Développement Android avec Kotlin

Cours - 06 - Stockage des données

Jordan Hiertz

Contact

hiertzjordan@gmail.com

jordan.hiertz@al-enterprise.com

Introduction au stockage des données en Android

Android gère les fichiers de manière similaire à d'autres systèmes d'exploitation.

Il existe différentes façons de stocker les données d'une application.

Choisir le bon type de stockage dépend des besoins de l'application (sécurisation, partage de données, persistance, etc.)

Stockage des données et des fichiers

Android offre plusieurs options pour enregistrer les données de votre application :

Stockage spécifique à l'application :

- Fichiers privés, stockés soit dans la mémoire interne (données sensibles), soit dans l'espace de stockage externe.

Stockage partagé :

- Fichiers à partager avec d'autres applications (ex. fichiers multimédias, documents).

Préférences (SharedPreferences) :

- Stockage de petites données privées sous forme de paires clé/valeur (ex. paramètres utilisateur).

Bases de données SQLite :

- Données structurées et relationnelles, stockées dans une base de données privée.

Stockage des données et des fichiers

	Type de contenu	Mode d'accès	Autorisations nécessaires	Accès autres applications	Suppression à la désinstallation

Fichiers spécifiques à l'application	Fichiers destinés à l'application	`getFilesDir()`
		`getCacheDir()`
		`getExternalFilesDir()`
		`getExternalCacheDir()`
			Aucunes	Non	Oui

Multimédia	Fichiers multimédias partageables	API `MediaStore`	Selon la version d'Android
			`READ_EXTERNAL_STORAGE`
			`WRITE_EXTERNAL_STORAGE`
				Oui	Non

Documents et autres fichiers	Autres fichiers partageables	Storage Access Framework	Aucunes	Oui	Non
Préférences de l'application	Paires clé/valeur	`SharedPreferences` / `DataStorePreferences`	Aucunes	Non	Oui
Base de données	Données structurées	Bibliothèque de persistance `Room`	Aucunes	Non	Oui

Comment choisir ?

De combien d'espace vos données ont-elles besoin ?

=> La mémoire de stockage interne est limité

Quel est le niveau de fiabilité nécessaire pour l'accès aux données ?

=> Stockage externe pas toujours accessible

Quel type de données devez-vous stocker ?

=> Clé/valeur, données structurées, contenu multimédia...

Les données doivent-elles être privées ?

=> Stockage interne empêche les applications et utilisateurs de voir les données

Emplacements de stockage

Stockage interne :

- **Caractéristiques** : Mémoire non amovible et généralement plus petite.
- **Fiabilité** : Toujours disponible, idéal pour stocker les données critiques de votre application.
- **Utilisation** : Enregistrer des données sensibles ou essentielles qui ne doivent pas être partagées avec d'autres applications.

Stockage externe :

- **Caractéristiques** : Mémoire amovible (par exemple, carte SD).
- **Fiabilité** : Moins fiable car elle peut être supprimée ou retirée par l'utilisateur.
- **Utilisation** : Stocker des données moins sensibles ou des fichiers multimédias qui peuvent être partagés entre plusieurs applications.

Par défaut, les applications sont stockées dans la mémoire de stockage interne.

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

Enregistrer dans l'espace de stockage de l'application

Accéder aux fichiers

- Le répertoire de l'application est accessible via **`filesDir`**.
- Utilisez l'API **`File`** pour effectuer des opérations sur les fichiers (lecture, écriture, suppression).

```kotlin
val file = File(context.filesDir, "example.txt")

// Vérifier si le fichier existe
file.exists()

// Vérifier si c'est un fichier ou un répertoire
file.isFile // true si c'est un fichier
file.isDirectory // true si c'est un répertoire

// Supprimer un fichier
file.delete()

// Lire le contenu d'un fichier
val content = file.readBytes()
```

Enregistrer dans l'espace de stockage de l'application

Utiliser un flux pour écrire dans un fichier

- **Différence avec l'utilisation directe de `File` :**
  - L'API **`openFileOutput`** offre une méthode simplifiée et sécurisée pour écrire dans des fichiers dans le répertoire interne de l'application.
  - Elle garantit que le fichier est ouvert avec les permissions correctes (par exemple, **privé par défaut** avec `MODE_PRIVATE`).

```
1 val filename = "myfile"
2 val fileContents = "SMB116!"
3
4 context.openFileOutput(filename, Context.MODE_PRIVATE).use {
5     it.write(fileContents.toByteArray())
6 }
```

Accéder à un fichier à l'aide d'un flux

Pourquoi utiliser un flux plutôt que manipuler le fichier directement ?

- **Optimisation mémoire** (lecture par petites portions, sans charger tout le fichier en mémoire).
- **Flexibilité** (lecture ligne par ligne ou byte par byte).
- **Sécurité** (fermeture automatique grâce à `use`, réduit les fuites de ressources).

```
1 val filename = "myfile"
2 val fileContents = "SMB116!"
3
4 context.openFileInput(filename).bufferedReader().useLines { lines ->
5     lines.fold("") { some, text ->
6         "$some\n$text"
7     }
8 }
```

Créer des fichiers dans le cache

- Les fichiers de cache sont stockés dans **`cacheDir`**, un espace dédié au cache de l'application.
- **Utilisation principale** : Stocker des données temporaires ou non essentielles.
- ⚠️ Ces fichiers sont automatiquement supprimés par Android lorsque l'espace est insuffisant.

```
1 // Créer un fichier temporaire dans le cache
2 File.createTempFile("tempFile", ".txt", context.cacheDir)
3
4
5 // Accéder à un fichier de cache
6 val cacheFile = File(context.cacheDir, "filename")
```

L'espace de stockage externe

L'espace de stockage externe est utilisé pour stocker des fichiers partageables ou volumineux.

- Cet espace peut être :
  - **Physique** : comme une carte SD amovible.
  - **Émulé** : une partie du stockage interne est simulée comme stockage externe (souvent accessible via le chemin `/sdcard`).

⚠️ Les fichiers de ces répertoires **ne sont pas toujours accessibles** (ex. : carte SD retirée ou espace émulé non monté).

```kotlin
// Vérifie si le stockage externe est accessible en lecture et écriture
fun isExternalStorageWritable(): Boolean {
    return Environment.getExternalStorageState() == Environment.MEDIA_MOUNTED
}

// Vérifie si le stockage externe est accessible en lecture seule
fun isExternalStorageReadable(): Boolean {
    return Environment.getExternalStorageState() in
        setOf(Environment.MEDIA_MOUNTED, Environment.MEDIA_MOUNTED_READ_ONLY)
}
```

Accéder aux fichiers du stockage externe

Pour accéder aux fichiers spécifiques à votre application depuis un espace de stockage externe, vous pouvez utiliser **`getExternalFilesDir()`** ou **`externalCacheDir()`**.

```
1 // Accéder au répertoire spécifique à l'application dans le stockage externe
2 val appSpecificExternalDir = File(context.getExternalFilesDir(null), filename)
3
4
5 // Accéder au répertoire cache spécifique à l'application dans le stockage externe
6 val externalCacheFile = File(context.externalCacheDir, filename)
```

Récapitulatif : Fichiers spécifiques à l'application

Stockage interne :

- Utilisé pour stocker des données sensibles et privées.
- Accédé via **`getFilesDir()`** et **`getCacheDir()`**.

Stockage externe :

- Accédé via **`getExternalFilesDir()`** et **`externalCacheDir()`**.
- Il peut être émulé (pas forcément lié à une carte SD physique).

Le stockage partagé

Permet de stocker des données accessibles à d'autres applications et enregistrées même si l'utilisateur désinstalle l'application.

Utilisations principales :

1. Partager des fichiers (photos, musiques, documents, etc.)
2. Fournir un accès centralisé à certains types de fichiers communs.

Types de données partageables

Contenu multimédia

- Emplacement commun pour les photos, vidéos, fichiers audio, etc.
- **Accès via :** API **`MediaStore`**.

Documents et autres fichiers

- Stockage pour les fichiers tels que **PDF**, **EPUB**, etc.
- **Accès via :** Framework **SAF (Storage Access Framework)**.

Ensembles de données volumineux (Android 11+)

- Cache partagé pour des données comme :
  - Modèles de **machine learning**.
  - Contenus multimédias. (non visibles par l'utilisateur)
- **Accès via :** API **BlobStoreManager**.

Types de données partageables

	Exemple de données	API
Contenu multimédia	Photos, vidéos	MediaStore
Documents	PDF, EPUB, TXT…	Storage Access Framework
Ensembles de données	Modèles ML	BlobStoreManager

MediaStore

Pour interagir avec l'abstraction du MediaStore, utilisez un objet **ContentResolver** récupéré depuis le contexte de votre application. Cela permet de gérer les requêtes et de parcourir les fichiers multimédias.

```
val collection = MediaStore.Images.Media.EXTERNAL_CONTENT_URI // Fournit l'URI pour accéder aux images

// Détermine les colonnes à récupérer
val projection = arrayOf(MediaStore.Images.Media._ID, MediaStore.Images.Media.DISPLAY_NAME)

// On exécute cette logique en dehors du thread principal pour que l'application reste réactive
context.contentResolver.query(collection, projection, null, null, null)?.use { cursor ->
    while (cursor.moveToNext()) {
        // Récupérer les données
        val id = cursor.getLong(cursor.getColumnIndexOrThrow(MediaStore.Images.Media._ID))
        val displayName = cursor.getString(cursor.getColumnIndexOrThrow(MediaStore.Images.Media.DISPLAY_NAME))

        // Construire un URI pour accéder à l'image
        val uri = ContentUris.withAppendedId(collection, id)
        println("Image trouvée : $displayName (URI: $uri)")
    }
}
```

Et les permissions dans tout ça ?

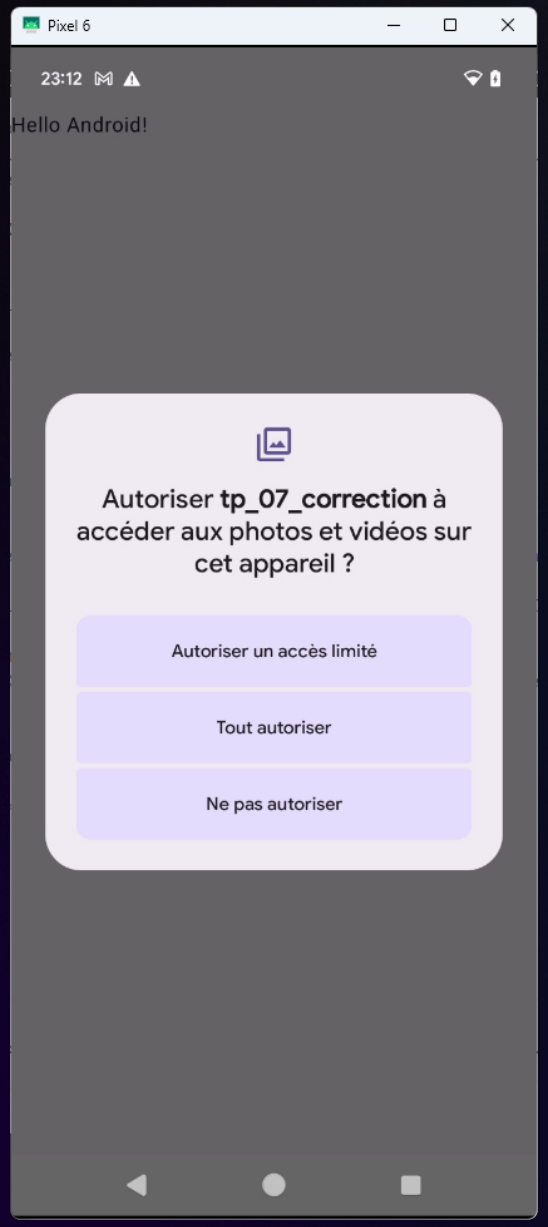
Lire des fichiers du **MédiaStore** nécessite des permissions à demander à l'utilisateur.

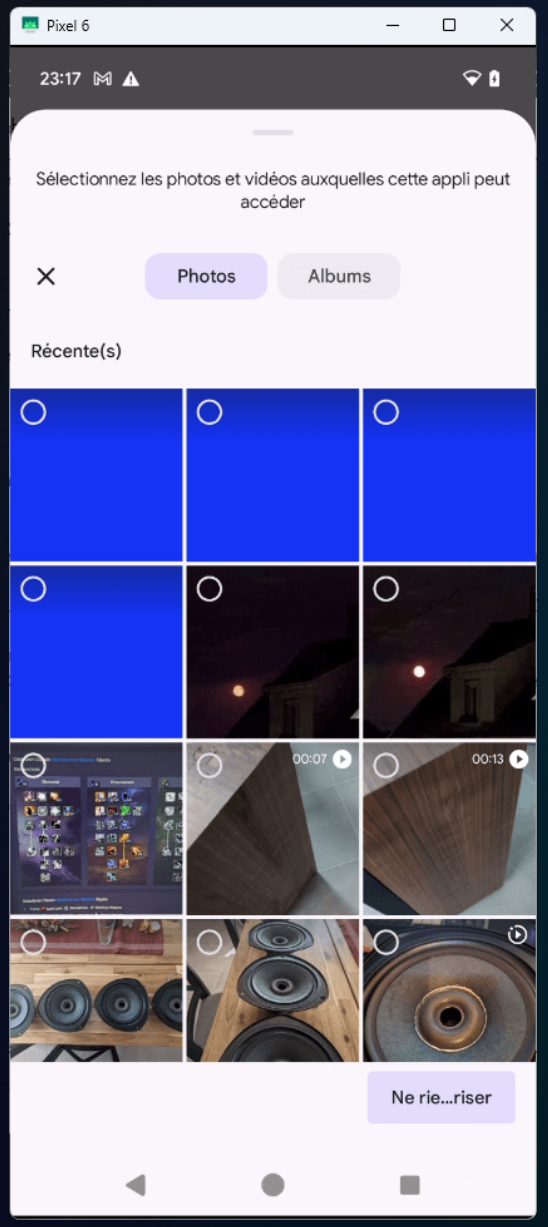
Ces permissions dépendent de la version d'Android et du type de contenu que vous souhaitez accéder…

```
<!-- Manifest.xml -->

<!-- Devices running Android 12L (API level 32) or lower  -->
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" android:maxSdkVersion="32" />

<!-- Devices running Android 13 (API level 33) or higher -->
<uses-permission android:name="android.permission.READ_MEDIA_IMAGES" />
<uses-permission android:name="android.permission.READ_MEDIA_VIDEO" />

<!-- To handle the reselection within the app on devices running Android 14
or higher if your app targets Android 14 (API level 34) or higher.  -->
<uses-permission android:name="android.permission.READ_MEDIA_VISUAL_USER_SELECTED" />
```

Et les permissions dans tout ça ?

```
1  private val requestPermissions = registerForActivityResult(ActivityResultContracts.RequestMultiplePermissions())
2  { results ->
3      // Check permissions granted
4  }
5
6
7  // Permission request logic
8  when {
9    Build.VERSION.SDK_INT >= Build.VERSION_CODES.UPSIDE_DOWN_CAKE -> {
10       requestPermissions.launch(arrayOf(READ_MEDIA_IMAGES, READ_MEDIA_VIDEO, READ_MEDIA_VISUAL_USER_SELECTED))
11   }
12   Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU -> {
13       requestPermissions.launch(arrayOf(READ_MEDIA_IMAGES, READ_MEDIA_VIDEO))
14   }
15   else -> {
16       requestPermissions.launch(arrayOf(READ_EXTERNAL_STORAGE))
17   }
18 }
```


Pixel 6
23:12
Hello Android!
Autoriser tp_07_correction à accéder aux photos et vidéos sur cet appareil ?
Autoriser un accès limité
Tout autoriser
Ne pas autoriser


Pixel 6
23:17
Sélectionnez les photos et vidéos auxquelles cette appli peut accéder
Photos
Albums
Récente(s)
00:07
00:13
Ne rie...riser

Et les permissions dans tout ça ?

Pour Android >= 10

➡️ **Aucune permission de stockage requise** pour accéder aux fichiers que votre application crée dans le MediaStore.

Exemple : Une application d'appareil photo peut accéder aux photos qu'elle prend, sans demander d'autorisation.

⚠️ **Attention** : Si l'utilisateur désinstalle et réinstalle l'application, le système considère que les fichiers appartiennent à l'ancienne version. Dans ce cas, il faut demander les **autorisations nécessaires** pour y accéder.

Sauvegarder un Média

Pour enregistrer un média dans le **MediaStore**, utilisez l'objet **ContentResolver**.

```
1  val resolver = applicationContext.contentResolver
2
3  val values = ContentValues().apply {
4      put(MediaStore.Images.Media.DISPLAY_NAME, "$imageName.png")
5      put(MediaStore.Images.Media.MIME_TYPE, "image/png")
6      put(MediaStore.Images.Media.RELATIVE_PATH, Environment.DIRECTORY_PICTURES)
7
8      // Seule votre application peut afficher le fichier jusqu'à qu'elle redéfinisse la valeur à 0
9      put(MediaStore.Images.Media.IS_PENDING, 1)
10 }
11
12 val uri = contentResolver.insert(MediaStore.Images.Media.EXTERNAL_CONTENT_URI, values)
13
14 uri?.let {
15     contentResolver.openOutputStream(it)?.use { outputStream ->
16         imageBitmap.compress(Bitmap.CompressFormat.PNG, 100, outputStream)
17     }
18
19     values.clear()
20     values.put(MediaStore.Images.Media.IS_PENDING, 0)
21     contentResolver.update(uri, values, null, null)
22 }
```

Supprimer un Média

La suppression d'un média dans le **MediaStore** se fait également via l'objet **ContentResolver**.

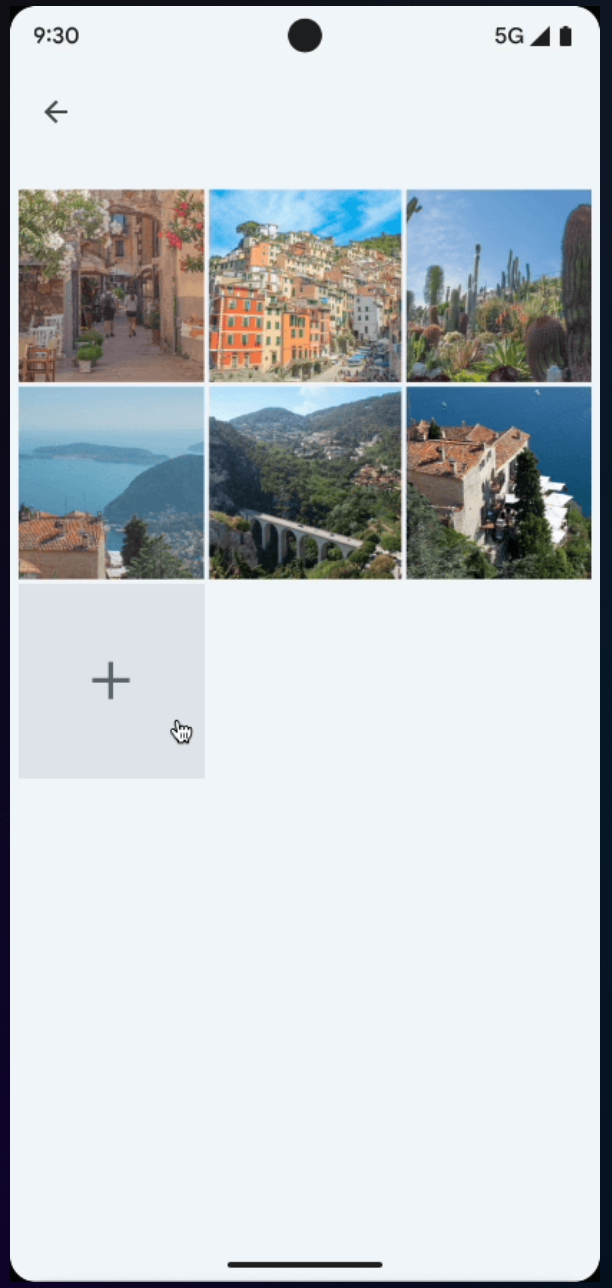
```
1  private fun deleteImageFromMediaStore(
2      contentResolver: ContentResolver,
3      imageName: String
4  ): Boolean {
5      val selection = "${MediaStore.Images.Media.DISPLAY_NAME} = ?"
6      val selectionArgs = arrayOf("$imageName.png")
7
8      val deletedRows = contentResolver.delete(
9          MediaStore.Images.Media.EXTERNAL_CONTENT_URI,
10         selection, // Clause WHERE
11         selectionArgs // Arguments
12     )
13
14     return deletedRows > 0
15 }
```

Sélecteur de Photos

Le **sélecteur de photos** est une méthode intégrée et sécurisée pour permettre à votre application d'accéder uniquement aux images et vidéos **sélectionnées par l'utilisateur**, au lieu de demander l'accès complet à la bibliothèque multimédia.

- **Disponible depuis Android 11 (API 30)**.
- **Fallback automatique** pour les versions antérieures d'Android.

```
1  val pickMultipleMedia = registerForActivityResult(PickMultipleVisualMedia(3)) { uris ->
2      if (uris.isNotEmpty()) {
3          Log.d("PhotoPicker", "Number of items selected: ${uris.size}")
4      } else {
5          Log.d("PhotoPicker", "No media selected")
6      }
7  }
8
9
10 pickMultipleMedia.launch(PickVisualMediaRequest(PickVisualMedia.ImageAndVideo))
```


9:30
5G

Les Documents et Autres Fichiers

Qu'est-ce que le Storage Access Framework (SAF) ?

- Le **Storage Access Framework (SAF)** permet aux utilisateurs de choisir des fichiers ou des répertoires auxquels votre application peut accéder.
- Ce mécanisme **ne nécessite aucune autorisation système** à partir d'Android 10 (API 29).
- **Android 9 (API 28) ou inférieur** nécessite la permission `READ_EXTERNAL_STORAGE`.

Étapes d'Utilisation du SAF

1 - L'application appelle un Intent avec une action liée au stockage (ex. : sélectionner un fichier).

2 - L'utilisateur interagit avec le sélecteur système, où il peut choisir un fichier ou un répertoire.

3 - L'application obtient un accès en lecture/écriture sur l'URI du fichier ou répertoire choisi par l'utilisateur.

⚠️ Attention

Sur **Android 9 ou moins**, il est nécessaire de demander la permission `READ_EXTERNAL_STORAGE` pour accéder au stockage externe via le SAF.

Créer un Fichier avec le Storage Access Framework (SAF)

```
1  private val createFileLauncher =
2  registerForActivityResult(ActivityResultContracts.CreateDocument("text/plain")) { uri ->
3      uri?.let {
4          // Handle URI
5      }
6  }
7
8  createFileLauncher.launch("example.txt")
9
10 contentResolver.openOutputStream(uri)?.use { outputStream ->
11     BufferedWriter(OutputStreamWriter(outputStream)).use { writer ->
12         writer.write("Hello SMB116 !")
13     }
14 }
```

Lire un Fichier avec le Storage Access Framework (SAF)

1 - Lancer un Intent pour sélectionner un fichier : Utilisez `ACTION_OPEN_DOCUMENT` pour ouvrir le sélecteur système et permettre à l'utilisateur de choisir un fichier.

2 - Obtenir un URI du fichier sélectionné : Le sélecteur renvoie un **URI** qui permet d'accéder au fichier.

3 - Lire le contenu du fichier : Ouvrez un **InputStream** à partir de l'URI et lisez les données.

```
1 private val openFileLauncher = registerForActivityResult(ActivityResultContracts.OpenDocument()) { uri ->
2     // Handle URI
3 }
4
5 openFileLauncher.launch(arrayOf("text/plain"))
6
7 contentResolver.openInputStream(uri)?.use { inputStream ->
8     inputStream.bufferedReader().use { it.readText() }
9 }
```

Accès Persistant aux Fichiers avec le SAF

Autorisation d'URI Temporaire

- Lorsqu'un fichier est ouvert via le SAF, le système accorde une autorisation temporaire d'accès (lecture/écriture) à cet URI.
- **Limitation** : Cette autorisation expire après le redémarrage de l'appareil.

Autorisations persistantes : Utilisez `ContentResolver.takePersistableUriPermission()` pour enregistrer l'autorisation persistante.

```
1 val contentResolver = applicationContext.contentResolver
2
3 val takeFlags: Int = Intent.FLAG_GRANT_READ_URI_PERMISSION or
4     Intent.FLAG_GRANT_WRITE_URI_PERMISSION
5
6 contentResolver.takePersistableUriPermission(uri, takeFlags)
```

Récapitulatif : Fichiers Partageables

Les fichiers partagés permettent aux applications d'échanger des données avec d'autres applications ou de les rendre accessibles à l'utilisateur même après la désinstallation de l'application.

- **Contenu multimédia** (photos, vidéos, musique) => Accès via **MediaStore**
- **Documents et autres fichiers** (PDF, EPUB, etc.) => Accès via Storage Access Framework
- **Ensembles de données** => Accès via **BlobStoreManager**.

Introduction aux SharedPreferences

Permet de stocker des **paires clé/valeur légères** pour des préférences utilisateur ou des données simples.

Quand les utiliser ?

- **Type de données** : Petites collections de données sous forme de paires clé/valeur.
- **Cas d'usage** : Sauvegarde de paramètres utilisateur, préférences, ou états simples de l'application.
- **Taille** : Limité à des données légères (pas adapté pour des fichiers volumineux ou des données complexes).

Fonctionnement des SharedPreferences

1 - Accéder aux SharedPreferences :

- Par **nom de fichier** ou via des préférences **par défaut**.

2 - Lire une valeur : Obtenir une valeur en fonction de sa clé.

3 - Écrire une valeur : Sauvegarder des données en modifiant l'objet via `apply()` ou `commit()`.

```
1 // 1️⃣ Accéder à un fichier nommé
2 val namedPrefs = context.getSharedPreferences("tp_07_preferences", Context.MODE_PRIVATE)
3
4 // 2️⃣ Utiliser les préférences par défaut (Depuis une Activité)
5 val defaultPreferences = getPreferences(Context.MODE_PRIVATE)
```

💡 **Note :** `Context.MODE_PRIVATE` signifie que les préférences sont privées à l'application.

Écrire dans les préférences partagées

Étapes pour écrire dans un fichier de préférences partagées :

1 - Créez un objet `SharedPreferences.Editor` en appelant `edit()` sur l'objet `SharedPreferences`.

2 - Ajoutez les paires clé/valeur avec des méthodes comme `putInt()`, `putString()`, etc.

3 - Enregistrez les modifications avec `apply()` (asynchrone) ou `commit()` (synchrone).

```
1 val sharedPref = activity?.getPreferences(Context.MODE_PRIVATE) ?: return
2
3 with(sharedPref.edit()) {
4     putString(getString(R.string.username), "$userName")
5     apply() // Enregistre les modifications
6 }
```

Lire à partir des préférences partagées

Pour récupérer les valeurs d'un fichier de préférences partagées, utilisez des méthodes comme `getInt()`, `getString()`, en fournissant la clé de la valeur recherchée, et éventuellement une valeur par défaut si la clé est absente.

```
1 // Récupérer les préférences partagées
2 val sharedPref = activity?.getSharedPreferences("tp_07_preferences", Context.MODE_PRIVATE) ?: return
3
4 // Lire une chaîne de caractères à partir des préférences
5 val username = sharedPref.getString(getString(R.string.username), null)
```

Migrer de SharedPreferences vers DataStore

Plus moderne et plus adapté aux besoins des applications modernes.

Thread-safe et non bloquant :

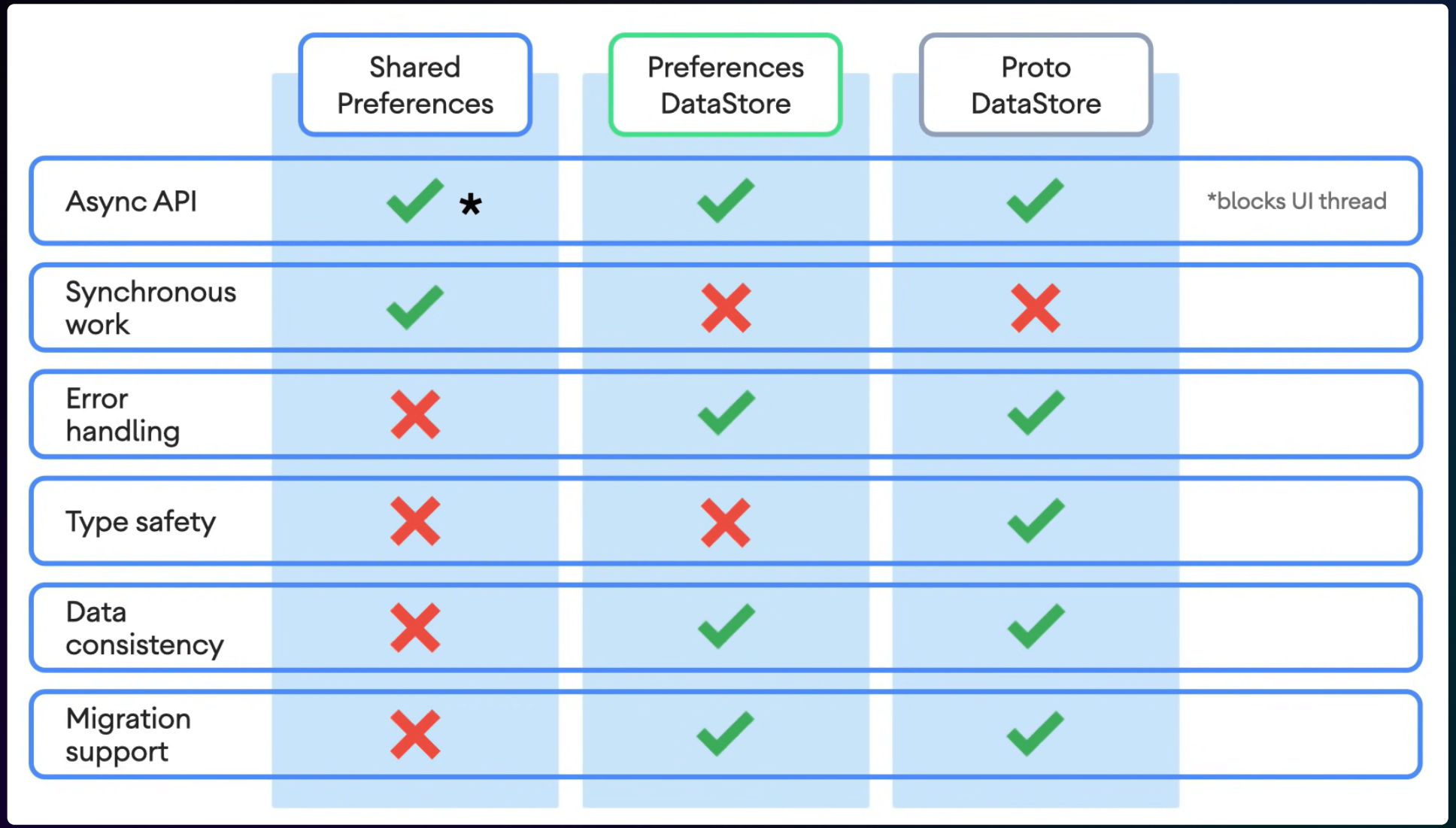
- DataStore fonctionne de manière asynchrone et ne bloque pas le thread principal, ce qui permet d'éviter des problèmes de performance et d'UI.

Exceptions difficiles à maîtriser avec SharedPreferences :

- DataStore gère mieux les exceptions grâce à son approche asynchrone et la gestion des erreurs est plus robuste.

Deux implémentations principales :

- **Preferences DataStore** => Idéale pour les données clé/valeur simples (comme SharedPreferences).
- **Proto DataStore** => Utilisé pour stocker des objets typés sérialisés.


Shared Preferences
Preferences DataStore
Proto DataStore
Async API
*
Synchronous work
Error handling
Type safety
Data consistency
Migration support
*blocks UI thread

Preferences DataStore

```
dependencies {
    implementation("androidx.datastore:datastore-preferences:1.1.1")
}
```

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

```
val USER_NAME = stringPreferencesKey("user_name")

val exampleUserNameFlow: Flow<String> = context.dataStore.data
  .map { preferences ->
    preferences[USER_NAME] ?: "Guest" // No type safety.
}
```

```
suspend fun updateUserName(userName: String) {
  context.dataStore.edit { settings ->
    settings[USER_NAME] = userName
  }
}
```

Conclusion

Le choix de la méthode de stockage dépend des besoins spécifiques de votre application :

- **Données légères** : Utilisez **SharedPreferences** ou **DataStore** pour des paires clé/valeur simples.
- **Contenu multimédia** : Préférez **MediaStore**.
- **Documents et fichiers** : Le **SAF** permet d'interagir avec le système de fichiers.
- **Données sensibles** : Stockez-les dans **le stockage interne** ou **DataStore** pour plus de sécurité.
- **Partage de données** : **MediaStore** et **SAF** sont adaptés pour le partage entre applications.

Chaque méthode a ses avantages, et il est crucial de choisir en fonction de la taille, de la sécurité, et de l'accessibilité des données.