Desenvolvimento para Dispositivos Móveis

Aula 14 - Criando App Blog

Material Didático do Instituto Metrópole Digital - IMD

Termo de uso

Apresentação

Nesta aula, vamos consolidar todo o conhecimento adquirido nas últimas aulas. Será construído um aplicativo que simula um blog.

Objetivos

- Conhecer como construir um aplicativo do início ao fim
- Como criar um aplicativo que simule um blog
- Conhecer como aplicar os conteúdos abordados até o momento

Apresentação do projeto

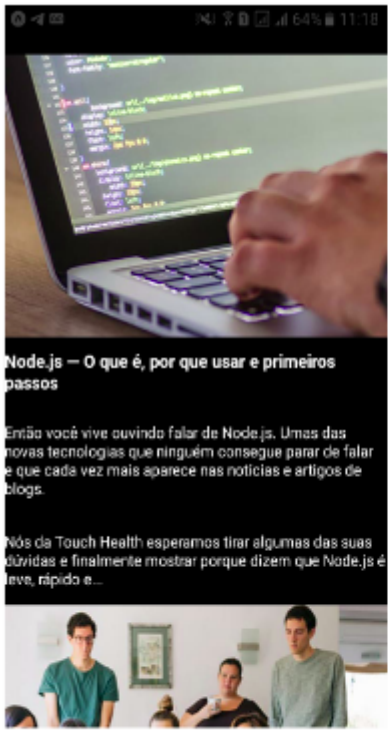
Link do video da aula: https://youtu.be/jthPdjCTYBo

Inicialmente, iremos apresentar o projeto que será construído durante toda a aula.

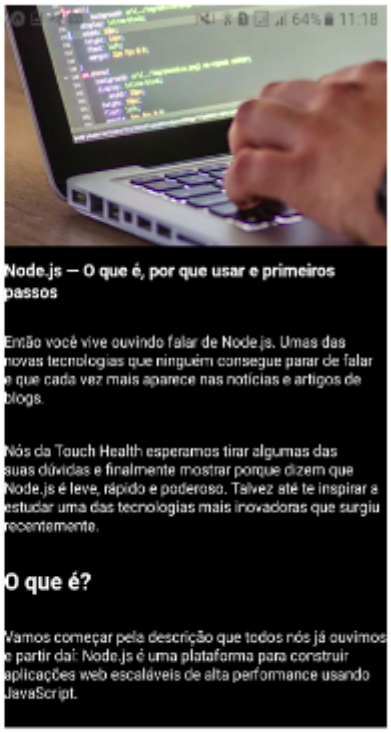
Conhecendo o projeto

Iremos construir um aplicativo que simula um blog e abordar a maior parte dos assuntos vistos até o momento.

Veja imagens do aplicativo abaixo:



Home



Post

Telas

Criando a estrutura

Link do video da aula: https://youtu.be/QkYGsXi3im8

Para iniciar, utilizaremos um projeto simples, resultado do comando abaixo:

```
expo init ProjectName
```

Instalando bibliotecas

Como pudemos ver na apresentação do projeto, serão criadas duas telas: uma com a listagem das postagens e outra com os detalhes de cada *post*. Então, será preciso criar a navegação entre as telas.

Antes, porém, precisamos instalar algumas bibliotecas para o bom funcionamento do código.

Execute em seu terminal os comandos abaixo:

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

Em seguida:

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

Navegação entre telas

Antes de tudo, para utilizar a navegação precisaremos iniciar a *Stack*. Para isso, faz-se necessário a importação de uma função e a chamada no código.

Veja o exemplo abaixo:

```
import { createNativeStackNavigator } from '@react-
navigation/native-stack';

const Stack = createNativeStackNavigator()
```

Como já visto na aula de navegação, temos de envolver toda a parte que desejamos que haja navegação com um componente denominado *NavigationContainer*. Esse componente envolverá as *Stack Screen*, que recebem como uma de suas propriedades o componente.

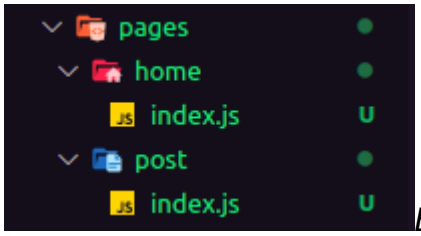
Ao serem exibidos na tela, os componentes, utilizando a navegação, exibem um cabeçalho com o nome da tela atual. Para remover essa informação, além das propriedades principais (*name* e *component*), adicionaremos uma *prop* chamada de *options* que setará o *headerShown* para falso.

Veja o exemplo abaixo:

```
<NavigationContainer>
      <Stack.Navigator>
          <Stack.Screen name="Home" component={Home}
options={{headerShown: false}}></Stack.Screen>
          <Stack.Screen name="BlogPost" component={PostPage}
options={{headerShown: false}}></Stack.Screen>
      </Stack.Navigator>
</NavigationContainer>
```

Criando os componentes

Foram passados os componentes para a *Stack Screen*, porém, ainda é necessário criá-los. Então, na raiz do projeto, crie uma pasta chamada *pages* e nela crie as pastas *home* e *post* com seus respectivos índices. Veja a estrutura abaixo:



Estrutura de pastas

Acompanhando a aula, implemente os arquivos como abaixo:

home/index.js

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

export default function Home() {
    return <View><Text>Home</Text></View>
}
```

post/index.js

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

```
export default function BlogPost() {
    return <View><Text>BlogPost</Text></View>
}
```

Listando os posts (Parte 1)

Link do video da aula: https://youtu.be/6cDKOI7oAq8

Agora vamos iniciar de maneira efetiva a construção da tela de *posts* carregando as informações que serão exibidas na tela. Os dados que serão utilizados serão carregados de uma *API* criada na disciplina de desenvolvimento *back-end*.

Armazenando em um estado

Os dados carregados precisam ser armazenados em um estado para serem utilizados em nosso aplicativo. Além disso, precisamos de outro estado que informe se os dados concluíram o carregamento.

Então, em seu arquivo, crie os estados abaixo:

```
const [posts, setPosts] = useState([])
const [isLoading, setLoading] = useState(true)
```

Carregando os dados

Para carregar dados, iremos criar uma função assíncrona e nela utilizaremos o *fetch*. Os dados carregados serão armazenados no estado criado anteriormente.

Acompanhe a aula e implemente a função abaixo:

```
const getPostsAPI = async () => {
        try {
            setLoading(true)
            const response = await
fetch('https://dev-backend-imd.herokuapp.com/api/posts')
            const posts = await response.json()
            setPosts(posts.posts)
        } catch (error) {
            alert('Falha ao carregar postagens. Tente novamente mais
tarde.')
            setPosts([])
        } finally {
            setLoading(false)
```

```
        }
}
```

Essa função criada deve ser chamada quando o aplicativo iniciar. Então, utilizaremos para isso o *Hook UseEffect( )*.

Veja abaixo:

```
useEffect(() => {
        getPostsAPI()
}, [])
```

Utilizando o *FlatList*

Com os dados já carregados, agora podemos iniciar a listagem dos itens. Nesse aplicativo, utilizaremos o *FlatList*.

Acompanhe a aula e produza o código abaixo:

```
import React, { useEffect, useState } from 'react'
import { View, StyleSheet, FlatList } from 'react-native'

export default function Home() {

    const [posts, setPosts] = useState([])
    const [isLoading, setLoading] = useState(true)

    const getPostsAPI = async () => {
        try {
            setLoading(true)
            const response = await
fetch('https://dev-backend-imd.herokuapp.com/api/posts')
            const posts = await response.json()
            setPosts(posts.posts)
        } catch (error) {
            alert('Falha ao carregar postagens. Tente novamente mais
tarde.')
            setPosts([])
        } finally {
            setLoading(false)
        }
    }

    useEffect(() => {
```

```
        getPostsAPI()
    }, [])

    return (
        <View style={styles.container}>
            <FlatList data={posts}></FlatList>
        </View>
    )
}

const styles = StyleSheet.create({
    container: {
        width: '100%',
        height: '100%',
        backgroundColor: 'red'
    }
})
```

Listando os posts (Parte 2)

Link do video da aula: https://youtu.be/BrMbAZ9T9gQ

Dando sequência à listagem dos itens, vamos agora finalizar a definição do FlatList.

Criando componente de renderização

Em nosso componente de listagem precisamos passar uma propriedade que seja um outro componente que represente a renderização de um item.

Acompanhando a aula, implemente o componente abaixo:

```
const Item = (props) => {
    return (
        <View>
            <Image style={styles.img} source={{ uri: props.item.foto
}}> </Image>
            <Text style={styles.titulo}>{props.item.titulo}</Text>
        </View>
    )
}
```

Passando o componente para o *FlatList*

Agora utilizaremos o componente criado no *FlatList* para listar os dados. Além de passar esse componente no *renderItem*, iremos passar a *keyExtractor* para identificação única de cada objeto da lista.

Veja abaixo:

```
<View style={styles.item}>
            <Image style={styles.img} source={{ uri: props.item.foto
}}> </Image>
            <Text style={styles.titulo}>{props.item.titulo}</Text>
</View>
```

Estilizando o componente

Com os itens listados, realizaremos agora uma estilização para melhoria da visualização dos *posts*. Acompanhe a aula e implemente o código.

Ao final, seu arquivo deve estar como abaixo:

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

const Item = (props) => {
    return (
        <View style={styles.item}>
            <Image style={styles.img} source={{ uri: props.item.foto
}}> </Image>
            <Text style={styles.titulo}>{props.item.titulo}</Text>
        </View>
    )
}

export default function Home() {

    const [posts, setPosts] = useState([])
    const [isLoading, setLoading] = useState(true)

    const getPostsAPI = async () => {
        try {
            setLoading(true)
            const response = await
```

```
fetch('https://dev-backend-imd.herokuapp.com/api/posts')
            const posts = await response.json()
            setPosts(posts.posts)
        } catch (error) {
            alert('Falha ao carregar postagens. Tente novamente mais
tarde.')
            setPosts([])
        } finally {
            setLoading(false)
        }
    }

    useEffect(() => {
        getPostsAPI()
    }, [])

    return (
        <View style={styles.container}>
            <FlatList style={styles.lista} data={posts}
renderItem={({ item }) => <Item item={item} />} keyExtractor={(item)
=> "#" + item.id}></FlatList>
        </View>
    )
}

const styles = StyleSheet.create({
    container: {
        width: '100%',
        height: '100%',
        backgroundColor: 'black'
    },
    item: {
        backgroundColor: 'black'
    },
    img: {
        width: '100%',
        height: 250
    },
    titulo: {
        fontSize: 16,
        fontWeight: '700',
        marginVertical: 10,
        color: 'white'
```

```
    },
    lista: {
        flex: 1,
        marginTop: 44
    }
})
```

Adicionando RenderHTML e Navegação

Link do video da aula: https://youtu.be/0qT2nABxtlI

Até o momento já estamos listando os itens na tela de *Home*, porém, ainda faltam alguns detalhes que daremos continuidade agora.

Loading da tela

Já foi criado o estado *isLoading* e ele já está sendo setado. Porém, ainda não o utilizamos na tela.

Para isso, utilizaremos o componente *ActivityIndicator* como se vê abaixo:

```
<View style={styles.container}>
            {isLoading
                ? <ActivityIndicator />
                : <FlatList style={styles.lista} data={posts}
renderItem={({ item }) => <Item item={item} />} keyExtractor={(item)
=> "#" + item.id}></FlatList>
            }
</View>
```

Instalando bibliotecas adicionais

Nosso texto do *post* vem no formato *HTML* e para manipulá-lo iremos precisar instalar uma biblioteca através do comando abaixo:

```
npm install react-native-render-html
```

Utilizando o *RenderHTML*

Com a biblioteca já instalada, utilizaremos esse componente em *Item*. Veja abaixo as modificações:

```
const Item = (props) => {

    const { width } = useWindowDimensions()
    const previewText = props.item.texto.substring(0, 300) + '...'
    const source = { html: `<div style='color:
white'>${previewText}</div>` }

    return (
        <View style={styles.item}>
            <Image style={styles.img} source={{ uri: props.item.foto
}}> </Image>
            <Text style={styles.titulo}>{props.item.titulo}</Text>
            <RenderHTML contentWidth={width}
source={source}></RenderHTML>
        </View>
    )
}
```

Navegando entre as telas

Agora queremos que ao clicar em uma postagem o usuário seja direcionado para uma tela nova contendo mais informações sobre o *post*.

Para isso, utilizaremos o componente *TouchableOpacity*. Veja abaixo como fazer o uso desse componente:

```
const Item = ({ item, navigation }) => {

    const { width } = useWindowDimensions()
    const previewText = item.texto.substring(0, 300) + '...'
    const source = { html: `<div style='color:
white'>${previewText}</div>` }

    return (
        <TouchableOpacity onPress={navigation.navigate('BlogPost', {
post: item })} style={styles.item}>
            <Image style={styles.img} source={{ uri: item.foto }}>
</Image>
            <Text style={styles.titulo}>{item.titulo}</Text>
            <RenderHTML contentWidth={width}
source={source}></RenderHTML>
        </TouchableOpacity>
    )
```

```
}
```

E no componente *FlatList*:

```
<FlatList style={styles.lista} data={posts} renderItem={({ item })
=> <Item navigation={navigation} item={item} />}
keyExtractor={(item) => "#" + item.id}></FlatList>
```

Refatoração para extrair um componente

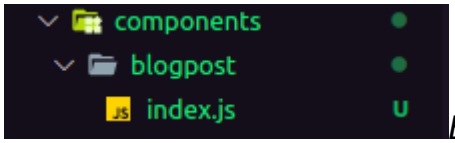
Link do video da aula: https://youtu.be/njFVi-p2SFQ

Nesta aula veremos como extrair as informações do *item*, fazendo dele um componente, para utilizar na tela de detalhes.

Criando arquivo

Na pasta raiz do projeto, crie uma pasta com o nome de *components*, e dentro dela uma pasta *blogpost* com o seu respectivo *index.js*.

Veja a estrutura de pastas abaixo:



Estrutura de pastas

Criando o componente

Agora, em seu arquivo *index.js* devemos ter o código do componente. Siga o passo a passo da aula e implemente o código abaixo:

```
import React from 'react'
import { Text, StyleSheet, Image, TouchableOpacity } from 'react-
native'
import RenderHTML from 'react-native-render-html'

export default function BlogPost({ item, navigation, width }) {

    const previewText = item.texto.substring(0, 300) + '...'
    const source = { html: `<div style='color:
white'>${previewText}</div>` }
```

```
    return (
        <TouchableOpacity onPress={() =>
navigation.navigate('BlogPost', { post: item })}
style={styles.item}>
            <Image style={styles.img} source={{ uri: item.foto
}}></Image>
            <Text style={styles.titulo}>{item.titulo}</Text>
            <RenderHTML contentWidth={width}
source={source}></RenderHTML>
        </TouchableOpacity>
    )

}

const styles = StyleSheet.create({
    item: {
        backgroundColor: 'black'
    },
    img: {
        width: '100%',
        height: 250
    },
    titulo: {
        fontSize: 16,
        fontWeight: '700',
        marginVertical: 10,
        color: 'white'
    }
})
```

Utilizando o componente *BlogPost*

Com o componente quase pronto, já podemos utilizá-lo no nosso *Home*. Para isso, faça as alterações abaixo no arquivo:

```
import React, { useEffect, useState } from 'react'
import { View, StyleSheet, FlatList, ActivityIndicator,
useWindowDimensions } from 'react-native'
import BlogPost from '../../components/blogpost'

export default function Home({ navigation }) {

    const [posts, setPosts] = useState([])
    const [isLoading, setLoading] = useState(true)
```

```
    const { width } = useWindowDimensions();

    const getPostsAPI = async () => {
        try {
            setLoading(true)
            const response = await
fetch('https://dev-backend-imd.herokuapp.com/api/posts')
            const posts = await response.json()
            setPosts(posts.posts)
        } catch (error) {
            alert('Falha ao carregar postagens. Tente novamente mais
tarde.')
            setPosts([])
        } finally {
            setLoading(false)
        }
    }

    useEffect(() => {
        getPostsAPI()
    }, [])

    return (
        <View style={styles.container}>
            {isLoading
                ? <ActivityIndicator />
                : <FlatList style={styles.lista} data={posts}
renderItem={({ item }) => <BlogPost width={width}
navigation={navigation} item={item} />} keyExtractor={(item) => "#"
+ item.id}></FlatList>
            }
        </View>
    )
}

const styles = StyleSheet.create({
    container: {
        width: '100%',
        height: '100%',
        backgroundColor: 'black'
    },
    lista: {
        flex: 1,
        marginTop: 44
```

```
    }
})
```

Criando página de detalhes

Link do video da aula: https://youtu.be/iC3uStcA6hQ

Nesta aula vamos dar continuidade à construção do *blog*, dessa vez, implementando a tela de detalhes.

Implementado a página

Em seu arquivo *index.js* vamos criar a página de *posts* utilizando o componente criado na aula anterior, o *BlogPost*. Então, acompanhando o passo a passo da aula, implemente o código abaixo:

```
import React from 'react'
import { View, StyleSheet, useWindowDimensions } from 'react-native'
import BlogPost from '../../components/blogpost'

export default function PostPage({ route, navigation }) {

    const { width } = useWindowDimensions()

    return <View style={styles.container}>
        <BlogPost width={width} item={route.params.post}
navigation={navigation}></BlogPost>
    </View>
}

const styles = StyleSheet.create({
    container: {
        flex: 1,
        backgroundColor: 'black'
    }
})
```

Ajustes do componente BlogPost

Link do video da aula: https://youtu.be/2w54_E6v_7k

Com a tela de detalhes já sendo exibida, é perceptível que ainda existem melhorias para serem feitas nela.

Preview Props

O componente está se comportando do mesmo modo na pré-visualização e no detalhamento. Porém, querem comportamentos distintos. Por exemplo: nos detalhes o componente não deve ser clicável e deve possuir a barra de rolagem.

Para resolver isso, vamos inserir uma nova *prop* chamada *preview*, que quando verdadeira se comporta como na tela de *Home*, mas caso seja falsa, se comporta como na tela de *Posts*.

Sendo assim, acompanhe o passo a passo da aula e altere seu componente. Ao final, ele deverá ficar como abaixo:

```
import React from 'react'
import { Text, StyleSheet, Image, TouchableOpacity, ScrollView, View
} from 'react-native'
import RenderHTML from 'react-native-render-html'

export default function BlogPost({ preview, item, navigation, width
}) {

    const previewText = preview ? item.texto.substring(0, 300) +
'...' : item.texto
    const source = { html: `<div style='color:
white'>${previewText}</div>` }

    const base = (
        <View><Image style={styles.img} source={{ uri: item.foto
}}></Image>
            <Text style={styles.titulo}>{item.titulo}</Text>
            <RenderHTML contentWidth={width}
source={source}></RenderHTML></View>)

    if (preview) {
        return (
            <TouchableOpacity onPress={() =>
navigation.navigate('BlogPost', { post: item })}
style={styles.item}>
                {base}
            </TouchableOpacity>)
```

```
    } else {
        return (<ScrollView>
            {base}
        </ScrollView>)
    }

}

const styles = StyleSheet.create({
    item: {
        backgroundColor: 'black'
    },
    img: {
        width: '100%',
        height: 250
    },
    titulo: {
        fontSize: 16,
        fontWeight: '700',
        marginVertical: 10,
        color: 'white'
    }
})
```

Utilizando a *prop*

Agora que já é possível que o componente se comporte de formas diferentes, devemos, ao utilizar o componente, enviar uma *prop* que diferencie esse comportamento.

Então, ao utilizar o componente *BlogPost*, é necessário passar o valor de *preview*. Veja um exemplo abaixo:

```
<BlogPost preview={false} width={width} item={route.params.post}
navigation={navigation}></BlogPost>
```

Projeto final

Sendo assim, chegamos ao final da construção do *blog*. Você pode encontrar os arquivos completos clicando aqui.

Resumo

Nesta aula vimos como criar um aplicativo que simule um *blog*, do início ao fim. Além disso, foi abordado como podemos combinar os componentes para trabalharem juntos.