**DAA - Développement d'applications Android**

# Persistance des données

*16 November 2025*

## Table des matières

# 1 Introduction

Android offre plusieurs solutions pour persister les données d'une application :

- **Fichiers** : stockage interne/externe, privé ou partagé
- **Préférences** : paires clé-valeur (SharedPreferences ou DataStore)
- **Bases de données** : SQLite via Room (recommandé)

> ⚠ **Warning**
>
> Le choix de la solution dépend du type et de la quantité de données à stocker.

# 2 Stockage de fichiers

## 2.1 Stockage interne privé

Accessible uniquement par l'application, automatiquement chiffré (API 29+), supprimé à la désinstallation.

```
// Répertoire principal
val filesDir = filesDir

// Cache (peut être supprimé par le système)
val cacheDir = cacheDir
```

## 2.2 Stockage externe privé

Permet d'utiliser une carte SD, émulé si non disponible. Vérifier la disponibilité :

```
fun isExternalStorageWritable() =
    Environment.getExternalStorageState() == Environment.MEDIA_MOUNTED

val externalRoot = getExternalFilesDir(null)
```

> ⚠ **Warning**
>
> Toujours utiliser les API pour obtenir les chemins, jamais de chemins en dur.

## 2.3 Fichiers média partagés

Pour images, vidéos, audio accessibles par d'autres apps. Utilisent le MediaStore et ne sont pas supprimés à la désinstallation.

# 3 Préférences

## 3.1 SharedPreferences

Stockage simple de paires clé-valeur dans un fichier XML.

```
// Obtenir les préférences
val prefs = getSharedPreferences("nom_fichier", Context.MODE_PRIVATE)

// Écriture
prefs.edit {
    putString("key", "value")
    putInt("count", 42)
}

// Lecture
val value = prefs.getString("key", "default")
```

> ⚠ **Warning**
>
> Google recommande de migrer vers Jetpack DataStore pour les nouveaux projets.

# 4 Bases de données avec Room

## 4.1 Architecture

Room est un ORM au-dessus de SQLite offrant :

- Vérification des requêtes à la compilation
- Génération de code via KSP
- Gestion des migrations
- Support des LiveData/Flow

**Entity ↔ DAO ↔ Database ↔ Repository ↔ ViewModel ↔ UI**

## 4.2 Composants principaux

**Entity** : data class représentant une table

```
@Entity
data class Person(
    @PrimaryKey(autoGenerate = true) var id: Long?,
    var name: String,
    var birthday: Calendar
)
```

**DAO** : interface définissant les requêtes

```
@Dao
interface PersonDao {
    @Query("SELECT * FROM Person")
    fun getAll(): LiveData<List<Person>>

    @Insert
    fun insert(person: Person): Long

    @Delete
    fun delete(person: Person)
}
```

**Database** : point d'entrée de la base

```
@Database(entities = [Person::class], version = 1)
@TypeConverters(CalendarConverter::class)
abstract class MyDatabase : RoomDatabase() {
    abstract fun personDao(): PersonDao

    companion object {
        @Volatile
        private var INSTANCE: MyDatabase? = null

        fun getDatabase(context: Context): MyDatabase {
            return INSTANCE ?: synchronized(this) {
                Room.databaseBuilder(
                    context.applicationContext,
                    MyDatabase::class.java,
                    "database.db"
                ).build().also { INSTANCE = it }
            }
        }
    }
}
```

## 4.3 Type Converters

Pour stocker des types non supportés nativement :

```
class CalendarConverter {
    @TypeConverter
    fun toCalendar(dateLong: Long) =
        Calendar.getInstance().apply { time = Date(dateLong) }

    @TypeConverter
```

```
    fun fromCalendar(date: Calendar) = date.time.time
}
```

## 4.4 Repository

Point d'entrée unique pour l'accès aux données :

```
class Repository(private val dao: PersonDao) {
    val allPersons = dao.getAll()

    fun insert(person: Person) {
        thread { dao.insert(person) }
    }
}
```

> ⚠ **Warning**
>
> Les opérations d'écriture doivent être asynchrones (thread ou coroutines).

## 4.5 Intégration avec ViewModel

```
class MyViewModel(repository: Repository) : ViewModel() {
    val persons = repository.allPersons

    fun addPerson(name: String) {
        repository.insert(Person(null, name, Calendar.getInstance()))
    }
}

// Dans l'Activity
private val viewModel: MyViewModel by viewModels {
    MyViewModelFactory((application as MyApp).repository)
}
```

## 4.6 Relations

### One-to-One / One-to-Many

```
data class PersonWithPhones(
    @Embedded val person: Person,
    @Relation(
        parentColumn = "personId",
        entityColumn = "ownerId"
    )
    val phones: List<Phone>
)
```

### Many-to-Many

```
@Entity(primaryKeys = ["playlistId", "songId"])
data class PlaylistSongCrossRef(
    val playlistId: Long,
    val songId: Long
)
```

## 4.7 Migrations

```
val MIGRATION_1_2 = object : Migration(1, 2) {
    override fun migrate(db: SupportSQLiteDatabase) {
        db.execSQL("ALTER TABLE Person ADD COLUMN email TEXT")
    }
}

Room.databaseBuilder(...)
    .addMigrations(MIGRATION_1_2)
    .build()
```

## 5 Bonnes pratiques

- Utiliser Room plutôt que SQLite directement
- Ne jamais faire d'opérations I/O sur le UI-thread
- Exposer uniquement des `LiveData` (lecture seule) depuis le Repository
- Stocker le singleton de la Database dans l'Application
- Utiliser `distinctUntilChanged()` sur les LiveData pour optimiser les performances
- Migrer vers DataStore pour les préférences
- Activer `exportSchema = true` et versionner les schémas

> ⚠ **Warning**
>
> Ne jamais référencer de View, Activity ou Context d'activité dans un ViewModel.

# 6 Dépendances

```
// Room
implementation("androidx.room:room-runtime:2.8.3")
implementation("androidx.room:room-ktx:2.8.3")
ksp("androidx.room:room-compiler:2.8.3")

// KSP plugin
plugins {
    id("com.google.devtools.ksp") version "2.2.20-2.0.4"
}
```