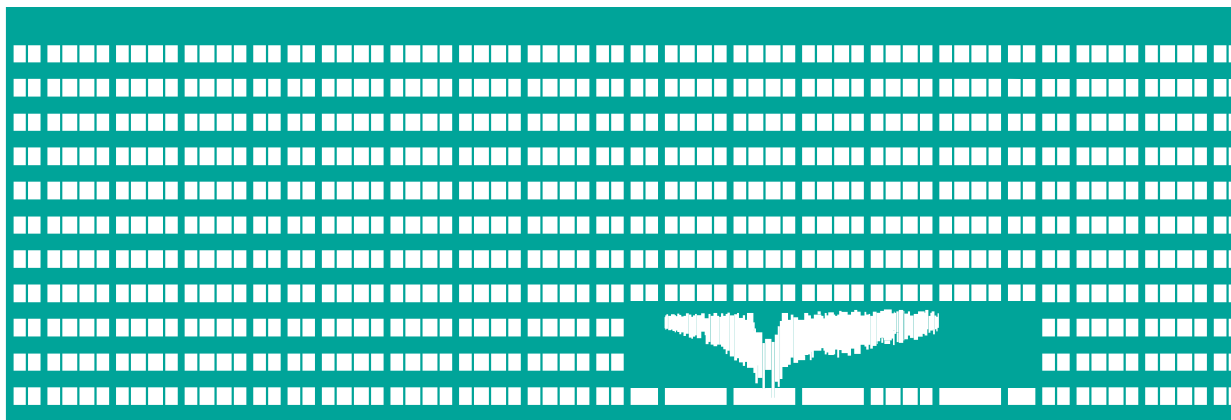


VŠB TECHNICKÁ
UNIVERZITA
OSTRAVA

VSB TECHNICAL
UNIVERSITY
OF OSTRAVA



EVROPSKÁ UNIE
Evropské strukturální a investiční fondy
Operační program Výzkum, vývoj a vzdělávání

MŠMT
MINISTERSTVO ŠKOLSTVÍ,
MLÁDEŽE A TĚLOVÝCHOVY

www.vsb.cz

Android – Data Storage

Michal Krumnikl



EVROPSKÁ UNIE
Evropské strukturální a investiční fondy
Operační program Výzkum, vývoj a vzdělávání



MINISTERSTVO ŠKOLSTVÍ,
MLÁDEŽE A TĚLOVÝCHOVY

Data Storage

- **Shared Preferences**

- Store private primitive data in key value pairs.

App preferences
Key-value pairs

- **Internal Storage**

- Store private data on the device memory.

Media

Shareable media files (images, audio files, videos)

- **External Storage**

- External storage for large data sets that are not private

App-specific files

Files meant for your app's use only

Data Storage

- **SQLite Databases**

- Store structured data in a private database.

- **Network Connection**

- Store data on the web with your own network server.

- **Content provider** is an optional component that exposes read/write access to your application data



Database
Structured data

Shared Preferences

- Framework that allows saving and retrieving persistent **key-value pairs** of **primitive data types**
- To get a SharedPreferences object for application, use one of two methods:
 - ***getSharedPreferences()***
 - Use this if you need multiple preferences files, which you specify with the first parameter.
 - ***getPreferences()***
 - Only one preferences file for your Activity.
- Writing values
 - Call ***edit()*** to get a ***SharedPreferences.Editor***
 - Use ***putBoolean()***, ***putString()***, etc.
 - Commit the values with ***commit()***
- Reading values
 - ***getBoolean()***, ***getString()***, etc.

***DataStore** is a modern data storage solution that you should use instead of SharedPreferences. It builds on Kotlin coroutines and Flow, and overcomes many of the drawbacks of SharedPreferences.*

Shared Preferences

- Obtain Shared Preferences for Activity
 - ***getSharedPreferences()***
- Reading values
 - ***getBoolean()***, ***getString()***, etc.
- Writing values
 - Call ***edit()*** to get a ***SharedPreferences.Editor***
 - Use ***putBoolean()***, ***putString()***, etc.
 - Commit the values with ***commit()***.

```

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    // Get the app's shared preferences
    // SharedPreferences app_preferences =
    //     PreferenceManager.getDefaultSharedPreferences(this);

    SharedPreferences app_preferences = getSharedPreferences(PREFS_NAME, 0);

    // Get the value for the run counter
    int counter = app_preferences.getInt("counter", 0);

    // Update the TextView
    TextView text = (TextView) findViewById(R.id.text);
    text.setText("This app has been started " + counter + " times.");

    // Increment the counter
    SharedPreferences.Editor editor = app_preferences.edit();
    editor.putInt("counter", ++counter);
    editor.commit(); // Very important
}
    
```

Shared Preferences Changes

- Receive a callback when a change happens to any one of the preferences
- ***SharedPreferences.OnSharedPreferenceChangeListener*** interface
- Register the listener ***registerOnSharedPreferenceChangeListener()***

```
public class SettingsActivity extends PreferenceActivity
    implements OnSharedPreferenceChangeListener {

    public void onSharedPreferenceChanged(
        SharedPreferences sharedPreferences,
        String key) { ... }

}
```

DataStore

- **Preferences DataStore** stores and accesses data using keys. Does not provide type safety.
- **Proto DataStore** stores data as instances of a custom data type. This implementation requires you to define a schema using protocol buffers, but it provides type safety.

<https://developer.android.com/topic/libraries/architecture/datastore>

```
// save
Single<Preferences> updateResult = datastore.updateDataAsync(prefsIn -> {
    MutablePreferences mutablePreferences = prefsIn.toMutablePreferences();
    Integer currentInt = prefsIn.get(INTEGER_KEY);
    mutablePreferences.set(INTEGER_KEY, currentInt != null ? currentInt + 1 : 1);
    return Single.just(mutablePreferences);
});
```


Internal Storage

- Save files directly on the **device's internal storage**
 - By default, files saved to the internal storage are **private to application**.
 - Other applications nor user can access them.
 - On Android 10 (API level 29) and higher, these **locations are encrypted**
- Available modes:
 - `MODE_PRIVATE`, `MODE_APPEND`, **`MODE_WORLD_READABLE/WRITEABLE`**.
- Apps themselves are stored within internal storage by default.
- ***FileOutputStream*** is specialized ***OutputStream*** that writes to a file in the file system.
 - For better performance a ***FileOutputStream*** is often wrapped by a ***BufferedOutputStream***
 - Access file with ***read()/write()*** and close the stream with ***close()***.
- Resource files
 - ***openRawResource()***

This constant was deprecated in API level 17.

Creating world-writable files is very dangerous, and likely to cause security holes in applications. It is strongly discouraged; instead, applications should use more formal mechanism for interactions such as [ContentProvider](#), [BroadcastReceiver](#), and [Service](#).

Internal Storage

- **File API** to access and store files.
- ***FileOutputStream*** is specialized ***OutputStream*** that writes to a file in the file system.
 - For better performance a ***FileOutputStream*** is often wrapped by a ***BufferedOutputStream***
 - Access file with ***read()/write()*** and close the stream with ***close()***.
 - **Catch exceptions.**
- Get names of all files within the filesDir directory by calling ***fileList()***

```
public class InternalStorage extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        String FILENAME = "hello_file";
        String string = "hello world!";

        FileOutputStream fos;
        try {
            fos = openFileOutput(FILENAME, Context.MODE_PRIVATE);
            fos.write(string.getBytes());
            fos.close();
        } catch (FileNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

Cache Files

- To cache some data, rather than store it persistently, use ***getCacheDir()*** to open a File that represents the **internal directory** where your application should save **temporary cache** files.
- If you need to cache some files, use ***createTempFile()***
file = File.createTempFile(fileName, null, context.getCacheDir());
- When the device is low on internal storage space, **Android may delete these cache files** to recover space.
- Maintain the cache files yourself and stay within a reasonable limit of space consumed, such as 1MB.
- Remove a file from the cache directory - ***cacheFile.delete()*** or ***context.deleteFile(cacheFileName)***
- When the user uninstalls your application, these files are removed.

External Storage

- Every Android device supports a shared “**external storage**” that you can use to save files.
 - This can be a **removable storage media** or an **internal storage**.
- Files saved to the external storage are **world-readable**.
- To give users more control over their files and to limit file clutter, apps that target Android 10 (API level 29) and higher are given scoped access into external storage, or **scoped storage**, by default.
- Before you do any work with the external storage, you should always call ***getExternalStorageState()*** to check whether the media is available.
 - The media might be mounted to a computer, missing, read-only, or in some other state.

*Note: **.nomedia** file will prevent Android's media scanner from reading your media files.*

File Access to External Storage

- For API Level 8 or greater, use ***getExternalFilesDir()*** to open a File that represents the external storage directory where you should save your files.
 - Method takes a type parameter that specifies the **type of subdirectory** you want
 - DIRECTORY_MUSIC
 - DIRECTORY_RINGTONES
 - ...
- If the user uninstalls your application, this **directory and all its contents will be deleted**.
- If you're using API Level 7 or lower, use ***getExternalStorageDirectory()***, to open a File representing the root of the external storage.

Permissions for External Storage

- To write to the external storage, you must request the **WRITE_EXTERNAL_STORAGE** permission.
 - If your app targets Android 11 (API level 30) or higher, the WRITE_EXTERNAL_STORAGE permission doesn't have any effect on your app's access to storage.
- To read the external storage (but not write), then you will need to declare the **READ_EXTERNAL_STORAGE**.
- Android 11 introduces the **MANAGE_EXTERNAL_STORAGE** permission, which provides write access to files outside the app-specific directory and MediaStore.

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

Manage all files on a storage device

- Some apps have a core use case that requires **broad access to files on a device**, but can't access them efficiently using the privacy-friendly storage best practices.
 - e.g., File managers, Backup and restore apps, Anti-virus apps, Document management apps, On-device file search, Disk and file encryption, Device-to-device data migration
- Request all-files access
 - Declare the **MANAGE_EXTERNAL_STORAGE** permission in the manifest.

```
adb shell appops set --uid PACKAGE_NAME MANAGE_EXTERNAL_STORAGE allow
```
 - **ACTION_MANAGE_ALL_FILES_ACCESS_PERMISSION** intent action to direct users to a system settings page where they can enable the following option for your app: Allow access to manage all files.
- To determine whether your app has been granted the **MANAGE_EXTERNAL_STORAGE** permission, call ***Environment.isExternalStorageManager()***.

Shared Files on External Storage

- Interact with the media store abstraction, use a ***ContentResolver*** object
- If you want to save files that are not specific to your application, save them to one of the public directories on the external storage.
 - **Images**, including photographs and screenshots, which are stored in the **DCIM/** and **Pictures/** directories. The system adds these files to the *MediaStore.Images* table.
 - **Videos**, which are stored in the **DCIM/**, **Movies/**, and **Pictures/** directories.
 - **Audio files**, which are stored in the **Alarms/**, **Audiobooks/**, **Music/**, **Notifications/**, **Podcasts/**, and **Ringtones/** directories. Additionally, the system recognizes audio playlists that are in the **Music/** or **Movies/** directories as well as voice recordings that are in the **Recordings/** directory. The system adds these files to the *MediaStore.Audio* table. The **Recordings/** directory isn't available on Android 11 (API level 30) and lower.
 - **Downloaded files**, which are stored in the **Download/** directory. On devices that run Android 10 (API level 29) and higher, these files are stored in the *MediaStore.Downloads* table. This table isn't available on Android 9 (API level 28) and lower.

Cache Files on External Storage

- For API Level 8 or greater, use ***getExternalCacheDir()*** to open a File that represents the external storage directory where you should save cache files.
- If the user uninstalls your application, these files will be automatically deleted. However, during the life of your application, you should manage these cache .
- For API Level 7 or lower, use ***getExternalStorageDirectory()*** to open a File that represents the root of the external storage, then write your cache data in the following directory:
 - ***/Android/data/<package_name>/cache/***

The <package_name> is Java style package name, such as "com.example.android.app".

Storage Space

- Current available space - ***getFreeSpace()***
- Total space in the storage volume - ***getTotalSpace()***

File removal

- Opened referenced file call ***delete()*** on itself.
- If the file is saved on internal storage, you can ask the Context to locate and delete a file by calling ***deleteFile()***

Files Summary

| Internal storage | External storage |
|---|---|
| Always available. | Not always available, because the user can mount the external storage as USB storage and in some cases remove it from the device. |
| Only your app can access files. Specifically, your app's internal storage directory is specified by your app's package name in a special location of the Android file system. | World-readable. Any app can read. |
| When the user uninstalls your app, the system removes all your app's files from internal storage. | When the user uninstalls your app, the system removes your app's files from here only if you save them in the directory from <i>getExternalFilesDir()</i> . |
| Internal storage is best when you want to be sure that neither the user nor other apps can access your files. | External storage is the best place for files that don't require access restrictions and for files that you want to share with other apps or allow the user to access with a computer. |

SQLite

- **Relational database management system** (RDBMS) contained in a C library
- SQLite is **not a client–server database**
- ACID-compliant and implements most of the SQL standard, generally following PostgreSQL syntax
- SQLite uses a **dynamically and weakly typed SQL syntax**
- SQLite is called **zero-conf** database
- A **standalone command-line utility** is provided in SQLite's distribution
- <https://www.sqlite.org>

SQL Databases

- SQL databases store data in tables of rows and columns:
 - The intersection of a row and column is called a **field**.
 - Fields contain data, references to other fields, or references to other tables.
 - **Rows** are identified by unique IDs.
 - **Columns** are identified by names that are unique per table.

| WORD_LIST_TABLE | | |
|-----------------|---------|-----------------|
| _id | word | definition |
| 1 | "alpha" | "first letter" |
| 2 | "beta" | "second letter" |
| 3 | "alpha" | "particle" |

SQL Databases

Data types

<https://www.sqlite.org/draft/datatype3.html>

SQL Commands

<https://www.sqlite.org/draft/lang.html>


| WORD_LIST_TABLE | | |
|-----------------|---------|-----------------|
| _id | word | definition |
| 1 | "alpha" | "first letter" |
| 2 | "beta" | "second letter" |
| 3 | "alpha" | "particle" |

Transactions

- A transaction is a sequence of operations performed as a single logical unit of work. A logical unit of work must exhibit four properties to qualify as a transaction:
 - **Atomicity.** Either all of a transaction's data modifications are performed, or none of them are performed. This is true even if the act of writing the change to the disk is interrupted by a program crash, operating system crash, or power failure.
 - **Consistency.** When completed, a transaction must leave all data in a consistent state.
 - **Isolation.** Modifications made by concurrent transactions must be isolated from the modifications made by any other concurrent transactions
 - **Durability.** After a transaction has completed, its effects are permanently in place in the system. The modifications persist even in the event of a system failure.

SQLite in Android

- Any databases you create will be accessible by name to any class in the application, but not outside the application.
- The recommended method to create a new SQLite database is to create a subclass of ***SQLiteOpenHelper*** and override the ***onCreate()*** method.
- To write to and read from the database, call ***getWritableDatabase()*** and ***getReadableDatabase()***, respectively. These both return a ***SQLiteDatabase*** object.
- You can execute SQLite queries using the SQLiteDatabase ***query()*** methods.
- **Room Persistence Library** as an abstraction layer for accessing information in your app's SQLite databases.

 **Caution:** Although these APIs are powerful, they are fairly low-level and require a great deal of time and effort to use:

- There is no compile-time verification of raw SQL queries. As your data graph changes, you need to update the affected SQL queries manually. This process can be time consuming and error prone.
- You need to use lots of boilerplate code to convert between SQL queries and data objects.

For these reasons, we **highly recommended** using the [Room Persistence Library](#) as an abstraction layer for accessing information in your app's SQLite databases.

Query Structure

SELECT columns FROM table WHERE column="value"

- **SELECT** columns: Select the columns to return. Use * to return all columns.
- **FROM** table: Specify the table from which to get results.
- **WHERE**: Keyword that precedes conditions that have to be met, for example column="value". Common operators are =, LIKE, <, and . To connect multiple conditions, use AND or OR.
- **ORDER BY**: Specify ASC for ascending, or DESC for descending. For default order, omit ORDER BY.
- **LIMIT**: Very useful keyword if you want to only get a limited number of results.

Query Structure

- ***SQLiteQueryBuilder*** provides several convenient methods for building complex queries.

```
String query = "SELECT * FROM WORD_LIST_TABLE";  
rawQuery(query, null);
```

- Every SQLite query will return a ***Cursor*** that points to all the rows found by the query.
- The ***Cursor*** is always the mechanism with which you can navigate results from a database query and read rows and columns.
 - A cursor is a pointer into a row of structured data.

Query Structure

- ***SQLiteQueryBuilder*** provides several convenient methods for building complex queries.

```
String table = "WORD_LIST_TABLE"
String[] columns = new String[]{"*"};
String selection = "word = ?"
String[] selectionArgs = new String[]{"alpha"};
String groupBy = null;
String having = null;
String orderBy = "word ASC"
String limit = "2,1"

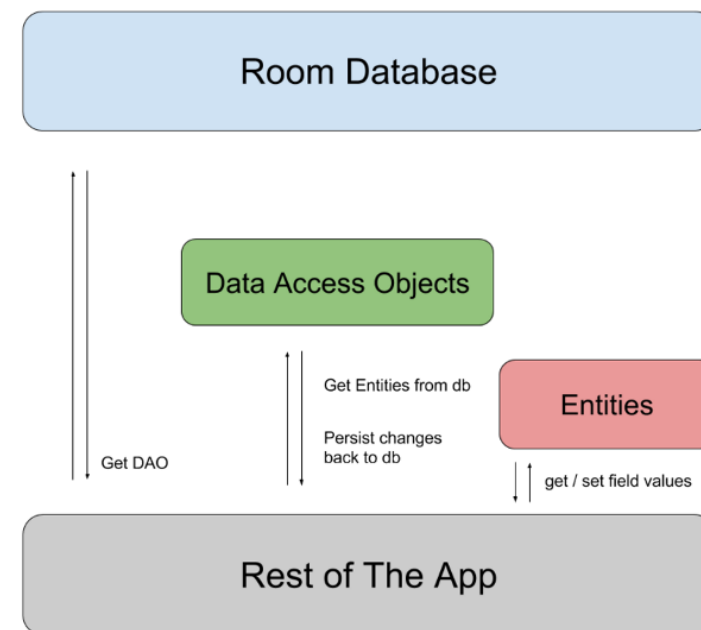
query(table, columns, selection, selectionArgs, groupBy, having, orderBy, limit);
```

Room persistence library

- Compile-time **verification of SQL queries**.
- **Convenience annotations** that minimize repetitive and error-prone boilerplate code.
- Streamlined database **migration paths**.

Quick Introduction

<https://developer.android.com/training/data-storage/room>

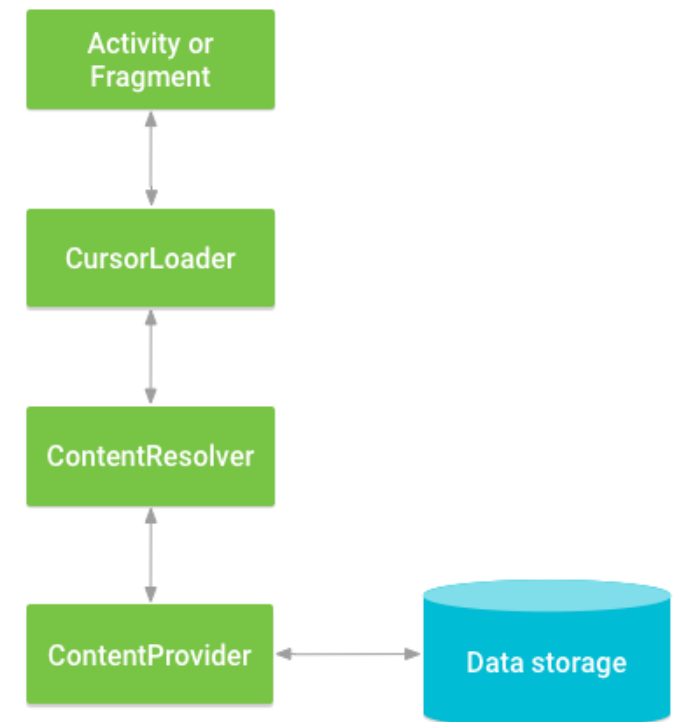


Content Providers

- Store and retrieve data and make it accessible to all applications.
- The only way to **share data across applications** -> there's no common storage area that all Android packages can access.
- Some of them listed in the *android.provider* package.
- You can query these providers for the data they contain.
- Some need the proper permission to access the data
- To make your own data public, you have two options:
 - Create your own content provider (a *ContentProvider* subclass)
 - Add the data to an existing provider — if there's one that controls the same type of data and you have permission to write to it.

Content Providers

- All content providers implement a common interface for querying the provider and returning results, as well as for adding, altering, and deleting data.
- It's an interface that clients use indirectly, generally through ***ContentResolver*** objects.
- Content providers expose their data as a simple table on a database model, where each row is a record and each column is data of a particular type and meaning.



Content Providers

- Every record includes a numeric **_ID** field that uniquely identifies the record within the table. IDs can be used to match records in related tables.
- A query returns a ***Cursor*** object that can move from record to record and column to column to read the contents of each field.
- Each content provider exposes a public URI (wrapped as a ***Uri*** object) that uniquely identifies its data set.

<https://developer.android.com/guide/topics/providers/content-provider-basics>

Content URI

- A. A **prefix** that the data is controlled by a content provider
- B. The **authority part** of the URI; it identifies the content provider.
- C. The **path** that the content provider uses to determine what kind of data is being requested.
- D. The **ID** of the specific record being requested, if any. This is the **_ID** value of the requested record.

`content://com.example.transportationprovider/trains/122`

The diagram shows the URI `content://com.example.transportationprovider/trains/122` with four labels below it: A, B, C, and D. Brackets connect the labels to parts of the URI: A is under `content:`, B is under `//com.example.transportationprovider`, C is under `/trains`, and D is under `/122`.

Querying a Content Provider

- To query a content provider:
 - The URI that identifies the provider
 - The names of the data fields you want to receive
 - The data types for those fields

```
import android.provider.Contacts.People;
import android.content.ContentUris;
import android.net.Uri;
import android.database.Cursor;

// Use the ContentUris method to produce the base URI for the contact with _ID == 23.
Uri myPerson = ContentUris.withAppendedId( People.CONTENT_URI, 23);

// Then query for this specific record:
Cursor cur = managedQuery( myPerson, null, null, null, null);
```

Querying a Content Provider

- The Cursor object returned by a query provides access to a recordset of results.
 - Cursor object has a separate method for reading each type of data — such as ***getString()***, ***getInt()***, and ***getFloat()***.

```
private void getColumnData( Cursor cur){
    if ( cur. moveToFirst()) {
        int nameColumn = cur. getColumnIndex( People. NAME);
        int phoneColumn = cur. getColumnIndex( People. NUMBER);
        do {
            // Get the field values
            String name = cur. getString( nameColumn);
            String phoneNumber = cur. getString( phoneColumn);
            // Do something with the values.
        } while ( cur. moveToNext());
    }
}
```

Inserting Record

- Map of key-value pairs in a `ContentValues` object, where each key matches the name of a column in the content provider and the value is the desired value for the new record in that column.
- Call ***ContentResolver.insert()*** and pass it the URI of the provider and the *ContentValues* map.

```
ContentValues values = new ContentValues();

// Add Abraham Lincoln to contacts and make him a favorite
values.put( People. NAME, "Abraham Lincoln");

// 1 = the new contact is added to favorites
// 0 = the new contact is not added to favorites
values.put( People. STARRED, 1);

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

Storing Binary Data

- Small amounts of binary data can be placed into the table using ***ContentValues.put()*** that takes a byte array.
- For large amount of binary data to add put a content: URI for the data in the table and call ***ContentResolver.openOutputStream()*** with the file's URI.
 - Content provider will store the data in a file and record the file path in a hidden field of the record.

Storing Binary Data

- For large amount of binary data to add put a content: URI for the data in the table and call ***ContentResolver.openOutputStream()*** with the file's URI.

```

ContentValues values = new ContentValues( 3);
values.put( Media.DISPLAY_NAME, "road_trip_1");
values.put( Media.DESRIPTION, "Day 1, trip to Los Angeles");
values.put( Media.MIME_TYPE, "image/jpeg");

Uri uri = getContentResolver().insert( Media.EXTERNAL_CONTENT_URI, values);

try {
    OutputStream outputStream = getContentResolver().openOutputStream( uri);
    sourceBitmap.compress( Bitmap.CompressFormat.JPEG, 50, outputStream);
    outputStream.close();
} catch ( Exception e) {
    Log.e( TAG, "exception while writing image", e);
}

```

Updating and Deleting Records

- To batch update a group of records call the ***ContentResolver.update()*** method with the columns and values to change.
- To delete a single record, call ***ContentResolver.delete()*** with the URI of a specific row.
- To delete multiple rows, call ***ContentResolver.delete()*** with the URI of the type of record to delete and an **SQL WHERE clause defining which rows to delete.**

```

public static final Uri CONTENT_URI = Uri. Parse("content://com.example.transportationprovider");

//Single records
content://com.example.transportationprovider/trains/122

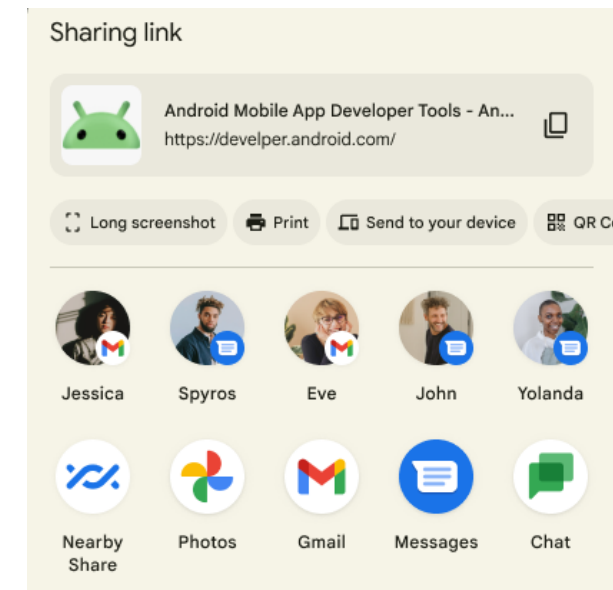
//Multiple records
content://com.example.transportationprovider/trains

```

Android Sharesheet

- For all types of sharing, create an intent and set its action to **Intent.ACTION_SEND**. To display the Android Sharesheet, call *Intent.createChooser()*, passing it your Intent object. It returns a version of your intent that always displays the Android Sharesheet.

```
Intent sendIntent = new Intent();  
sendIntent.setAction(Intent.ACTION_SEND);  
sendIntent.putExtra(Intent.EXTRA_TEXT, "This is my text to send.");  
sendIntent.setType("text/plain");  
  
Intent shareIntent = Intent.createChooser(sendIntent, null);  
startActivity(shareIntent);
```



References

- <https://developer.android.com>
- <https://www.sqlite.org/draft/>
- <https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/unit-4-saving-user-data/lesson-10-storing-data-with-room/10-0-c-sqlite-primer/10-0-c-sqlite-primer.html>

Thank you for your attention

Mgr. Ing. **Michal Krumnikl**, Ph.D.

+420 597 325 867

michal.krumnikl@vsb.cz

www.vsb.cz