Programmation Android
Données et fichiers

F. Michel

Plan

1. Solutions disponibles
2. App location
3. App files
4. Shared storage
5. Paires *Key-value*
6. Base de données locales avec l'API Room
7. Content providers

Plan

1. Solutions disponibles
2. App location
3. App files
4. Shared storage
5. Paires *Key-value*
6. Base de données locales avec l'API Room
7. Content providers

Quel support ?
internal vs. external storage

Deux localisations de stockage local sont distinguées :

Internal storage

- stockage inclus dans le matériel
- toujours disponible

External storage

- stockage amovible, ajouté après la fabrication, e.g. carte SD
- peut ne pas être disponible
- chemins démarrant de la racine, e.g. `/sdcard`

Les chemins des fichiers varient suivant les appareils
$\Rightarrow$ Ne jamais utiliser de chemin hardcodé !

Quel est le type de donnée concerné ?

1. Stockage applicatif privé

- fichiers utiles uniquement pour l'application

2. Préférences ⇒ librairie *Jetpack Preferences*

- stockage applicatif privé sous forme de paires **clé** → **valeur**

3. Base de données structurée ⇒ librairie Room

- Données privées structurées et persistantes (> 2 colonnes)

4. Stockage partagé

- fichiers utilisateurs potentiellement utilisés par d'autres applis
- ⇒ Fichiers médias : audio, images et vidéos
- ⇒ Autres : pdf, téléchargements, etc.

Taille / fiabilité / sensibilité des données ?

Quelle taille pour les données ?

- *Internal storage* : place limitée
- ⇒ Il est possible mixer les emplacements

Quelle est la fiabilité requise ?

- Certaines données peuvent être vitales pour l'appli, par exemple pour le démarrage
- ⇒ utiliser un répertoire sur le *internal storage* ou une BD

Données sensibles ?

- Certaines données ne doivent pas être accessible depuis d'autres app
- ⇒ *internal storage* / préférences / BD
- ⇒ *internal storage* est (1) caché pour l'utilisateur et (2) inaccessible aux autres app

Plan

1. Solutions disponibles
2. **App location**
3. App files
4. Shared storage
5. Paires *Key-value*
6. Base de données locales avec l'API Room
7. Content providers

Stockage d’une application

Par défaut, les applications sont stockées sur le *internal storage*.

Si l’APK est très gros ⇒

```
<manifest ...
  android:installLocation="preferExternal">
  ...
</manifest>
```

Accès des applications au *external storage*

Les permissions nécessaires évoluent avec les versions

Android versions 10-, API 29-

- READ_EXTERNAL_STORAGE ou WRITE_EXTERNAL_STORAGE
- ⇒ requises pour écrire en dehors du répertoire de l'appli

Scoped storage : API 29+

- accès *Scoped storage* par défaut : accès aux répertoires de l'app sur *external storage* et aux fichiers médias créés par l'app

Android 11+, API 30+

- Repose sur des permissions à la demande
- ⇒ WRITE_EXTERNAL_STORAGE n'a plus d'effet
- ⇒ MANAGE_EXTERNAL_STORAGE est introduit
  - permet l'accès en écriture en dehors du répertoire de l'appli et du *MediaStore*
  - ne nécessite pas forcément une déclaration préalable dans le manifest

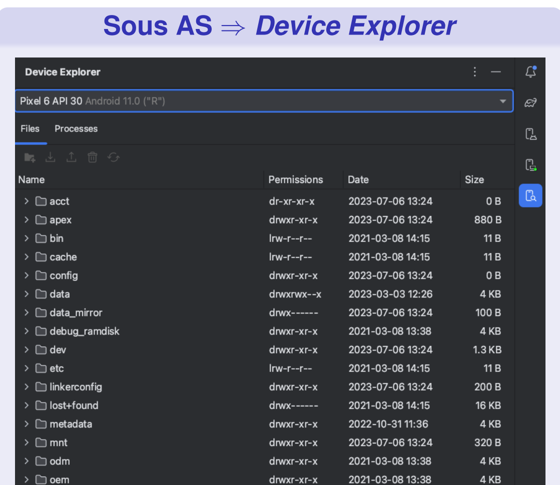
Accès / Permissions / Persistance

	Type de contenu	Méthodes	Permissions requises	Accès depuis autre app	Suppression lors de la désinstallation
Fichiers App	App seule	internal storage : getFilesDir() getCacheDir() external storage : getExternalFilesDir() getExternalCacheDir()	internal storage : Aucune external storage : Aucune API 19+	Aucun	Oui
Médias	images audio vidéos	API MediaStore	Depuis une autre app : API 30+ : READ_EXTERNAL_STORAGE API 29 : READ_EXTERNAL_STORAGE WRITE_EXTERNAL_STORAGE API 28- : Permissions pour tous les fichiers	Suivant les permissions	Non
Documents utilisateurs et autres	autre contenu partagé et téléchargements	*Storage Access Framework*	Aucune	via le *system file picker*	Non
Préférences	Paires clé-valeur	Jetpack Preferences SharedPreferences API	Aucune	Aucun	Oui
Base de données	données structurées	Room *persistence library*	Aucune	Aucun	Oui

Utilisation du *Device Explorer*

La plupart des données d'une device ne sont pas visibles, sauf sous AS avec les images adéquates sur l'émulateur ou avec l'*USB debugging* activé

Sous AS ⇒ *Device Explorer*



Utilisation du *Device Explorer*

Les fichiers ouverts avec le *Device Explorer* sont copiés puis ouverts dans un répertoire temporaire
$\Rightarrow$ Leur modification n'est répercutée sur la device : il faut les retéléverser

Répertoires utiles pour le debuggage

- `data/data/app_name/` : contient les fichiers de l'app sur le *internal storage*
- `sdcard/` : les fichiers stockés sur le *external storage*

Certains fichiers ne sont jamais visibles (e.g. présents dans `data/data/`), même avec le *Device Explorer*

Plan

1. Solutions disponibles
2. App location
3. **App files**
4. Shared storage
5. Paires *Key-value*
6. Base de données locales avec l'API Room
7. Content providers

App files : *Internal storage*

Deux localisations possibles

- fichiers persistants : `getFilesDir()`
- fichiers pour le cache de l'application : `getCacheDir()`

Fichiers persistants : création

Création avec l'API `java.io.File`

```
File file = new File(context.getFilesDir(), filename);
```

Avec les méthodes des objets de type `android.content.Context`

```
String filename = "myfile";
String fileContents = "Hello world!";
try (FileOutputStream fos = context.openFileOutput(filename, Context.MODE_PRIVATE)) {
    fos.write(fileContents.toByteArray());
}
```

Ce code, issu de la doc officielle ne compile pas :
`java.lang.String.toByteArray()` n'existe pas !

To allow other apps to access files stored in this directory within internal storage, use a `FileProvider` with the `FLAG_GRANT_READ_URI_PERMISSION` attribute.

Fichiers persistants : lecture

android.content.Context.openFileInput

```
FileInputStream fis = context.openFileInput("hello.txt");
InputStreamReader inputStreamReader =
        new InputStreamReader(fis, StandardCharsets.UTF_8);
StringBuilder stringBuilder = new StringBuilder();
try (BufferedReader reader = new BufferedReader(inputStreamReader)) {
    String line = reader.readLine();
    while (line != null) {
        stringBuilder.append(line).append('\n');
        line = reader.readLine();
    }
} catch (IOException e) {
    // Error occurred when opening raw file for reading.
} finally {
    String contents = stringBuilder.toString();
}
```

★ **Note:** If you need to access a file as a stream at install time, save the file in your project's /res/raw directory. You can open these files with `openRawResource()`, passing in the filename prefixed with `R.raw` as the resource ID. This method returns an `InputStream` that you can use to read the file. You cannot write to the original file.

Autres méthodes usuelles

Lister les fichiers de l'appli

```
Array<String> files = context.fileList();
```

Création d'un sous répertoire

```
File directory = context.getFilesDir();
File file = new File(directory, filename);
```

Création d'un fichier cache

```
File.createTempFile(filename, null, context.getCacheDir());
```

Accès à un fichier cache

```
File cacheFile = new File(context.getCacheDir(), filename);
```

Effacer un fichier: `cacheFile.delete()` ou `context.deleteFile(cacheFileName)`

App files : *External storage*

Même principe que pour le *internal storage*, mais avec des routines pour vérifier la disponibilité :

You can query the volume's state by calling `Environment.getExternalStorageState()`. If the returned state is `MEDIA_MOUNTED`, then you can read and write app-specific files within external storage. If it's `MEDIA_MOUNTED_READ_ONLY`, you can only read these files.

For example, the following methods are useful to determine the storage availability:

Kotlin Java

```
// Checks if a volume containing external storage is available
// for read and write.
private boolean isExternalStorageWritable() {
    return Environment.getExternalStorageState().equals(Environment.MEDIA_MOUNTED);
}

// Checks if a volume containing external storage is available to at least read.
private boolean isExternalStorageReadable() {
     return Environment.getExternalStorageState().equals(Environment.MEDIA_MOUNTED) ||
            Environment.getExternalStorageState().equals(Environment.MEDIA_MOUNTED_READ_(
}
```

App files : External storage

Access persistent files

To access app-specific files from external storage, call `getExternalFilesDir()`.

To help maintain your app's performance, don't open and close the same file multiple times.

The following code snippet demonstrates how to call `getExternalFilesDir()`:

★ **Note:** On Android 11 (API level 30) and higher, apps cannot create their own app-specific directory on external storage.

Vérifier la place disponible

The following code snippet shows an example of how your app can query free space on the device:

Kotlin Java

```
// App needs 10 MB within internal storage.
private static final long NUM_BYTES_NEEDED_FOR_MY_APP = 1024 * 1024 * 10L;

StorageManager storageManager =
        getApplicationContext().getSystemService(StorageManager.class);
UUID appSpecificInternalDirUuid = storageManager.getUuidForPath(getFilesDir());
long availableBytes =
        storageManager.getAllocatableBytes(appSpecificInternalDirUuid);
if (availableBytes >= NUM_BYTES_NEEDED_FOR_MY_APP) {
    storageManager.allocateBytes(
            appSpecificInternalDirUuid, NUM_BYTES_NEEDED_FOR_MY_APP);
} else {
    // To request that the user remove all app cache files instead, set
    // "action" to ACTION_CLEAR_APP_CACHE.
    Intent storageIntent = new Intent();
    storageIntent.setAction(ACTION_MANAGE_STORAGE);
}
```

Demander à l'utilisateur de faire de la place

Ask user to remove some device files

To request that the user choose files on the device to remove, invoke an intent that includes the `ACTION_MANAGE_STORAGE` action. This intent displays a prompt to the user. If desired, this prompt can show the amount of free space available on the device. To show this user-friendly information, use the result of the following calculation:

```
StorageStatsManager.getFreeBytes() / StorageStatsManager.getTotalBytes()
```

Ask user to remove all cache files

Alternatively, you can request that the user clear the cache files from **all** apps on the device. To do so, invoke an intent that includes the `ACTION_CLEAR_APP_CACHE` intent action.

Caution: The `ACTION_CLEAR_APP_CACHE` intent action can substantially affect device battery life and might remove a large number of files from the device.

Plan

1. Solutions disponibles
2. App location
3. App files
4. **Shared storage**
5. Paires *Key-value*
6. Base de données locales avec l'API Room
7. Content providers

Shared storage

Le *Shared storage* doit être utilisé pour les données utilisateurs accessibles par les autres app, et qui ne doivent pas être effacées lors de la désinstallation.

Android fournit des API pour chacun des types suivants :

- ***Media*** : Le système fournit des répertoires publiques standards : photos, vidéos, audio ⇒ API `MediaStore`
- **Autres documents** (pdf, epub, etc.) : accès via le *Storage Access Framework* (explorateur de fichiers)
- **Datasets** : Android 11+, API 30+ fournit une API pour leur gestion : BlobStoreManager API

Media files

Les fichiers médias sont stockés dans des répertories standards ⇒ faiclite l'indexation et l'utilisation par les apps

Access media files from shared storage

To provide a more enriched user experience, many apps let users contribute and access media that's available on an external storage volume. The framework provides an optimized index into media collections, called the *media store*, that lets users retrieve and update these media files more easily. Even after your app is uninstalled, these files remain on the user's device.

Note: If your app works with media files that provide value to the user only within your app, it's best to store them in app-specific directories within external storage.

Photo picker

As an alternative to using the media store, the Android photo picker tool provides a safe, built-in way for users to select media files without needing to grant your app access to their entire media library. This is only available on supported devices. For more information, see the photo picker guide.

Exemple d'utilisation de l'API `MediaStore`

Media store

To interact with the media store abstraction, use a `ContentResolver` object that you retrieve from your app's context:

Kotlin Java

```
String[] projection = new String[] {
        media-database-columns-to-retrieve
};
String selection = sql-where-clause-with-placeholder-variables;
String[] selectionArgs = new String[] {
        values-of-placeholder-variables
};
String sortOrder = sql-order-by-clause;

Cursor cursor = getApplicationContext().getContentResolver().query(
    MediaStore.media-type.Media.EXTERNAL_CONTENT_URI,
    projection,
    selection,
    selectionArgs,
    sortOrder
);

while (cursor.moveToNext()) {
    // Use an ID column from the projection to get
    // a URI representing the media item itself.
}
```

Exemple d’utilisation de l’API `MediaStore`

1/2

```java
private void getPicturesFiles() {
    // IMAGES
    // Define the columns you want to retrieve
    String[] projection = new String[]{
            MediaStore.Images.Media._ID,
            MediaStore.Images.Media.DISPLAY_NAME,
            MediaStore.Images.Media.DATE_ADDED
    };

// Define the selection criteria (SQL where clause) with placeholder variables
    String selection = MediaStore.Images.Media.RELATIVE_PATH + " LIKE ?";

// Define the values to replace the placeholders
    String[] selectionArgs = new String[]{
            "%" + Environment.DIRECTORY_PICTURES + "%"
    };

// Define how you want the results sorted
    String sortOrder = MediaStore.Images.Media.DATE_ADDED + " DESC";


// Get the content URI for the "Pictures" directory on the primary external storage volume
    Uri contentUri = MediaStore.Images.Media.getContentUri(MediaStore.VOLUME_EXTERNAL_PRIMARY);
```

Exemple d'utilisation de l'API `MediaStore`

2/2

```
// Perform the query
        Uri externalContentUri = MediaStore.Images.Media.EXTERNAL_CONTENT_URI;
        Cursor cursor = getApplicationContext().getContentResolver().query(
                externalContentUri,
                projection: null,
                selection: null,
                selectionArgs: null,
                sortOrder: null
        );

// Iterate over the results
        while (cursor.moveToNext()) {
            // Retrieve the ID, display name, and date added from the cursor
            int columnIndex = cursor.getColumnIndex(MediaStore.Images.Media._ID);
            long id = cursor.getLong(columnIndex);
            columnIndex = cursor.getColumnIndex(MediaStore.Images.Media.DISPLAY_NAME);
            String displayName = cursor.getString(columnIndex);
            columnIndex = cursor.getColumnIndex(MediaStore.Images.Media.DATE_ADDED);
            long dateAdded = cursor.getLong(columnIndex);

            // Do something with the ID, display name, and date added, like print them to the console
            Log.i( tag: "MediaStoreExample", msg: "ID: " + id + ", Display Name: " + displayName + ", Date Added: " + dateAc
        }
// Always remember to close the cursor when you're done with it
        cursor.close();
    }
```

Access documents and other files from shared storage

On devices that run Android 4.4 (API level 19) and higher, your app can interact with a documents provider, including external storage volumes and cloud-based storage, using the Storage Access Framework. This framework allows users to interact with a system picker to choose a documents provider and select specific documents and other files for your app to create, open, or modify.

Because the user is involved in selecting the files or directories that your app can access, this mechanism doesn't require any system permissions, and user control and privacy is enhanced. Additionally, these files, which are stored outside of an app-specific directory and outside of the media store, remain on the device after your app is uninstalled.

Using the framework involves the following steps:

1. An app invokes an intent that contains a storage-related action. This action corresponds to a specific use case that the framework makes available.
2. The user sees a system picker, allowing them to browse a documents provider and choose a location or document where the storage-related action takes place.
3. The app gains read and write access to a URI that represents the user's chosen location or document. Using this URI, the app can perform operations on the chosen location.

★ **Note:** If your app accesses media files, consider using the photo picker, which provides a convenient interface for accessing photos and videos.

Création et accès se font via des Intents

Use cases for accessing documents and other files

The Storage Access Framework supports the following use cases for accessing files and other documents.

Create a new file

The `ACTION_CREATE_DOCUMENT` intent action allows users to save a file in a specific location.

Open a document or file

The `ACTION_OPEN_DOCUMENT` intent action allows users to select a specific document or file to open.

Grant access to a directory's contents

The `ACTION_OPEN_DOCUMENT_TREE` intent action, available on Android 5.0 (API level 21) and higher, allows users to select a specific directory, granting your app access to all of the files and sub-directories within that directory.

Création par l’utilisateur d’un fichier partagé

```
// Request code for creating a PDF document.
private static final int CREATE_FILE = 1;

private void createFile(Uri pickerInitialUri) {
    Intent intent = new Intent(Intent.ACTION_CREATE_DOCUMENT);
    intent.addCategory(Intent.CATEGORY_OPENABLE);
    intent.setType("application/pdf");
    intent.putExtra(Intent.EXTRA_TITLE, "invoice.pdf");

    // Optionally, specify a URI for the directory that should be opened in
    // the system file picker when your app creates the document.
    intent.putExtra(DocumentsContract.EXTRA_INITIAL_URI, pickerInitialUri);

    startActivityForResult(intent, CREATE_FILE);
}
```

Accès par l'utilisateur d'un fichier partagé

```
// Request code for selecting a PDF document.
private static final int PICK_PDF_FILE = 2;

private void openFile(Uri pickerInitialUri) {
    Intent intent = new Intent(Intent.ACTION_OPEN_DOCUMENT);
    intent.addCategory(Intent.CATEGORY_OPENABLE);
    intent.setType("application/pdf");

    // Optionally, specify a URI for the file that should appear in the
    // system file picker when it loads.
    intent.putExtra(DocumentsContract.EXTRA_INITIAL_URI, pickerInitialUri);

    startActivityForResult(intent, PICK_PDF_FILE);
}
```

Access restrictions

On Android 11 (API level 30) and higher, you cannot use the `ACTION_OPEN_DOCUMENT` intent action to request that the user select individual files from the following directories:

- The `Android/data/` directory and all subdirectories.
- The `Android/obb/` directory and all subdirectories.

Récupération de l’URI du fichier créé/ouvert/sélectionné

```
@Override
public void onActivityResult(int requestCode, int resultCode,
        Intent resultData) {
    if (requestCode == your-request-code
            && resultCode == Activity.RESULT_OK) {
        // The result data contains a URI for the document or directory that
        // the user selected.
        Uri uri = null;
        if (resultData != null) {
            uri = resultData.getData();
            // Perform operations on the document using its URI.
        }
    }
}
```

By getting a reference to the selected item's URI, your app can perform several operations on the item. For example, you can access the item's metadata, edit the item in place, and delete the item.

Plan

1. Solutions disponibles
2. App location
3. App files
4. Shared storage
5. **Paires *Key-value***
6. Base de données locales avec l'API Room
7. Content providers

SharedPreferences API

Save simple data with SharedPreferences

If you have a relatively small collection of key-values that you'd like to save, you can use the `SharedPreferences` APIs. A `SharedPreferences` object points to a file containing key-value pairs and provides simple methods to read and write them. Each `SharedPreferences` file is managed by the framework and can be private or shared.

This page shows you how to use the `SharedPreferences` APIs to store and retrieve simple values.

Caution: `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`.

Read the DataStore guide for more information.

Note: The `SharedPreferences` APIs are for reading and writing key-value pairs, and you shouldn't confuse them with the `Preference` APIs, which help you build a user interface for your app settings (although they also use `SharedPreferences` to save the user's settings). For information about the `Preference` APIs, see the Settings developer guide.

`android.content.SharedPreferences`

SharedPreferences API

Get a handle to shared preferences

You can create a new shared preference file or access an existing one by calling one of these methods:

- `getSharedPreferences()`: Use this if you need multiple shared preference files identified by name, which you specify with the first parameter. You can call this from any `Context` in your app.
- `getPreferences()`: Use this from an `Activity` if you need to use only one shared preference file for the activity. Because this retrieves a default shared preference file that belongs to the activity, you don't need to supply a name.

For example, the following code accesses the shared preferences file that's identified by the resource string `R.string.preference_file_key` and opens it using the private mode so the file is accessible by only your app:

Kotlin Java

```java
Context context = getActivity();
SharedPreferences sharedPref = context.getSharedPreferences(
        getString(R.string.preference_file_key), Context.MODE_PRIVATE);
```

SharedPreferences API

On peut avoir un fichier de préférences pour chaque activité

When naming your shared preference files, you should use a name that's uniquely identifiable to your app. A good way to do this is prefix the file name with your application ID. For example: `"com.example.myapp.PREFERENCE_FILE_KEY"`

Alternatively, if you need just one shared preference file for your activity, you can use the `getPreferences()` method:

Kotlin Java

```
SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE);
```

Pour stocker des données communes à toute l'appli ⇒ `getDefaultSharedPreferences()`

SharedPreferences API

Write to shared preferences

To write to a shared preferences file, create a `SharedPreferences.Editor` by calling `edit()` on your `SharedPreferences`.

Pass the keys and values you want to write with methods such as: `putInt()` and `putString()`. Then call `apply()` or `commit()` to save the changes. For example:

Kotlin Java

```java
SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE);
SharedPreferences.Editor editor = sharedPref.edit();
editor.putInt(getString(R.string.saved_high_score_key), newHighScore);
editor.apply();
```

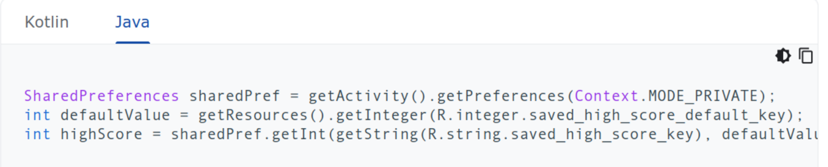
`apply()` changes the in-memory `SharedPreferences` object immediately but writes the updates to disk asynchronously. Alternatively, you can use `commit()` to write the data to disk synchronously. But because `commit()` is synchronous, you should avoid calling it from your main thread because it could pause your UI rendering.

Ce fonctionnement explique pourquoi cette API est en passe d'être remplacée

SharedPreferences API

Read from shared preferences

To retrieve values from a shared preferences file, call methods such as `getInt()` and `getString()`, providing the key for the value you want, and optionally a default value to return if the key isn't present. For example:



DataStore API ⇒ librairie Jetpack

Jetpack DataStore is a data storage solution that allows you to store key-value pairs or typed objects with protocol buffers. DataStore uses Kotlin coroutines and Flow to store data asynchronously, consistently, and transactionally.

If you're currently using `SharedPreferences` to store data, consider migrating to DataStore instead.

Note: If you need to support large or complex datasets, partial updates, or referential integrity, consider using Room instead of DataStore. DataStore is ideal for small, simple datasets and does not support partial updates or referential integrity.

Preferences DataStore and Proto DataStore

DataStore provides two different implementations: Preferences DataStore and Proto DataStore.

- **Preferences DataStore** stores and accesses data using keys. This implementation does not require a predefined schema, and it 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.

DataStore API ⇒ librairie Jetpack

Précautions d'usage

Using DataStore correctly

In order to use DataStore correctly always keep in mind the following rules:

1. **Never create more than one instance of `DataStore` for a given file in the same process.** Doing so can break all DataStore functionality. If there are multiple DataStores active for a given file in the same process, DataStore will throw `IllegalStateException` when reading or updating data.
2. **The generic type of the DataStore must be immutable.** Mutating a type used in DataStore invalidates any guarantees that DataStore provides and creates potentially serious, hard-to-catch bugs. It is strongly recommended that you use protocol buffers which provide immutability guarantees, a simple API and efficient serialization.
3. **Never mix usages of `SingleProcessDataStore` and `MultiProcessDataStore` for the same file.** If you intend to access the `DataStore` from more than one process always use `MultiProcessDataStore`.

DataStore API ⇒ librairie Jetpack

Ajout des dépendances dans le `build.gradle` de l'app

Setup

To use Jetpack DataStore in your app, add the following to your Gradle file depending on which implementation you want to use:

Preferences DataStore

Groovy Kotlin

```
// Preferences DataStore (SharedPreferences like APIs)
dependencies {
    implementation("androidx.datastore:datastore-preferences:1.0.0")

    // optional - RxJava2 support
    implementation("androidx.datastore:datastore-preferences-rxjava2:1.0.0")

    // optional - RxJava3 support
    implementation("androidx.datastore:datastore-preferences-rxjava3:1.0.0")
}

// Alternatively - use the following artifact without an Android dependency.
dependencies {
    implementation("androidx.datastore:datastore-preferences-core:1.0.0")
}
```

DataStore API $\Rightarrow$ librairie Jetpack

Proto DataStore

Groovy Kotlin

```
// Typed DataStore (Typed API surface, such as Proto)
dependencies {
    implementation("androidx.datastore:datastore:1.0.0")

    // optional - RxJava2 support
    implementation("androidx.datastore:datastore-rxjava2:1.0.0")

    // optional - RxJava3 support
    implementation("androidx.datastore:datastore-rxjava3:1.0.0")
}

// Alternatively - use the following artifact without an Android dependency.
dependencies {
    implementation("androidx.datastore:datastore-core:1.0.0")
}
```

Comparaison des différentes API

Feature	SharedPreferences	PreferencesDataStore	ProtoDataStore
Async API	✅ (only for reading changed values, via listener)	✅ (via `Flow` and RxJava 2 & 3 `Flowable`)	✅ (via `Flow` and RxJava 2 & 3 `Flowable`)
Synchronous API	✅ (but not safe to call on UI thread)	❌	❌
Safe to call on UI thread	❌1	✅ (work is moved to `Dispatchers.IO` under the hood)	✅ (work is moved to `Dispatchers.IO` under the hood)
Can signal errors	❌	✅	✅
Safe from runtime exceptions	❌2	✅	✅
Has a transactional API with strong consistency guarantees	❌	✅	✅
Handles data migration	❌	✅	✅
Type safety	❌	❌	✅ with Protocol Buffers

Preferences DataStore API

Store key-value pairs with Preferences DataStore

The Preferences DataStore implementation uses the `DataStore` and `Preferences` classes to persist simple key-value pairs to disk.

Create a Preferences DataStore

Use the property delegate created by `preferencesDataStore` to create an instance of `Datastore<Preferences>`. Call it once at the top level of your kotlin file, and access it through this property throughout the rest of your application. This makes it easier to keep your `DataStore` as a singleton. Alternatively, use `RxPreferenceDataStoreBuilder` if you're using RxJava. The mandatory `name` parameter is the name of the Preferences DataStore.

Kotlin Java

```java
RxDataStore<Preferences> dataStore =
  new RxPreferenceDataStoreBuilder(context, /*name=*/ "settings").build();
```

Preferences DataStore API

Read from a Preferences DataStore

Because Preferences DataStore does not use a predefined schema, you must use the corresponding key type function to define a key for each value that you need to store in the `DataStore<Preferences>` instance. For example, to define a key for an int value, use `intPreferencesKey()`. Then, use the `DataStore.data` property to expose the appropriate stored value using a `Flow`.

Kotlin   Java

```
Preferences.Key<Integer> EXAMPLE_COUNTER = PreferencesKeys.int("example_counter");

Flowable<Integer> exampleCounterFlow =
  dataStore.data().map(prefs -> prefs.get(EXAMPLE_COUNTER));
```

Preferences DataStore API

Write to a Preferences DataStore

Preferences DataStore provides an `edit()` function that transactionally updates the data in a `DataStore`. The function's `transform` parameter accepts a block of code where you can update the values as needed. All of the code in the transform block is treated as a single transaction.

Kotlin Java

```
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);
});
// The update is completed once updateResult is completed.
```

Proto DataStore API

Store typed objects with Proto DataStore

The Proto DataStore implementation uses DataStore and protocol buffers to persist typed objects to disk.

Define a schema

Proto DataStore requires a predefined schema in a proto file in the `app/src/main/proto/` directory. This schema defines the type for the objects that you persist in your Proto DataStore. To learn more about defining a proto schema, see the protobuf language guide.

```
syntax = "proto3";

option java_package = "com.example.application";
option java_multiple_files = true;

message Settings {
  int32 example_counter = 1;
}
```

★ **Note:** The class for your stored objects is generated at compile time from the `message` defined in the proto file. Make sure you rebuild your project.

protobuf language guide

Exemple d'utilsiation de l'API *Proto DataStore*

Plan

1 Solutions disponibles

2 App location

3 App files

4 Shared storage

5 Paires *Key-value*

6 **Base de données locales avec l'API Room**

7 Content providers

Room API ⇒ librairie Jetpack

Utile pour manipuler des données structurées nécessitant une base données (plus de deux colonnes)

Save data in a local database using Room

Part of Android Jetpack.

Apps that handle non-trivial amounts of structured data can benefit greatly from persisting that data locally. The most common use case is to cache relevant pieces of data so that when the device cannot access the network, the user can still browse that content while they are offline.

The Room persistence library provides an abstraction layer over SQLite to allow fluent database access while harnessing the full power of SQLite. In particular, Room provides the following benefits:

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

Because of these considerations, we highly recommend that you use Room instead of using the SQLite APIs directly.

Room API ⇒ librairie Jetpack

Ajout des dépendances dans le `build.gradle` de l'app

To use Room in your app, add the following dependencies to your app's `build.gradle` file:

Groovy Kotlin

```
dependencies {
    val room_version = "2.6.1"

    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:$room_version")
    // To use Kotlin Symbol Processing (KSP)
    ksp("androidx.room:room-compiler:$room_version")

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

    // optional - RxJava2 support for Room
    implementation("androidx.room:room-rxjava2:$room_version")

    // optional - RxJava3 support for Room
    implementation("androidx.room:room-rxjava3:$room_version")

    // optional - Guava support for Room, including Optional and ListenableFuture
    implementation("androidx.room:room-guava:$room_version")

    // optional - Test helpers
    testImplementation("androidx.room:room-testing:$room_version")

    // optional - Paging 3 Integration
    implementation("androidx.room:room-paging:$room_version")
}
```

Room API $\Rightarrow$ librairie Jetpack

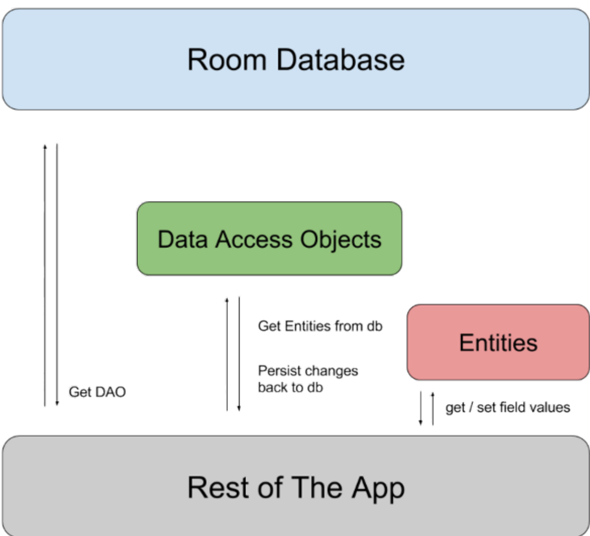
Trois concepts principaux sont utilisés :

There are three major components in Room:

- The database class that holds the database and serves as the main access point for the underlying connection to your app's persisted data.
- Data entities that represent tables in your app's database.
- Data access objects (DAOs) that provide methods that your app can use to query, update, insert, and delete data in the database.

The database class provides your app with instances of the DAOs associated with that database. In turn, the app can use the DAOs to retrieve data from the database as instances of the associated data entity objects. The app can also use the defined data entities to update rows from the corresponding tables, or to create new rows for insertion. Figure 1 illustrates the relationship between the different components of Room.

Room API ⇒ librairie Jetpack



Définition d'une entité

Data entity

The following code defines a `User` data entity. Each instance of `User` represents a row in a `user` table in the app's database.

Kotlin Java

```
@Entity
public class User {
    @PrimaryKey
    public int uid;

    @ColumnInfo(name = "first_name")
    public String firstName;

    @ColumnInfo(name = "last_name")
    public String lastName;
}
```

Accès aux entités de la BD

Data access object (DAO)

The following code defines a DAO called `UserDao`. `UserDao` provides the methods that the rest of the app uses to interact with data in the `user` table.

Kotlin Java

```
@Dao
public interface UserDao {
    @Query("SELECT * FROM user")
    List<User> getAll();

    @Query("SELECT * FROM user WHERE uid IN (:userIds)")
    List<User> loadAllByIds(int[] userIds);

    @Query("SELECT * FROM user WHERE first_name LIKE :first AND " +
           "last_name LIKE :last LIMIT 1")
    User findByName(String first, String last);

    @Insert
    void insertAll(User... users);

    @Delete
    void delete(User user);
}
```

Définition de la base de données

Database

The following code defines an `AppDatabase` class to hold the database. `AppDatabase` defines the database configuration and serves as the app's main access point to the persisted data. The database class must satisfy the following conditions:

- The class must be annotated with a `@Database` annotation that includes an `entities` array that lists all of the data entities associated with the database.
- The class must be an abstract class that extends `RoomDatabase`.
- For each DAO class that is associated with the database, the database class must define an abstract method that has zero arguments and returns an instance of the DAO class.

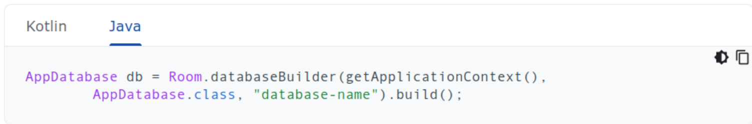
Kotlin Java

```
@Database(entities = {User.class}, version = 1)
public abstract class AppDatabase extends RoomDatabase {
    public abstract UserDao userDao();
}
```

★ **Note:** If your app runs in a single process, you should follow the singleton design pattern when instantiating an `AppDatabase` object. Each `RoomDatabase` instance is fairly expensive, and you rarely need access to multiple instances within a single process.

If your app runs in multiple processes, include `enableMultiInstanceInvalidation()` in your database builder invocation. That way, when you have an instance of `AppDatabase` in each process, you can invalidate the shared database file in one process, and this invalidation automatically propagates to the instances of `AppDatabase` within other processes.

Instanciation de la base de données



You can then use the abstract methods from the `AppDatabase` to get an instance of the DAO. In turn, you can use the methods from the DAO instance to interact with the database:

Kotlin Java

```
UserDao userDao = db.userDao();
List<User> users = userDao.getAll();
```

Documentation :

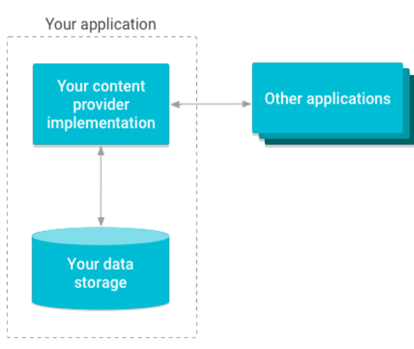
Plan

1. Solutions disponibles
2. App location
3. App files
4. Shared storage
5. Paires *Key-value*
6. Base de données locales avec l'API Room
7. **Content providers**

Content providers

Intérêts ⇒ faciliter l'accès aux données par d'autres app

- encapsuler les données
- assurer la sécurité des données
- faciliter la connexion aux données depuis l'extérieur



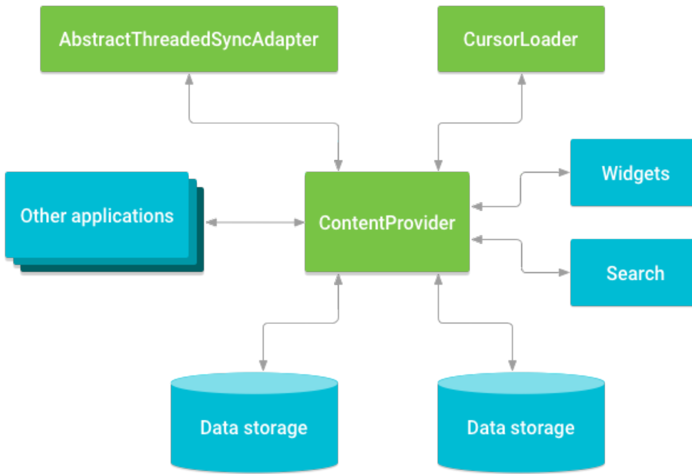
Cas d'utilisation

A content provider presents data to external applications as one or more tables that are similar to the tables found in a relational database. A row represents an instance of some type of data the provider collects, and each column in the row represents an individual piece of data collected for an instance.

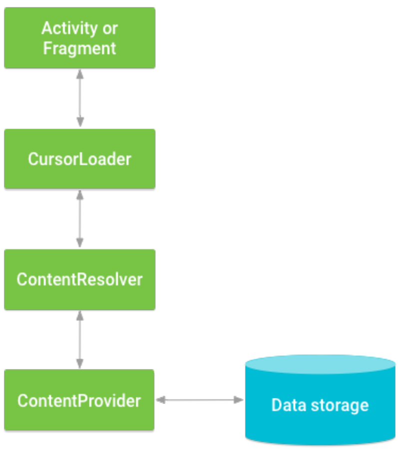
A content provider coordinates access to the data storage layer in your application for a number of different APIs and components. As illustrated in figure 1, these include the following:

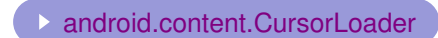
- Sharing access to your application data with other applications
- Sending data to a widget
- Returning custom search suggestions for your application through the search framework using `SearchRecentSuggestionsProvider`
- Synchronizing application data with your server using an implementation of `AbstractThreadedSyncAdapter`
- Loading data in your UI using a `CursorLoader`

Cas d'utilisation



Accès à un content provider





android.content.ContentProvider

Utilisation du *User Dictionary Provider*

To get a list of the words and their locales from the User Dictionary Provider, you call `ContentResolver.query()`. The `query()` method calls the `ContentProvider.query()` method defined by the User Dictionary Provider. The following lines of code show a `ContentResolver.query()` call:

Kotlin Java

```
// Queries the UserDictionary and returns results
cursor = getContentResolver().query(
    UserDictionary.Words.CONTENT_URI,  // The content URI of the words table
    projection,                        // The columns to return for each row
    selectionClause,                   // Selection criteria
    selectionArgs,                     // Selection criteria
    sortOrder);                        // The sort order for the returned rows
```

Content provider URI

In the previous lines of code, the full URI for the `Words` table is:

```
content://user_dictionary/words
```

- The `content://` string is the *scheme*, which is always present and identifies this as a content URI.
- The `user_dictionary` string is the provider's authority.
- The `words` string is the table's path.

Many providers let you access a single row in a table by appending an ID value to the end of the URI. For example, to retrieve a row whose `_ID` is `4` from the User Dictionary Provider, you can use this content URI:

Kotlin Java

```
Uri singleUri = ContentUris.withAppendedId(UserDictionary.Words.CONTENT_URI,4);
```

```
// Queries the UserDictionary and returns results
cursor = getContentResolver().query(
    UserDictionary.Words.CONTENT_URI,   // The content URI of the words table
    projection,                         // The columns to return for each row
    selectionClause,                    // Selection criteria
    selectionArgs,                      // Selection criteria
    sortOrder);                         // The sort order for the returned rows
```

Table 2: `query()` compared to SQL query.

`query()` argument	SELECT keyword/ parameter	Notes
`Uri`	`FROM` *`table_name`*	`Uri` maps to the table in the provider named *table_name*.
`projection`	*`col,col,col,...`*	`projection` is an array of columns that is included for each row retrieved.
`selection`	`WHERE` *`col`* `=` *`value`*	`selection` specifies the criteria for selecting rows.
`selectionArgs`	No exact equivalent. Selection arguments replace `?` placeholders in the selection clause.	
`sortOrder`	`ORDER BY` *`col,col,...`*	`sortOrder` specifies the order in which rows appear in the returned `Cursor`.

Construction de la requête

Construct the query

The next step in retrieving data from a provider is to construct a query. The following snippet defines some variables for accessing the User Dictionary Provider:

Kotlin Java

```
// A "projection" defines the columns that are returned for each row
String[] mProjection =
{
    UserDictionary.Words._ID,    // Contract class constant for the _ID column name
    UserDictionary.Words.WORD,   // Contract class constant for the word column name
    UserDictionary.Words.LOCALE  // Contract class constant for the locale column name
};

// Defines a string to contain the selection clause
String selectionClause = null;

// Initializes an array to contain selection arguments
String[] selectionArgs = {""};
```

```
/*
 * This defines a one-element String array to contain the selection argument.
 */
String[] selectionArgs = {""};

// Gets a word from the UI
searchString = searchWord.getText().toString();

// Remember to insert code here to check for invalid or malicious input

// If the word is the empty string, gets everything
if (TextUtils.isEmpty(searchString)) {
    // Setting the selection clause to null returns all words
    selectionClause = null;
    selectionArgs[0] = "";

} else {
    // Constructs a selection clause that matches the word that the user entered
    selectionClause = UserDictionary.Words.WORD + " = ?";

    // Moves the user's input string to the selection arguments
    selectionArgs[0] = searchString;

}

// Does a query against the table and returns a Cursor object
mCursor = getContentResolver().query(
    UserDictionary.Words.CONTENT_URI, // The content URI of the words table
    projection,                       // The columns to return for each row
    selectionClause,                  // Either null or the word the user entered
    selectionArgs,                    // Either empty or the string the user entered
    sortOrder);                       // The sort order for the returned rows

// Some providers return null if an error occurs, others throw an exception
if (null == mCursor) {
    /*
     * Insert code here to handle the error. Be sure not to use the cursor! You can
     * call android.util.Log.e() to log this error.
     *
     */
// If the Cursor is empty, the provider found no matches
} else if (mCursor.getCount() < 1) {

    /*
     * Insert code here to notify the user that the search is unsuccessful. This isn't
     * an error. You can offer the user the option to insert a new row, or re-type the
     * search term.
     */

} else {
    // Insert code here to do something with the results

}
```

Résumé

	Type de contenu	**Méthodes**	**Permissions requises**	**Accès depuis autre app**	**Suppression lors de la désinstallation**
Fichiers App	App seule	internal storage : getFilesDir() getCacheDir() external storage : getExternalFilesDir() getExternalCacheDir()	internal storage : Aucune external storage : Aucune API 19+	Aucun	Oui
Médias	images audio vidéos	API MediaStore	Depuis une autre app : API 30+ : READ_EXTERNAL_STORAGE API 29 : READ_EXTERNAL_STORAGE WRITE_EXTERNAL_STORAGE API 28- : Permissions pour tous les fichiers	Suivant les permissions	Non
Documents utilisateurs et autres	autre contenu partagé et téléchargements	*Storage Access Framework*	Aucune	via le *system file picker*	Non
Préférences	Paires clé-valeur	Jetpack Preferences SharedPreferences API	Aucune	Aucun	Oui
Base de données	données structurées	Room *persistence library*	Aucune	Aucun	Oui

Ce cours reprend largement la documentation de Google: ▸ App data and files