
     }
 }

 const styles = StyleSheet.create({
     container: {
     flex: 1,
     alignItems: 'center',
     justifyContent: 'center',
     },
     box: {
     width: 200,
     height: 200,
     backgroundColor: 'orange',
     },
     button: {
     backgroundColor: 'black',
     paddingHorizontal: 20,
     paddingVertical: 15,
     marginTop: 15,
     },
     buttonText: {
     color: '#fff',
     fontWeight: 'bold',
     },
 });
```

O código anterior exibe na tela dois elementos: um quadrado laranja e um botão preto contendo o texto “Pressione para diminuir o quadrado”. Cada vez que o botão é pressionado, esse quadrado diminui de tamanho. Faremos três comentários sobre o código:

Comentário 1

O primeiro ponto de atenção no código diz respeito a este fragmento:

```
UIManager.setLayoutAnimationEnabledExperimental
UIManager.setLayoutAnimationEnabledExperimental(true);
```

Essa instrução deve ser inserida para que a animação funcione na plataforma Android.

Comentário 2

Nesse exemplo, é utilizado state (estado), que é responsável por definir a largura e a altura setados inicialmente para o quadrado. O state, a exemplo do props, é um objeto JavaScript utilizado para guardar informações mutáveis, sendo gerenciado dentro de cada componente. Podemos, por ora e de forma mais simplista, dizer que o state é semelhante a uma variável.

Comentário 3

Em relação à animação, o pressionamento do botão faz com que a função _onPress seja executada. Essa função utiliza o LayoutAnimation para animar o efeito de diminuição das dimensões do quadrado.

Quando utilizar a LayoutAnimation API

Como o próprio nome sugere, tal recurso de animação é voltado para a animação do layout do aplicativo ao longo do seu ciclo de renderização (inicialização e atualização). Sob esse viés, vê-se que a LayoutAnimation API é bastante útil nas situações em que um conteúdo inicialmente escondido, ao ser exibido, afeta o dimensionamento de seus elementos anteriores ou posteriores. Como exemplo, temos os links de “veja mais”.

Atenção

Embora seja um recurso poderoso, a LayoutAnimation API oferece menos controle que a Animated API.

Estilização e animação do React Native

Neste vídeo, abordaremos os recursos de estilização e animação disponíveis no React Native.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Vem que eu te explico!

Os vídeos a seguir abordam os assuntos mais relevantes do conteúdo que você acabou de estudar.

A propriedade Style

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Animated API

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Verificando o aprendizado

Questão 1

A estilização de um aplicativo mobile é bastante similar à de uma página web, definindo-se qual elemento se quer estilizar e as propriedades que se deseja modificar de forma inline, interna ou externa. Selecione a alternativa correta na estilização de aplicativos mobile.

A

Embora bastante similares, as páginas web contam com uma gama muito maior de propriedades de estilo, já que as páginas são maiores, uma vez que são exibidas em desktops.

B

As similaridades na aplicação de estilos entre esses dois ambientes nos permitem copiar um código CSS utilizado em uma página web e aplicá-lo diretamente em um elemento no aplicativo para que o código funcione sem nenhum tipo de adaptação.

C

A estilização de um aplicativo escrito com o framework React Native é feito com a utilização de propriedades CSS semelhantes àquelas utilizadas no ambiente web, tendo como única diferença a necessidade de escrever cada propriedade no padrão camelCase.

D

O recurso de estilização nativo Style é muito poderoso. Entretanto, apenas com sua utilização não é possível estilizar um aplicativo por completo, sendo necessário, como complemento, incluir uma biblioteca externa, como a Styled Components.

E

A estilização *inline* deve ser evitada em React Native, pois produz um componente com muitas linhas de código, fato esse que gera lentidão na renderização de tal elemento.

A alternativa C está correta.

Em React Native, de forma nativa, estão disponíveis as mesmas propriedades presentes na CSS. Tais propriedades podem ser aplicadas em quaisquer componentes. Além disso, ainda existe a possibilidade de utilizar, isoladamente ou até mesmo de forma combinada, bibliotecas externas, como a Styled Components, que possui recursos avançados, como permitir, entre outros exemplos, a aplicação dinâmica/condicional de estilos.

Questão 2

A utilização de animações é um recurso importante em aplicativos, porque elas tendem a melhorar a experiência do usuário, oferecendo interações mais fluidas no lugar de elementos simplesmente estáticos. Nesse contexto, da utilidade das animações e de sua implementação em React Native, é correto afirmar que:

A

Embora importantes, devemos limitar a quantidade de animações utilizada em um aplicativo a fim de resguardar os recursos do dispositivo, além de melhorar a performance.

B

Em React Native, limitações técnicas nos impedem de animar o layout de nosso aplicativo, sendo possível apenas aplicar efeitos de fade.

C

As animações podem adicionar dicas visuais que informam aos usuários sobre o que está acontecendo no aplicativo, sendo bastante úteis nas mudanças de estado, como, por exemplo, quando um novo conteúdo é carregado a partir do click em um botão ou por meio do scroll ao longo da tela.

D

O framework React Native não possui nenhum recurso nativo que permita a aplicação de animações. Para utilizar esse recurso, é necessária a instalação de dependências externas, como a lib Styled Components.

E

A Animated API permite que alguns componentes nativos sejam animados. A sintaxe de sua utilização consiste em usar o nome do componente seguido da palavra “animated”, como, por exemplo, em .

A alternativa C está correta.

As animações são um importante recurso em um aplicativo mobile, sobretudo quando usadas em resposta a interações ou ações do usuário, como o click em um botão ou a alternância entre uma tela e outra. Tal importância vai além de se tornar a aplicação visualmente mais atrativa. No React Native, estão disponíveis nativamente duas APIs para animar os componentes e o layout como um todo: a Animated API e a LayoutAnimation API.

Considerações finais

Neste conteúdo, apresentamos alguns recursos importantes do React Native de forma prática e teórica, a começar pela esquematização dos passos para a definição da interface interativa de um aplicativo mobile. Dos passos apresentados, conseguimos introduzir alguns componentes do React voltados para a construção de layout e a conteinerização de conteúdos e outros elementos.

Além desses componentes, vimos, na sequência, alguns com funções mais específicas: os componentes de lista e multivalorados. Em seguida, descrevemos os três principais modelos de navegação em um aplicativo mobile disponíveis no framework.

Por fim, abordamos os recursos de estilização e animação. Esses recursos são essenciais para o desenvolvedor mobile tirar o melhor proveito das funcionalidades de construção de interface interativa do React Native.

Podcast

Ouça o podcast. Nele, falamos sobre o processo de definição da interface de um aplicativo mobile, componentes de lista, modelos de navegação e os recursos de estilização e animação do React Native.

Conteúdo interativo

Acesse a versão digital para ouvir o áudio.

Explore +

Explore outros recursos sobre animações e transições em aplicativos mobile no site do Android Developer.

DEVELOPERS. **Animações e transições**. Consultado na internet em: 12 ago. 2021.

Outra fonte rica em informações sobre o desenvolvimento mobile é a página do Material Design.

MATERIAL DESIGN. **Foundation**. Consultado na internet em: 12 ago. 2021.

Referências

REACT NATIVE. **Docs**. Consultado na internet em: 12 ago. 2021.