soluções mobile

prof. Thyerri Mezzari

Android: Persistência de dados, Preferences e SQLite + Room Persistance Library

Persistência de Dados

Conceitualizando

Persistência de Dados

Além de toda a interatividade que um aplicativo Android Nativo deve fornecer ao usuário, em boa parte das situações alguns dados deverão "durar" além do uso atual do app. 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

O Android fornece várias opções para salvar os dados de aplicativos persistentes. A solução que for escolhida irá depender de necessidades específicas, como se os dados devem ser privados de sua aplicação ou acessíveis a outras aplicações (e de usuário) e quanto espaço os seus dados requerem.

Mas basicamente, as principais maneiras de persistir informações no Android com API nativa são: utilizando um par de chave valor, chamado de *SharedPreference*, ou usando um banco de dados relacional convencional, com o *SQLite*.

Persistência de Dados

Detalhamento de soluções clássicas

- **SharedPreferences**: Armazenar dados particulares primitivos em pares chave-valor.
- **Internal Storage**: Armazenar dados privados na memória do dispositivo.
- **External Storage**: Armazenar dados públicos sobre o armazenamento externo compartilhado.
- **SQLite Databases**: Armazenar dados estruturados em um banco de dados privado.

Persistência de Dados

Outras soluções

- **Firebase**: banco de dados noSQL na nuvem compatível com Android e iOS
- **REALM**: banco de dados noSQL semelhante ao firebase/mongodb também com possibilidade de sincronização na nuvem
- ***Room Persistance Library***: evolução do uso do SQLite tradicional em Android

Por fim, nesta aula iremos nos ater ao básico do assunto focando em ***SharedPreferences*** e ***SQLite/ROOM***.

Preferences

`getPreferences x getSharedPreferences`

SharedPreferences

A classe *SharedPreferences* permite salvar e recuperar pares de chave/valor de tipos de dados primitivos. Podemos usar o *SharedPreferences* para salvar os dados primitivos: *booleans*, *floats*, *ints*, *longs*, *e strings*. 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 a instalação atual do app. Por exemplo, os dados do usuário logado no momento em seu app, poderia ser salvos de alguma forma como Preference, bem como as preferências dele em relação ao "tema escuro ou claro" do aplicativo.

SharedPreferences

Na teoria cada *Activity* possui seu próprio "arquivo de configurações" a disposição, mas também é possível criar um arquivo que possa ser "compartilhado" entre todas as activities do seu app, por isso que chamamos de *SharedPreferences* (shared = compartilhado).

Para obter um objeto *SharedPreferences* para sua aplicação, utilize um dos dois métodos:

getSharedPreferences(String nome, int modo) – Utilize se você precisa de vários arquivos de preferências identificados pelo nome que será passado no primeiro parâmetro.

getPreferences(int modo) – Utilize se você só precisa de um arquivo de preferências para a sua atividade.

SharedPreferences

Para escrever valores

- Chamar o método *edit()* para obter uma *SharedPreferences.Editor*;
- Adicionar valores com métodos tais como *putBoolean()* e *putString()*;
- Persistir os novos valores com *commit()*.

E para **ler os valores** do *SharedPreferences* utilize os métodos como ***getBoolean()*** e ***getString()***.

Escrevendo - *SharedPreferences*

Para escrever um valor em um arquivo de *SharedPreferences*, precisamos definir um nome comum para este "banco de dados". E também será necessário acessar o mesmo através de uma activity válida, por tanto se você estiver acessando o método de dentro de um Fragment, terá que utilizar getActivity().

Para o nome do banco de dados recomenda-se atrelar o nome do mesmo ao package de sua aplicação, por exemplo:

```
com.satc.exemploapp.preferences.PERFIL
```

Escrevendo - *SharedPreferences*

Na sequência, na primeira vez que o método ***getSharedPreferences*** for chamado o banco será criado e caso ele já exista será recuperado para acesso.

Não se esqueça que para iniciar o "modo de edição" com o objetivo de salvar dados nesse arquivo de preferences será necessário instanciar um ***SharedPreferences.Editor***.

Escrevendo - *SharedPreferences*

O código abaixo escreve um número (***int*** = *número inteiro)* no banco de *SharedPreferences* cujo o nome é *com.satc.exemploapp.preferences.PERFIL*.

```kotlin
// kotlin
val sharedPref = activity?.getSharedPreferences("com.satc.exemploapp.preferences.PERFIL", Context.MODE_PRIVATE) ?: return
with (sharedPref.edit()) {
    putInt("IDADE_USUARIO", 33)
    commit()
}
```

Escrevendo - *SharedPreferences*

O código abaixo escreve um número (***int*** = *número inteiro)* no banco de *SharedPreferences* cujo o nome é *`com.satc.exemploapp.preferences.PERFIL`*.

```
// java
SharedPreferences sharedPref = getActivity()
                                .getSharedPreferences("com.satc.exemploapp.preferences.PERFIL", Context.MODE_PRIVATE);
SharedPreferences.Editor editor = sharedPref.edit();
editor.putInt("IDADE_USUARIO", 33);
editor.commit();
```

Escrevendo - *SharedPreferences*

Vale observar que para cada tipo de dado a ser salvo na memória, precisamos usar um método específico:

- **putInt**: salva um número inteiro
- **putFloat:** salva um número não-inteiro (com vírgula)
- **putString**: salva um text/string
- **putBoolean**: salva um valor true ou false

Escrevendo - *SharedPreferences*

Caso você precise salvar dados complexos em seu arquivo de *SharedPreference* como uma lista de itens, recomendo que você converta seu valor complexo para uma **string JSON** e salve com o método *putString*.

Para lhe auxiliar nesta tarefa recomendo os seguintes links:

https://github.com/google/gson

https://medium.com/@evancheese1/shared-preferences-saving-arraylists-and-more-with-json-and-gson-java-5d899c8b0235

Lendo - *SharedPreferences*

Seguindo nosso exemplo de escrita anterior, caso precisássemos recuperar o valor da "idade do usuário" que salvamos como um int, poderíamos fazer assim:

```kotlin
// kotlin
val sharedPref = activity?.getSharedPreferences("com.satc.exemploapp.preferences.PERFIL", Context.MODE_PRIVATE) ?: return
val idadeUsuario = sharedPref.getInt("IDADE_USUARIO", 16)
```

*PS.: O segundo argumento das funções **getXXX** é o valor padrão caso ainda não tenha sido setado valor anteriormente.*

Lendo - *SharedPreferences*

Seguindo nosso exemplo de escrita anterior, caso precisássemos recuperar o valor da "idade do usuário" que salvamos como um int, poderíamos fazer assim:

```java
// java
SharedPreferences sharedPref = getActivity()
                                .getSharedPreferences("com.satc.exemploapp.preferences.PERFIL",  Context.MODE_PRIVATE);
int idadeUsuario = sharedPref.getInt("IDADE_USUARIO", 16);
```

*PS.: O segundo argumento das funções **getXXX** é o valor padrão caso ainda não tenha sido setado valor anteriormente.*

SQLite

SQLiteOpenHelper


SQLite

SQLite - O banco de dados mobile

O **SQLite** é uma lib de funcionalidades escrita em C que também possui um banco de dados relacional embutido em sua estrutura.

SQLite não é um *client* usado para conectar com um grande servidor externo de banco de dados, mas sim o próprio servidor em si. A biblioteca SQLite lê e escreve diretamente no arquivo de banco de dados no disco.

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 - O banco de dados mobile

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***.

Inicialmente o uso **SQLite** junto ao *Android* se dava de maneira bem clássica, trabalhando manualmente com uma série de pontos, inclusive a criação de tabelas no banco de dados. De certa forma é possível trabalhar desta maneira até os dias de hoje, principalmente se você estiver trabalhando com um app de pequeno a médio porte.

Porém em contraponto ao parágrafo anterior, atualmente com a evolução da plataforma uma nova maneira de lidar com o SQLite fora surgindo junto das últimas versões da Android SDK, **o banco de dados ROOM.** *Nos próximos slides iniciaremos pela implementação tradicional / clássica.*

SQLite - Criando o banco

No Android o banco de dados que você cria em uma aplicação só é acessível para a mesma, a não ser que você utilize um provedor de conteúdo. Uma vez criado o banco de dados, ele é armazenado no diretório *"/data/data/{nome do pacote}/databases/{nome do banco}"*, além de gerenciar o banco por código você pode fazê-lo pelo adb utilizando a ferramenta ***sqlite3***.

O caminho mais fácil para iniciar seus trabalhos com **SQLite** em android é utilizando a classe ***SQLiteOpenHelper***.

Com ela podemos iniciar um novo banco, controlar suas versões (mudanças de estrutura) e fornecer acesso rápido a *querys*, *inserts*, *updates e deletes*.

SQLite - Criando o banco

Criando sua própria classe auxiliar, poderemos customizar um banco **SQLite** de acordo com as necessidades de sua aplicação. Sua classe deverá estender a classe herdada da Android SDK chamada ***SQLiteOpenHelper***.

Não se esqueça de customizar os métodos *onCreate*, *onUpgrade* e *onDowngrade*.

O método **onCreate** será executado apenas na primeira vez que seu banco for criado, logo suas tabelas deverão ser criadas dentro desse método. Os métodos **onUpgrade** e **onDowngrade** se referem a mudança de versão de seu banco *(explicarei melhor adiante)*.

SQLite - Criando o banco

Por se tratar de 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.

Depois dessas querys talvez seja válido criar um conceito de *DROP TABLE* para caso o banco precise ser recriado ou destruído, eliminar as tabelas atuais antes de prosseguir.

SQLite - Criando o banco

Começaremos nossa classe *SQLiteOpenHelper* criando a string que contém toda a estrutura de nossas tabelas. Neste exemplo apenas uma tabela chamada usuarios:

```kotlin
// kotlin
private const val SQL_CREATE_ENTRIES =
  "CREATE TABLE IF NOT EXISTS usuarios (" +
    "id INTEGER PRIMARY KEY autoincrement," +
    "name varchar(255) NOT NULL," +
    "email varchar(255) NOT NULL)"
```

SQLite - Customizando a Classe SQLiteOpenHelper

```kotlin
// kotlin
class AppDbHelper(context: Context) : SQLiteOpenHelper(context, DATABASE_NAME, null, DATABASE_VERSION)
{
    override fun onCreate(db: SQLiteDatabase) {
        // esse código irá rodar na criação do banco
        db.execSQL(SQL_CREATE_ENTRIES)
    }
    override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {
        // implementaremos isso caso necessário
    }
    override fun onDowngrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {
        // implementaremos isso caso necessário
    }
    companion object {
        // nome e versão do banco
        const val DATABASE_NAME = "SatcDemo.db"
        const val DATABASE_VERSION = 1
    }
}
```

SQLite - Criando o banco

Depois a ideia é *instanciarmos* nossa classe sempre que uma **Activity** ou **Fragment** precisar usar/acessar algo no banco de dados.

De uma Activity:

```kotlin
// kotlin
val dbHelper = AppDbHelper(this)
```

```java
// java
AppDbHelper dbHelper = new AppDbHelper(this);
```

SQLite - Criando o banco

Depois a ideia é *instanciarmos* nossa classe sempre que uma **Activity** ou **Fragment** precisar usar/acessar algo no banco de dados.

De um Fragment precisaremos acessar a Activity (ou seja o contexto) no qual o Fragment está inserido:

```kotlin
// kotlin
val dbHelper = AppDbHelper(getActivity().getApplicationContext())
```

```java
// java
AppDbHelper dbHelper = new AppDbHelper(getActivity().getApplicationContext());
```

SQLite - Manipulando o banco

Após criarmos nossa classe ***SQLiteOpenHelper***, definirmos nossas tabelas e o nome do banco, já podemos começar a trabalhar com a mesma em uma *Activity* ou *Fragment*.

Vale observar que precisamos definir que tipo de operação iremos executar antes de ir a diante.

SQLite - Manipulando o banco

Por exemplo se você vai inserir, atualizar ou deletar um registro do banco, precisará de uma instância do tipo *`writableDatabase`*:

```
// kotlin
val db = dbHelper.writableDatabase
```

Se você precisar apenas ler/recuperar dados salvos no banco, precisará de uma instância do tipo *`readableDatabase`*:

```
// kotlin
val db = dbHelper.readableDatabase
```

SQLite - Inserindo (INSERT) registros no banco

Para inserir registros no banco precisaremos de um "mapa" de *chave-valor* relacionando o nome da coluna e qual o valor dela para cada linha inserida. Para isto podemos usar a classe ***ContentValues*** disponível tanto em Java quanto em Kotlin.

```kotlin
// kotlin
val dadosUsuario = ContentValues().apply {
    put("name", "Thyerri Mezzari")
    put("email","thyerri.mezzari@satc.edu.br"
    )
}
```

SQLite - Inserindo (INSERT) registros no banco

Na sequência poderemos usar o método ***insert*** de nossa variável db:

```
// Obtemos a instância do banco em modo "escrita/edição"
val db = dbHelper.writableDatabase

// inserindo a nova linha no banco, na tabela "usuarios" (primeiro argumento)
// o retorno da execução de sucesso da função "insert" é o novo ID (chave-primária)
// que este novo usuário salvo no banco acaba de receber
val newRowId = db?.insert("usuarios", null, dadosUsuario)
```

SQLite - Obtendo (SELECT) informações no banco

Para obtermos registros salvos em um banco SQLite no Android podemos usar dois métodos mais comuns, ***query*** e ***rawQuery*** a fim de executarmos "*SELECTs*" no banco.

O primeiro método (***query***) é uma espécie de auxiliar no processo de montagem de query.

```kotlin
// kotlin
val db = dbHelper.readableDatabase

val campos = arrayOf("id", "name", "email")
val criterio = "name LIKE ?"
val criterioValores = arrayOf("%Lucas%")
val ordem = "name ASC, email ASC"

// montando a query e criando um Cursor para acesso dos registros
val cursor = db.query(
    "usuarios",          // nome da tabela a ser consultada
    campos,              // Campos a serem selecionados, passar null para pegar todos
    criterio,            // Parte "WHERE" desssa query de SELECT
    criterioValores,     // Argumentos dinâmicos para a montagem do WHERE
    null,                // GROUP BY - se precisar
    null,                // HAVING - se precisar
    ordem                // Ordenação dos resultados (ORDER BY)
)
```

SQLite - Obtendo (SELECT) informações no banco

A outra forma que teríamos de fazer isso é usando o método ***rawQuery*** a fim de montarmos o *SELECT* 100% nós mesmos:

```kotlin
// kotlin
val db = dbHelper.readableDatabase

// primeiro parâmetro é a query, o segundo são os argumentos dinâmicos para a montagem do WHERE
val cursor = db.rawQuery(
  "SELECT id, name, email FROM usuarios WHERE name LIKE ?",
  arrayOf("%Thyerri%")
)
```

SQLite - Obtendo (SELECT) informações no banco

Ambos os métodos (***query*** e ***rawQuery***) retornam um "***cursor***" também conhecido como ponteiro referente a localização dos registros no banco.

Para extrair os dados deste cursor, precisamos usar as funções `moveToXXX`, como por exemplo a função ***moveToFirst*** para obter apenas um registro encontrado:

```
// kotlin
if (cursor.moveToFirst()) {
  val email = cursor.getString(cursor.getColumnIndex("email"));
}
cursor.close()
```

SQLite - Atualizando (UPDATE) informações no banco

As funcionalidades de atualização, são bem semelhantes a de INSERT, sendo que a função recomendada é a ***update***. O que vai além da função de insert é que na função de update temos que passar um critério *(WHERE)* de quais registros serão atualizados.

```kotlin
// kotlin
// Obtemos a instância do banco em modo "escrita/edição" val db
= dbHelper.writableDatabase

val dadosAtualizados = ContentValues().apply {
    put("name", "Thyerri Mezzari")
}

val criterio = "email = ?"
val criterioValores = arrayOf("thyerri.mezzari@satc.edu.br")

// o retorno de execução de sucesso da função update
// te diz quantos registros foram atualizados na tabela val
count = db?.update(
  "usuarios",          // tabela a ser atualizada
  dadosAtualizados,    // novos valores
  criterio,            // WHERE para restringir quais linhas serão atualizadas
  criterioValores      // argumentos dinâmicos para a montagem do WHERE
)
```

SQLite - Obtendo (SELECT) informações no banco

Ou no caso de uma query em que você precise listar mais de uma linha, usar a função ***moveToNext*** combinada de um `while` (laço de repetição):

```
// kotlin
with(cursor) {
    // cada vez que esse while "girar" teremos o e-mail
    // de um usuário de cada vez de todos encontrados na query
    while (moveToNext()) {
        val email = cursor.getString(cursor.getColumnIndex("email"));
    }
}
```

SQLite - Deletando (DELETE) informações no banco

A funcionalidade de delete também é bem semelhante a de INSERT, sendo que a função recomendada é a ***delete***. O que vai além da função de insert é que na função de delete temos que passar um critério *(WHERE)* de quais registros serão atualizados.

```kotlin
// kotlin
// Obtemos a instância do banco em modo "escrita/edição"
val db = dbHelper.writableDatabase

val criterio = "email = ?"
val criterioValores = arrayOf("thyerri.mezzari@satc.edu.br")

// o retorno de execução de sucesso da função update
// te diz quantos registros foram atualizados na tabela val
count = db?.delete(
  "usuarios",          // tabela a ser atualizada
  criterio,           // WHERE para restringir quais linhas serão atualizadas
  criterioValores     // argumentos dinâmicos para a montagem do WHERE
)
```

SQLite - Conexão persistente do banco de dados

Por fim, visto que é caro chamar ***getWritableDatabase()*** e g***etReadableDatabase()*** quando o banco de dados está fechado, deixe a conexão do banco de dados aberta durante todo o período de tempo em que possivelmente será necessário acessá-lo. Normalmente, é ideal fechar o banco de dados no ***onDestroy()*** da *Activity* de chamada para liberar memória.

```
// kotlin
override fun onDestroy() {
    dbHelper.close()
    super.onDestroy()
}
```

Room Persistence Library

RoomDatabase

Room Persistence Library

A biblioteca de persistência **Room** oferece uma camada de abstração sobre o SQLite para permitir um acesso mais robusto ao banco de dados, aproveitando toda a capacidade do SQLite.

É altamente recomendado pelo Google usar o **Room** em vez do SQLite, mas sendo que o **ROOM** fora introduzido junto com as novas bibliotecas do *Android Jetpack* será necessário configurar seu projeto para que o mesmo rode.

Room Persistence Library

Para usar a **Room** no app, adicione as seguintes dependências ao arquivo *build.gradle* do app:

```
dependencies {
  def room_version = "2.3.0"

  implementation "androidx.room:room-runtime:$room_version"
  annotationProcessor "androidx.room:room-compiler:$room_version"

  // optional - Kotlin Extensions and Coroutines support for Room
  implementation "androidx.room:room-ktx:$room_version"
}
```

***Observação**: para apps baseados em Kotlin, use kapt em vez de annotationProcessor.*

Room Persistence Library

Para usar a **Room** no app, adicione as seguintes dependências ao arquivo *build.gradle* do app:

```
// para Kotlin
dependencies {
  def room_version = "2.3.0"

  implementation "androidx.room:room-runtime:$room_version"
  annotationProcessor "androidx.room:room-compiler:$room_version"

  // To use Kotlin annotation processing tool (kapt)
  kapt("androidx.room:room-compiler:$room_version")

  // optional - Kotlin Extensions and Coroutines support for Room
  implementation("androidx.room:room-ktx:$room_version")
}
```

Entidade - Room Persistence Library

Seguindo nosso último exemplo de SQLite puro/nativo, reproduzindo a ideia de uma "tabela de usuários" com os campos name e email, iniciaremos nossa implementação usando **ROOM** criando o arquivo ***Entidade*** que irá representar cada linha de usuário em nosso banco. Seria algo como um Modelo do modelo MVC:

```kotlin
// kotlin
@Entity(tableName = "usuarios")
data class Usuario(
    @PrimaryKey val id: Int,
    @ColumnInfo(name = "name") val name: String?,
    @ColumnInfo(name = "email") val email: String?
)
```

DAO - Room Persistence Library

Na sequência iremos criar a interface ***DAO*** responsável por definir quais métodos iremos utilizar para interação com a tabela usuarios. Coisas como todos os INSERTS previstos, possíveis tipos de SELECT com mais ou menos critérios, e até a funções para deletar usuários:

```
@Dao
interface UsuarioDao {
  @Query("SELECT * FROM usuarios WHERE id = :usuarioId")
  fun getUsuario(usuarioId: String): Usuario

  @Query("SELECT * FROM usuarios")
  fun getAll(): List<Usuario>

  @Query("SELECT * FROM usuarios WHERE name LIKE :name")
  fun loadAllByName(name: String): List<Usuario>

  @Insert
  fun insertAll(vararg usuarios: Usuario)

  @Delete
  fun delete(usuario: Usuario)
}
```

RoomDatabase - Room Persistence Library

Por fim criaremos nossa classe que irá representar nosso banco de dados como um todo e também fornecer acesso rápido a nosso aplicativo a todas as instâncias do tipo DAO de cada tabela:

```kotlin
// kotlin
@Database(entities = arrayOf(Usuario::class), version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun usuarioDao(): UsuarioDao
}
```

Room - Acessando o banco

Sempre que precisarmos usar banco de dados em alguma *Activity* ou *Fragment* precisaremos usar a função estática ***databaseBuilder*** da classe ***Room***:

```
// kotlin
val db = Room.databaseBuilder(
  this,                                  // contexto, em fragment usar getActivity()
  AppDatabase::class.java,               // nossa classe que representa o banco
  "satc-demo-room"                       // nome de nosso banco SQLite
).build()
```

Room - Inserindo (INSERT) registros no banco

Após instanciarmos nosso banco em uma *Activity* ou *Fragment*, poderemos acessar os **DAOs** disponíveis e utilizar os comandos escritos na interface.

Um destes comandos definidos por nós (desenvolvedores) foi o ***insertAll*** que serve para inserir novos usuários na tabela:

```
// kotlin
val novoUsuario:Usuario = Usuario(1, "Thyerri Mezzari",
"thyerri.Mezzari@satc.edu.br") db.usuarioDao().insertAll(novoUsuario)
```

Room - Obtendo (SELECT) informações no banco

Para obtermos registros já salvos no banco, basta continuarmos usando nosso ***DAO*** e um dos métodos baseados em *SELECT*:

```kotlin
// kotlin
val usuarioLucas:Usuario = db.usuarioDao().getUsuario("1")
```

Na execução do método acima iríamos encontrar um usuário cujo o id (chave-primária) é "1".

Room - Outras necessidades

Podemos utilizar todos os métodos que nós mesmos tenhamos definido na interface DAO de cada tabela, UPDATE, DELETE e etc.

No mais recomendo dar uma olhada na documentação completa do **ROOM**:
https://developer.android.com/training/data-storage/room

E neste **tutorial** bem completo aqui:
https://developer.android.com/codelabs/android-room-with-a-view-kotlin?hl=pt-br#0

obrigado 🚀