Fundamentação Teórica: React e React Native

Este artigo apresenta uma fundamentação teórica sobre o React e o React Native, cobrindo definições, arquitetura, principais conceitos (componentização, JSX, props e state), pré-requisitos de linguagem (ES6+), ferramentas (Expo) e a comunicação nativa (Bridge). Também inclui orientações práticas para criar um projeto base (Gerenciador de Perfil Interativo) que utiliza navegação, gerenciamento de estado, fetch/AJAX e layout com Flexbox.

1. Introdução ao React e React Native

1.1 O que é React?

React é uma biblioteca JavaScript criada pelo Facebook para construir interfaces de usuário (UIs) de forma declarativa e baseada em componentes. Seu objetivo principal é facilitar a construção de interfaces complexas a partir de pequenos componentes reutilizáveis. React abstrai a manipulação direta do DOM (no caso da web) por meio do conceito de Virtual DOM, permitindo atualizações eficientes quando o estado da aplicação muda.

1.2 O que é React Native?

React Native é um framework que permite desenvolver aplicações móveis nativas usando JavaScript e a filosofia de componentes do React. Em vez de renderizar HTML em uma WebView, o React Native traduz componentes React em componentes nativos (Views, Text, Image etc.) para iOS e Android. Isso resulta em melhor performance e em uma aparência/experiência mais próxima do comportamento nativo.

Relação entre React e React Native:

- Ambos compartilham os conceitos centrais: componentização, conceito de props/estado, ciclo de vida e (quando aplicável) Hooks.
- Enquanto o React (web) renderiza elementos DOM, o React Native mapeia componentes para widgets nativos da plataforma.

1.3 Vantagens competitivas do React Native:

- **Reuso de lógica**: grande parte da lógica de negócio e de componentes pode ser reaproveitada entre plataformas.
- **Performance:** diferente de soluções baseadas em WebView, o React Native renderiza elementos nativos, o que melhora responsividade e performance em muitos cenários.
- **Desenvolvimento mais rápido:** por usar JavaScript/React, equipes com experiência web adaptam-se rapidamente.
- **Ecossistema e bibliotecas maduras:** React Navigation, Expo, e uma comunidade grande fornecem ferramentas e plugins prontos.

Limitações:

- Algumas features muito específicas da plataforma podem requerer código nativo (Java/Kotlin/Swift/Objective-C).
- Diferenças sutis entre plataformas que demandam ajustes de UI/UX.

1.4 Componentização

A componentização é a filosofia de quebrar a interface em unidades pequenas, independentes e reutilizáveis. Em React/React Native, componentes podem ser:

- **Funcionais (funções que retornam JSX**): muitas vezes com Hooks para gerenciar estado e efeitos.
- **Baseados em classes (class components)**: usam “this.state” e métodos do ciclo de vida como “componentDidMount()”.

Benefícios:

- Reutilização de código e facilidade para testar e manter.
- Separação clara entre responsabilidade de apresentação e lógica.

2. Sintaxe e Estrutura de Componentes

2.1 JSX

JSX é uma extensão de sintaxe do JavaScript que permite escrever elementos em uma aparência semelhante ao HTML/XML dentro de arquivos JavaScript. Em React Native, JSX descreve árvores de elementos que serão transformadas em componentes nativos.

Exemplo de JSX (React Native):

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


export default function Hello() {
 return (
  <View>
   <Text>Olá, mundo!</Text>
  </View>
 );
}
```

JSX não é HTML: trata-se de sintaxe que será compilada para chamadas de API do React. No React Native não existem tags HTML (como “div”), mas componentes nativos como “View”, “Text”, “Image”.

2.2 Props (Propriedades)

Props são argumentos recebidos por um componente — equivalem a parâmetros de função. Elas são imutáveis do ponto de vista do componente que as recebe. O padrão é passar dados e callbacks do componente pai para o filho via props.

Exemplo:

```
function Saudacao({ nome }) {
 return <Text>Olá, {nome}!</Text>;
}
```

Nesse exemplo, “nome” é uma prop que o componente “Saudacao” recebe e usa para renderizar texto.

2.3 State (Estado)

O state é um objeto (ou conjunto de valores) que representa dados internos de um componente que podem mudar ao longo do tempo. Em class components, o state é inicializado em “this.state” e atualizado por “this.setState()”. Em componentes funcionais usamos o Hook “useState”.

Class component (exemplo mínimo):

```
class Contador extends React.Component {
 constructor(props) {
  super(props);
  this.state = { count: 0 };
 }
 incrementar() {
  this.setState({ count: this.state.count + 1 });
 }
 render() {
  return <Text onPress={() => this.incrementar()}>{this.state.count}</Text>;
 }
}
```

Em um componente funcional com Hooks:

```
function Contador() {
 const [count, setCount] = React.useState(0);
 return <Text onPress={() => setCount(count + 1)}>{count}</Text>;
}
```

Importante: atualizar o state provoca re-render do componente e de seus filhos quando necessário — este é o mecanismo que mantém a UI sincronizada com os dados.

3. Pré-requisitos e Arquitetura

3.1 Funcionalidades ES6+ importantes

Recursos ES6+ frequentemente usados em projetos React/React Native:

- “import” / “export” — módulos claros e organização de código.
- “class” — sintaxe para componentização baseada em classes.
- Arrow functions (“() => {}”) — sintaxe curta e binding léxico do “this”.
- Template literals (““ “texto ${var}” ““) — concatenação legível.
- Array helpers (“map”, “filter”, “reduce”) — processamento declarativo de listas.

Esses recursos melhoram legibilidade e expressividade do código.

3.2 Ferramentas e Ambiente: Expo

Expo é um conjunto de ferramentas e serviços que simplifica o desenvolvimento com React Native. Entre as vantagens do Expo:

- Inicialização rápida de projeto (“npx create-expo-app” ou “expo init”).
- Hot reloading / live reload facilitado durante o desenvolvimento.
- Expo Go (aplicativo) permite testar a app no telefone escaneando um QR code (sem necessidade de build nativo imediato).
- Kits de APIs prontos (câmera, localização, notificações) que funcionam “out-of-the-box” em muitos casos.
- Limitações:
- Plugins nativos personalizados podem exigir “eject”/”prebuild” (ou usar EAS para builds nativos customizados).

3.3 Comunicação Nativa (Bridge)

A Bridge (ponte) é o mecanismo que permite ao JavaScript executar chamadas que atingem APIs nativas e vice-versa. Em React Native, o código JS roda em uma thread JavaScript separada (geralmente usando Hermes ou outro motor JS) e comunica-se com as APIs nativas por meio da Bridge. Operações pesadas nativas ou componentes customizados podem ser escritos em Java/Kotlin (Android) e Objective-C/Swift (iOS) e expostos ao JS.

Observações:

A Bridge pode ser um gargalo se houver muitas chamadas síncronas entre JS e nativo; por isso recomenda-se minimizar round-trips e trabalhar com lotes quando possível.

4. Criação do Projeto Base: "Gerenciador de Perfil Interativo" (aplicação prática)

Este tópico descreve a arquitetura prática necessária para atender à atividade proposta.

4.1 Requisitos básicos

- Node.js (versão LTS recomendada: 18.x ou 20.x).
- Expo (usar “npx expo” sem necessidade de instalar globalmente).
- React Navigation para gerenciamento de telas.

Passos iniciais (resumo):

```
npx create-expo-app GerenciadorPerfil

cd GerenciadorPerfil

npx expo start
```

4.2 Layout e estilização com Flexbox

No React Native usamos “StyleSheet.create” para declarar estilos. O layout baseia-se em Flexbox. Exemplos de propriedades:

- “flexDirection” — define o eixo principal (“column” por padrão).
- “justifyContent” — posicionamento ao longo do eixo principal.
- “alignItems” — alinhamento no eixo secundário.
- “flex” — define como um elemento cresce/ocupa espaço disponível.

Exemplo:

```
const styles = StyleSheet.create({

 container: { flex: 1, justifyContent: 'center', alignItems: 'center' },

 header: { flexDirection: 'row', justifyContent: 'space-between' }

});
```

Usar “flex: 1” em um contêiner geralmente indica que ele deve expandir para preencher o espaço disponível, o que é essencial para layouts responsivos.

4.3 Gerenciamento de Estado e Componentes Controlados

A atividade pede um “TextInput” controlado por estado — ou seja, o valor do input vem do state e cada alteração atualiza o state.

Exemplo em class component (para demonstrar “this.state” e “this.setState”):

```
class SearchScreen extends React.Component {
 constructor(props) {
  super(props);
  this.state = { usuario: '' };
 }

 onChangeText = (text) => {
  this.setState({ usuario: text });
 }

 render() {
  return (
   <TextInput value={this.state.usuario} onChangeText={this.onChangeText} />
  );
 }
}
```

Em componente funcional com Hooks:

```
function SearchScreen() {
 const [usuario, setUsuario] = React.useState('');
 return (
  <TextInput value={usuario} onChangeText={setUsuario} />
 );
}
```

4.4 Integração com API (AJAX)

A implementação deve incluir uma função “fetchDados” que realiza requisições à API (por exemplo, GitHub Users API) e atualiza o state com os dados recebidos.

Em class components, o ponto recomendado para iniciar a requisição é “componentDidMount()” — é chamado logo após o componente ser montado na árvore e garante que a UI esteja pronta antes de processar os dados recebidos.

```
Exemplo (class):
componentDidMount() {
 this.fetchDados('octocat');
}

async fetchDados(username) {
 const res = await fetch(“https://api.github.com/users/${username}”);
 if (!res.ok) throw new Error('Usuário não encontrado');
 const json = await res.json();
 this.setState({ dados: json });
}
```

Em componentes funcionais usamos “useEffect” com array de dependências vazio para emular “componentDidMount”:

```
React.useEffect(() => {
 fetchDados('octocat');
}, []);
```

Boas práticas:

- Tratar erros (try/catch) e estados de loading para melhorar a experiência do usuário.
- Evitar atualizar o state após o componente ser desmontado — cancele/promessa ou use flags quando apropriado.

4.5 Renderização condicional e navegação

Renderização condicional permite exibir diferentes componentes com base no estado (“loading”, “dados”, “error”). O operador ternário e guard clauses são úteis.

Exemplo:

```
{loading ? (
  <ActivityIndicator />
) : error ? (
  <Text>{error}</Text>
) : dados ? (
  <ProfileCard dados={dados} />
) : (
  <Text>Digite um usuário</Text>
)}
```

Para navegação, o React Navigation (Stack) é indicado:

```
import { NavigationContainer } from '@react-navigation/native';
import { createStackNavigator } from '@react-navigation/stack';
const Stack = createStackNavigator();


<NavigationContainer>
  <Stack.Navigator>
    <Stack.Screen name="Search" component={SearchScreen} />
    <Stack.Screen name="Profile" component={ProfileScreen} />
  </Stack.Navigator>
</NavigationContainer>
```

Passe dados entre telas via “navigation.navigate('Profile', { profile: dados })” e recupere-os em “route.params” na tela de destino.

5. Boas práticas e casos de borda

- Validação de entrada: tratar strings vazias, caracteres inválidos.
- Controle de estados “loading” e “error” para feedback ao usuário.
- Debounce em buscas se for permitir busca enquanto digita (para reduzir número de requisições).
- Lidar com limites de rate da API (GitHub impõe limites para requests não autenticadas).
- Considerar uso de cache local (AsyncStorage) em cenários offline ou para evitar requisições repetidas.

6. Uso de Hooks (conceito avançado)

Hooks (introduzidos no React 16.8) permitem usar estado e outros recursos do React em componentes funcionais. Os Hooks mais comuns:

- “useState” : estado local.
- “useEffect”: Efeitos colaterais (requisitions, timers, subscriptions).
- “useReducer”: gerenciamento de estado mais complexo.

Exemplo simples de “useState”:

```
const [contador, setContador] = React.useState(0);

return <Button onPress={() => setContador(c => c + 1)} title={“Contador: ${contador}”} />;
```

Quando usar classes vs Hooks:

- Hooks são o padrão atual e recomendável. Eles reduzem boilerplate em muitos cenários.
- Entretanto, a atividade pede demonstrar “componentDidMount()”; isso pode ser feito via class components ou através de “useEffect(() => {...}, [])” para obter comportamento equivalente.

7. Conclusão e mapa de entregas

Este documento cobriu os conceitos exigidos pela atividade: definição de React Native, vantagens, componentização, JSX, props e state, pré-requisitos ES6+, papel do Expo e a Bridge. Também forneceu orientações práticas para implementar o projeto base (Gerenciador de Perfil Interativo) com exemplos de código para “TextInput” controlado, fetch/AJAX, renderização condicional e navegação por pilha.

Referências:

Documentação React: https://reactjs.org

Documentação React Native: https://reactnative.dev

Documentação Expo: https://docs.expo.dev

React Navigation: https://reactnavigation.org