React Native - Primeiros passos

Adonai Pinheiro

Primeiros passos

> **Se você é novo no desenvolvimento móvel** , a maneira mais fácil de começar é com o Expo Go. Expo é um conjunto de ferramentas e serviços construídos em torno do React Native e, embora tenha muitos recursos , o recurso mais relevante para nós agora é que ele pode fazer você escrever um aplicativo React Native em minutos. Você precisará apenas de uma versão recente do Node.js e de um telefone ou emulador.

> **Se você já está familiarizado com o desenvolvimento móvel**, talvez queira usar o React Native CLI. Requer Xcode ou Android Studio para começar. Se você já tiver uma dessas ferramentas instaladas, poderá colocá-la em funcionamento em alguns minutos. Se eles não estiverem instalados, você deve esperar cerca de uma hora para instalá-los e configurá-los.

1- Preparando o ambiente

Para desenvolver aplicativos mobile utilizando precisamos configurar minimamente nosso ambiente de desenvolvimento com as ferramentas adequadas.

Basta seguir a documentação fornecida pela documentação oficial do React Native: https://reactnative.dev/docs/environment-setup?guide=native

2- RN CLI x Expo

React Native CLI e Expo são duas abordagens para o desenvolvimento de aplicativos móveis com React Native. Ambas têm suas vantagens e desvantagens, e a escolha entre elas depende dos requisitos e das metas do seu projeto. Aqui estão algumas razões pelas quais você ainda pode querer usar o React Native CLI em vez do Expo:

1. Flexibilidade e Personalização:

- React Native CLI oferece maior flexibilidade e controle sobre o projeto. Você pode adicionar nativamente módulos personalizados ou bibliotecas nativas que não estão disponíveis no Expo.
- Com o React Native CLI, você pode acessar diretamente os arquivos nativos do projeto, o que é essencial para tarefas avançadas de personalização.

2. Dependência de Bibliotecas Nativas:

- Se o seu aplicativo requer bibliotecas nativas específicas que não são suportadas pelo Expo ou precisa de código nativo personalizado, você precisará do React Native CLI para acomodar essas necessidades.

3. Tamanho do Aplicativo:

- Os aplicativos Expo tendem a ser maiores devido à inclusão de muitas dependências e bibliotecas predefinidas, mesmo que você não as utilize. Se o tamanho do aplicativo for crítico, o React Native CLI permite um controle mais preciso sobre as dependências.

4. Módulos Nativos Não Suportados:

- Se o seu aplicativo precisa de módulos nativos que não são suportados pelo Expo ou não têm equivalente no repositório de pacotes Expo, o React Native CLI é a única opção.

5. Ejeção do Expo:

- O Expo permite que você inicie seu projeto de maneira rápida e fácil, mas se você decidir que precisa das funcionalidades do React Native CLI, pode ser necessário "ejetar" seu projeto do Expo, o que é um processo irreversível e pode ser complexo.

6. Integração com o Backend e APIs Específicas:

- Se o seu aplicativo requer integração com serviços ou APIs específicas que não são compatíveis com as políticas de segurança do Expo, você pode encontrar limitações.

7. Necessidades de Publicação:

- O processo de publicação de aplicativos Expo é mais simples, mas também é mais limitado em termos de configurações avançadas. Se você precisa de um controle mais preciso sobre o processo de publicação, o React Native CLI oferece essa flexibilidade.

8. Ambientes de Desenvolvimento Específicos:

- Para cenários de desenvolvimento que envolvem necessidades específicas, como integração contínua ou automação avançada, o React Native CLI pode ser mais adequado devido à sua maior flexibilidade.

Em resumo, tanto o React Native CLI quanto o Expo têm suas próprias vantagens e desvantagens. A escolha entre eles deve ser baseada nas necessidades específicas do seu projeto. Se você precisa de controle total, personalização e acesso a funcionalidades nativas avançadas, o React Native CLI é a escolha apropriada. No entanto, se você deseja um ambiente de desenvolvimento rápido e simplificado, o Expo pode ser a opção ideal para começar.

3- JSDoc x Typescript

JSDoc:

JSDoc é uma convenção de comentários em JavaScript para adicionar anotações de tipo e documentação ao código JavaScript. É uma maneira de adicionar informações extras ao código, mas não afeta o sistema de tipos do JavaScript, uma vez que o JavaScript é uma linguagem dinamicamente tipada*.

Aqui está um exemplo de como usar JSDoc para adicionar anotações de tipo a uma função em JavaScript:

```
/**
 * Retorna a soma de dois números.
 * @param {number} a - O primeiro número.
 * @param {number} b - O segundo número.
 * @returns {number} A soma dos dois números.
 */
```

```
function soma(a, b) {
  return a + b;
}
```

Nesse exemplo, usamos JSDoc para indicar que os parâmetros `a` e `b` são números e que a função `soma` retorna um número.

TypeScript:

TypeScript é um superconjunto de JavaScript que adiciona um sistema de tipos estático à linguagem. Ele permite que você defina tipos para variáveis, parâmetros de função, retornos de função e outros elementos do código. Essas anotações de tipo são verificadas em tempo de compilação para garantir que o código esteja sendo usado corretamente.

Aqui está um exemplo de como usar TypeScript para definir tipos para uma função:

```
function soma(a: number, b: number): number {
  return a + b;
}
```

Nesse exemplo, usamos TypeScript para definir os tipos dos parâmetros `a` e `b` como números e o tipo de retorno da função como número.

A principal diferença entre JSDoc e TypeScript é que o JSDoc é apenas uma convenção de comentários e não realiza verificação de tipo em tempo de compilação, enquanto o TypeScript é um sistema de tipos estático que verifica o código em tempo de compilação e ajuda a evitar erros de tipo.

Portanto, o TypeScript oferece uma abordagem mais robusta e segura para lidar com tipos em comparação com o JSDoc.

Javascript:

Uma linguagem dinamicamente tipada, como JavaScript, tem suas vantagens, como flexibilidade e facilidade de prototipagem rápida. No entanto, também pode apresentar algumas desvantagens, especialmente em projetos maiores e mais complexos. Aqui estão algumas razões pelas quais a natureza dinamicamente tipada pode ser considerada desvantajosa:

1. Falta de detecção de erros em tempo de compilação: Em linguagens dinamicamente tipadas, os erros de tipo só são descobertos durante a execução do programa. Isso significa que você pode encontrar erros que poderiam ter sido detectados facilmente em tempo de compilação em uma linguagem estaticamente tipada, como o TypeScript. A detecção precoce de erros ajuda a evitar problemas e facilita a manutenção do código.
2. Refatoração complexa: Em projetos grandes, onde várias partes do código interagem entre si, a ausência de um sistema de tipos pode dificultar a refatoração do código. Sem tipos definidos, é difícil saber como uma mudança em um lugar pode afetar outras partes do código. O TypeScript, com seu sistema de tipos estático, facilita a refatoração, pois ele pode identificar e atualizar automaticamente referências a um determinado símbolo.
3. Menor clareza e documentação: Em um código JavaScript sem anotações de tipo adequadas ou documentação explícita, pode ser difícil entender o que uma função espera como argumento, qual é o formato do objeto retornado ou quais operações são permitidas em determinadas variáveis. A falta de informações claras sobre tipos torna a leitura e a compreensão do código mais desafiadoras, especialmente para desenvolvedores que não estão familiarizados com o projeto.
4. Menos ajuda do ambiente de desenvolvimento: As IDEs (Integrated Development Environments) modernas fornecem recursos poderosos, como sugestões de código, autocompletar, navegação rápida e recursos de refatoração. Essas funcionalidades são mais eficazes em linguagens estaticamente tipadas, pois podem analisar o código com base em informações de tipo para oferecer sugestões e detecção de erros mais precisas. Em linguagens dinamicamente tipadas, esses recursos podem ser limitados ou menos precisos.