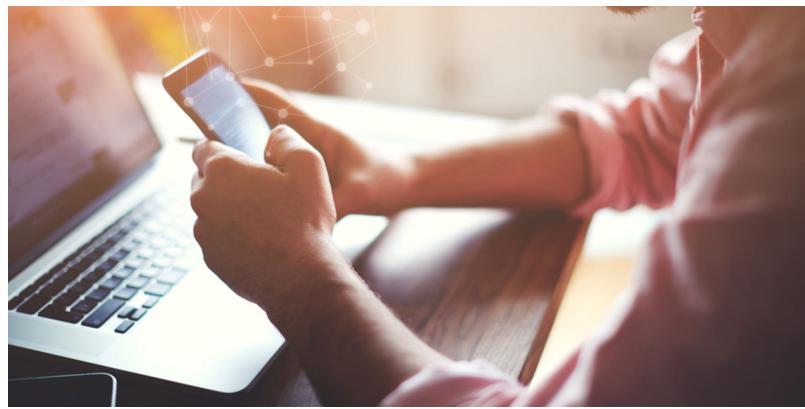


Programação para dispositivos móveis

Aula 8: Persistência de Dados no Android

INTRODUÇÃO



Quando desenvolvemos aplicativos, é muito comum termos alguma forma de armazenamento e recuperação de dados. Esta aula visa demonstrar a persistência de dados em aplicações Android através de Streams (Java.io) ou através de um banco de dados relacional (SQLite).

OBJETIVOS



Explicar o conceito de persistência de dados em aplicações Android através de Streams (Java.io).

Demonstrar o conceito de persistência de dados através de um banco de dados relacional (SQLite).

PERSISTÊNCIA DE DADOS

A persistência de dados é uma necessidade na maioria das aplicações.

O Android nos permite persistir dados de formas e em locais diferentes, como em arquivos usando o conceito de Java.io ou em banco de dados, usando o SQLite.

A escolha da forma e do local dependerá de suas necessidades, como, por exemplo, se os dados serão privados ou não, o espaço necessário para armazenamento etc.

Esses assuntos serão discutidos a seguir.



PERSISTÊNCIA DE DADOS EM ANDROID

Devemos escolher o tipo de persistência em função de algumas necessidades de armazenamento, como, por exemplo:

- Segurança de dados (Os dados serão privados?)

- Acesso aos dados (Os dados podem ser acessados por outras aplicações/usuários?)

- Espaço em disco (Qual a necessidade de espaço de armazenamento?)

Em Android, podemos persistir dados de nossa aplicação de várias formas diferentes.

Dentre as opções de armazenamento de dados, podemos escolher:

SHARED PREFERENCES

É muito comum que nossas aplicações precisem armazenar pequenas quantidades de dados. A classe Shared Preferences se mostra como uma alternativa ao uso de banco de dados.

Ela permite salvar e recuperar pares de chave/valor de tipos primitivos de dados (booleans, floats, ints, longs, e strings) em um arquivo denominado Arquivo de Preferências, pois associa um “nome” a uma determinada informação para que depois se possa recuperá-la através desse nome.

Normalmente, usamos essa opção para poucas informações. Como exemplo desse tipo de armazenamento temos:

- Data do último acesso ao servidor;

- Pontuação de um jogo;
-
- Configurações, como o nível em que um jogo será iniciado (Easy/Hard) e senha default do jogo.
-

Para criarmos um arquivo de preferências, podemos escolher formas:

getSharedPreferences()

- Se precisar acessar múltiplos arquivos de preferências em sua Activity;
- Tem como parâmetros o nome da Shared Preference e o modo de abertura.

Exemplo:

```
SharedPreferences sharedpreferences =
    getSharedPreferences(MinhasPreferencias,
    Context.MODE_PRIVATE);
```

getPreferences()

- Se precisar de um único arquivo de preferências para a Activity;
- Tem como parâmetro o modo de abertura;
- Não é definido o nome da Shared Preference por se tratar de um único arquivo.

Exemplo:

```
SharedPreferences sharedpreferences =
    getPreferences(Context.MODE_PRIVATE);
```

Além de privado, a classe Context nos oferece outros modos que estão listados abaixo:

Modo	Descrição
MODE_APPEND	Se o arquivo já existe, então deve-se escrever dados no final do arquivo existente em vez de apagá-lo.
MODE_ENABLE_WRITE_AHEAD_LOGGING	Aplicado a banco de dados.
MODE_MULTI_PROCESS	Verifica se há modificação de preferências, mesmo que a instância Shared preference já tenha sido carregada (deprecated).
MODE_PRIVATE	É o modo padrão, onde o arquivo criado só pode ser acessado pelo aplicativo de chamada (ou todos os aplicativos que compartilham o mesmo ID de usuário).
MODE_WORLD_READABLE	Permite que outro aplicativo leia as preferências (deprecated).
MODE_WORLD_WRITEABLE	Permite que outro aplicativo escreva as preferências (deprecated).

Para salvar, usamos a classe SharedPreferences.Editor, que nada mais é do que uma classe auxiliar que escreve no arquivo.

Veja o exemplo abaixo:

```
Editor editor = sharedpreferences.edit();
editor.putString("chave", "valor");
editor.commit();
```

Não podemos nos esquecer do método commit que efetiva a escrita no arquivo.

Além do método putString(), a classe Editor possui vários outros métodos. Entre eles podemos destacar:

Modo	Descrição
apply()	Confirma as alterações para o objeto sharedPreference.
clear()	Remove todos os valores
remove (String key)	Remove o valor da chave que foi passado como um parâmetro
PutLong (String key, long value)	Salva um valor long
PutInt (String key, int value)	Salva um valor inteiro
PutFloat (String key, float value)	Salva um valor float/real
PutString (String key, String value)	Salva um valor String
PutBoolean (String key, boolean value)	Salva um valor booleano
GetLong (String key, long value)	Retorna o valor Long, referente à chave ou a um valor pré-definido, caso a chave não seja encontrada
GetInt (String key, int value)	Retorna o valor Int, referente à chave, ou a um valor pré-definido, caso a chave não seja encontrada
GetFloat (String key, float value)	Retorna o valor Float, referente à chave ou a um valor pré-definido, caso a chave não seja encontrada
GetString (String key, String value)	Retorna o valor String, referente à chave ou a um valor pré-definido, caso a chave não seja encontrada
GetBoolean (String key, boolean value)	Retorna o valor Boolean, referente à chave ou a um valor pré-definido, caso a chave não seja encontrada
Commit()	Salva as preferências no objeto SharedPreferences associado e em disco

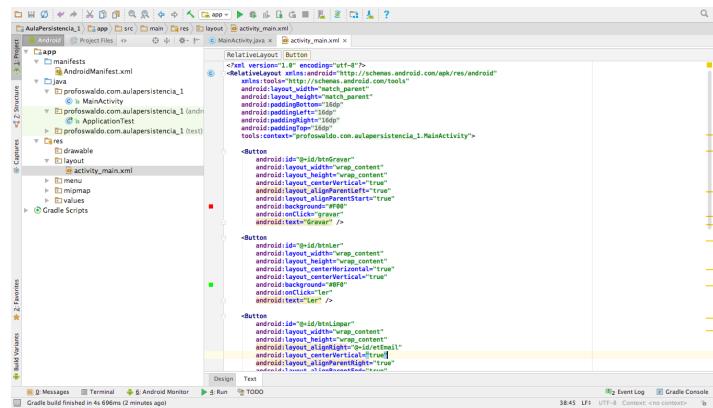
Para maiores informações, acesse o link:

<https://developer.android.com/reference/android/content/SharedPreferences.html> (glossário)

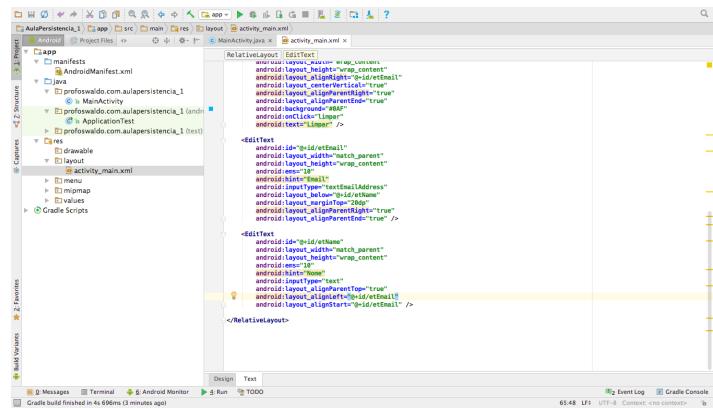
EXEMPLO - Shared Preferences

Embora estejamos estudando Share Preferences, vamos implementar o layout de nossa Main Activity para podemos visualizar os dados persistidos.

Crie, então, um arquivo de layout chamado activity_main.xml e implemente o código XML abaixo:



Não foi possível visualizar o arquivo na íntegra em uma só tela. Por isso, a tela abaixo exibe o resto do código.



O nome desse arquivo pode ser qualquer um. Se você estiver usando o Android Studio, provavelmente já tenha esse arquivo criado na pasta layout que se encontra dentro da pasta res.

Para executar esse aplicativo, precisamos implementar a Classe MainActivity.java. Seu código-fonte é demonstrado na tela abaixo:

```

package profoswido.com.ulapersistencia_1;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.content.SharedPreferences;
import android.view.Menu;
import android.view.View;
import android.widget.EditText;
public class MainActivity extends AppCompatActivity {
    private SharedPreferences sharedpreferences;
    private EditText nome;
    private EditText email;
    public static final String MINHASPREFERENCIAS = "minhaspreferencias";
    public static final String CHAVENOME = "chavennome";
    public static final String CHAVEEMAIL = "chaveemail";
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        nome = (EditText) findViewById(R.id.editText);
        email = (EditText) findViewById(R.id.editText2);
        sharedpreferences = getSharedPreferences(MINHASPREFERENCIAS, Context.MODE_PRIVATE);
        if (sharedpreferences.contains(CHAVENOME)) {
            nome.setText(sharedpreferences.getString(CHAVENOME, ""));
        }
        if (sharedpreferences.contains(CHAVEEMAIL)) {
            email.setText(sharedpreferences.getString(CHAVEEMAIL, ""));
        }
    }
    public void gravar(View view) {
        String n = nome.getText().toString();
        String e = email.getText().toString();
        SharedPreferences.Editor editor = sharedpreferences.edit();
        editor.putString(CHAVENOME, n);
        editor.putString(CHAVEEMAIL, e);
        editor.commit();
    }
    public void limpar(View view) {
        nome = (EditText) findViewById(R.id.editText);
        email = (EditText) findViewById(R.id.editText2);
        nome.setText("");
        email.setText("");
    }
    public void sair(View view) {
        nome = (EditText) findViewById(R.id.editText);
        email = (EditText) findViewById(R.id.editText2);
        SharedPreferences preferences = getSharedPreferences(MINHASPREFERENCIAS, Context.MODE_PRIVATE);
        if (sharedpreferences.contains(chavennome)) {
            nome.setText(sharedpreferences.getString(CHAVENOME, ""));
        }
        if (sharedpreferences.contains(chaveemail)) {
            email.setText(sharedpreferences.getString(CHAVEEMAIL, ""));
        }
    }
}

```

Esse código também é bastante extenso e, por isso, foi demonstrado em duas telas. Segue o complemento do código abaixo:

```

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.menu_main, menu);
    return true;
}

```

Observe a linha a seguir:

```
sharedpreferences = getSharedPreferences(MINHASPREFERENCIAS, Context.MODE_PRIVATE);
```

Estamos selecionando um arquivo de preferências específico através da constante que criamos, denominada MINHASPREFERENCIAS, e definindo o modo de acesso como sendo MODE_PRIVATE.

Para efetuar a gravação dos dados, estamos implementando as linhas abaixo:

```

SharedPreferences.Editor editor = sharedpreferences.edit();
editor.putString(CHAVENOME, n);
editor.putString(CHAVEEMAIL, e);
editor.commit();

```

O método edit() retorna a classe Editor da Shared Preferences, que nos oferece vários métodos para efetuar gravação de dados primitivos.

No exemplo acima, vamos implementar o método putString(), que possui 2 parâmetros:

1º - Chave

2º - Valor propriamente dito

É bastante familiar ao método put da interface Map na linguagem Java.

Para efetuar a leitura, é mais fácil ainda. Observe o trecho de código abaixo:

```
nome.setText(sharedpreferences.getString(CHAVENOME, ""));
email.setText(sharedpreferences.getString(CHAVEEMAIL, ""));
```

Assim como para gravação, existem vários métodos para leitura. Em nosso exemplo, estamos fazendo uso do getString(), que também possui dois parâmetros:

1º - Chave do dado que desejamos recuperar

2º - Define um valor padrão caso a chave não seja encontrada (opcional).

Para visualizar o aplicativo em ação, basta executar seu programa. A tela exibida é demonstrada abaixo:



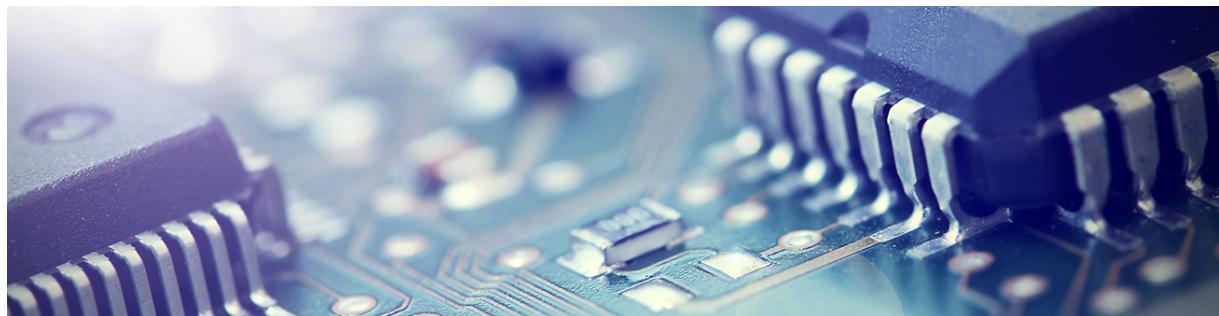
- 1) Digite o seu nome e e-mail;
- 2) Clique em Gravar;
- 3) Para verificar a recuperação dos dados, clique em Limpar e depois em Ler.

INTERNAL STORAGE

Nossa segunda opção de persistência de dados possibilita que sejam salvos arquivos na memória interna do aparelho.

É bastante oportuno ressaltar que essa memória interna é a mesma onde são armazenados os arquivos da Shared Preferences e que, quando o aplicativo é desinstalado, seus dados são apagados, pois são dados privados de seu aplicativo.

Esse tipo de persistência faz uso da API Java I/O e cria os arquivos no diretório /data/data/pacote.do.aplicativo/files.



Entre as várias classes que nos permitem efetuar esse tipo de persistência podemos destacar:

Gravação:

FileWriter:

Implementa a gravação de streams de caracteres.

Exemplo de gravação em arquivo texto:

```
File arquivo = new File("meu_arquivo.txt")
FileWriter fileWriter = new FileWriter( arquivo );
BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);
bufferedWriter.write("Oswaldo Borges Peres");
```

Como você pode observar, estamos trabalhando com Java.io da mesma forma que fazemos com programas para Desktop em Java.

No exemplo acima, vamos efetuar a gravação em um arquivo texto através da classe **FileWriter**. Poderíamos, a partir daí, usar o método **append()** dessa classe para efetuar a gravação, mas optamos por efetuar uma gravação bufferizada.

Para isso, faremos uso da classe **BufferedWriter** que recebe como parâmetro um **FileWriter**. Agora, basta usar o **método write** dessa classe para efetuar a gravação.

FileOutputStream:

Implementa a gravação de fluxos de bytes.

Exemplo de gravação em arquivo binário:

```
FileOutputStream fOut = openFileOutput("arquivoTeste", MODE_APPEND);
String str = "data";
fOut.write(str.getBytes());
fOut.close();
```

Esse exemplo faz uso do método **openFileOutput()**, que abre um arquivo de escrita, retornando um **FileOutputStream**.

Nele são passados dois parâmetros:

1º - O nome do arquivo

**2º - O modo de abertura (no exemplo
MODE_APPEND, para apenas acrescentar novas
informações ao arquivo)**

- Leitura:
 - FileReader:
 - Implementa a leitura de streams de caracteres.

Exemplo de leitura em arquivo texto:

```
File arquivo = new File("meu_arquivo.txt")
FileReader fileReader = new FileReader( arquivo );
BufferedReader bufferedReader = new BufferedReader(fileReader);
String linha= null;
while(bufferedReader.ready() ){
    linha += bufferedReader.readLine();
}
```

Vamos efetuar a leitura do arquivo texto gravado em nosso primeiro exemplo de gravação de dados. Para isso, faremos o uso da classe FileReader.

Implementaremos também a classe BufferedReader, para efetuar a leitura bufferizada, isto é, uma leitura de linha por linha, através de seu método readLine(). Esta recebe por parâmetro um FileReader.

- FileInputStream:
 - Implementa a leitura de fluxos de bytes.

Exemplo de leitura em arquivo binário:

```
FileInputStream fileInputStream = openFileInput("arquivoTeste");
int meucaracter;
String aux="";
while( (meucaracter = fileInputStream.read()) != -1){
    aux = aux + Character.toString((char) meucaracter);
}
fin.close();
```

Já nesse exemplo fizemos uso do método openFileInput(), que retorna um FileInputStream.

Agora, usaremos o seu método read(), que efetua a leitura byte a byte.

Para mais detalhes consulte os links:

[https://developer.android.com/reference/java/io/FileOutputStream.html \(glossário\)](https://developer.android.com/reference/java/io/FileOutputStream.html)

[https://developer.android.com/reference/java/io/FileInputStream.html \(glossário\)](https://developer.android.com/reference/java/io/FileInputStream.html)

[https://developer.android.com/reference/java/io/FileWriter.html \(glossário\)](https://developer.android.com/reference/java/io/FileWriter.html)

[https://developer.android.com/reference/java/io/FileReader.html \(glossário\)](https://developer.android.com/reference/java/io/FileReader.html)

EXEMPLO - Internal Storage

Nosso exemplo envolve dois arquivos.

Vamos implementar o primeiro deles: arquivo **activity_main.xml**.

Clique no vídeo e conheça.

EXTERNAL STORAGE

Muitas aplicações precisam efetuar sua persistência em algum tipo de memória externa, como um SD Card, que é removível, ou até mesmo em uma partição interna não removível do dispositivo.

Esses arquivos podem ser acessados por qualquer aplicação, inclusive pelo usuário. Felizmente, o Android oferece suporte nativo para esse tipo de persistência, porém precisamos verificar o estado em que se encontra a memória auxiliar.

Para tanto, podemos implementar o método `Environment.getExternalStorageState()`, que, em síntese, verifica se a mídia está disponível ou não.



Este pode retornar, entre outras constantes:

ENVIRONMENT.MEDIA_MOUNTED:	ENVIRONMENT.MEDIA_MOUNTED_READ_ONLY:
Se a mídia está montada e disponível para leitura e gravação.	Se a mídia está montada e disponível somente para leitura.
ENVIRONMENT.MEDIA_BAD_REMOVAL:	ENVIRONMENT.MEDIA_NOFS:
Se a mídia foi removida antes de ser desmontada.	Se a mídia está presente, mas está em branco ou usando um sistema de arquivos incompatível.
ENVIRONMENT.MEDIA_REMOVED:	ENVIRONMENT.MEDIA_UNMOUNTABLE:
Se a mídia não está presente.	Se a mídia está presente, mas não pode ser montada.

Para acessar os arquivos, devemos implementar:

API 7 ou Inferior:	API 8 ou Superior:
<code>Environment.getExternalStorageDirectory()</code> :	<code>Environment.getExternalStorageDir()</code>
Abrir um diretório que representa a raiz do armazenamento externo.	A principal diferença do anterior é que este requer um parâmetro <code>type</code> que especifica o tipo de subdiretório, já pré-configurado, que você deseja criar.

Como exemplo de tipos temos as constantes:

ENVIRONMENT DIRECTORY_MUSIC:	ENVIRONMENT DIRECTORY_RINGTONES:
Pasta padrão para músicas	Pasta padrão para áudios de seus ringtones
ENVIRONMENT DIRECTORY_ALARMS:	ENVIRONMENT DIRECTORY_NOTIFICATIONS:
Pasta padrão para áudios de seus alarmes	Pasta padrão para áudios da sua lista de notificações
ENVIRONMENT DIRECTORY_PICTURE:	ENVIRONMENT DIRECTORY_MOVIES:
Pasta padrão para fotos	Pasta padrão para vídeos
ENVIRONMENT DIRECTORY_DOWNLOADS:	ENVIRONMENT DIRECTORY_DCIM:
Pasta padrão para downloads	Pasta padrão para fotos e vídeos para dispositivos de câmera

Segue abaixo um exemplo:

Não podemos esquecer de habilitar, no arquivo `AndroidManifest.xml`, a permissão “`WRITE_EXTERNAL_STORAGE`”.

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

O trecho de código abaixo verifica se a mídia está montada e disponível somente para leitura:

```
private static boolean isExternalStorageReadOnly() {
    String extStorageState = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED_READ_ONLY.equals(extStorageState)) {
        return true;
    }
    return false;
}
```

O trecho de código abaixo verifica se a mídia está montada:

```
private static boolean isExternalStorageAvailable() {
    String extStorageState = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(extStorageState)) {
        return true;
    }
    return false;
}
```

EXEMPLO - External Storage

Como nos exemplos anteriormente desenvolvidos, vamos iniciar nosso aplicativo desenvolvendo o layout.

Clique no vídeo.

SQLITE DATABASES

O SQLite, ao contrário da maioria de gerenciadores de banco de dados, não está sob o controle de um programa servidor em separado do programa do cliente. O programa do usuário possui uma biblioteca SQLite compacta, que lida com todo o acesso ao banco de dados, eliminando a necessidade do usuário instalar, configurar e manter informações de bancos de dados complexos.

Outra característica do SQLite é que cada aplicação pode criar um ou mais bancos de dados que são visíveis somente para a aplicação que o criou.

Esse pequeno e notável banco de dados tem se tornado cada vez mais popular, principalmente por desenvolvedores Android e IOS, uma vez que essas plataformas oferecem suporte nativo.

Podemos criar o banco de dados em SQLite de três formas diferentes:



SQLITE DATABASES - CLASSE SQLITEOPENHELPER

No Android, os bancos criados serão acessíveis, pelo nome, somente pelas classes da aplicação. Salvo se for utilizado um provedor de conteúdo (Content Provider).

Para criar um banco de dados SQLite, precisamos implementar uma subclasse de SQLiteOpenHelper, que tem por responsabilidade criar o banco de dados, assim como gerenciar a versão do mesmo.

Para tanto, precisamos implementar os métodos:

onCreate():	onUpgrade():
Método chamado quando o banco é criado pela primeira vez	Método chamado quando a versão do banco de dados que está sendo aberto é diferente da versão existente

A tabela abaixo destaca alguns métodos da classe SQLiteOpenHelper:

Método	Descrição
<code>SQLiteOpenHelper(Context context, String name, SQLiteDatabase.CursorFactory factory, int version)</code>	Construtor responsável por criar um objeto capaz de gerenciar o banco de dados
<code>SQLiteDatabase getReadableDatabase()</code>	Método responsável por criar ou abrir um banco de dados apenas para leitura
<code>SQLiteDatabase getWritableDatabase()</code>	Método responsável por criar ou abrir um banco de dados para leitura e escrita
<code>public abstract void onCreate(SQLiteDatabase db)</code>	Método chamado apenas uma vez, quando a base de dados é criada pela primeira vez.
<code>public abstract void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)</code>	Método chamado quando o banco de dados precisa ser atualizado
<code>void onOpen(SQLiteDatabase db)</code>	Método chamado quando o banco de dados é aberto
<code>public void close()</code>	Método que fecha o banco de dados

Para mais detalhes consulte o link:

[\(glossário\)](https://developer.android.com/reference/android/database/sqlite/SQLiteOpenHelper.html)

SQLITE DATABASES - CLASSE SQLITEDATABASE

Essa classe contém métodos a serem executados nas tabelas em SQLite, como criar, atualizar, excluir, selecionar etc.

Existem muitos métodos na classe SQLiteDatabase. Alguns deles são:

Método	Descrição
<code>void execSQL(String sql)</code>	Executa sentenças que INCLuem SELECT, CREATE, TABLE, INSERT, UPDATE etc. Insere um registro e retorna o id
<code>long insert(Table, null, ContentValues values)</code>	Insere um registro(s) e retorna a quantidade de linhas modificadas
<code>int update(Table, ContentValues values, whereClause, whereArgs)</code>	Atualiza registro(s) e retorna a quantidade de linhas modificadas
<code>int delete(Table, whereClause, whereArgs)</code>	Deleta registro(s) e retorna a quantidade de linhas modificadas
<code>boolean isOpen()</code>	Verifica se está aberto
<code>Cursor query(Table, String selection, String[] selectionArgs, String groupBy, String having, String orderBy)</code>	Mostra e executa um SQL de consulta
<code>Cursor query(Table, String selection, String[] selectionArgs, String groupBy, String having, String orderBy, int limit)</code>	Mostra e executa um SQL de consulta
<code>Cursor query(Boolean distinct, Table, String selection, String[] selectionArgs, String groupBy, String having, String orderBy, String limit)</code>	Mostra e executa um SQL de consulta
<code>static SQLiteDatabase openDatabase(String path, CursorFactory factory, int flags)</code>	Abrir banco de dados de acordo com os flags: OPEN, READWRITE, OPEN, READONLY, EXCLUSIVE, FILE_PATH, NO_LOCALIZED_COLLATORS
<code>static SQLiteDatabase openOrCreateDatabase(String path, CursorFactory factory)</code>	Abrir ou cria banco de dados
<code>static SQLiteDatabase openOrCreateDatabase(File file, CursorFactory factory)</code>	Abrir ou cria banco de dados

É importante lembrar que, para a manipulação de dados do banco, podemos efetuar através dos métodos execSQL() e rawQuery() usando SQL ou através dos métodos insert(), delete(), update() e query().

Exemplo de execSQL:

```
sqliteDatabase.execSQL("INSERT INTO ALUNO (nome, idade) values ('Oswaldo', 50);",
```

Exemplo de rawSQL:

```
Cursor result = database.rawQuery("SELECT * FROM ALUNO", null);
```

Para mais detalhes consulte o link:

[\(glossário\)](https://developer.android.com/reference/android/database/sqlite/SQLiteDatabase.html)

SQLITE DATABASES - CLASSE CONTENTVALUES

Se você já implementou alguma vez um HashMap, não vai sentir nenhuma dificuldade de entender a classe ContentValues.

Através dessa classe, podemos armazenar o conteúdo de um registro para uma operação de persistência. Ela permite definir pares de chave-valor, onde a chave representa o identificador da coluna e o valor representa o conteúdo de uma coluna da tabela, como é demonstrado no código abaixo:

```
ContentValues contentValues = new ContentValues();
contentValue.put(DBHelper.NOME, name);
contentValue.put(DBHelper.EMAIL, desc);
database.insert(DBHelper.NOME_TABELA, null, contentValues);
```

SQLITE DATABASES - CURSOR

Assim como o ContentValues se assemelha ao HasMap, a classe Cursor se assemelha ao ResultSet do JDBC.

Ela é capaz de referenciar as linhas resultantes de uma consulta e, além disso, possui funções de navegação, tais como:

int getCount()	boolean moveToFirst()
Retorna o número de registros resultantes da consulta	Aponta para o primeiro registro
boolean moveToNext()	String getColumnNames (int columnIndex):
Aponta para o próximo registro	Retorna o nome da coluna a partir de seu índice
int getColumnIndex(String columnName) Retorna o índice da coluna a partir de seu nome, ou -1, caso a coluna não exista	

EXEMPLO - SQLite

Primeiramente vamos implementar o arquivo chamado activity_main.xml, que representa o layout de nossa MainActivity.

Clique no vídeo e conheça.

CONTENT PROVIDER

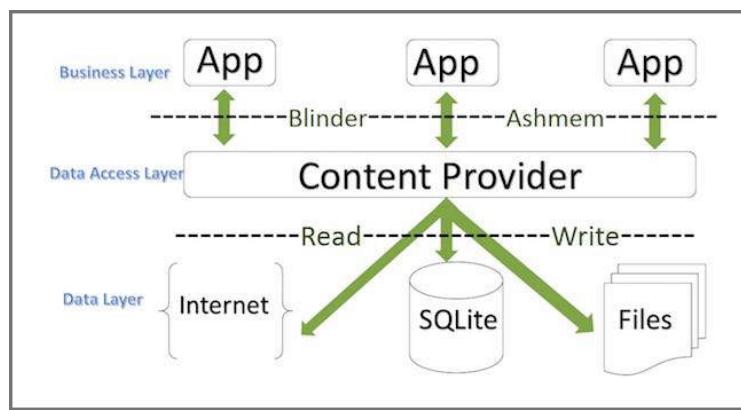
Normalmente, os aplicativos não podem acessar arquivos de outros aplicativos. Mas se isso for uma necessidade?



O Android disponibiliza o componente Content Provider, que permite o compartilhamento de dados entre aplicações.

Podemos usá-lo para ler e gravar dados armazenados nas preferências, arquivos ou até mesmo banco de dados.

A figura abaixo ilustra bem o nosso componente:



<https://www.tutorialspoint.com/android/images/content.jpg>

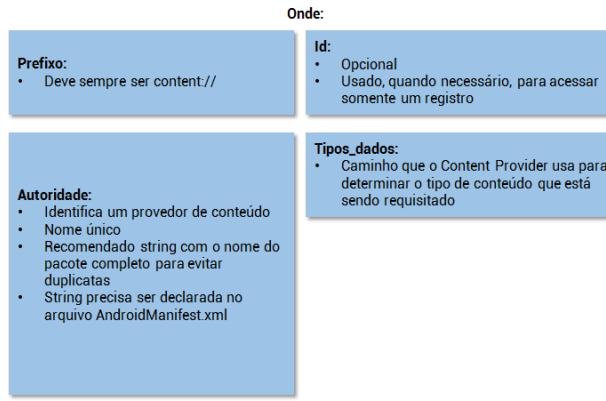
Um exemplo clássico desse componente é a lista de contatos, que pode fornecer nomes, endereços e telefones a vários aplicativos.

Podemos destacar como principais características do Content Provider:



Para que possamos acessar um Content Provider, ele possui um CONTENT_URI único, que identifica o conteúdo que vai gerenciar. Sua sintaxe é bastante simples:

< prefixo>://< Autoridade>/< tipo_dados>/



Exemplo:

Acessando uma lista de produtos:

content://profoswaldo.com.produtos/produtos/

Acessando o produto com id 1:

content://profoswaldo.com.produtos/produtos/1

O Android possui uma série de Content Providers nativos.

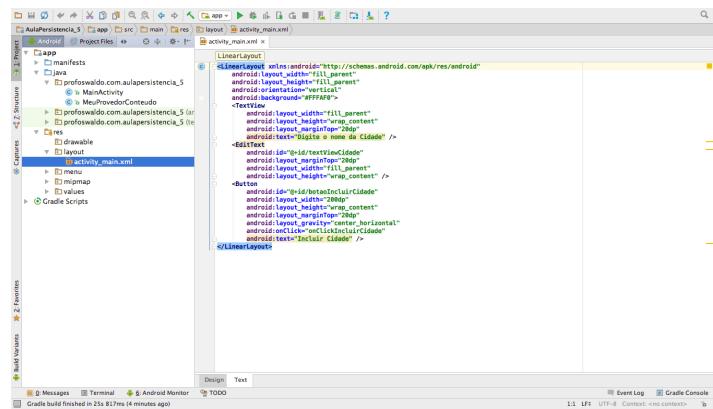
Clique [aqui \(glossário\)](#) e veja a relação de algumas URLs que podem ser utilizadas para ler informações públicas do Android.

Para facilitar o entendimento, vamos exemplificar.



EXEMPLO - Content Provider

Vamos começar pelo arquivo activity_main.xml que implementa nosso layout:



Esse layout é referente a tela:



Nossa MainActivity é bastante simples, conforme ilustrado abaixo:



The screenshot shows the Android Studio interface with the project structure on the left and the code editor on the right. The code editor displays the `MeuProviderConteudo.java` file, which extends `ContentProvider`. It includes imports for various Android packages and defines static final variables for the provider name, URL, and authority. The `insert()`, `update()`, and `delete()` methods are implemented to interact with a SQLite database.

```
import java.util.HashMap;
import android.content.ContentProvider;
import android.content.ContentValues;
import android.content.UriMatcher;
import android.database.Cursor;
import android.database.SQLException;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.database.sqlite.SQLiteQueryBuilder;
import android.net.Uri;

public class MeuProviderConteudo extends ContentProvider {
    static final String PROVIDER_NAME = "profowaldo.com.meuprovivedeConteudo";
    static final String AUTHORITY = "com.profowaldo.com.meuprovivedeConteudo";
    static final Uri CONTENT_URI = Uri.parse("content://" + AUTHORITY);
    static final String nome = "nome";
    static final UriMatcher urimatcher;
    private static HashMap<String, String> values;
    private static ContentValues conteudos;
    urimatcher = new UriMatcher(UriMatcher.NO_MATCH);
    urimatcher.addURI(AUTHORITY, "conteudo", conteudos);
    urimatcher.addURI(AUTHORITY, "conteudo#", conteudos);
}

@Override
public int delete(Uri uri, String selection, String[] selectionArgs) {
    int count = 0;
    boolean b = false;
    switch (urimatcher.match(uri)) {
        case conteudos:
            db.delete(TABELA, selection, selectionArgs);
            break;
        default:
            throw new IllegalArgumentException("Unknown Uri " + uri);
    }
    getContext().getContentResolver().notifyChange(uri, null);
    return count;
}
```

Quando efetuarmos um click no botão incluir cidade, o método `onClickIncluirCidade()` será chamado. Nele, criaremos o objeto do tipo `ContentValues`, visando encapsular o par chave-valor, onde a chave é nome e o valor é a cidade digitada.

Isso pode ser observado nas linhas abaixo:

```
ContentValues values = new ContentValues();
    values.put(MeuProvedorConteudo.nome, ((EditText)
        findViewById(R.id.textViewCidade)).getText().toString());
```

Para acessar o Content Provider, usamos a classe `ContentResolver`, que pode ser acessada através do método `getContentResolver()`, conforme demonstrado abaixo:

```
Uri uri = getContentResolver().insert(MeuProvedorConteudo.CONTENT_URI, values);
```

Finalmente, vamos implementar o nosso provedor de conteúdo. Isso será ilustrado nas telas abaixo, pois não foi possível exibir o código somente em uma tela:

```

@Override
public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder) {
    Cursor cursor = qb.query(db, projection, selection, selectionArgs, null,
        null, sortOrder);
    Context context = qb.getContext();
    ContentResolver resolver = context.getContentResolver();
    return cursor;
}

@Override
public Uri insert(Uri uri, ContentValues values) {
    long rowId = db.insert(TABLE, "", values);
    if (rowId == -1)
        throw new SQLException("Error inserting " + uri);
    ContentResolver resolver = getContext().getContentResolver();
    Uri newUri = ContentUris.withAppendedId(CONTENT_URI, rowId);
    resolver.notifyChange(newUri, null);
    return newUri;
}

@Override
public int delete(Uri uri, String selection, String[] selectionArgs) {
    int count = db.delete(TABLE, selection, selectionArgs);
    ContentResolver resolver = getContext().getContentResolver();
    resolver.notifyChange(uri, null);
    return count;
}

@Override
public int update(Uri uri, ContentValues values, String selection, String[] selectionArgs) {
    int count = db.update(TABLE, values, selection, selectionArgs);
    ContentResolver resolver = getContext().getContentResolver();
    resolver.notifyChange(uri, null);
    return count;
}

```

```

private static class DatabaseHelper extends SQLiteOpenHelper {
    DatabaseHelper(Context context) { super(context, BANCO_DADOS, null, VERSAO_BANCO_DADOS); }

    @Override
    public void onCreate(SQLiteDatabase db) { db.execSQL(SCRIPPTY_CREATE_TABLE); }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        db.execSQL("DROP TABLE IF EXISTS " + TABLE);
        onCreate(db);
    }
}

```

Como você pode observar, grande parte do código é referente à persistência no SQLite, já discutido anteriormente. Contudo, é necessário enfatizar alguns pontos.

Observe as linhas abaixo:

```
static final String PROVEDOR =
        "profoswaldo.com.MeuProvedorConteudo";
```

Essa string define a autoridade para o nosso Content Provider. Não podemos esquecer de que deve ser registrado no arquivo `AndroidManifest.xml`.

```
static final String URL = "content://" + PROVEDOR + "/conteudo";
static final Uri CONTENT_URI = Uri.parse(URL);
```

Vamos definir a `CONTENT_URI` personalizada, através do método estático `parse()` da classe `URI`.

Observe também as linhas abaixo:

```
static final int uriCode = 1;
static final UriMatcher uriMatcher;

private static HashMap values;
static {
    uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
    uriMatcher.addURI(PROVEDOR, "conteudo", uriCode);
    uriMatcher.addURI(PROVEDOR, "conteudo/*", uriCode);
}
```

O Content Provider utiliza um UriMatcher para determinar a operação a ser efetuada em seus métodos `query`, `insert`, `update` e `delete`.

A linha `uriMatcher = new UriMatcher(UriMatcher.NO_MATCH)` indica se certa URL é do tipo lista ou do tipo item.

O parâmetro `UriMatcher.NO_MATCH` informa ao aplicativo qual valor padrão retornará para uma correspondência.

A linha `uriMatcher.addURI(PROVEDOR, "conteudo", uriCode)` define qualquer URL que use a autoridade `profoswaldo.com.MeuProvedorConteudo` e tem um caminho denominado “**conteúdo**” retornando o valor `uriCode`, que, em nosso exemplo, foi definido como 1.

De forma similar à anterior, a linha `uriMatcher.addURI(PROVEDOR, "conteudo/*", uriCode)` define qualquer URL que use a autoridade `profoswaldo.com.MeuProvedorConteudo` e tem um caminho parecido com “**conteúdo/***”, onde * é um número e retornando o valor `uriCode`, que, em nosso exemplo, foi definido como 1.

Vamos observar também nosso método `insert`:

```
public Uri insert(Uri uri, ContentValues values) {
    long rowID = db.insert(TABELA, "", values);
    if (rowID > 0) {
        Uri _uri = ContentUris.withAppendedId(CONTENT_URI, rowID);
        getContext().getContentResolver().notifyChange(_uri, null);
        return _uri;
    }
    throw new SQLException("Falha na inclusão de cidade na uri " + uri);
}
```

A linha `long rowID = db.insert(TABELA, "", values)` efetua a gravação de dados em nosso banco de dados retornando o id de nossa linha no banco.

Através da linha `Uri _uri = ContentUris.withAppendedId(CONTENT_URI, rowID)`, construímos um URI para um conteúdo específico, pois adiciona um id específico.

É importante chamar o método `notifyChange()`. Isso fará com que as aplicações que estejam utilizando esse conjunto de dados sejam notificadas, permitindo a elas atualizar também os dados mostrados ao usuário.

Isso é feito através da linha `getContext().getContentResolver().notifyChange(_uri, null)`.

ATIVIDADES

1. Marque a opção correta sobre Shared Preferences:

- Normalmente usado para armazenar dados de configuração.
- Suporta a persistência de objetos. Quando implementamos a interface `Serializable` da Java.io, podemos gravar qualquer objeto em disco.
- Armazena dados públicos em uma memória externa do dispositivo (Ex: SD Card).
- Armazena dados estruturados em banco de dados privado. O Android inclui suporte nativo ao banco de dados relacional chamado SQLite.

Justificativa

2. Observe a imagem abaixo e desenvolva um pequeno aplicativo, conforme demonstrado na tela acima, que seja capaz de gravar usuário e senha digitados. Ele também deverá ser capaz de recuperar esses dados, assim como limpar os campos da tela quando desejado. É compulsório o uso de Shared Preferences.



Resposta Correta

Glossário