

Stockage des données sur Android

Emmanuel BRUNO

Université de Toulon, LIS UMR CNRS 7020

emmanuel.bruno@univ-tln.fr

2026-10-14

Table des matières

1 Pourquoi stocker des données dans une application mobile ?	1
2 Stockage à court terme vs stockage persistant	2
3 Stockage local vs stockage cloud	2
4 Comparaison	2
5 Sélectionner la solution adaptée	3
6 Préférences partagées (SharedPreferences)	3
6.1 Exemple d'utilisation des SharedPreferences	3
6.2 JetPack DataStore	3
7 Exemple d'utilisation de Preferences DataStore	4
8 Stockage interne de fichiers (1/2)	4
8.1 Stockage interne de fichiers (2/2)	4
8.2 Exemple d'utilisation	5
9 Stockage externe	5
9.1 Media Store	5
9.2 Permissions et évolution	5
9.3 Accès moderne au stockage externe .	6
10 Room Database	6
10.1 Composants principaux de Room ..	7
10.2 Migrations de schéma	7
10.3 Exemple d'utilisation de Room	7
11 Stockage dans le cloud avec Firebase	8
11.1 Exemple d'utilisation de Firebase Cloud Firestore	8
12 Choix de la solution de stockage	9
13 Bonnes pratiques	9
14 Alternatives et compléments aux solutions de stockage cloud	9
14.1 API REST pour le stockage distant .	9
14.2 Communication WebSocket pour données en temps réel	9
14.3 GraphQL pour requêtes flexibles ...	9
14.4 Stratégies hybrides pour applications robustes	10
15 A vous de jouer !	10

1 Pourquoi stocker des données dans une application mobile ?

- **Persistance entre sessions** : Conserver les informations même après fermeture de l'application
- **Expérience utilisateur fluide** : Éviter de redemander les mêmes informations à chaque démarrage
- **Fonctionnement hors-ligne** : Permettre l'utilisation de l'application sans connexion internet
- **Personnalisation** : Adapter l'application aux préférences et comportements de l'utilisateur
- **Performances** : Mettre en cache des données pour améliorer la réactivité de l'application
- **Réduction du trafic réseau** : Limiter les requêtes vers des services distants

2 Stockage à court terme vs stockage persistant

Stockage à court terme	Stockage persistant
- Variables en mémoire - Bundle d'instance d'activité - Cache applicatif	- SharedPreferences - Fichiers internes/externes - Base de données (SQLite/Room) - Stockage dans le cloud

3 Stockage local vs stockage cloud

Stockage local	Stockage cloud
Avantages : - Disponibilité immédiate - Fonctionne hors-ligne - Pas de coûts supplémentaires - Accès rapide aux données - Contrôle total sur les données	Avantages : - Synchronisation multi-appareils - Sauvegarde automatique - Collaboration en temps réel - Espace de stockage évolutif - Persistance au-delà du cycle de vie de l'appareil
Inconvénients : - Limité à l'appareil - Risque de perte en cas de désinstallation/réinitialisation - Capacité de stockage limitée - Pas de partage natif entre appareils - Vulnérable aux problèmes matériels	Inconvénients : - Nécessite une connexion internet - Coûts potentiels (abonnements, consommation data) - Complexité de mise en œuvre - Latence possible - Risques de confidentialité et sécurité

4 Comparaison

Solution de stockage	Performance	Complexité	Volumétrie de données	Cas d'usage typiques
SharedPreferences	Élevées	Simple	Faible	Paramètres utilisateur, états d'UI
Fichiers internes	Bonnes	Moyennée	Moyen	Documents, fichiers temporaires, cache structuré
Fichiers externes	Bonnes	Moyennée+	Élevé	Médias, exports, partage de fichiers
SQLite/Room	Moyennée	Élevée	Élevé	Données structurées complexes, relations
Firebase	Variable	Élevée	Variable	Synchronisation multi-appareils,

Solu- tion de sto- ckage	Per- formances	Com- plexité	Vo- lume de don- nées	Cas d'usage ty- piques
				temps réel

* Dépend de la qualité de la connexion internet

5 Sélectionner la solution adaptée

Questions clés à se poser :

- Quelle est la nature des données à stocker ?
- Quelle quantité de données faut-il gérer ?
- Faut-il partager ces données avec d'autres applications ?
- Les données doivent-elles être synchronisées entre appareils ?
- Quelle importance ont la sécurité et la confidentialité ?

6 Préférences partagées (SharedPreferences)

- Présentation des SharedPreferences
 - Définition : stockage de paires clé-valeur primitives
 - Persistance des données à travers les sessions
- Cas d'usage appropriés
 - Paramètres utilisateur
 - Petites quantités de données structurées
- Limitations et contraintes
- Implémentation en Kotlin

6.1 Exemple d'utilisation des SharedPreferences

```
// Récupérer les SharedPreferences
val sharedPref =
    getSharedPreferences("fitness_app",
```

```
Context.MODE_PRIVATE)
```

```
// Écriture de données
```

```
val editor = sharedPref.edit()
    editor.putString("username",
"JohnDoe")
    editor.putBoolean("notifications",
true)
    editor.putInt("daily_steps_goal",
10000)
    editor.apply()
```

```
// Lecture de données
```

```
val username =
sharedPref.getString("username", "")
val notificationsEnabled = sharedPref.getBoolean("noti
false)
val stepsGoal = sharedPref.getInt("daily_steps_goal",
8000)
```

6.2 JetPack DataStore

- Successeur moderne de SharedPreferences
- Solution de stockage de données développée par Google
- Partie de la bibliothèque Android Jetpack
- API entièrement basée sur Kotlin Coroutines
- Deux implémentations
 - **Preferences DataStore** : stockage de paires clé-valeur (similaire à SharedPreferences)
 - **Proto DataStore** : stockage d'objets typés avec Protocol Buffers
- Avantages par rapport à SharedPreferences
 - Opérations asynchrones par défaut (pas de blocage du thread UI)
 - Accès sûr aux types avec des API fortement typées

- Gestion des transactions et cohérence des données
- Notifications de changements via Flow
- Migrations intégrées depuis SharedPreferences

7 Exemple d'utilisation de Preferences DataStore

```
// Exemple d'utilisation de
// Preferences DataStore
val FITNESS_PREFERENCES_NAME =
    "fitness_preferences"

// Création du DataStore
val Context.dataStore by
    preferencesDataStore(name =
        FITNESS_PREFERENCES_NAME)

// Définition des clés
val USERNAME
    = stringPreferencesKey("username")
val DAILY_GOAL = intPreferencesKey("daily_steps_goal")
val NOTIFICATIONS_ENABLED = booleanPreferencesKey("notifications_enabled")

// Lecture de données avec Flow
val userPreferencesFlow:
    Flow<UserPreferences>
    = dataStore.data
        .catch { exception ->
            if (exception is IOException) {
                emit(emptyPreferences())
            } else {
                throw exception
            }
        }
        .map { preferences ->
            UserPreferences(
                username =
                    preferences[USERNAME] ?: "",
                dailyStepsGoal =
                    preferences[DAILY_GOAL] ?: 10000,
                notificationsEnabled = preferences[NOTIFICATIONS_ENABLED] ?>
```

```
true
    )
}

// Écriture de données
suspend fun updateDailyGoal(newGoal:
    Int) {
    datastore.edit { preferences ->
        preferences[DAILY_GOAL] =
            newGoal
    }
}
```

8 Stockage interne de fichiers (1/2)

• Définition

- Espace de stockage privé à l'application
- Accès limité à l'application propriétaire
- Localisation: `/data/data/[package_name]/files/`
- Supprimé automatiquement lors de la désinstallation

• Avantages

- Pas de permissions requises
- Sécurité garantie (sandbox)
- Accès rapide
- Idéal pour données sensibles

8.1 Stockage interne de fichiers (2/2)

• Limitations

- Non visible via gestionnaire de fichiers
- Non partageable facilement
- Limité à la durée de vie de l'app

• Cas d'usage

- Fichiers texte (journaux, logs)
- Données structurées (JSON, XML)
- Cache persistant

- Configurations applicatives

8.2 Exemple d'utilisation

```
// Écriture dans un fichier
fun saveNote(content: String) {
    try {
        // Mode privé: accessible
        // uniquement par cette app
        openFileOutput("note.txt",
            Context.MODE_PRIVATE).use { stream ->
            stream.write(content.toByteArray())
        }
        Toast.makeText(this, "Note
sauvegardée",
            Toast.LENGTH_SHORT).show()
    } catch (e: Exception) {
        Toast.makeText(this, "Erreur:
${e.message}",
            Toast.LENGTH_SHORT).show()
    }
}

// Lecture d'un fichier
fun loadNote(): String {
    return try {
        openFileInput("note.txt").use
{ stream ->
            String(stream.readBytes())
        }
    } catch (e: FileNotFoundException)
{
        "Aucune note sauvegardée"
    } catch (e: Exception) {
        "Erreur: ${e.message}"
    }
}

// Utilisation dans une activité
binding.btnSave.setOnClickListener {
    val noteContent =
binding.editNote.text.toString()
    saveNote(noteContent)
}
```

```
binding.btnLoad.setOnClickListener {
    binding.textNote.text = loadNote()
}
```

9 Stockage externe

• Stockage public

- Accessible à toutes les applications
- Visible à l'utilisateur dans l'explorateur de fichiers
- Requiert des permissions spécifiques
- Conservation après désinstallation de l'app

• Stockage spécifique à l'app

- Réserve à l'application (sauf root)
- Supprimé à la désinstallation
- Ne nécessite pas de permissions spéciales
- Stockage dans `/Android/data/[package_name]/`

9.1 Media Store

- API dédiée aux médias (images, vidéos, audio)
- Accès standardisé via ContentProvider
- Recherche et filtrage avancés
- Indexation automatique

9.2 Permissions et évolution

Version Android	An-	Approche d'accès au stockage
-----------------	-----	------------------------------

- | | | |
|------------------|--|--|
| < Android 10 (Q) | | <ul style="list-style-type: none"> • Permissions larges: <code>READ_EXTERNAL_STORAGE</code> et <code>WRITE_EXTERNAL_STORAGE</code>- Accès complet au stockage externe partagé |
| Android 10 (Q) | | <ul style="list-style-type: none"> • Introduction du Scoped Storage- Possibilité d'opt-out temporaire via <code>requestLegacyExternalStorage</code> |
| Android 11+ | | <ul style="list-style-type: none"> • Scoped Storage obligatoire- Accès limité aux |

Version Android	An-	Approche d'accès au stockage
		répertoires spécifiques via MediaStore- Permission <code>MANAGE_EXTERNAL_STORAGE</code> pour cas exceptionnels

9.3 Accès moderne au stockage externe

```
// Stockage spécifique à l'application (ne nécessite pas de permissions)
fun saveTrackingDataToAppStorage(context: Context, filename: String, data: String) {
    // Obtention du répertoire "Documents" spécifique à l'application
    val appDocumentsDir = context.getExternalFilesDir(null)
    val file = File(appDocumentsDir, filename)

    try {
        FileOutputStream(file).use { output ->
            output.write(data.toByteArray())
        }
        Log.d("StorageExample", "Fichier sauvegardé: ${file.absolutePath}")
    } catch (e: Exception) {
        Log.e("StorageExample", "Erreur de sauvegarde", e)
    }
}

// Accès au MediaStore pour sauvegarder une image dans Photos
@RequiresApi(Build.VERSION_CODES.Q)
fun saveImageToMediaStore(context: Context, bitmap: Bitmap, filename:
```

```
String) {
    val contentValues =
        ContentValues().apply {
            put(MediaStore.Images.Media.DISPLAY_NAME, filename)
            put(MediaStore.Images.Media.MIME_TYPE, "image/jpeg")
            put(MediaStore.Images.Media.RELATIVE_PATH, Environment.DIRECTORY_PICTURES + "/" + FitnessApp())
            put(MediaStore.Images.Media.IS_PENDING, 1)
        }

    val resolver = context.contentResolver
    val imageUri = resolver.insert(MediaStore.Images.Media.EXTERNAL_CONTENT_URI, contentValues)

    imageUri?.let { uri ->
        resolver.openOutputStream(uri)?.use { output ->
            bitmap.compress(Bitmap.CompressFormat.JPEG, 90, output)
        }

        contentValues.clear()
        contentValues.put(MediaStore.Images.Media.IS_PENDING, 0)
        resolver.update(uri, contentValues, null, null)

        Toast.makeText(context, "Image sauvegardée dans la galerie", Toast.LENGTH_SHORT).show()
    }
}
```

10 Room Database

- **Définition**
 - Bibliothèque de persistance fournie par Android Jetpack

- Abstraction de SQLite facilitant l'accès aux bases de données
- Solution recommandée par Google pour le stockage structuré
- **Avantages principaux**
 - Vérification des requêtes SQL à la compilation
 - Réduction du code boilerplate et des erreurs d'exécution
 - Intégration native avec LiveData, Flow et Coroutines
 - Mapping objet-relationnel (ORM) moderne et optimisé
 - Migrations de schéma simplifiées
- **Architecture des composants**
 - Annotations pour générer le code d'accès à la base de données
 - Séparation claire des responsabilités (entités, DAO, DB)
 - Support des relations entre tables (One-to-One, One-to-Many, Many-to-Many)

10.1 Composants principaux de Room

Composant	Description	Annotation
Entités	Classes Kotlin représentant les tables Définition des colonnes, index et contraintes	@Entity
DAOs	Interfaces définissant les méthodes d'accès Opérations CRUD et requêtes SQL personnalisées	@Dao

Composant	Description	Annotation
Database	Classe abstraite qui connecte les DAOs Gestion de la création et migration de la base	@Database

10.2 Migrations de schéma

- Gestion du versionnement de la base de données
- Mise à jour des schémas sans perte de données utilisateur
- Stratégies de test pour valider les migrations

10.3 Exemple d'utilisation de Room

```
// Définir une entité
@Entity(tableName = "workouts")
data class Workout(
    @PrimaryKey(autoGenerate = true)
    val id: Int = 0,
    val exerciseName: String,
    val sets: Int,
    val repetitions: Int,
    val weight: Float,
    val date: Long,
    val duration: Int,
    val caloriesBurned: Int
)

// Définir un DAO
@Dao
interface WorkoutDao {
    @Insert
    fun insertWorkout(workout: Workout): Long

    @Update
```

```

        fun updateWorkout(workout:
Workout)

        @Delete
        fun deleteWorkout(workout:
Workout)

        @Query("SELECT * FROM workouts
ORDER BY date DESC")
        fun getAllWorkouts():
LiveData<List<Workout>>
    }

    // Définir la base de données
    @Database(entities =
[Workout::class], version = 1)
    abstract class WorkoutDatabase :
RoomDatabase() {
        abstract fun workoutDao():
WorkoutDao
    }

```

11 Stockage dans le cloud avec Firebase

- Présentation de Firebase
 - Services offerts par Firebase
 - Avantages d'un backend as a service (BaaS)
- Firebase Realtime Database
 - Structure de données en temps réel
 - Synchronisation automatique
- Firebase Cloud Firestore
 - Base de données NoSQL orientée documents
 - Modèle de requêtes avancées
- Firebase Authentication
 - Gestion des utilisateurs
 - Sécurisation des données

11.1 Exemple d'utilisation de Firebase Cloud Firestore

```

// Configurer Firebase
// (Configuration préalable dans
le fichier build.gradle et google-
services.json)

// Sauvegarder un workout dans
Firestore
fun saveWorkoutToCloud(workout:
Workout) {
    val db =
FirebaseFirestore.getInstance()
    val userId = FirebaseAuth.getInstance().currentUser?.

    if (userId != null) {
        val workoutMap = hashMapOf(
            "exerciseName" to
workout.exerciseName,
            "sets" to workout.sets,
            "repetitions" to
workout.repetitions,
            "weight" to workout.weight,
            "date" to workout.date,
            "duration" to
workout.duration,
            "caloriesBurned" to
workout.caloriesBurned
        )

        db.collection("users").document(userId)
            .collection("workouts").add(workoutMap)
            .addOnSuccessListener
{ documentReference ->
                Log.d("Firebase",
"DocumentSnapshot added with ID:
${documentReference.id}")
            }
            .addOnFailureListener { e
                ->
                Log.w("Firebase",
"Error adding document", e)
            }
    }
}

```

```
}  
}
```

12 Choix de la solution de stockage

- Critères de sélection
 - Volume et structure des données
 - Performances requises
 - Besoins de synchronisation
 - Contraintes de confidentialité
- Tableau comparatif des solutions
 - Cas d'usage
 - Avantages et inconvénients
- Stratégies de stockage hybride
 - Combiner plusieurs solutions
 - Cache local + synchronisation cloud

13 Bonnes pratiques

- Sécurisation des données
 - Chiffrement des données sensibles
 - Protection contre les accès non autorisés
- Performances et optimisation
 - Opérations asynchrones
 - Mise en cache intelligente
- Gestion des versions
 - Migration de schéma de base de données
 - Rétrocompatibilité

14 Alternatives et compléments aux solutions de stockage cloud

14.1 API REST pour le stockage distant

- **Principes fondamentaux**
 - Architecture client-serveur standardisée
 - Communication via requêtes HTTP(S) (GET, POST, PUT, DELETE)
 - Formats d'échange : principalement JSON, parfois XML

- **Intégration avec Android**

- Bibliothèques populaires : Retrofit, OkHttp, Volley
- Facilité d'intégration avec Kotlin Coroutines
- Gestion du cache et des requêtes conditionnelles

- **Applications**

- Synchronisation des données avec un serveur personnel
- Intégration avec des services tiers
- Accès à des catalogues de données

14.2 Communication WebSocket pour données en temps réel

- **Avantages par rapport au REST**

- Communication bidirectionnelle permanente
- Réduction de la latence pour les mises à jour fréquentes
- Idéal pour les fonctionnalités collaboratives et temps réel

- **Cas d'usage dans les applications fitness**

- Suivi d'entraînement en direct entre coach et client
- Compétitions et défis en temps réel entre utilisateurs
- Notifications instantanées (nouveaux records, rappels)

14.3 GraphQL pour requêtes flexibles

- **Concept fondamental**

- Langage de requête pour API développé par Facebook
- Alternative à REST avec requêtes précises et structurées
- Client qui spécifie exactement les données dont il a besoin

- **Avantages par rapport à REST**

- Diminution du sur-chargement et sous-chargement de données

- Réduction du nombre de requêtes réseau (requêtes combinées)
- Introspection automatique (documentation auto-générée)
- Typage fort des données
- **Intégration avec Android**
 - Bibliothèques clientes : Apollo GraphQL, graphql-kotlin
 - Génération automatique de classes Kotlin à partir du schéma
 - Support pour les fragments et les opérations complexes

14.4 Stratégies hybrides pour applications robustes

- **Approche offline-first**
 - Stockage local principal (Room) + synchronisation cloud
 - Gestion des conflits et réconciliation des données
 - Expérience utilisateur fluide même sans connexion
- **Architecture évolutive**
 - Séparation des préoccupations (repository pattern)
 - Sources de données interchangeables et combinables
 - Migration facilitée entre solutions de stockage

15 A vous de jouer !

- Apprenez en détails et mettez en Datastore et room
 - <https://developer.android.com/courses/android-basics-compose/unit-6?hl=fr>
- En option, explorez ce que propose Firebase pour stocker les données
 - <https://firebase.google.com/docs/guides>