

Architecture et composants Android

Activités

Cycle de vie (voir Ressources)

États d'une Activité

- **Active (Running)** : au premier plan, utilisateur interagit → onResume()
- **Visible (Paused)** : visible mais sans focus (dialog, split-screen) → onPause()
- **Arrêtée (Stopped)** : invisible, état conservé en mémoire, peut être tuée → onStop()
- **Détruite (Destroyed)** : doit être recréée, mémoire libérée → onDestroy()

Méthodes de cycle de vie

- onCreate() : création, initialisation, setContentView()
- onStart() : activité devient visible
- onResume() : au premier plan, interactive, reprendre animations
- onPause() : perd focus, sauvegarder brouillons, pause opérations
- onStop() : invisible, libérer ressources lourdes
- onDestroy() : destruction définitive, nettoyage final

Transitions courantes

- Lancement : onCreate() → onStart() → onResume()
- Rotation : onPause() → onStop() → onDestroy() → onCreate() → onStart() → onResume()
- Dialog affiché : onPause() (activité visible mais sans focus)
- Retour arrière : onPause() → onStop() → onDestroy()

Activity vs AppCompatActivity : Activity offre une rétrocompatibilité limitée et supporte Material Design uniquement sur API 21+, tandis que AppCompatActivity garantit une excellente compatibilité sur toutes versions avec support complet de Material Design, ActionBar moderne et Fragments. Toujours utiliser AppCompatActivity.

Pourquoi hériter d'AppCompatActivity plutôt que d'Activity ?

On n'hérite jamais directement de la classe Activity du SDK car elle offre une compatibilité limitée. AppCompatActivity (de AndroidX) fournit :

- **Rétrocompatibilité** : fonctionnalités modernes sur anciennes versions Android
- **Material Design** : composants visuels modernes sur toutes versions (API 14+)
- **ActionBar moderne** : getSupportActionBar() avec fonctionnalités étendues
- **Support des Fragments** : FragmentManager moderne et fiable
- **Vector Drawables** : support images vectorielles sur anciennes versions
- **Thèmes AppCompat** : thèmes cohérents multi-versions
- **Bug fixes** : corrections de bugs sans attendre mise à jour système

Principe : AppCompatActivity encapsule Activity et ajoute une couche de compatibilité, permettant d'utiliser les dernières fonctionnalités Android tout en restant compatible avec d'anciennes versions.

Démarrer une Activité

On ne crée jamais directement une instance d'Activité. Android gère le cycle de vie. Les différentes manières :

1. **Activité principale** : déclarée avec LAUNCHER dans le manifeste, point d'entrée de l'app
2. **Intent explicite** : startActivity(Intent(this, SecondActivity::class.java))
3. **Intent implicite** : startActivityForResult(Intent(Intent.ACTION_VIEW, Uri.parse("https://...")))

View Binding

- Activer : buildFeatures { viewBinding = true }
- Usage : binding = ActivityMainBinding.inflate(layoutInflater)
- Accès : binding.myButton.setOnClickListener { }

Intents Explicite vs Implicite : Un Intent explicite cible une classe précise pour la navigation interne de l'app avec une sécurité contrôlée (Intent(this, Activity::class.java)), alors qu'un Intent implicite définit une action générique pour appeler des apps externes comme le navigateur ou la caméra, où le système choisit l'app (Intent(ACTION_VIEW)).

Les 2 types d'Intents avec exemples

1. **Intent Explicite** : spécifie la classe exacte de l'Activity à démarrer

```
// Exemple : ouvrir un écran de détails dans votre app
val intent = Intent(this, DetailActivity::class.java)
intent.putExtra("USER_ID", userId)
startActivity(intent)
```

2. **Intent Implicite** : spécifie une action, le système choisit l'app appropriée

```
// Exemple : ouvrir une URL dans le navigateur
val intent = Intent(Intent.ACTION_VIEW, Uri.parse("https://heig-vd.ch"))
startActivity(intent)
```

```
// Exemple : composer un numéro de téléphone
val intent = Intent(Intent.ACTION_DIAL, Uri.parse("tel:+41123456789"))
startActivity(intent)
```

Actions implicites courantes

- ACTION_VIEW : afficher URL/image
- ACTION_DIAL : composer numéro
- ACTION_SEND : partager contenu
- ACTION_IMAGE_CAPTURE : prendre photo

Activity Result API

```
val launcher = registerForActivityResult(StartActivityForResult()) { result
    if (result.resultCode == RESULT_OK) {
        val data = result.data?.getStringExtra("KEY")
    }
}
```

```
}
```

Sauvegarde état

- onSaveInstanceState(Bundle) : sauvegarder avant rotation
- onRestoreInstanceState(Bundle) : restaurer après récréation
- Sauvegarder : position scroll, texte saisie, sélections
- Ne pas sauvegarder : objets complexes, préférences

Cas de destruction de l'Activité

1. **Rotation écran** : onSaveInstanceState() appelé → Activity détruite et recréée → état restauré
2. **Manque de mémoire** : système tue l'Activity en arrière-plan → onSaveInstanceState() appelé → état sauvegardé
3. **Multi-fenêtres** : Activity visible mais pas au premier plan → onSaveInstanceState() appelé si risque de destruction
4. **Retour arrière (Back)** : utilisateur quitte définitivement → onSaveInstanceState() NON appelé car destruction intentionnelle
5. **finish() explicite** : code appelle finish() → onSaveInstanceState() NON appelé car destruction programmée
6. **Processus tué par système** : app en arrière-plan, système libère mémoire → onSaveInstanceState() peut ne pas être appelé si kill brutal

Principe : onSaveInstanceState() est appelé pour les **déstructions temporaires/involontaires** où l'utilisateur s'attend à retrouver l'état. Elle n'est **pas appellée** pour les **déstructions définitives/intentionnelles**.

⚠ Warning

Dans le cas où on utilise un view model pour sauvegarder l'état, il n'est pas nécessaire d'utiliser onSaveInstanceState et onRestoreInstanceState car le view model survit aux changements de configuration comme la rotation de l'écran.

Fragments

Cycle de vie (voir Ressources)

- onAttach() : attachement à activité
- onCreate() : initialisation fragment
- onCreateView() : création interface (inflate layout)
- onViewCreated() : vues accessibles, initialisation
- onDestroyView() : destruction vues
- onDestroy() : nettoyage ressources
- onDetach() : détachement activité

Caractéristiques

- Portion réutilisable d'UI avec cycle de vie propre
- Plusieurs fragments par activité
- Modularité et réutilisabilité
- Back stack indépendant

Transactions

```
fragmentManager.commit {
    add(R.id.container, fragment)
    replace(R.id.container, fragment)
    remove(fragment)
    addToBackStack(null)
}
```

Communication

- Fragment → Activité : interfaces callback
- Activité → Fragment : méthodes publiques
- Fragment ↔ Fragment : ViewModel partagé (activityViewModels())

⚠ Warning

Observer LiveData avec viewLifecycleOwner dans fragments, jamais this.

Services

Services Started vs Bound vs Foreground : Un Started Service démarre avec startService() pour une durée indéfinie (upload, sync) sans notification, un Bound Service utilise bindService() et vit lié au client pour une communication bidirectionnelle sans notification, tandis qu'un Foreground Service nécessite startForeground() avec une notification obligatoire pour des tâches longues comme la musique ou GPS avec haute priorité.

Cycle de vie

- onCreate() : création, initialisation
- onStartCommand() : chaque appel startService()
- onBind() : retourne IBinder pour communication
- onDestroy() : nettoyage ressources

WorkManager (recommandé)

- Tâches différées/périodiques garanties
- Contraintes : réseau, batterie, stockage
- Survit aux redémarrages
- Remplace JobScheduler et AlarmManager

⚠ Warning

Services exécutent sur UI-Thread. Créer thread séparé pour tâches longues.

Permissions

Types

- **Normal** : accordées automatiquement (Internet, vibration)
- **Dangerous** : demande explicite (localisation, caméra, contacts)
- **Spéciales** : paramètres système (overlay, usage stats)

Workflow runtime permissions

```
// 1. Vérifier
if (checkSelfPermission(permission) == PERMISSION_GRANTED) { }

// 2. Demander
val launcher = registerForActivityResult(RequestPermission()) { granted ->
    if (granted) { /* utiliser */ }
}
launcher.launch(permission)
```

Best practices

- Demander au moment d'usage, pas au lancement
- Expliquer pourquoi permission nécessaire
- Fonctionnalité dégradée si refusée
- Respecter refus définitif

Interface utilisateur de base

Ressources et classe R

Manifest (AndroidManifest.xml)

- Déclare composants : activités, services, receivers, providers
- Liste permissions requises
- Définit point d'entrée : LAUNCHER activity

Ressources (res/)

- values/ : strings, dimensions, couleurs
- drawable/ : images (bitmap, vector)
- layout/ : interfaces XML
- Qualificateurs : -fr, -night, -sw600dp, -land

Classe R

La classe **R** permet d'accéder de manière typesafe aux ressources définies dans le projet Android. Par exemple, pour accéder à une chaîne de caractères définie dans `res/values/strings.xml`, on utilise `R.string.nom_de_la_chaine`. De même, pour une image dans `res/drawable/`, on utilise `R.drawable.nom_de_l_image`. Cela évite les erreurs de frappe et facilite la maintenance du code. Cela permet aussi de gérer facilement les ressources pour différentes configurations (langues, tailles d'écran, modes sombre/clair) en utilisant des qualificateurs de dossiers.

Unités de mesure : dp (density-independent pixels) est une unité **indépendante de la densité écran** ($1\text{dp} = 1\text{px}$ à 160dpi) pour les **dimensions de layout**, tandis que **sp (scale-independent pixels)** s'adapte en plus aux **préférences de taille de texte** de l'utilisateur et doit toujours être utilisé pour le texte.

Drawables Bitmap vs Vector : Les **Bitmap** (PNG, WEBP, JPEG) nécessitent **plusieurs densités** (`mdpi=1x`, `hdpi=1.5x`, `xhdpi=2x`, `xxhdpi=3x`, `xxxhdpi=4x`) et se **pixellisent** lors du redimensionnement, alors que les **Vector** (SVG Android) sont **scalables sans perte de qualité** avec un **fichier unique** pour toutes densités.

Afficher une image dans une Activité

Trois approches principales :

1. **XML avec ImageView** : définir `android:src="@drawable/image"` dans le layout
2. **Programmatique** : `imageView.setImageResource(R.drawable.image)` pour ressources ou `imageView.setImageBitmap(bitmap)` pour images dynamiques
3. **Librairies** : Glide, Picasso, Coil pour chargement asynchrone, cache, et gestion mémoire

Précautions JPEG/PNG : Les images bitmap (JPEG, PNG) doivent être fournies en **plusieurs densités** (`drawable-mdpi`, `drawable-hdpi`, `drawable-xhdpi`, etc.) pour éviter la **pixellisation** ou la **surconsommation de mémoire**. Android sélectionne automatiquement la densité appropriée selon l'écran. JPEG est préféré pour photos (compression), PNG pour transparence et graphiques nets.

Autres types

- **Nine-Patch** : zones extensibles sans distorsion
- **State List** : change selon état (pressed, focused)
- **Level List** : varie selon niveau numérique

Composants graphiques (Views)

Visibilité

- **VISIBLE** : visible et occupe espace
- **INVISIBLE** : invisible mais espace réservé
- **GONE** : invisible sans espace

Composants de base

- **TextView** : affichage texte avec `text`, `textSize`, `textColor`
- **EditText** : saisie avec `inputType` (`text`, `number`, `email`, `password`), `hint`
- **Button** : déclenchement actions avec `setOnClickListener`
- **ImageView** : affichage image avec `scaleType` (`fitCenter`, `centerCrop`)
- **CheckBox/Switch** : choix multiples avec `isChecked`
- **RadioGroup** : choix unique parmi options
- **Spinner** : menu déroulant avec `ArrayAdapter`
- **ProgressBar** : indéterminée ou déterminée (0-100)
- **SeekBar** : curseur ajustable

Layouts et widgets

LinearLayout vs RelativeLayout vs ConstraintLayout : **LinearLayout** offre une **direction unique** (vertical ou horizontal) avec `layout_weight` pour répartition proportionnelle, simple pour formulaires. **RelativeLayout** permet un **positionnement relatif** entre vues (above, below, alignParent) plus flexible mais complexe.

ConstraintLayout (recommandé) utilise des **contraintes multiples** par vue avec chains, guidelines et barrières pour un **layout plat performant qui remplace les deux autres**.

FrameLayout

- Vues empilées
- Bon pour fragments
- Un enfant généralement

ScrollView

- Défilement vertical
- Un seul enfant direct
- NestedScrollView pour coordination

Vues personnalisées

Création

- Hériter View ou sous-classe existante
- `onDraw(Canvas)` : dessin custom
- `onMeasure()` : calcul dimensions
- `onTouchEvent()` : gestion tactile

Attributs XML personnalisés

```
// res/values/attrs.xml
<declare-styleable name="MyView">
    <attr name="customColor" format="color"/>
</declare-styleable>

// Constructeur
val attrs = context.obtainStyledAttributes(attrs, R.styleable.MyView)
val color = attrs.getColor(R.styleable.MyView_customColor, defaultColor)
attrs.recycle()
```

Material Design

Composants principaux

- **MaterialButton** : bouton avec variantes (outlined, text)
- **TextInputLayout** : label flottant, erreurs, compteur
- **Chip** : étiquettes interactives (filtres, tags)
- **FloatingActionButton** : action principale écran
- **BottomNavigationView** : navigation barre basse
- **TabLayout** : onglets horizontaux
- **CardView** : conteneur avec ombres

TextInputLayout

```
// Afficher erreur
inputLayout.error = "Message d'erreur"
// Effacer
inputLayout.error = null
// Compteur
inputLayout.isCounterEnabled = true
inputLayout.counterMaxLength = 100
```

Interaction utilisateur

Gestures et animations

Types de gestes

- **Tap** : toucher simple
- **Double tap** : double toucher
- **Long press** : appui long
- **Swipe** : glisser
- **Pinch** : pincer (zoom)
- **Fling** : projection rapide

GestureDetector

```
val detector = GestureDetector(context, object : SimpleOnGestureListener() {
    override fun onDown(e: MotionEvent) = true
    override fun onSingleTapUp(e: MotionEvent) = true
    override fun onLongPress(e: MotionEvent) { }
    override fun onFling(e1: MotionEvent, e2: MotionEvent,
        velocityX: Float, velocityY: Float) = true
})
view.setOnTouchListener { detector.onTouchEvent(it) }
```

Animations

```
// Programmatique
view.animate()
    .alpha(0f)
    .translationY(100f)
    .rotation(360f)
    .setDuration(1000)
    .start()

// XML (res/anim/)
val anim = AnimationUtils.loadAnimation(context, R.anim.fade_in)
view.startAnimation(anim)
```

Feedback utilisateur (Toast, Snackbar, Dialog)

Toast vs Snackbar vs Dialog : **Toast** affiche un message court (2-3.5s) sans interaction de manière flottante pour des **infos simples**, **Snackbar** apparaît en **bas d'écran** avec une **action optionnelle** (undo, retry) et **durée configurable**, tandis que **Dialog** au centre

nécessite une **interaction obligatoire** jusqu'à action utilisateur pour des **confirmations importantes**.

Toast
Toast.makeText(context, "Message", Toast.LENGTH_SHORT).show()

Snackbar
Snackbar.make(view, "Message", Snackbar.LENGTH_LONG)
.setAction("Annuler") { /* action */ }
.show()

AlertDialog
AlertDialog.Builder(context)
.setTitle("Titre")
.setMessage("Message")
.setPositiveButton("OK") { _, _ -> }
.setNegativeButton("Annuler", null)
.show()

ActionBar et Menu

Configuration ActionBar

```
setSupportActionBar(toolbar)
supportActionBar?.apply {
    title = "Titre"
    setDisplayHomeAsUpEnabled(true)
}
```

Menu

```
// onCreateOptionsMenu
menuInflater.inflate(R.menu.menu_main, menu)

// onOptionsItemSelected
when (item.itemId) {
    R.id.action_search -> true
    android.R.id.home -> { finish(); true }
    else -> super.onOptionsItemSelected(item)
}
```

Menu XML

- showAsAction : always (toujours visible), ifRoom (si place), never (overflow)

Notifications

Structure

```
val notification = NotificationCompat.Builder(context, CHANNEL_ID)
    .setSmallIcon(R.drawable.icon)
    .setContentTitle("Titre")
    .setContentText("Message")
    .setContentIntent(pendingIntent)
    .setAutoCancel(true)
    .build()
```

notificationManager.notify(notificationId, notification)

Channels (API 26+)

```
val channel = NotificationChannel(
    CHANNEL_ID,
    "Nom du channel",
    NotificationManager.IMPORTANCE_DEFAULT
)
notificationManager.createNotificationChannel(channel)
```

Styles étendus

- BigTextStyle : texte long déployable
- BigPictureStyle : affiche grande image
- InboxStyle : liste de lignes (5-6 max)
- MessagingStyle : conversation avec messages

Affichage de données

Listes (ListView et RecyclerView)

ListView vs RecyclerView : ListView offre des **performances moyennes** avec un **ViewHolder optionnel** et uniquement une **disposition verticale** avec des **animations limitées**, tandis que RecyclerView garantit d'**excellentes performances** avec un **ViewHolder obligatoire**, supporte **Linear, Grid et Staggered** layouts, propose des **animations riches** et des **ItemDecoration**, rendant RecyclerView le **standard recommandé**.

RecyclerView composants

- Adapter : lie données aux vues, crée ViewHolders
- ViewHolder : cache références vues (performances)
- LayoutManager : disposition items (Linear, Grid, Staggered)
- ItemDecoration : séparateurs, espacements
- ItemAnimator : animations ajout/suppression

Adapter méthodes

```
class MyAdapter : RecyclerView.Adapter<MyAdapter.ViewHolder>() {
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
        val view = LayoutInflater.from(parent.context)
            .inflate(R.layout.item, parent, false)
        return ViewHolder(view)
    }

    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        holder.bind(items[position])
    }
}
```

```
    override fun getItemCount() = items.size
}
```

ScrollView vs RecyclerView : ScrollView charge **toutes les vues** en mémoire sans recyclage, offre de **mauvaises performances** pour les longues listes, supporte uniquement le **vertical** avec des **animations manuelles**, idéal pour du **contenu statique court** (< 20 éléments). RecyclerView utilise un **recyclage automatique** des vues visibles uniquement, garantit d'**excellentes performances**, supporte **Linear, Grid et Staggered** avec des **animations intégrées**, parfait pour les **listes dynamiques longues** (feeds, catalogues).

Exemples d'utilisation

- ScrollView : formulaire de connexion avec quelques champs, page "À propos" avec texte et images, écran de paramètres avec 10-15 options
- RecyclerView : liste de contacts (centaines d'entrées), feed d'actualités, galerie photos, catalogue produits e-commerce, historique de messages

Optimisations

- setHasFixedSize(true) : si taille ne change pas
- setItemViewCacheSize(n) : augmenter cache
- Librairies images : Glide, Picasso, Coil

Adaptateurs

DiffUtil vs ListAdapter : DiffUtil nécessite un **setup manuel** avec calculateDiff() puis dispatchUpdatesTo(adapter) offrant un **contrôle total** mais un **code plus complexe**, tandis que ListAdapter intègre DiffUtil **automatiquement** avec submitList(newList) pour un **code simplifié** adapté aux **cas standards**.

DiffUtil

```
class DiffCallback : DiffUtil.ItemCallback<Item>() {
    override fun areItemsTheSame(old: Item, new: Item) =
        old.id == new.id

    override fun areContentsTheSame(old: Item, new: Item) =
        old == new
}
```

ListAdapter

```
class MyAdapter : ListAdapter<Item, ViewHolder>(DiffCallback()) {
    // Pas de liste interne, utiliser getItem(position)
}

// Usage
adapter.submitList(newList)
```

Architecture MVVM

LiveData

Caractéristiques

- Observable respectant cycle de vie
- Notifie seulement si composant actif (STARTED/RESUMED)
- Évite fuites mémoire (dés-observation automatique)
- Thread-safe
- Notifie sur UI-thread

Utilisation

```
// ViewModel
private val _data = MutableLiveData<String>()
val data: LiveData<String> = _data

fun updateData(value: String) {
    _data.value = value // UI thread
    _data.postValue(value) // any thread
}

// Activity
viewModel.data.observe(this) { newValue ->
    textView.text = newValue
}

// Fragment
viewModel.data.observe(viewLifecycleOwner) { newValue ->
    textView.text = newValue
}
```

⚠ Warning

Exposer LiveData immuable publiquement, MutableLiveData privée pour encapsulation.

ViewModel

Rôle

- Stocke données UI
- Survit rotations/recreations
- Sépare logique métier de logique UI
- Partage données entre fragments

Cycle de vie et relation avec Activity

- Créé au premier accès via ViewModelProvider
- Persiste pendant toute la durée de vie de l'Activity (rotations, recreations)
- Détruit uniquement au finish() définitif de l'activité
- onCleared() : appelé juste avant destruction, nettoyage ressources

L'affirmation est CORRECTE : Le ViewModel permet effectivement de se passer de `onSaveInstanceState()` pour sauvegarder l'état car il **survit aux destructions temporaires** (rotation écran). L'Activity peut être détruite et recréée (`onDestroy() → onCreate()`), mais le **ViewModel reste en mémoire** et conserve les données. Cependant, le ViewModel est détruit si le processus est tué par le système (manque mémoire), donc `onSaveInstanceState()` reste utile pour sauvegarder l'état critique dans ce cas.

Création

```
class MyViewModel : ViewModel() {
    private val _data = MutableLiveData<String>()
    val data: LiveData<String> = _data

    override fun onCleared() {
        // Nettoyage
    }
}

// Activity/Fragment
val viewModel: MyViewModel by viewModels()

// Fragment partagé
val sharedViewModel: SharedViewModel by activityViewModels()
```

Règles importantes

- **✗** Jamais référencer View, Activity, Fragment
- **✗** Pas de Context activité (fuites mémoire)
- **✓** Exposer LiveData immuables
- **✓** MutableLiveData privée
- **✓** Logique métier dans ViewModel

Observers et transformations

map : transformation 1-à-1 : **map** transforme une **valeur** en une autre **valeur** de manière **synchrone** et **immédiate** (formater date, extraire propriété), alors que **switchMap** transforme une **valeur** en une nouvelle **LiveData** en **annulant l'observation** précédente automatiquement (recherche dynamique, chargement dépendant d'ID).

Transformations

```
// map
val userNames = users.map { list -> list.map { it.name } }

// switchMap
val userData = userId.switchMap { id ->
    repository.getUserById(id)
}

// distinctUntilChanged
val filtered = data.distinctUntilChanged()
```

MediatorLiveData

```
val fullName = MediatorLiveData<String>().apply {
    addSource(firstName) { value = "$it ${lastName.value}" }
    addSource(lastName) { value = "${firstName.value} $it" }
}
```

Persistance des données

Stockage de fichiers

Interne vs Externe : Le stockage interne dans `filesDir` est **privé à l'app**, chiffré **automatiquement** (API 29+), **toujours disponible** et **supprimé à la désinstallation**, tandis que le stockage externe sur `getExternalFilesDir()` peut être sur **carte SD, non chiffré, accessible par d'autres apps**, nécessite de vérifier le **montage** et est **supprimé si privé**.

Stockage interne

- `filesDir` : données persistantes
- `cacheDir` : cache, système peut supprimer

Stockage externe

- `getExternalFilesDir(type)` : fichiers app
- `externalCacheDir` : cache externe
- Vérifier : `Environment.MEDIA_MOUNTED`

Préférences (SharedPreferences)

Types supportés

- String, Int, Long, Float, Boolean, Set

Point commun des 3 approches : Toutes permettent de **persister des données** localement sur l'appareil qui **survivent à la fermeture de l'app** et aux redémarrages. Elles offrent un **accès hors ligne** aux données.

Quand utiliser chaque approche

Fichiers (filesDir, cacheDir) : pour stocker des **documents, images, vidéos, logs** ou tout contenu volumineux non structuré.

- Exemple : télécharger et sauvegarder un PDF, stocker des photos capturées par l'utilisateur, sauvegarder des logs d'application pour debugging

SharedPreferences : pour des **préférences simples clé-valeur** comme paramètres utilisateur, settings d'app.

- Exemple : sauvegarder le thème choisi (clair/sombre), langue de l'interface, préférence "rester connecté", dernière position de scroll

Room : pour des **données structurées relationnelles** nécessitant requêtes complexes, relations entre tables.

- Exemple : liste de tâches avec catégories, contacts avec numéros multiples, historique de transactions avec filtres/recherches, inventaire avec relations produit-catégorie

Opérations

```
// Écriture
getSharedPreferences("nom", MODE_PRIVATE).edit {
    putString("key", "value")
   .putInt("count", 42)
    putBoolean("enabled", true)
}

// Lecture
val prefs = getSharedPreferences("nom", MODE_PRIVATE)
val value = prefs.getString("key", "default")
val count = prefs.getInt("count", 0)
```

Commit vs Apply

- `commit()` : synchrone, retourne succès
- `apply()` : asynchrone, recommandé

Bases de données (Room)

Architecture : Entity ↔ DAO ↔ Database ↔ Repository ↔ ViewModel ↔ UI

Avantages

- Vérification requêtes à la compilation
- Conversion automatique objets ↔ tables
- Support LiveData/Flow
- Gestion migrations

Entity

```
@Entity(tableName = "users")
data class User(
    @PrimaryKey(autoGenerate = true) val id: Int = 0,
    @ColumnInfo(name = "user_name") val name: String,
    val age: Int,
    @Ignore val temporary: String = "")
```

DAO

```
@Dao
interface UserDao {
    @Query("SELECT * FROM users WHERE age > :minAge")
    fun getUsersOlderThan(minAge: Int): LiveData<List<User>>

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insert(user: User)

    @Update
    suspend fun update(user: User)

    @Delete
    suspend fun delete(user: User)
}
```

Database

```
@Database(entities = [User::class], version = 1, exportSchema = true)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao

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

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

exportSchema = true

Active l'exportation du schéma de la base de données au format JSON dans le dossier `schemas/` du projet.

Intérêt d'activer `exportSchema`

- **Historique des versions** : conserve un fichier JSON pour chaque version de la DB (`v1.json`, `v2.json`, etc.)
- **Migrations facilitées** : permet de comparer les différences entre versions lors de l'écriture de migrations
- **Documentation** : sert de documentation technique de la structure de la base
- **Debugging** : aide à identifier les problèmes de structure
- **Contrôle de version** : peut être versionné avec Git pour suivre l'évolution du schéma
- **Tests** : permet de valider que les migrations transforment correctement le schéma

Configuration Gradle require :

```
ksp {
    arg("room.schemaLocation", "$projectDir/schemas")
}
```

TypeConverter

```
class Converters {
    @TypeConverter
    fun fromTimestamp(value: Long?): Date? = value?.let { Date(it) }

    @TypeConverter
    fun dateToTimestamp(date: Date?): Long? = date?.time
}
```

```
@Database(..., )
@TypeConverters(Converters::class)
abstract class AppDatabase : RoomDatabase()
```

Repository pattern

Rôle

- Abstraction accès données
- Comme sources multiples (Room + API)
- Logique métier (cache, validation)
- Gère threads (coroutines)

Implémentation

```
class UserRepository(private val userDao: UserDao) {
    val allUsers: LiveData<List<User>> = userDao.getAllUsers()

    suspend fun insert(user: User) {
        userDao.insert(user)
    }

    suspend fun refreshUsers() {
        // Récupérer du réseau et sauvegarder
        val users = apiService.fetchUsers()
        userDao.insertAll(users)
    }
}
```

Pattern recommandé

- Room est **single source of truth**
- Réseau met à jour Room
- UI observe Room uniquement
- **Offline-first** : app fonctionne sans réseau

Gestion de la concurrence

UI-Thread

Règle d'or

- Modifications UI → uniquement UI-Thread
- Opérations longues → thread séparé

Conséquences violations

- Modifier UI hors UI-Thread → CalledFromWrongThreadException
- Bloquer UI-Thread → ANR (Application Not Responding)

Solutions

```
// runOnUiThread
runOnUiThread {
    textView.text = "Updated"
}

// Handler
Handler(Looper.getMainLooper()).post {
    textView.text = "Updated"
}
```

Opérations asynchrones

Couroutines (recommandé)

```
// ViewModel
viewModelScope.launch {
    val data = withContext(Dispatchers.IO) {
        // Opération background (réseau, DB)
        repository.fetchData()
    }
    // Retour automatique sur Main thread
    _data.value = data
}
```

Dispatchers

- **Main** : UI thread
- **IO** : réseau, fichiers, base de données
- **Default** : calculs intensifs CPU
- **Unconfined** : non confiné (avancé)

WorkManager

```
val workRequest = OneTimeWorkRequestBuilder<MyWorker>()
    .setConstraints(Constraints.Builder()
        .setRequiredNetworkType(NetworkType.CONNECTED)
    .build())
    .build()

WorkManager.getInstance(context).enqueue(workRequest)
```

Ressources



