soluções mobile

prof. Thyerri Mezzari

React Native:
reta final

Persistência de Dados

Além de toda a interatividade que um aplicativo deve fornecer ao usuário, em boa parte das situações alguns dados deverão "durar" além do uso atual do app. E assim como em um app Nativo (android ou iOS), em um app feito em ***React Native*** não poderia ser diferente.

Uma lista de preferências do usuário, uma configuração a ser salva, registros do usuário logado no momento e etc. Uma hora ou outra você como desenvolvedor terá que garantir que "coisas" permaneçam "vivas" mesmo depois da aplicação estar "morta".

Persistência de Dados

Em **React Native *(e Expo)*** temos algumas soluções disponíveis para trabalho:

- AsyncStorage
- expo-sqlite
- Firebase
- Realm *(RN puro apenas)*
- react-native-sqlite-storage *(RN puro apenas)*

Como viemos trabalhando nas últimas aulas, darei preferências em demonstrações de soluções que funcionam tanto em Expo quanto em RN padrão, sendo assim nesta aula iremos ver ***`AsyncStorage`*** e ***`expo-sqlite`***.

A persistência mais básica

AsyncStorage

AsyncStorage

Em uma analogia direta ao *Android Nativo*, o pacote **AsyncStorage** seria o equivalente ao que conhecemos por *SharedPreferences*. Esta lib permite salvar e recuperar pares de chave/valor de tipos de dados do tipo string. Estes dados irão persistir na sessão do usuário (mesmo que sua aplicação seja morta).

Este tipo de armazenamento seria o caminho ideal para salvar pequenos dados e configurações que não precisam ser sincronizadas na nuvem e pertence apenas à instalação atual do app. Por exemplo, os dados do usuário logado no momento em seu app, poderiam ser salvos de alguma forma com **AsyncStorage**, bem como as preferências dele em relação ao "tema escuro ou claro" do aplicativo.

Instalando o AsyncStorage

Sendo uma lib externa (que não vem de fábrica habilitada) o pacote `**@react-native-community/async-storage**` necessita ser instalado em nosso projeto sempre que pretendermos usar o mesmo para alguma necessidade.

Instalando no **Expo**:

```
npx expo install @react-native-async-storage/async-storage
```

Utilizando AsyncStorage

O uso do ***AsyncStorage*** é bem simples, basta importar a lib usando a seguinte linha no começo de seu arquivo:

```
import AsyncStorage from '@react-native-async-storage/async-storage';
```

Depois temos dois métodos principais para trabalho: **`setItem`** e **`getItem`**:

https://react-native-async-storage.github.io/async-storage/docs/usage

Utilizando AsyncStorage

Para **salvar um item** em nosso projeto, podemos seguir este caminho:

```
const salvarNome = async (value) => {
  try {
    await AsyncStorage.setItem('@nomePessoa', value)
  } catch (e) {
    // saving error
  }
}
```

Utilizando AsyncStorage

Caso seja necessário salvar um objeto complexo, composto de várias propriedades, teremos que dar um jeito de "transformá-lo" em string. O jeito mais fácil de fazer isso com JavaScript é converter o objeto para o formato JSON antes de salvá-lo.

```
const salvarPessoa = async (pessoa) => {
  try {
    const jsonValue = JSON.stringify(pessoa)
    await AsyncStorage.setItem('@pessoa', jsonValue)
  } catch (e) {
    // saving error
  }
}
```

Utilizando AsyncStorage

Após salvar um item com sucesso, em um segundo momento quando necessitarmos **recuperar** o valor (por exemplo, depois de reiniciar o aplicativo) podemos usar o método getItem chamando o item pelo mesmo nome de chave usado para salvar:

```
const getNome = async () => {
  try {
    const value = await AsyncStorage.getItem('@nomePessoa')
    if(value !== null) {
      // se o value for diferente de null, quer dizer que já havia sido salvo anteriormente.
      return value;
    }
  } catch(e) {
    // error reading value
  }
}
```

Utilizando AsyncStorage

Caso o valor recuperado seja complexo (um objeto por exemplo) devemos lembrar que o mesmo foi salvo no formato JSON, para restaurá-lo ao formato de Objeto novamente, logo após o processamento da função getItem devemos usar o método **JSON.parse**:

```
const getPessoa = async () => {
  try {
    const jsonValue = await AsyncStorage.getItem('@pessoa')
    if(jsonValue !== null) {
      // se o jsonValue for diferente de null, quer dizer que já havia sido salvo anteriormente.
      return JSON.parse(jsonValue);
    }
  } catch(e) {
    // error reading value
  }
}
```

Utilizando AsyncStorage

Se porventura em algum momento você necessitar limpar ou remover o valor de uma chave, a lib disponibiliza um método chamado **removeItem**:

```
const removePessoa = async () => {
  try {
    await AsyncStorage.removeItem('@pessoa')
  } catch(e) {
    // remove error
  }

  console.log('Removido com Sucesso')
}
```

Banco de Dados *"de Verdade"*

expo-sqlite

SQLite

Por ser altamente portável, autocontido e extremamente leve, o **SQLite** se tornou a solução nativa de banco de dados para muitas plataformas móveis como o iOS e também o Android. Sendo uma biblioteca, o **SQLite** lê e escreve diretamente no arquivo de banco de dados no *storage* do aparelho.

O uso do **SQLite** é recomendado onde a simplicidade da administração, implementação e manutenção são mais importantes que incontáveis recursos que bancos de dados tradicionais, mais voltados para aplicações complexas, possivelmente implementam *(ex: Postgres ou Oracle)*.

SQLite

No caso do ***React Native,*** as duas alternativas mais populares de adicionar recursos de *SQLite* em nossa aplicação são as bibliotecas **expo-sqlite** e **react-native-sqlite-storage**.

Como já mencionado iremos prosseguir nossos estudos focando no pacote **expo-sqlite**, que diferente de sua nomenclatura *não funciona apenas em Expo*.

Instalando o expo-sqlite

Por ser uma lib original do mesmo time responsável pelo core do Expo, o processo de instalação nesta plataforma será bem fácil:

```
npx expo install expo-sqlite
```

Usando o expo-sqlite

Para iniciarmos o uso do *expo-sqlite* em nosso projeto devemos começar importando a lib e seus métodos no começo de nosso arquivo:

```
import * as SQLite from 'expo-sqlite';
```

Após importado, devemos lembrar que estamos trabalhando com um banco de dados tradicional baseado em SQL Ansi, em alguns momentos teremos que escrever nossas **querys na unha**.

Uma das querys mais importantes para iniciarmos nosso banco são as querys relacionadas com **CREATE TABLE** usadas para criar todas as tabelas de banco de dados que poderão ser usadas em nosso aplicativo.

Usando o expo-sqlite

Primeiro podemos criar uma variável do tipo string template (usando aspas invertidas ``) contendo o SQL básico de criação para nossa primeira tabela:

```
const SQL_CREATE_ENTRIES = `
  CREATE TABLE IF NOT EXISTS usuarios (
    id INTEGER PRIMARY KEY autoincrement,
    name varchar(255) NOT NULL,
    email varchar(255) NOT NULL
  )
`;
```

Usando o expo-sqlite

Na sequência devemos abrir nosso banco dando um nome para o arquivo .sqlite que será salvo no disco do aparelho do usuário:

```
const db = SQLite.openDatabaseSync("exemploApp.sqlite");
```

Eu gosto de pensar sempre que no momento que eu "aceito" que meu app vai precisar de um banco de dados, diversas telas podem precisar usar o mesmo, por tanto eu costumo criar um arquivo chamado "db.js" na raiz do meu projeto a fim de facilitar o acesso a essa database que devemos conectar sempre que precisar fazer um SELECT ou INSERT.

Usando o expo-sqlite

Neste arquivo ***db.js*** eu costumo fazer algo assim:

```
import * as SQLite from 'expo-sqlite';

const DATABASE_NAME = "exemploApp.sqlite";
const SQL_CREATE_ENTRIES = `...`;

let _db = null;

export default function openDB() {
  if(!_db) {
    _db = SQLite.openDatabaseSync(DATABASE_NAME);

    // primeira vez que iremos abrir a conexão,
    // tentaremos criar nossas tabelas
    _db.withTransactionSync(() => {
      _db.execSync(SQL_CREATE_ENTRIES);
    });
  }

  return _db;
}
```

Usando o expo-sqlite

Assim no futuro sempre que eu precisar usar o banco de dados de qualquer tela eu posso importar o arquivo "db.js":

```
import openDB from "./db";

const db = openDB();

function MinhaTela() {

    useEffect(() => {
        const rows = db.getAllSync("select * from usuarios", []);
        console.log(JSON.stringify(rows));
    }, []);

  return ...;
}
```

Usando o expo-sqlite

Voltando ao funcionamento do **expo-sqlite**, após abrirmos nossa conexão com o banco *(usando openDatabaseSync)* teremos a nossa disposição diversos métodos para inserir dados ou realizar consultas, sempre usando SQL.

- db.execAsync ou db.execSync
- db.runAsync ou db.runSync
- db. getFirstAsync..
- db. getAllAsync..

https://docs.expo.dev/versions/latest/sdk/sqlite/

Usando o expo-sqlite

O método `db.**withTransactionSync**` inicia uma "transação" em nosso banco imbuída de uma proteção relacionada a "auto-rollback".

Ao iniciar uma nova transação você poderá executar uma série de comandos SQL e caso algum comando de erro, a transação será revertida e o estado do banco voltará para antes de você ter começado a operar na transação.

```
Promise.all([
  // 1. A new transaction begins
  db.withTransactionAsync(async () => {
    // 2. The value "first" is inserted into the test table and we wait 2
    //    seconds
    await db.execAsync('INSERT INTO test (data) VALUES ("first")');
    await sleep(2000);

    // 4. Two seconds in, we read the latest data from the table
    const row = await db.getFirstAsync<{ data: string }>('SELECT data FROM test');

    // ❌ The data in the table will be "second" and this expectation will fail.
    //    Additionally, this expectation will throw an error and roll back the
    //    transaction, including the `UPDATE` query below since it ran within
    //    the transaction.
    expect(row.data).toBe('first');
  }),
  // 3. One second in, the data in the test table is updated to be "second".
  //    This `UPDATE` query runs in the transaction even though its code is
  //    outside of it because the transaction happens to be active at the time
  //    this query runs.
  sleep(1000).then(async () => db.execAsync('UPDATE test SET data = "second"')),
]);
```

Usando o expo-sqlite

O db disponibiliza os métodos ***`execAsync, runAsync, getAllAsync`*** e seus variantes SYNC, e estes são basicamente os métodos que precisamos. Esses métodos recebem os seguintes argumentos nesta ordem de preenchimento:

1. **sqlStatement**: seu código de query SQL-ansi
2. **arguments**: argumentos para preencher uma query "preparada"

Vamos ver os retornos desses métodos...

Usando o expo-sqlite

O método execAsync não possui nenhum retorno. Usamos quando queremos executar algum SQL sem necessidade de avaliar o retorno da operação.

```
// `execAsync()` is useful for bulk queries when you want to execute altogether.
// Please note that `execAsync()` does not escape parameters and may lead to SQL injection.
await db.execAsync(`
PRAGMA journal_mode = WAL;
CREATE TABLE IF NOT EXISTS test (id INTEGER PRIMARY KEY NOT NULL, value TEXT NOT NULL, intValue INTEGER);
INSERT INTO test (value, intValue) VALUES ('test1', 123);
INSERT INTO test (value, intValue) VALUES ('test2', 456);
INSERT INTO test (value, intValue) VALUES ('test3', 789);
`);
```

Usando o expo-sqlite

Com o método runAsync podemos pegar, por exemplo, o id do item inserido.

```
// `runAsync()` is useful when you want to execute some write operations.
const result = await db.runAsync('INSERT INTO test (value, intValue) VALUES (?, ?)', 'aaa', 100);
console.log(result.lastInsertRowId, result.changes);
await db.runAsync('UPDATE test SET intValue = ? WHERE value = ?', 999, 'aaa'); // Binding unnamed parameters from variadic arguments
await db.runAsync('UPDATE test SET intValue = ? WHERE value = ?', [999, 'aaa']); // Binding unnamed parameters from array
await db.runAsync('DELETE FROM test WHERE value = $value', { $value: 'aaa' }); // Binding named parameters from object
```

Usando o expo-sqlite

Com o método getAllAsync podemos pegar o retorno de uma consulta

```
// `getAllAsync()` is useful when you want to get all results as an array of objects.
const allRows = await db.getAllAsync('SELECT * FROM test');
for (const row of allRows) {
  console.log(row.id, row.value, row.intValue);
}
```

Usando o expo-sqlite

Óbvio que dentro do React Native, devemos seguir algumas normas do ecossistema React, como o uso de ***useState***, ***useEffect*** e outros hooks que irão ditar o estado dos nossos dados e também em que momento nossos códigos serão executados.

Pedido permissão

Permissions

Por que pedir permissões?

Um dos pontos de projeto centrais da arquitetura de segurança do **SO Android** é que, por padrão, nenhum app tem permissão de realizar nenhuma operação que prejudique outros apps, o sistema operacional ou o usuário.

Isso abrange a leitura e a gravação de dados privados do usuário (como contatos ou e-mails), leitura ou gravação dos arquivos, realização de acesso de rede (internet), manter a tela do dispositivo ativo etc.

Apps para Android e IOS precisam solicitar permissão para acessar dados confidenciais do usuário (como contatos e SMS), bem como recursos específicos do sistema (como câmera, galeria de fotos e Internet). Dependendo do recurso, o sistema pode conceder a permissão automaticamente ou pedir ao usuário que aprove a solicitação.

Permissions

Se estivermos usando **Expo** o módulo a ser usado é o `*expo-permissions*` que deve ser instalado assim:

```
npx expo install expo-permissions
```

Depois devemos listar as permissões necessitadas no arquivo **app.json** conforme orientações neste link:

https://docs.expo.dev/versions/latest/config/app/#permissions

Permissions

Porém, no **EXPO** atualmente *(e principalmente)* cada módulo chave possui seu método exclusivo para solicitar as permissões adequadas para seu funcionamento e na atual situação acabamos por usar o módulo `expo-permissions` apenas quando não estamos usando módulos fornecidos pela documentação do Expo.

Por exemplo o módulo de Câmera do Expo possui uma função exclusiva para solicitar as permissões necessárias para podermos usar a câmera de um aparelho:

```
const [permission, requestPermission] =Camera.useCameraPermissions();
```

Permissions

Assim como o módulo **ImagePicker**:

```
ImagePicker.requestCameraPermissionsAsync()
ImagePicker.requestMediaLibraryPermissionsAsync()
```

E outro exemplo o módulo de *Localização* (**expo-location**):

```
let { status } = await Location.requestForegroundPermissionsAsync();
```

Permissions

No caso de um aplicativo **React Native padrão/puro** *(sem expo)*, obrigatoriamente um app precisa divulgar as permissões necessárias incluindo tags `<uses-permission>` no arquivo manifesto do app.

Por exemplo um aplicativo que precise enviar mensagens SMS de maneira programática teria que adicionar a seguinte linha em seu ***AndroidManifest.xml***:

```
<uses-permission android:name="android.permission.SEND_SMS" />
```

Recomendo consultar a documentação oficial do módulo a ser usado junto da configuração acima:

https://reactnative.dev/docs/permissionsandroid

Exercício persistência e permissões

- Desenvolva um App com React Native, expo, expo-sqlite e react-native-async-storage
- Use React Native pages para os componentes visuais
- Crie um switch button para determinar a configuração do modo escuro (habilitado/desabilitado). Salve/recupere essa informação usando react-native-async-storage
- Crie um banco de dados com expo-sqlite para armazenar uma lista de localizações (coordenadas latitude, longitude)
- Insira um botão “registrar localização”. Sempre que o botão for pressionado a localização do usuário deverá ser registrada e salva no DB.
- Exiba na tela a lista com todas as localizações salvas.

obrigado 🚀