Gestione dati in Android

Introduzione ai diversi modi per gestire i dati e Preferences

Lezione 9.1

App data e files: Overview

- Android utilizza un file system simile ai file system basati su disco di altre piattaforme
- Il sistema offre diverse opzioni per salvare i dati dell'app:
  1. **App-specific storage (1)**
     - archivia i file destinati esclusivamente all'uso dell'app, in directory dedicate all'interno di un volume di archiviazione interno o in diverse directory dedicate all'interno dei volumi di archiviazione esterna. NB: Utilizzare le directory all'interno della memoria interna per salvare informazioni riservate a cui altre app non dovrebbero accedere
  2. **Shared storage (media, files): (2)**
     - archivia i file che la tua app intende condividere con altre app, inclusi file multimediali, documenti e altri file
  3. **App preferences: (3)**
     - archivia i dati primitivi, privati, in coppie chiave-valore (key-value data)
  4. **Database: (4)**
     - archivia i dati strutturati in un database privato utilizzando la libreria di persistenza Room (che forniscono un livello di astrazione sopra SQLite)

	Type of content	Access method	Permissions needed	Can other apps access?	Files removed on app uninstall?
App-specific files	Files meant for your app's use only	From internal storage, `getFilesDir()` or `getCacheDir()` From external storage, `getExternalFilesDir()` or `getExternalCacheDir()`	Never needed for internal storage Not needed for external storage when your app is used on devices that run Android 4.4 (API level 19) or higher	No	Yes

	Type of content	Access method	Permissions needed	Can other apps access?	Files removed on app uninstall?
Media	Shareable media files (images, audio files, videos)	`MediaStore` API	`READ_EXTERNAL_STORAGE` when accessing other apps' files on Android 11 (API level 30) or higher `READ_EXTERNAL_STORAGE` or `WRITE_EXTERNAL_STORAGE` when accessing other apps' files on Android 10 (API level 29) Permissions are required for **all** files on Android 9 (API level 28) or lower	Yes, though the other app needs the `READ_EXTERNAL_STORAGE` permission	No

	Type of content	Access method	Permissions needed	Can other apps access?	Files removed on app uninstall?
Documents and other files	Other types of shareable content, including downloaded files	Storage Access Framework	None	Yes, through the system file picker	No
App preferences	Key-value pairs	Jetpack Preferences library	None	No	Yes
Database	Structured data	Room persistence library	None	No	Yes

Scelta in base agli specifici bisogni

- **Quanto spazio richiedono i tuoi dati?**
  La memoria interna ha uno spazio limitato per App-specific data. Utilizzare altri tipi di archiviazione se è necessario salvare una notevole quantità di dati.
- **Quanto deve essere affidabile l'accesso ai dati?**
  Se la funzionalità di base della tua app richiede determinati dati, ad esempio all'avvio della tua app, posizionali nella directory di archiviazione interna o in un database. I file specifici dell'app (1) che vengono archiviati nella memoria esterna non sono sempre accessibili perché alcuni dispositivi consentono agli utenti di rimuovere un dispositivo fisico corrispondente alla memoria esterna
- **Che tipo di dati devi archiviare?**
  Se disponi di dati significativi solo per la tua app, utilizza l'archiviazione specifica per l'app (app-specific storage) (1). Per contenuti multimediali condivisibili, usa l'archiviazione condivisa (shared storage)(2) in modo che altre app possano accedere al contenuto. Per i dati strutturati, utilizzare le preferenze (3) (per dati valore-chiave - key-value data) o un database (4) (per i dati che contengono più di 2 colonne).
- **I dati devono essere privati alla tua app?**
  Quando si archiviano dati riservati, dati che non dovrebbero essere accessibili da altre app, utilizzare la memoria interna (1), le preferences (3) o un database (4). La memoria interna ha l'ulteriore vantaggio di nascondere i dati agli utenti.

Categorie di storage locations

- Due tipi
  - *internal storage*
    - Sempre disponibile anche se spesso più piccola di quella esterna (ma può non essere vero..)
  - *external storage*
    - Volumi rimovibili, come SD card
    - Sono rappresentati in Android usando il path: es. /sdcard
    - NOTA: La posizione esatta in cui è possibile salvare i file può variare a seconda del dispositivo. Per questo motivo, non vanno utilizzati percorsi di file codificati

preferExternal

- Le app vengono archiviate nella memoria interna per impostazione predefinita
- Se le dimensioni dell'APK sono molto grandi, tuttavia, si può indicare una preferenza nel file manifest della tua app per installare invece l'app su memoria esterna:

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

Permessi ed accesso a external storage

- Android definisce le seguenti autorizzazioni per l'accesso in lettura e scrittura alla memoria esterna:
  - READ_EXTERNAL_STORAGE
  - WRITE_EXTERNAL_STORAGE
  - MANAGE_EXTERNAL_STORAGE (Da Android 11)

Permessi ed accesso a external storage

- Nelle versioni precedenti di Android, le app dovevano dichiarare l'autorizzazione READ_EXTERNAL_STORAGE per accedere a qualsiasi file al di fuori delle directory specifiche dell'app sulla memoria esterna. Inoltre, le app dovevano dichiarare l'autorizzazione WRITE_EXTERNAL_STORAGE per scrivere su qualsiasi file al di fuori della directory specifica dell'app
- Le versioni più recenti di Android si basano più sullo **scopo/ambito** (scope – vedi prossima slide) di un file che sulla sua posizione per determinare la capacità di un'app di accedere e scrivere in un determinato file.
  - In particolare, se la tua app ha come target Android 11 (livello API 30) o superiore, l'autorizzazione WRITE_EXTERNAL_STORAGE non ha alcun effetto sull'accesso della tua app allo spazio di archiviazione
- Questo modello di archiviazione basato sullo scopo migliora la privacy dell'utente perché alle app viene concesso l'accesso solo alle aree del file system del dispositivo che effettivamente utilizzano
- Android 11 introduce l'autorizzazione **MANAGE_EXTERNAL_STORAGE**, che fornisce l'accesso in scrittura ai file al di fuori della directory specifica dell'app e di MediaStore.
  - Per ulteriori informazioni su questa autorizzazione e sul motivo per cui la maggior parte delle app non deve dichiararla per soddisfare i propri casi d'uso, vedere la guida su come gestire tutti i file su un dispositivo di archiviazione https://developer.android.com/training/data-storage/manage-all-files

Scoped storage

- Per offrire agli utenti un maggiore controllo sui propri file e per limitare il disordine, per impostazione predefinita (by default) le app destinate ad Android 10 (livello API 29) e versioni successive hanno accesso con ambito (scoped access) alla memoria esterna (scoped storage), **di default**
- Tali app hanno accesso solo alla directory specifica dell'app (app-specific directory) su memoria esterna, nonché a tipi specifici di media creati dall'app
- Aggiornamenti in Android 11: https://developer.android.com/about/versions/11/privacy/storage

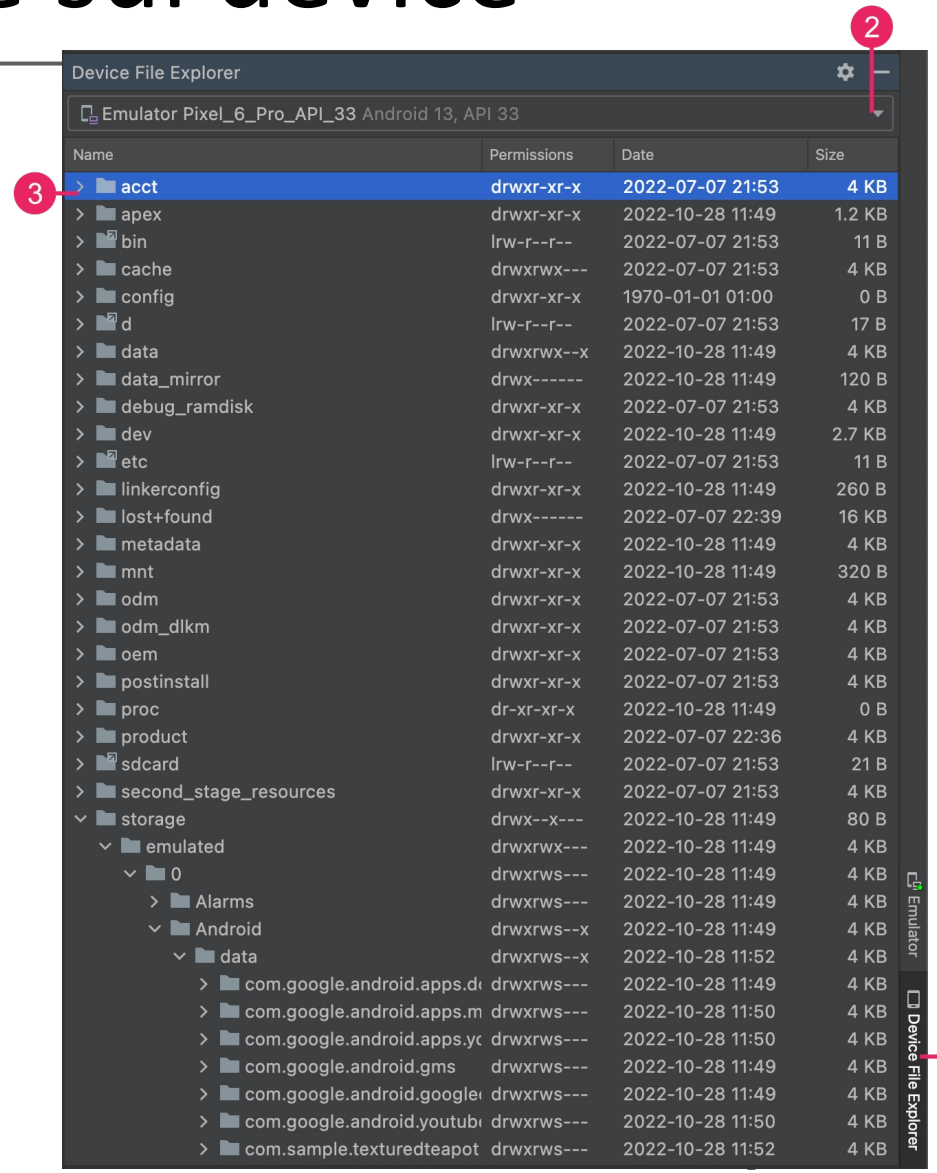
Scoped storage

- NOTA generale: meglio utilizzare l'archiviazione con ambito (scoped storage) a meno che l'app non abbia bisogno di accedere a un file archiviato all'esterno di una directory specifica dell'app e all'esterno di una directory a cui le API *MediaStore* possono accedere
- Se si archiviano file specifici per app su memoria esterna, è possibile semplificare l'adozione della memorizzazione con ambito inserendo questi file in una directory app-specific su memoria esterna
  - In questo modo, l'app mantiene l'accesso a questi file quando è abilitata la memorizzazione con ambito (scoped storage)

    Approfondimento: Preparing for Scoped Storage (Android Dev Summit '19)

Parentesi: Vedere i file sul device

- Per visualizzare i file archiviati su un dispositivo, puoi utilizzare Device File Explorer di Android Studio
- Info:
- https://developer.android.com/studio/debug/device-file-explorer



1. App-specific storage

- Il più delle volte l'app crea file che le altre app non devono accedere. Ci sono 2 possibilità in cui salvare questi file
  - **Internal storage directories**
    - Queste directory includono sia una posizione dedicata per l'archiviazione dei file persistenti, sia un'altra posizione per la memorizzazione dei dati della cache. Il sistema impedisce ad altre app di accedere a queste posizioni e, su Android 10 (livello API 29 e successive), queste posizioni sono crittografate. Queste caratteristiche rendono queste posizioni un buon posto per archiviare dati sensibili a cui solo la tua app può accedere
  - **External storage directories**
    - queste directory includono sia una posizione dedicata per l'archiviazione dei file persistenti, sia un'altra posizione per la memorizzazione dei dati della cache. Sebbene sia possibile che un'altra app acceda a queste directory se quell'app ha le autorizzazioni appropriate, i file memorizzati in queste directory sono pensati per essere utilizzati solo dalla tua app. Se si intende specificamente creare file a cui altre app dovrebbero poter accedere, l'app dovrebbe archiviare questi file nella parte di archiviazione condivisa (shared storage) della memoria esterna

App-specific storage

- Quando l'utente disinstalla l'app, i file salvati nella memoria specifica dell'app (app-specific storage) vengono rimossi
- A causa di questo comportamento, non dovresti usare questo spazio di archiviazione per salvare tutto ciò che l'utente si aspetta che persista indipendentemente dalla tua app
  - Ad esempio, se l'app consente agli utenti di acquisire foto, l'utente si aspetterebbe di poter accedere a tali foto anche dopo aver disinstallato l'app. Quindi dovresti invece usare l'archiviazione condivisa (shared storage (3)) per salvare quei tipi di file nella raccolta multimediale appropriata
- Nota: per proteggere ulteriormente app-specific file, si può utilizzare la Security library (https://developer.android.com/topic/security/data) per criptare i file. La chiave di crittografia è specifica per la tua app

Internal storage

- Android, per ogni app fornisce due directory accedibili solo dall'app stessa (per permesso di lettura e scrittura)
  - Una per contenere app persistent file (**filesDir**)
  - Una per contenere l'app cache (**cacheDir**)
- PRO
  - Le altre app non posso accedere a queste directory
- CONTRO
  - Solitamente lo spazio riservato è piccolo (conviene sempre controllare lo spazio prima di creare app-specific file per la memoria interna)

Persistent files: **filesDir**

- Per accedere e morizzare file persisenti si può usare *File* API

```
val file = File(context.filesDir, filename)
```

- Alternativamente openFileOutput() per ottenere un FileOutputStream e **scrivere** in *filesDir*

```
val filename = "myfile"
val fileContents = "Hello world!"
context.openFileOutput(filename, Context.MODE_PRIVATE).use {
        it.write(fileContents.toByteArray())
}
```

Esempio di codice per **scrivere** del testo in un file

Persistent files: **filesDir**

- Accedere ad un file attraverso uno stream

```
context.openFileInput(filename).bufferedReader().useLines {
lines ->
    lines.fold("") { some, text ->
        "$some\n$text"
    }
}
```

Gestione file filesDir

- Si può ottenere un array contenente i nomi di tutti i file nella directory *filesDir* chiamando ***fileList()***

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

- Si possono creare sotto directory

```
context.getDir(dirName, Context.MODE_PRIVATE)
```

File di cache

- Se occorre archiviare dati sensibili solo temporaneamente, si può utilizzare la directory *cache* dell'app che si trova nella memoria interna per salvare i dati
- Come per tutti gli archivi specifici dell'app (app-specific storage), i file archiviati in questa directory vengono rimossi quando l'utente disinstalla l'app, sebbene i file in questa directory possano essere rimossi prima
- Per sapere quanto spazio si ha a disposizione, metodo getCacheQuotaBytes()

Creare ed accedere ad un file di cache

- Per creare un file di cache

```
File.createTempFile(filename, null, context.cacheDir)
```

- L'app accede a un file in questa directory utilizzando la proprietà ***cacheDir*** di un oggetto context e l'API *File*:

```
val cacheFile = File(context.cacheDir, filename)
```

Attenzione: quando il dispositivo ha poco spazio di archiviazione interno, Android può eliminare questi file di cache per recuperare spazio. Quindi è sempre meglio controllare l'esistenza dei file di cache prima di leggerli

Cancellare la cache

- Per rimuovere un file dalla directory della cache all'interno della memoria interna, utilizzare uno dei seguenti metodi:

```
cacheFile.delete()
//cacheFile rappresenta il file da cancellare
```

Oppure

```
context.deleteFile(cacheFileName)
```

External Storage

- Se non c'è abbastanza spazio nelle memoria interna di app-specific files, si può usare quella esterna
  - Il sistema operativo fornisce directory dove è possibile gestire file di interesse all'utente
- Le due directory servono per
  - app's persistent files (***externalFilesDir***)
  - app's cached files (***externalCacheDir***)
- I file in queste directory vengono cancellati quando l'app viene disinstallata
- Attenzione: i file in queste directory non sono garantiti come accessibili, ad esempio quando una scheda SD rimovibile viene estratta dal dispositivo. Se la funzionalità della tua app dipende da questi file, dovresti invece archiviarli nella memoria interna

Versione 9 e 10

- Sui dispositivi che eseguono Android 9 (livello API 28) o inferiore, qualsiasi app può accedere ad app-specific files all'interno dell'archiviazione esterna, a condizione che l'altra app disponga delle autorizzazioni di archiviazione appropriate
- Per offrire agli utenti un maggiore controllo sui propri file e limitare «il disordine» dei file, per impostazione predefinita le app destinate ad Android 10 (livello API 29) e superiori hanno accesso mirato allo spazio di archiviazione esterno (external storage) o scoped storage
  - Quando l'archiviazione con ambito è abilitata (scoped storage), le app non possono accedere alle directory specifiche dell'app (app-specific directories) che appartengono ad altre app
  - Aggiornamenti versioni 11: https://developer.android.com/about/versions/11/privacy/storage

Controllare la presenza della memoria esterna

```
// Checks if a volume containing external storage is available
// for read and write.
fun isExternalStorageWritable(): Boolean {
    return Environment.getExternalStorageState() ==
Environment.MEDIA_MOUNTED
}

// Checks if a volume containing external storage is available to at
least read.
fun isExternalStorageReadable(): Boolean {
     return Environment.getExternalStorageState() in
        setOf(Environment.MEDIA_MOUNTED,
Environment.MEDIA_MOUNTED_READ_ONLY)
}
```

Selezionare una physical storage location

- Un device può avere diversi extenal storage
- Per accedere alle diverse locazioni, si può usare ContextCompat.getExternalFilesDirs()

```
val externalStorageVolumes: Array<out File> =
        ContextCompat.getExternalFilesDirs(applicationContext,
null)
val primaryExternalStorage = externalStorageVolumes[0]
```

//in posizione **0** c'è il primary external storage volume
//questo è il volume da utilizzare a meno che sia pieno o non disponibile

Accedere a file persistenti

- Per accedere ai app-specific files nell'archiviazione esterna, si usa getExternalFilesDir()

```
val appSpecificExternalDir = File(context.getExternalFilesDir(null), filename)
```

- Attenzione (da ricordare sempre!): per mantenere le prestazioni della tua app, evita di aprire e chiudere più volte lo stesso file

Gestione cache files

- Per creare e aggiungere un file specifico dell'app alla cache all'interno della memoria esterna, serve un riferimento a *externalCacheDir*

```
val externalCacheFile = File(context.externalCacheDir, filename)
```

- Per rimuove cache file

```
externalCacheFile.delete()
  //externalCacheFile è un oggetto File
```

Contenuti multimediali app-specific

- Se la tua app ha bisogno di file multimediali per funzionare che servono all'utente (ma solo all'interno della tua app), è meglio archiviarli in directory specifiche dell'app all'interno di una memoria esterna

```
fun getAppSpecificAlbumStorageDir(context: Context, albumName: String): File? {
    // Get the pictures directory that's inside the app-specific directory on
    // external storage.
    val file = File(context.getExternalFilesDir(
            Environment.DIRECTORY_PICTURES), albumName)
    if (!file?.mkdirs()) {
        Log.e(LOG_TAG, "Directory not created")
    }
    return file
}
```

Nota sui contenuti multimediali

- È importante utilizzare nomi di directory forniti da costanti API come DIRECTORY_PICTURES. Questi nomi di directory assicurano che i file siano trattati correttamente dal sistema
- Se nessuno dei nomi di sottodirectory predefiniti soddisfa i tuoi file, puoi invece passare *null* in getExternalFilesDir(). Ciò restituisce la directory specifica dell'app root nella memoria esterna

Richiedere più spazio

- Per sapere quando spazio libero il device può fornire alla tua app, chiamare *getAllocatableBytes*()
  - NOTA: questo spazio può essere maggiore di quello davvero disponibile, perché Android può aver identificato spazio che si può eventualmente liberare
- Con *allocateBytes()* si può richiedere più spazio
- Alternativamente, si può usare un Intent con l'azione ACTION_MANAGE_STORAGE che visualizza un prompt per l'utente, chiedendo di scegliere i file sul dispositivo da rimuovere in modo che l'app possa disporre dello spazio necessario. Se lo si desidera, questo prompt può mostrare la quantità di spazio libero disponibile sul dispositivo (`StorageStatsManager.getFreeBytes() / StorageStatsManager.getTotalBytes()`)

Richiedere più spazio con Intent

```
// App needs 10 MB within internal storage.
const val NUM_BYTES_NEEDED_FOR_MY_APP = 1024 * 1024 * 10L;

val storageManager = applicationContext.getSystemService<StorageManager>()!!
val appSpecificInternalDirUuid: UUID = storageManager.getUuidForPath(filesDir)
val availableBytes: Long =
        storageManager.getAllocatableBytes(appSpecificInternalDirUuid)
if (availableBytes >= NUM_BYTES_NEEDED_FOR_MY_APP) {
    storageManager.allocateBytes(
        appSpecificInternalDirUuid, NUM_BYTES_NEEDED_FOR_MY_APP)
} else {
    val storageIntent = Intent().apply {
        // To request that the user remove all app cache files instead, set
        // "action" to ACTION_CLEAR_APP_CACHE.
        action = ACTION_MANAGE_STORAGE
    }
}
```

Richiedere più spazio con Intent

```
// App needs 10 MB within internal storage.
const val NUM_BYTES_NEEDED_FOR_MY_APP = 1024 * 1024 * 10L;

val storageManager = a                                    orageManager>()!!
val appSpecificInternalD                                  idForPath(filesDir)
val availableBytes: Long
      storageManager.ge                                   irUuid)
if (availableBytes >= NU
   storageManager.alloc
      appSpecificInterna                                  MY_APP)
} else {
   val storageIntent = In
      // To request that t                                ead, set
      // "action" to ACTI
      action = ACTION_M
   }
}
```

Se non sapete a priori quanto spazio vi serve,
Potete provare a scrivere il file, e usare un catch di IOException se serve gestire l'eccezione

2. Shared storage

- Utilizza l'archiviazione condivisa (shared storage) per i dati utente che possono o devono essere accessibili ad altre app e salvati anche se l'utente disinstalla l'app
- Android fornisce API per l'archiviazione e l'accesso ai seguenti tipi di dati condivisibili:
  - **Contenuti multimediali (media content)**: il sistema fornisce directory pubbliche standard per questo tipo di file, quindi l'utente ha una posizione comune per tutte le sue foto, un'altra posizione comune per tutta la loro musica e file audio e così via. La tua app può accedere a questi contenuti utilizzando l'API *MediaStore* della piattaforma
  - **Documenti e altri file**: il sistema ha una directory speciale per contenere altri tipi di file, come documenti PDF e libri che utilizzano il formato EPUB. La tua app può accedere a questi file utilizzando *Storage Access Framework* della piattaforma
  - **Datasets** Su Android 11 (livello API 30) e versioni successive, il sistema memorizza nella cache set di dati di grandi dimensioni che potrebbero essere utilizzati da più app. Questi set di dati possono supportare casi d'uso come machine learning e la riproduzione multimediale. Le app possono accedere a questi set di dati condivisi utilizzando l'API BlobStoreManager

Shared storage: Media

- Per fornire un'esperienza utente arricchita, molte app consentono agli utenti di accedere e contribuire ai media disponibili su un volume di archiviazione esterno
- Il framework fornisce un indice ottimizzato nelle raccolte multimediali, chiamato *media store*, che consente di recuperare e aggiornare più facilmente questi file multimediali
- Anche dopo la disinstallazione dell'app, questi file rimangono sul dispositivo dell'utente
- Nota: se invece i contenuti multimediali sono di interesse solo all’interno della vostra app, allora usare app-specific directories con external storage

Media

- Per interagire con media store, si può utilizzare un oggetto **ContentResolver** recuperato dal contesto della tua app
- Il sistema esegue automaticamente la scansione di un volume di archiviazione esterno e aggiunge file multimediali alle seguenti raccolte ben definite:
  - **Images** comprese fotografie e screenshot, che sono archiviati nelle directory DCIM/ e Pictures/. Il sistema aggiunge questi file alla tabella MediaStore.Images
  - **Videos** archiviati nelle directory DCIM /, Movies / e Pictures /. Il sistema aggiunge questi file alla tabella MediaStore.Video
  - **Audio files** memorizzati nelle directory Alarms/, Audiobooks/, Music/, Notifications/, Podcasts/ e Ringtones/, nonché playlist audio che si trovano nelle directory Music/ o Movies/. Il sistema aggiunge questi file alla tabella MediaStore.Audio.
  - **Downloaded files** che sono archiviati nella directory Download/. Sui dispositivi che eseguono Android 10 (livello API 29) e versioni successive, questi file vengono archiviati nella tabella MediaStore.Downloads. Questa tabella non è disponibile su Android 9 (livello API 28) e inferiore

Media

- Il media store include anche una raccolta chiamata MediaStore.Files
- Il suo contenuto dipende dal fatto che l'app utilizzi l'archiviazione con ambito, disponibile nelle app che hanno come target Android 10 o versioni successive: Se l'archiviazione con ambito (scope) è abilitata, la raccolta mostra solo le foto, i video e i file audio creati dall'app.
  - La maggior parte degli sviluppatori non avrà bisogno di utilizzare MediaStore.Files per visualizzare i file multimediali da altre app, ma se hai un requisito specifico per farlo, puoi dichiarare l'autorizzazione READ_EXTERNAL_STORAGE (o MANAGE_EXTERNAL_STORAGE)
    - Si consiglia, tuttavia, di utilizzare le API MediaStore per aprire i file che l'app non ha creato
- Se l'archiviazione con ambito (scoped storage) non è disponibile (versioni vecchie di Android) o non viene utilizzata, la raccolta mostra tutti i tipi di file multimediali

Media

- Prima di eseguire operazioni sui file multimediali, bisogna assicurarsi che l'app abbia dichiarato le autorizzazioni necessarie per accedere a questi file
  - Tieni presente, tuttavia, che la tua app non deve dichiarare autorizzazioni di cui non necessita o non utilizza
- Il modello di autorizzazioni per l'accesso ai file multimediali nella tua app dipende dal fatto che la tua app utilizzi l'archiviazione con ambito (scoped storage), disponibile nelle app destinate ad Android 10 o versioni successive

Media

- Se la tua app usa l'archiviazione con ambito (scoped storage), dovrebbe richiedere le autorizzazioni relative all'archiviazione solo per i dispositivi che eseguono Android 9 (livello API 28) o inferiore
- Puoi applicare questa condizione aggiungendo l'attributo android: maxSdkVersion alla dichiarazione di autorizzazione nel file manifest della tua app:

```
<uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE"
        android:maxSdkVersion="28" />
```

Shared storage

- Se siete interessati ad avere più informazioni su questa parte, potete guardare nei seguenti link
- Media
  - https://developer.android.com/training/data-storage/shared/media
- Photo picker
  - https://developer.android.com/training/data-storage/shared/photopicker
- Documenti, PDF e ebook
  - https://developer.android.com/training/data-storage/shared/documents-files
- Dataset (da Android 11)
  - https://developer.android.com/training/data-storage/shared/datasets

3. Preferences: Save key-value data

- Se hai una raccolta relativamente piccola di valori-chiave (key-value) che desideri salvare, puoi utilizzare le API *SharedPreferences*
- Un oggetto *SharedPreferences* punta a un file contenente coppie chiave-valore e fornisce metodi semplici per leggerli e scriverli
- Ogni file *SharedPreferences* è gestito dal framework e può essere privato o condiviso

Novità!

- Android ha sviluppato **DataStore**
  - DataStore è una soluzione di archiviazione dati nuova e migliorata volta a sostituire SharedPreferences, fortemente basato su Kotlin e Flow
  - DataStore fornisce due diverse implementazioni:
    - Proto DataStore, che memorizza oggetti tipizzati (supportati da buffer di protocollo)
    - Preferences DataStore, che memorizza coppie chiave-valore. I dati vengono archiviati in modo asincrono, coerente e transazionale, superando alcuni degli svantaggi di SharedPreferences

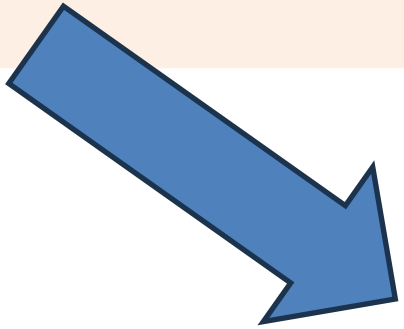
Novità!

- https://developer.android.com/training/data-storage/shared-preferences

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.



DataStore

Part of Android 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.

Regole da seguire

- Per utilizzare correttamente DataStore tieni sempre presenti le seguenti regole:
  - Non creare mai più di un'istanza di DataStore per un determinato file nello stesso processo. Ciò potrebbe interrompere tutte le funzionalità di DataStore. Se sono presenti più DataStore attivi per un determinato file nello stesso processo, DataStore lancerà IllegalStateException durante la lettura o l'aggiornamento dei dati
  - Il tipo generico di DataStore deve essere immutabile. La modifica di un tipo utilizzato in DataStore invalida qualsiasi garanzia fornita da DataStore e crea bug potenzialmente gravi e difficili da individuare. Si consiglia vivamente di utilizzare buffer di protocollo che forniscano garanzie di immutabilità, un'API semplice e una serializzazione efficiente
  - Non mescolare mai gli utilizzi di SingleProcessDataStore e MultiProcessDataStore per lo stesso file. Se intendi accedere al DataStore da più di un processo utilizza sempre MultiProcessDataStore

Setup: Preferences DataStore (Gradle file)

```
// 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")
    }
```

Store key-value pairs - Preferences DataStore

- L'implementazione Preferences DataStore utilizza le classi DataStore e Preferences per rendere persistenti semplici coppie chiave-valore sul disco

- Per creare un **Preferences DataStore**

```
// At the top level of your kotlin file:
val Context.dataStore: DataStore<Preferences> by
preferencesDataStore(name = "settings")
```

Leggere da un Preferences DataStore

```
val EXAMPLE_COUNTER = intPreferencesKey("example_counter")
val exampleCounterFlow: Flow<Int> = context.dataStore.data
   .map { preferences ->
     // No type safety.
     preferences[EXAMPLE_COUNTER] ?: 0
}
```

In coroutines, a *flow* is a type that can emit multiple values sequentially,

Scrivere un Preferences DataStore

```
suspend fun incrementCounter() {
  context.dataStore.edit { settings ->
    val currentCounterValue =
settings[EXAMPLE_COUNTER] ?: 0
    settings[EXAMPLE_COUNTER] = currentCounterValue +
1
  }
}
```

4. Database locale usando Room

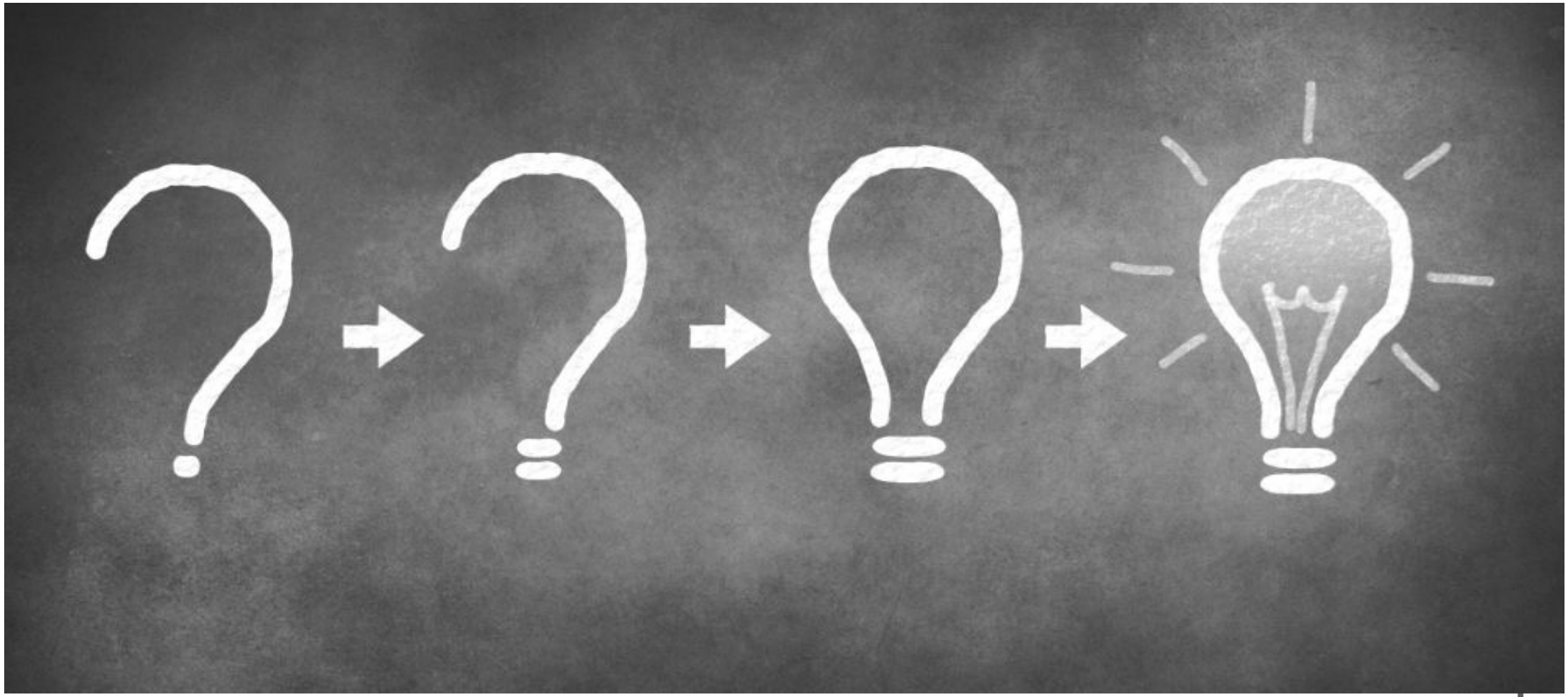
- *Room* offre uno strato di astrazione su SQLite per consentire un accesso fluido al database sfruttando la piena potenza di SQLite
- Le app che gestiscono quantità non banali di dati strutturati possono trarre grandi vantaggi dal memorizzare tali dati localmente. Il caso d'uso più comune è memorizzare nella cache parti di dati rilevanti
  - In questo modo, quando il dispositivo non è in grado di accedere alla rete, l'utente può comunque sfogliare quel contenuto mentre è offline. Eventuali modifiche al contenuto avviate dall'utente vengono quindi sincronizzate con il server dopo che il dispositivo è tornato online
- Poiché Room si occupa di queste gestioni per te, è consigliato fortemente di utilizzare Room anziché SQLite

4. Database locale usando Room

- *Room* offre uno strato di astrazi su te per consentire un accesso fluido al database sfrut lo la za SQLite
- Le app che gestiscono q rati possono trarre grandi vantaggi te. Il caso d'uso più comune è memori levanti
  - In questo modo, qu di accedere alla rete, l'utente può entre è offline. Eventuali modifiche al con uindi sincronizzate con il server dopo che il
- Poiché Room si oni per te, è consigliato fortemente di utilizzare Room anzi SQLite

Lo vedremo in dettaglio!

Domande?



References

- https://developer.android.com/training/data-storage
- https://developer.android.com/training/data-storage/app-specific
- https://developer.android.com/training/data-storage/shared
- https://developer.android.com/training/data-storage/shared-preferences
- https://developer.android.com/topic/libraries/architecture/datastore