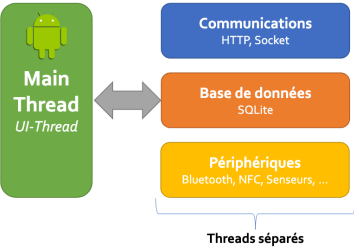


Threads & Coroutines

Threads

- UI-Thread et opérations bloquantes**
- Thread principal responsable de l’UI, ne doit jamais être bloqué
 - Opérations réseau → exception si sur UI-Thread
 - Opérations longues → thread séparé obligatoire



Handler

- Communication entre threads
- Post tâches sur thread principal depuis background

```
val handler = Handler(Looper.getMainLooper())
handler.post { myImage.setImageBitmap(bmp) }
```

Exemple complet avec Handler

```
class MyDownloader {
    private val url: URL,
    private val myImage: ImageView,
    private val handler: Handler
} : Thread() {
    override fun run() {
        val bmp = BitmapFactory.decodeStream(
            url.openConnection().getInputStream()
        )
        handler.post {
            myImage.setImageBitmap(bmp)
        }
    }
}

// Utilisation
val handler = Handler(Looper.getMainLooper())
val downloader = MyDownloader(
    URL("https://example.com/image.png"),
    myImageView,
    handler
)
downloader.start()
```

runOnUiThread dans Activity

```
runOnUiThread { myImage.setImageBitmap(bmp) }
```

- Limites des threads**
- Non conscients du cycle de vie Android
 - Risque de fuites mémoire (références Activity)
 - Gestion concurrence complexe
 - Solution : WeakReference pour éviter memory leaks

Exemple WeakReference

```
class MyImageDownloader(private val url: URL,
    private val handler: Handler) : Thread() {
    lateinit var callback: WeakReference<Bitmap>
```

```
Bitmap?) -> Unit>

    fun start(callBack: (bitmap: Bitmap?) -> Unit)
    {
        this.callback = WeakReference(callBack)
        super.start()
    }

    override fun run() {
        handler.post { callback.get()?.let
        { it(bmp) } } }
    }
```

Exemple téléchargement

```
thread {
    val url = URL("https://example.com/image.png")
    val bmp =
    BitmapFactory.decodeStream(url.openStream())
    runOnUiThread {
        myImage.setImageBitmap(bmp)
    }
}
```

Download dans thread séparé, affichage dans UI-Thread

- Alternatives modernes**
- Coroutines Kotlin (léger, cycle de vie)
 - WorkManager (tâches garanties)

Coroutines

- Caractéristiques**
- Unité légère d’exécution asynchrone
 - Suspendent/reprennent sans bloquer thread
 - Code asynchrone séquentiel (lisible)
 - Mot-clé suspend
 - Coroutine = partage de threads

Suspending vs Bloquantes : Fonctions **suspendues** libèrent le thread pendant l’attente (améliore réactivité), fonctions **bloquantes** empêchent le thread de continuer. L’approche suspendue permet d’écrire du **code asynchrone** de manière **séquentielle**, facilitant la lecture et la maintenance.

⚠ Warning

Suspend = sucre syntaxique, compilateur convertit en callbacks (Continuation).

Exemple :

```
suspend fun loadImage(url: String): ByteArray
{
    return withContext(Dispatchers.IO) {
        URL(url).readBytes() // bloquant mais hors
        UI-Thread
    }
}
```

Dispatchers (contextes d’exécution)

Problématique résolue par les Dispatchers :

- Les Dispatchers permettent de :
- **Choisir le type de thread approprié** selon la nature de la tâche
 - **Éviter de surcharger** certains pools de threads
 - **Optimiser les performances** en utilisant le bon nombre de threads
 - **Protéger l’UI-Thread** des opérations bloquantes

Types de dispatchers : Chaque dispatcher utilise un pool de threads adapté à son usage.

- **Dispatchers.Main** : UI thread, interactions UI
 - Thread unique pour l’interface utilisateur
 - Pour mettre à jour les TextView, ImageView, etc.
 - Ne JAMAIS y faire d’opérations bloquantes
- **Dispatchers.IO** : I/O (réseau, fichiers, DB)
 - Pool jusqu’à 64 threads dynamiques
 - Optimisé pour opérations bloquantes (lecture/écriture)
 - S’adapte automatiquement à la charge
 - Exemples : téléchargements, requêtes API, accès DB, lecture fichiers
- **Dispatchers.Default** : calculs CPU intensifs
 - Nombre de threads = nombre de coeurs CPU
 - Pour traitement d’images, calculs mathématiques, parsing JSON volumineux
 - Partage le pool avec d’autres coroutines Default

⚠ Warning

Toujours utiliser withContext(Dispatchers.IO) pour I/O, jamais sur Main. Un appel réseau sur Main provoque une NetworkOnMainThreadException.

Exemple d’utilisation correcte :

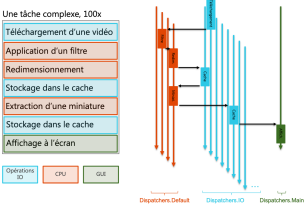
```
suspend fun loadImage(url: URL): Bitmap? =
withContext(Dispatchers.IO) {
    // Téléchargement réseau sur IO dispatcher
    BitmapFactory.decodeStream(
        url.openConnection().getInputStream()
    )
}
```

```
suspend fun processImage(bitmap: Bitmap): Bitmap
=
withContext(Dispatchers.Default) {
    // Traitement CPU intensif sur Default
    dispatcher
    applyFilters(bitmap)
}
```

```
fun displayImage(bitmap: Bitmap) {
    // Affichage UI sur Main dispatcher
    (automatique dans lifecycleScope)
    imageView.setImageBitmap(bitmap)
}
```

// Usage complet

```
lifecycleScope.launch { // Main par défaut
    val downloaded = loadImage(url) // Bascule
    vers IO
    val processed = processImage(downloaded) //
    Bascule vers Default
    displayImage(processed) // Retour sur Main
}
```



```
suspend fun downloadImage(url: URL): ByteArray? = withContext(Dispatchers.IO) {
    try {
        url.readBytes()
    } catch (e: IOException) {
        Log.e(TAG, "Exception while downloading image", e)
    }
    null
}

suspend fun decodeImage(bytes: ByteArray?): Bitmap? = withContext(Dispatchers.Default) {
    try {
        BitmapFactory.decodeByteArray(bytes, 0, bytes.size / 4)
    } catch (e: IOException) {
        Log.e(TAG, "Exception while decoding image", e)
    }
    null
}

suspend fun displayImage(bitmap: Bitmap?) = withContext(Dispatchers.Main) {
    if (bitmap != null) {
        myImage.setImageBitmap(bitmap)
    } else {
        myImage.setImageResource(R.drawable.error_placeholder)
    }
}
```

Dispatcher custom

```
// Limiter à 4 threads max pour contrôler la concurrence
val myDispatcher = Executors
    .newFixedThreadPool(4)
    .asCoroutineDispatcher()
```

```
// Usage
withContext(myDispatcher) {
    // Tâche spécifique avec pool dédié
}
```

```
// Ne pas oublier de fermer le dispatcher !
myDispatcher.close()
```

Utile pour des besoins spécifiques (ex: limiter uploads simultanés).

withContext - Changement de dispatcher

withContext change temporairement le dispatcher pour un bloc de code :

- Suspend la coroutine actuelle
- Exécute le bloc sur le dispatcher spécifié
- Reprend sur le dispatcher original avec le résultat
- Thread-safe et optimisé par le compilateur

```
lifecycleScope.launch { // Sur Main
    val data = withContext(Dispatchers.IO) {
        // Sur IO
        downloadData()
    }
    // Retour automatique sur Main
    textView.text = data
}
```

Scopes

Problématique résolue par les Scopes :

- Les Scopes permettent de :
- **Gérer automatiquement le cycle de vie** des coroutines
 - **Éviter les memory leaks** en annulant les coroutines avec leur composant
 - **Organiser hiérarchiquement** les coroutines (parent-enfant)
 - **Propager l’annulation** : annuler le parent annule tous les enfants
 - **Définir le contexte d’exécution** par défaut (dispatcher, gestion d’erreurs)

Le scope définit la durée de vie des coroutines et leur contexte d’exécution.

- Types de scopes Android :**
- **GlobalScope** : scope application (à ÉVITER)
 - Durée de vie = toute l’application
 - Risque de memory leaks (coroutines continuent après destruction composants)
 - Utiliser uniquement pour tâches vraiment globales
 - Exemple : logger, analytics (mais préférer des alternatives)

- **lifecycleScope** : lié Activity/Fragment

- Annulation automatique quand Activity/Fragment est détruit
- Idéal pour opérations UI liées au cycle de vie
- Se met en pause/reprend avec le lifecycle
- Exemple : téléchargements UI, mise à jour écran

- **viewModelScope** : lié ViewModel
 - Annulation automatique quand ViewModel est cleared
 - Survit aux rotations d’écran (contrairement à lifecycleScope)
 - Idéal pour logique métier indépendante de l’UI
 - Exemple : requêtes API, transformation données

Exemple lifecycleScope dans Activity :

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        lifecycleScope.launch {
            // Coroutine liée au cycle de vie de l'Activity
            val bytes = downloadImage(url)
            val bmp = decodeImage(bytes)
            displayImage(bmp)
        }
        // Annulation automatique si Activity détruite
    }
}
```

Exemple viewModelScope dans ViewModel :

```
class MyViewModel : ViewModel() {
    private val _data =
MutableStateFlow<String>("")
    val data = _data.asStateFlow()

    fun loadData() {
        viewModelScope.launch {
            // Coroutine liée au ViewModel
            try {
                val result = repository.fetchData()
                _data.value = result
            } catch (e: Exception) {
                Log.e("VM", "Erreur: ${e.message}")
            }
        }
        // Survit à la rotation, annulée quand ViewModel cleared
    }
}
```

Lancement avec dispatcher spécifique :

```
lifecycleScope.launch(Dispatchers.Default) {
    // Coroutine sur Default dispatcher
    val result = heavyComputation()
}
```

On peut combiner scope et dispatcher pour un contrôle précis.

Hiérarchie parent-enfant :

```
lifecycleScope.launch { // Parent
    val job1 = launch { task1() } // Enfant 1
    val job2 = launch { task2() } // Enfant 2

    // Si parent annulé → enfants annulés automatiquement
    // Si enfant échoue → parent peut gérer ou propager
}
```

Exemple complet multi-dispatchers

```
// Téléchargement (IO)
suspend fun downloadImage(url: URL) =
    withContext(Dispatchers.IO) {
```

```
url.openConnection().getInputStream().readBytes()
    }

    // Décodage (CPU)
    suspend fun decodeImage(bytes: ByteArray) =
        withContext(Dispatchers.Default) {
            BitmapFactory.decodeByteArray(bytes, 0,
bytes.size)
        }

    // Affichage (UI)
    suspend fun displayImage(bmp: Bitmap?) =
        withContext(Dispatchers.Main) {
            myImage.setImageBitmap(bmp)
        }

    // Utilisation
    lifecycleScope.launch {
        val bytes = downloadImage(url)
        val bmp = decodeImage(bytes)
        displayImage(bmp)
    }
```

Chaque fonction utilise le dispatcher approprié.

delay vs Thread.sleep

- delay() : suspend coroutine, libère thread
- Thread.sleep() : bloque thread

i Info

Toujours préférer delay() dans les coroutines pour ne pas bloquer le thread.

suspendCoroutine() - Pont avec APIs à callbacks

Rôle de suspendCoroutine() :

suspendCoroutine() permet de convertir des APIs basées sur callbacks en fonctions suspensives. C’est le pont entre l’ancien monde (callbacks) et le nouveau (coroutines).

Fonctionnement détaillé :

- Suspend la coroutine en cours d’exécution
- Fournit un objet Continuation pour reprendre l’exécution ultérieurement
- cont.resume(value) : reprend l’exécution avec un résultat de succès
- cont.resumeWithException(e) : reprend l’exécution avec une erreur
- La coroutine reste suspendue jusqu’à ce qu’une de ces méthodes soit appelée

Cas d’utilisation typique : Volley avec coroutines

Volley utilise des callbacks, on veut l’utiliser avec des coroutines :

```
suspend fun downloadHTMLVolley(urlParam: String): String =
    suspendCoroutine { cont ->
        val textRequest = StringRequest(
            Request.Method.GET, urlParam,
            { response ->
                // Succès : on reprend avec la réponse
                cont.resume(response)
            },
            { error ->
                // Erreur : on reprend avec une exception
                cont.resumeWithException(error)
            }
        )
        queue.add(textRequest)
    }
```

Utilisation dans une coroutine :

```
lifecycleScope.launch {
    try {
        val html = downloadHTMLVolley("https://www.heig-vd.ch")
        // Traiter le résultat
        textView.text = html
    } catch (e: Exception) {
        // Gérer l'erreur
        Log.e("Download", "Erreur: ${e.message}")
    }
}
```

Autre exemple : LocationManager avec coroutines

```
suspend fun getCurrentLocation(): Location =
    suspendCoroutine { cont ->
        val locationManager =
            getSystemService(Context.LOCATION_SERVICE) as
            LocationManager

        locationManager.requestSingleUpdate(
            LocationManager.GPS_PROVIDER,
            object : LocationListener {
                override fun onLocationChanged(location:
                    Location) {
                    cont.resume(location) // Succès
                }
                override fun onProviderDisabled(provider:
                    String) {
                    cont.resumeWithException(Exception("GPS
                    désactivé"))
                }
            },
            null
        )
    }
```

Avantages de suspendCoroutine :

- Transforme du code asynchrone à callbacks en code séquentiel lisible
- Gestion d’erreur naturelle avec try/catch (pas de callback d’erreur séparé)
- Compatible avec les bibliothèques existantes sans les réécrire
- Pas besoin de modifier les APIs tierces
- Code plus maintenable et testable

⚠ Warning

Ne jamais appeler resume() ou resumeWithException() plusieurs fois. Une seule reprise est autorisée par suspension.

Points d’attention :

- Toujours gérer le cas d’erreur dans le callback
- La coroutine reste suspendue jusqu’à l’appel de resume
- Ne pas oublier de gérer les timeouts si nécessaire
- Attention aux fuites mémoire si la callback n’est jamais appelée

Annulation des Coroutines

```
val job = lifecycleScope.launch { /* ... */ }
job.cancel() // Demande annulation
job.join() // Attend fin effective
```

Méthodes d’annulation :

1. **Appel régulier fonction suspensive :**

```
suspend fun countdown(max: Int = 10) =
    withContext(Dispatchers.Default) {
        repeat(max) { i ->
            delay(1000) // vérifie automatiquement
            annulation
```

```
        }
    }
}

2. Vérification explicite :
while(value > 0 && isActive) {
    // travail
}

3. yield() pour coopération :
while(value > 0) {
    yield() // donne la main et vérifie annulation
    // travail
}
```

WorkManager

Périmètre

Pour tâches persistantes devant s’exécuter même quand :

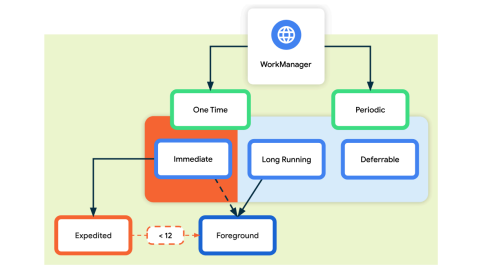
- L’application est en arrière-plan
- L’application est fermée (tuée par l’utilisateur)
- Le téléphone redémarre

Cas d’usage typiques :

- Synchronisation périodique de données
- Upload de fichiers en arrière-plan
- Nettoyage de cache/données temporaires
- Envoi de logs/analytics
- Traitement différé de données

Avantages de WorkManager :

- Gestion automatique des contraintes système (réseau, batterie, stockage)
- Persistance dans SQLite (les tâches survivent au redémarrage)
- Résiste aux redémarrages du téléphone
- Respect des optimisations batterie Android (Doze, App Standby)
- Retry automatique en cas d’échec avec backoff exponentiel
- Compatible avec toutes versions Android (API 14+)



Types de tâches WorkManager :

- **Immédiates** : exécution ASAP (mais peut être différée par le système)
 - OneTimeWorkRequest pour tâche unique
 - Pas de garantie de temps d’exécution exact
- **Longue durée** : > 10 minutes (notification obligatoire)
 - Notification pour informer utilisateur
 - setForeground() pour éviter kill système
- **Différables** : programmées et/ou périodiques
 - PeriodicWorkRequest avec intervalle minimum 15 minutes
 - Contraintes système appliquées

Contraintes système

Android Doze (API 23+)

Mode Doze : économie batterie agressive

- Activé quand écran verrouillé + appareil immobile + non chargé
- Veille profonde : désactive réseau, syncs, jobs, GPS
- Fenêtres de maintenance périodiques (durée de plus en plus espacée)
 - Première : 15 min après Doze
 - Puis espacement exponentiel : 30min, 1h, 2h, 4h...
- Les tâches WorkManager s'exécutent durant ces fenêtres
- App importante : fenêtres plus fréquentes



Light Doze (appareil en mouvement) :

- Restrictions moins sévères
- Fenêtres plus fréquentes
- Permet certaines activités réseau

App Standby Buckets (API 28+)

Classification automatique selon utilisation : Android classe les apps selon leur fréquence d'utilisation pour optimiser batterie.

- **Active** : app actuellement utilisée ou utilisée très récemment
 - Aucune restriction
 - Jobs et syncs normaux

- **Working set** : app utilisée régulièrement (quotidiennement)
 - Restrictions légères
 - Jobs différés de quelques minutes

- **Frequent** : app utilisée fréquemment (quelques fois par semaine)
 - Restrictions modérées
 - Jobs limités à 10 par jour

- **Rare** : app rarement utilisée
 - Restrictions importantes
 - Jobs limités à 5 par jour
 - Fenêtre quotidienne de 10 minutes max

- **Restricted** : app consomme beaucoup de ressources
 - Restrictions maximales
 - Pratiquement pas de jobs en arrière-plan
 - Nécessite interaction utilisateur pour sortir

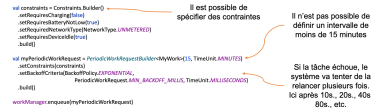
Impact sur WorkManager : Plus une app est dans un bucket restrictif, plus l'exécution des tâches est différée. Le système attend des conditions optimales (charge, WiFi, etc.) avant d'exécuter.

App Hibernation (API 30+)

Hibernation automatique après plusieurs mois sans interaction :

- Si l'utilisateur n'a pas ouvert l'app depuis 3 mois :
- **Révocation permissions runtime** L'utilisateur devra réaccorder les permissions sensibles
- **Arrêt des tâches programmées** WorkManager ne s'exécute plus du tout
- **Arrêt notifications push (FCM)** Plus de messages Firebase reçus
- **Vidage du cache** Libération d'espace de stockage
- **Reset de l'app à l'état initial** Comme si elle venait d'être installée

Sortie d'hibernation : L'utilisateur doit ouvrir l'app manuellement pour la réactiver.



Exemple d'utilisation WorkManager

Définition du Worker :

```
class SyncWorker(
    appContext: Context,
    workerParams: WorkerParameters
) : Worker(appContext, workerParams) {

    override fun doWork(): Result {
        return try {
            // Logique métier ici (téléchargement,
            sync, etc..)
            val data = fetchDataFromServer()
            saveToLocalDatabase(data)

            Result.success() // Tâche réussie
        } catch (e: Exception) {
            Log.e("SyncWorker", "Erreur: ${e.message}")
            Result.retry() // Réessayer plus tard avec
            backoff
        } // ou Result.failure() pour échec définitif
    }
}
```

La logique métier va dans doWork(). Cette méthode s'exécute automatiquement sur un thread background.

Tâche unique (OneTimeWorkRequest) :

```
val workManager =
    WorkManager.getInstance(applicationContext)

val myWorkRequest =
    OneTimeWorkRequestBuilder<SyncWorker>().build()
```

workManager.enqueue(myWorkRequest)

S'exécute une seule fois dès que possible (selon contraintes système).

Tâche périodique avec contraintes (PeriodicWorkRequest) :

```
// Définir les contraintes d'exécution
val constraints = Constraints.Builder()
    .setRequiredNetworkType(NetworkType.CONNECTED) //
    WiFi ou mobile data
    .setRequiresBatteryNotLow(true) // Batterie >
    15%
    .setRequiresCharging(false) // Pas besoin de
    charge
    .setRequiresStorageNotLow(true) // Stockage
    suffisant
    .build()
```

```
// Créer la requête périodique
val periodicWork =
    PeriodicWorkRequestBuilder<SyncWorker>(
        15, // Intervalle : 15 minutes minimum
        TimeUnit.MINUTES,
        10, // Flex interval : peut s'exécuter dans
        les 10 dernières minutes
        TimeUnit.MINUTES
    )
    .setConstraints(constraints) // Appliquer les
    contraintes
    .setBackoffCriteria( // Stratégie en cas
    d'échec
        BackoffPolicy.EXPONENTIAL,
```

```
10,
    TimeUnit.SECONDS
)
.build()
```

workManager.enqueue(periodicWork)

Flex interval expliqué :

- Intervalle = 15 min, Flex = 10 min
- La tâche peut s'exécuter entre la 5ème et 15ème minute
- Permet au système d'optimiser en groupant plusieurs tâches ensemble
- Économise batterie en évitant réveils fréquents de l'appareil

Backoff en cas d'échec : Si Result.retry() est retourné, WorkManager réessaie avec délai croissant :

- 1er essai : immédiat
- 2ème essai : après 10s
- 3ème essai : après 20s
- 4ème essai : après 40s
- 5ème essai : après 80s
- Etc. (backoff exponentiel jusqu'à maximum)

Contraintes disponibles :

- NetworkType.CONNECTED : n'importe quelle connexion réseau
- NetworkType.UNMETERED : WiFi uniquement (pas de data mobile)
- NetworkType.NOT_ROAMING : pas en roaming
- NetworkType.METERED : data mobile acceptée
- setRequiresBatteryNotLow() : batterie > 15%
- setRequiresCharging() : appareil en charge
- setRequiresDeviceIdle() : appareil inactif (rare)
- setRequiresStorageNotLow() : stockage suffisant

⚠ Warning

Intervalle **minimum : 15 minutes** pour PeriodicWorkRequest. Limite Android non contournable pour économiser la batterie.

Observer l'état d'un Worker :

```
workManager.getWorkInfoByIdLiveData(periodicWork.id)
    .observe(this) { workInfo ->
        when (workInfo?.state) {
            WorkInfo.State.ENQUEUED -> Log.d("Work",
            "En attente")
            WorkInfo.State.RUNNING -> Log.d("Work", "En
            cours d'exécution")
            WorkInfo.State.SUCCEEDED -> Log.d("Work",
            "Réussi")
            WorkInfo.State.FAILED -> Log.d("Work",
            "Échoué")
            WorkInfo.State.CANCELLED -> Log.d("Work",
            "Annulé")
            else -> {}
        }
    }
```

Annuler un Worker :

```
workManager.cancelWorkById(periodicWork.id) //
Annuler par ID
workManager.cancelAllWork() // Tout annuler
```

Bonnes pratiques

- Threads : uniquement tâches courtes, WeakReference obligatoire
- Coroutines : préférer aux threads, bon dispatcher, lifecycleScope/viewModelScope
- Rendre coroutines annulables (yield/isActive)

- WorkManager : tâches devant survivre à l'app, minimum 15min périodiques

Communication Web

Connectivité

Technologies

- Wi-Fi : 2.4 GHz (portée) vs 5 GHz (débit)
- Réseaux mobiles : 2G, 3G, 4G, 5G

Norme	Nom	Date	Fréquences	Débit maximum
802.11	N/A	1997	2.4 GHz	2 Mbps
802.11b	Wi-Fi 1	1999	2.4 GHz	11 Mbps
802.11a	Wi-Fi 2	1999	5 GHz	54 Mbps
802.11g	Wi-Fi 3	2003	2.4 GHz	54 Mbps
802.11n	Wi-Fi 4	2009	2.4 GHz ou 5 GHz	72 Mbps 450 Mbps
802.11ac	Wi-Fi 5	2014	5 GHz	1'000 Mbps
802.11ax	Wi-Fi 6	2019	2.4 et 5 GHz	2'400 Mbps
802.11ax	Wi-Fi 6E	2021	2.4, 5 et 6 GHz	4'800 Mbps
802.11be	Wi-Fi 7	2024	2.4, 5 et 6 GHz	30'000 Mbps
802.11bn	Wi-Fi 8	2028 ?	2.4, 5 et 6 GHz	100'000 Mbps

Génération	Année	Données (pointe)	Latence	Remarques
1G	1983	ø	ø	Système analogique
2G	1992	Kbit / s	< 1000 ms	Premier réseau numérique, initialement uniquement voix
3G	2003	Mbit / s	< 500 ms	Intégration des données dès la conception
4G	2010	Gbit / s	< 100 ms	Réseau données uniquement, intégration ultérieure de la voix
5G	2019+	Gbit / s	< 5 ms	Ouverture à l'Internet des Objets (IoT)

Permissions

```
<manifest xmlns:android="http://schemas.android.
com/apk/res/android">
    <uses-permission
    android:name="android.permission.INTERNET" />
    <uses-permission
    android:name="android.permission.ACCESS_NETWORK_STATE" />
</manifest>
```

i Info

INTERNET n'est pas une permission dangereuse, pas besoin d'acceptation utilisateur.

NetworkSecurityConfig Fichier de configuration réseau optionnel :

```
<application
    android:networkSecurityConfig="@xml/
network_security_config">
</application>
```

Permet de définir :

- Certificats auto-signés acceptés
- Domaines autorisés
- Configuration HTTPS

ConnectivityManager

```
val connectivityManager = getSystemService(
    Context.CONNECTIVITY_SERVICE
) as ConnectivityManager
```

```
val networkCapabilities = connectivityManager
    .getNetworkCapabilities(
        connectivityManager.activeNetwork
    )
```

```
val hasInternet = networkCapabilities?
    .hasCapability(
        NetworkCapabilities.NET_CAPABILITY_INTERNET
    ) ?: false
```

```
val isFreeToUse = networkCapabilities?
```



```
.hasCapability()
NetworkCapabilities.NET_CAPABILITY_NOT_METERED
) ?: false
```

```
val notRoaming = networkCapabilities?.hasCapability()
```

```
NetworkCapabilities.NET_CAPABILITY_NOT_ROAMING
) ?: false
```

Capabilities importantes :

- NET_CAPABILITY_INTERNET : connexion Internet disponible
- NET_CAPABILITY_NOT_METERED : réseau illimité (WiFi)
- NET_CAPABILITY_NOT_ROAMING : pas en itinérance

Accès Socket TCP/IP Toutes les API réseau Java

disponibles :

```
thread {
    val resolution = Inet4Address
        .getByName("www.heig-vd.ch").first()
    val socketAddress = InetSocketAddress(
        resolution.hostAddress, 80
    )
    val socket = Socket()
    try {
        socket.connect(socketAddress)
        Log.d("MainActivity", "Host reachable")
    } catch (_: Exception) {}
    finally { socket.close() }
}
```

Services REST

Appel GET avec java.net.URL

```
val url = URL("https://api.example.com/data")
thread {
    val json = url.readText(Charsets.UTF_8)
}
```

Désérialisation JSON avec Gson

```
val type = object : TypeToken<List<FruitDTO>>()
{}.type
val fruits =
Gson().fromJson<List<FruitDTO>>(json, type)
```

Appel POST/PUT

```
val connection = url.openConnection() as
HttpURLConnection
connection.requestMethod = "PUT"
connection.doOutput = true
connection.setRequestProperty("Content-Type",
"application/json")
connection.outputStream.bufferedWriter(Charsets.UTF_8).use
{
    it.append(Gson().toJson(newFruit))
}
val responseCode = connection.responseCode
```

⚠ Warning

- Méthodes synchrones/bloquantes
- Connexion lors de outputStream, inputStream ou responseCode
- Réutilisation automatique connexion

Alternatives à java.net.URL

- OkHttp : client HTTP performant, intercepteurs, cache
- Volley : file requêtes, annulation, retry automatique
- Retrofit : REST client type-safe, conversion auto JSON

Avantages librairies

- Planification requêtes ordonnées
- Annulation requêtes
- Cache intégré

- Retry automatique en cas d'échec
- Sérialisation/désérialisation facilitée

Comparaison java.net.URL vs Volley

```
java.net.URL :
val url = "https://www.heig-vd.ch"
thread {
    val html = url.readText(Charsets.UTF_8)
    runOnUiThread { textView.text = html }
}

Volley :
val url = "https://www.heig-vd.ch"
val queue = Volley.newRequestQueue(this)
val textRequest = StringRequest(Request.Method.GET, url,
{ response -> textView.text = response },
{ textView.text = "error" })
queue.add(textRequest)
```

- Gestion «manuelle» des threads
 - On utilise readText car on s'attend à recevoir du texte
 - Pas de gestion des erreurs
- Pas de gestion des threads, on donne deux callbacks (succès et erreur) que la librairie appellera dans l'Ui-Thread
 - On précise vouloir traiter du texte en retour en utilisant une StringRequest
 - Utilisation d'une queue qui va gérer l'exécution des requêtes

```
java.net.URL :
val url = URL("https://www.heig-vd.ch/logo.png")
thread {
    val bytes = url.readBytes()
    val bmp = BitmapFactory
        .decodeByteArray(bytes, bytes.size)
    runOnUiThread { imageView.setImageBitmap(bmp) }
}

Volley :
val url = "https://www.heig-vd.ch/logo.png"
val queue = Volley.newRequestQueue(this)
val imgRequest = ImageRequest(url,
{ bmp -> imageView.setImageBitmap(bmp) },
1024, 1024,
ScaleType.CENTER_INSIDE,
Bitmap.Config.ARGB_8888,
{ error -> Log.d("MainActivity", "${error.message}") })
queue.add(imgRequest)
```

- On utilise readBytes pour obtenir le payload brut
 - L'image est décodée puis affichée
- Utilisation d'une ImageRequest pour traiter un payload contenant une image, la librairie va convertir l'image chargée (changement de l'encodage, redimensionnement, crop)

```
java.net.URL :
val url = URL("https://www.fruityvice.com/api/fruit/all")
thread {
    val json = url.readText(Charsets.UTF_8)
    val type = object : TypeToken<List<FruitDTO>>() {}
    val fruits = Gson().fromJson<List<FruitDTO>>(json, type)
    runOnUiThread { textView.text = fruits.toString() }
}

Volley :
val url = "https://www.fruityvice.com/api/fruit/all"
val queue = Volley.newRequestQueue(this)
val jsonRequest = JsonRequest(Request.Method.GET, url, null,
{ json -> textView.text = json.toString() },
{ error -> Log.d("MainActivity", "${error.message}") },
queue.add(jsonRequest)
```

- On utilise readText pour obtenir le json brut, puis Gson pour le désérialiser
- L'objet json qui est passé au callback est du type org.json.JSONArray qui n'est pas directement exploitable

java.net.URL avec coroutines

```
suspend fun downloadHTML(urlParam: String):
String = withContext(Dispatchers.IO) {
    URL(urlParam).readText(Charsets.UTF_8)
}
```

Volley avec coroutines

```
suspend fun downloadHTMLVolley(urlParam: String):
String = suspendCoroutine { cont ->
    val textRequest =
StringRequest(Request.Method.GET, urlParam,
{ response -> cont.resume(response) },
{ e -> cont.resumeWithException(e) })
    queue.add(textRequest)
}
```

Transformer callback en coroutine

- suspendCoroutine : suspend jusqu'à réponse
- cont.resume(value) : reprend avec valeur
- cont.resumeWithException(e) : reprend avec erreur

Ktor

Framework asynchrone client/serveur pour HTTP développé par JetBrains.

Configuration

DTOs sérialisables :

```
@Serializable
data class FruitDTO(
    val id: Int,
    val name: String,
    val nutritions: NutritionsDTO
)
```

```
@Serializable
data class NutritionsDTO(
    val calories: Int,
    val sugar: Double
)
```

Configuration du client :

```
val client = HttpClient {
    install(ContentNegotiation) {
        json()
    }
}
```

Exemple d'utilisation

```
suspend fun getAllFruits(): List<FruitDTO> {
    return client.get("https://api.example.com/fruits")
        .body()
}
```

```
suspend fun createFruit(fruit: FruitDTO):
FruitDTO {
    return client.post("https://api.example.com/fruits") {
        contentType(ContentType.Application.Json)
        setBody(fruit)
    }.body()
}
```

Utilisation dans ViewModel

```
class FruitViewModel : ViewModel() {
    private val _fruits =
MutableStateFlow<List<FruitDTO>>(emptyList())
    val fruits = _fruits.asStateFlow()

    fun loadFruits() {
        viewModelScope.launch {
            try {
                _fruits.value = getAllFruits()
            } catch (e: Exception) {
                Log.e("FruitVM", "Error: ${e.message}")
            }
        }
    }
}
```

Avantages Ktor

- Intégration coroutines native (tout est suspend)
- Désérialisation automatique JSON vers objets
- Configuration modulaire
- Type-safe (erreurs compilation)
- Multiplateforme (JVM, JS, Native)

```
private val ktorClient = HttpClient(Android) {
    // Configuration du moteur Android
    engine {
        connectTimeout = 5_000
        socketTimeout = 5_000
        dispatcher = Dispatchers.IO
    }
}

// Plugin pour parser le JSON
install(ContentNegotiation) {
    json {
        ignoreUnknownKeys = true
        prettyPrint = true
        isLenient = true
    }
}

private suspend fun requestFruits(): List<FruitDTO> {
    // 1. On lance la requête GET
    // 2. body() convertit automatiquement le JSON en List<FruitDTO>
    return ktorClient
        .get("https://www.fruityvice.com/api/fruit/all")
        .body()
}

fun getFruits() {
    // La requête s'exécute dans le scope du ViewModel
    viewModelScope.launch {
        val allFruits = requestFruits()
    }
}
```

Comparaison des approches

- java.net.URL : bas niveau, verbeux, synchrone
- Volley : callbacks, cache, retry auto, queue requêtes
- Ktor : coroutines natives, type-safe, moderne, recommandé

Synchronisation données

Contexte

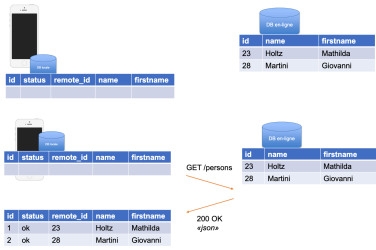
- App fonctionne hors-ligne
- DB locale synchronisée avec serveur
- Algorithme sync complexe (conflits multiples devices)

Structure DB locale

- id : identifiant local unique (auto-incrément)
- remoteId : identifiant serveur (null si nouveau)
- status : ok, new, mod, del (suivi modifications)

Cycle de synchronisation

1. Fetch : télécharger nouvelles données serveur
2. Apply : appliquer changements locaux vers serveur
3. Update : mettre à jour status et remoteId



```
try {
    val response = api.createPerson(person)
    person.remoteId = response.id
    person.status = "ok"
    dao.update(person)
} catch (e: Exception) {
    // Reste en status "new" pour sync
    ultérieure
}
```

```
fun update(person: Person) {
    person.status = "mod"
    dao.update(person)
    lifecycleScope.launch {
        try {
            api.updatePerson(person.remoteId!!,
            person)
            person.status = "ok"
            dao.update(person)
        } catch (e: Exception) {
            // Reste en status "mod"
        }
    }
}
```

DAO avec filtre

```
@Query(
    "SELECT * FROM Person WHERE status != 'del'"
)
fun getAllPersons(): LiveData<List<Person>>
```

L'UI ne voit que les données non supprimées.

Limitations approche simple

- Approche 1 client uniquement
- Conflits multiples clients non gérés
- Modifications serveur non détectées

Solutions avancées :

- Timestamps + dernière modification gagne
- Versioning (ETags HTTP)
- Résolution manuelle utilisateur
- Operational Transform (Google Docs)
- CRDTs (Conflict-free Replicated Data Types)

Bonnes pratiques synchronisation

- Privilégier expérience offline
- Indiquer état sync dans UI
- Utiliser WorkManager pour sync arrière-plan
- Retry intelligent avec backoff exponentiel
- Logger échecs synchronisation
- Bouton “forcer synchronisation”

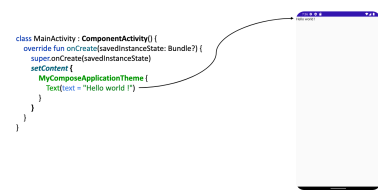
Jetpack Compose

Caractéristiques

- API déclarative pour UI (Kotlin uniquement)
- Approche “Quoi” vs “Comment”
- Composants réutilisables et testables
- BOM (Bill of Materials) pour gestion versions compatibles

Optimisations recomposition

- Évite recomposer composants non modifiés
- Recompositions parallèles (multi-threading)
- Smart recomposition (seules parties changées)



Fonctions @Composable

Règles importantes

- Exécution très fréquente (animations 60fps = 16.6ms/frame)
- Optimisation recomposition automatique
- Doit être rapide, sans effets de bord
- Idempotent (même entrée = même sortie)
- Éviter I/O, modifications variables externes, opérations longues

⚠ Warning

Ne jamais modifier de variables externes dans @Composable.

Cycle de vie fonction composable

1. **Composition initiale** : première exécution, création UI
2. **Recomposition** : ré-exécution quand état change
3. **Suppression** : quand composable plus nécessaire

Fonction composable à état

Une fonction stateful :

- Contient et gère son propre état avec remember
- Utilise mutableStateOf pour état observable
- Se recompose automatiquement quand état change
- Responsable de la logique métier

```
@Composable
fun Counter() { // Fonction À ÉTAT
    var count by remember { mutableStateOf(0) }
    Button(onClick = { count++ }) {
        Text("Compteur: $count")
    }
}
```

Différence stateful vs stateless

Stateful (à état) :

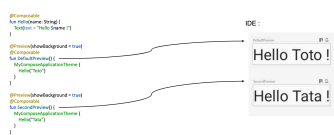
- Gère son propre état interne
- Contient logique métier
- Moins réutilisable
- Difficile à tester

Stateless (sans état) :

- Reçoit état en paramètre
- Pas de logique métier
- Hautement réutilisable
- Facile à tester

Prévisualisation

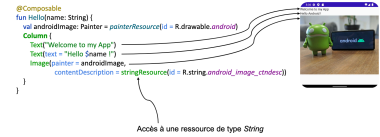
```
@Preview
@Composable
fun Hello(name: String = "World") {
    Text(text = "Hello $name!")
}
```



Layouts

Column

```
Column(
    verticalArrangement = Arrangement.SpaceBetween,
    horizontalAlignment =
    Alignment.CenterHorizontally
) {
    Text("Élément 1")
    Text("Élément 2")
}
```



Row

```
Row(
    horizontalArrangement =
    Arrangement.SpaceBetween,
    verticalAlignment = Alignment.CenterVertically
) { }
```



Box

```
Box(
    contentAlignment = Alignment.Center
) {
    Image(...)
    Text("Superposé") // au-dessus de l'image
}
```

Positionnement libre avec superposition (équivalent RelativeLayout).

ConstraintLayout

Nécessite définir identifiants pour références :

```
val (oneButton, twoButton, threeButton) =
    createRefs()
```

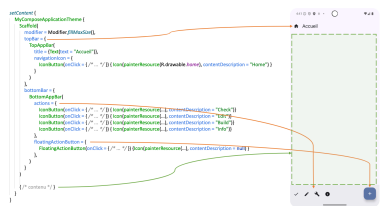
```
Button(
    modifier = Modifier.constrainAs(oneButton) {
        top.linkTo(parent.top)
        start.linkTo(parent.start)
    }
) { Text("1") }
```

```
Button(
    modifier = Modifier.constrainAs(twoButton) {
        top.linkTo(oneButton.bottom)
        start.linkTo(oneButton.end)
    }
) { Text("2") }
```

Composants de base

Scaffold

```
Scaffold(
    topBar = { TopAppBar(...) },
    floatingActionButton =
    { FloatingActionButton(...) },
    content = { }
)
```



Listes scrollables : LazyColumn et LazyRow

LazyColumn/LazyRow : listes optimisées pour grandes quantités de données

Les LazyColumn et LazyRow sont des composants Compose pour afficher des listes scrollables performantes. Ils ne rendent que les éléments visibles à l'écran, économisant ainsi mémoire et ressources.

Différence avec RecyclerView :

- **Plus simple** : pas d'Adapter, ViewHolder, LayoutManager nécessaires
- **Pas de recyclage manuel** : Compose crée/détruit les vues à la volée automatiquement
- **Recomposition automatique** : seuls les éléments modifiés sont recalculés
- **Meilleure intégration** : s'intègre naturellement avec l'état Compose
- **Performance** : seuls les éléments visibles sont rendus (lazy loading)

LazyColumn : liste verticale scrollable

```
LazyColumn(
    modifier = Modifier.fillMaxSize(),
    verticalArrangement =
    Arrangement.spacedBy(8.dp),
    contentPadding = PaddingValues(16.dp)
) {
    items(listOfItems) { item ->
        ItemCard(item = item)
    }
    // ou avec index
    itemsIndexed(listOfItems) { index, item ->
        Text("$index: ${item.name}")
    }
}
```

LazyRow : liste horizontale scrollable

```
LazyRow(
    horizontalArrangement =
    Arrangement.spacedBy(8.dp),
    contentPadding = PaddingValues(horizontal =
    16.dp)
) {
    items(contacts) { contact ->
        ContactCard(contact)
    }
}
```

LazyVerticalGrid : grille verticale scrollable

```
LazyVerticalGrid(
    columns = GridCells.Fixed(2), // 2 colonnes
    // ou GridCells.Adaptive(minSize = 128.dp) //
    taille adaptative
    verticalArrangement =
    Arrangement.spacedBy(8.dp),
    horizontalArrangement =
```

```
Arrangement.spacedBy(8.dp)
) {
    items(photos) { photo ->
        ImageCard(photo)
    }
}
```

- Caractéristiques importantes :**
- Seuls les **éléments visibles** sont rendus (pas tous)
 - **Défilement automatique** sans configuration supplémentaire
 - **Performance optimale** même avec milliers d'éléments
 - Pas de recyclage de vues comme RecyclerView (architecture différente)
 - Support de sticky headers, arrangements personnalisés, etc.

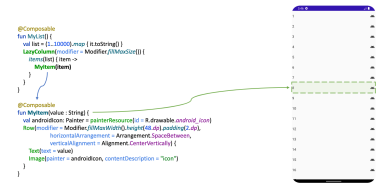
Exemple complet avec données observables :

```
@Composable
fun ContactList(viewModel: ContactViewModel =
viewModel()) {
    val contacts by
viewModel.allContacts.collectAsStateWithLifecycle()
```

```
LazyColumn(
    modifier = Modifier.fillMaxSize(),
    verticalArrangement =
Arrangement.spacedBy(8.dp),
    contentPadding = PaddingValues(16.dp)
) {
    items(
        items = contacts,
        key = { it.id } // Optimisation : clé
        unique par item
    ) { contact ->
        ContactCard(
            contact = contact,
            onClick =
{ viewModel.selectContact(contact) }
        )
    }
}
```

⚠ Warning

Toujours utiliser key = { it.id } pour optimiser les recompositions. Sans clé, Compose ne peut pas identifier les éléments modifiés/déplacés.



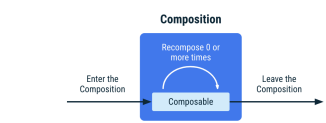
Gestion événements

```
Row(modifier = Modifier.clickable {
    Toast.makeText(context, "Cliqué",
    Toast.LENGTH_SHORT).show()
}) { }
```

- Gestion états**
- State / MutableState**
- remember : conserve état entre recompositions
 - rememberSaveable : survit recréation Activity (rotation)

⚠ Warning

remember ne survit PAS à la rotation. Utiliser rememberSaveable si nécessaire.



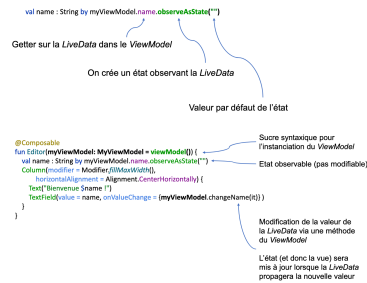
Compteur

```
@Composable
fun Counter() {
    var counter by remember { mutableStateOf(0) }
    Button(onClick = { ++counter }) {
        Text("+")
    }
    Text("$counter")
}
```

TextField

```
@Composable
fun Editor() {
    var name by remember { mutableStateOf("") }
    TextField(value = name, onValueChange = {name =
it})
}
```

ViewModel et LiveData



Flux exécution

1. Saisie TextField
2. Appel changeName() ViewModel
3. Modification _name.value
4. observeAsState() détecte changement
5. Recomposition avec nouvelle valeur

```
La ViewModel et sa Factory:
class MyViewModel(private val viewModelFactory: ViewModelFactory) {
    private val _name = MutableLiveData<String>()
    val name: LiveData<String> get() = _name
    fun changeName(newName: String) {
        _name.value = newName
    }
}

La fonction composable utilisant la Factory:
@Composable
fun Editor(viewModelFactory: ViewModelFactory) {
    val name by remember { mutableStateOf(viewModelFactory.create().name) }
    TextField(value = name, onValueChange = {viewModelFactory.changeName(it)})
}
```

State Hoisting

Déplacer gestion état vers composant parent.

- Avantages**
- **Single Source of Truth** : état géré à un seul endroit
 - **Encapsulation** : enfant ne gère pas son état
 - **Interceptable** : événements peuvent être ignorés
 - **Partage** : état partageable entre composants
 - **Découplage** : facilite tests unitaires

```
@Composable (stateful)
fun EditorScreen() {
    var name by remember { mutableStateOf("") }
    EditorContent(name = name, onNameChange = {name = it})
}

@Composable (stateless)
fun EditorContent(name: String, onNameChange: (String) -> Unit) {
    Column(modifier = Modifier.fillMaxWidth(),
    horizontalAlignment = Alignment.CenterHorizontally) {
        Text("Bienvenue $name")
        TextField(value = name, onValueChange = onNameChange)
    }
}
```



Exemple stateless (enfant)

```
@Composable
fun CounterDisplay(
    count: Int,
    onIncrement: () -> Unit
) {
    Column {
        Text("Count: $count")
        Button(onClick = onIncrement) {
            Text("+1")
        }
    }
}
```

Exemple stateful (parent)

```
@Composable
fun CounterScreen() {
    var count by remember { mutableStateOf(0) }
    CounterDisplay(
        count = count,
        onIncrement = { count++ }
    )
}
```

⚠ Warning

Créer nouvelle instance avec copy() pour détecter changements (comparaison par référence).

Problème mise à jour LiveData

```
fun changePerson(name: String? = null, firstName:
String? = null) {
    val p = _person.value!!.copy() // COPIE
    nouvelle instance
    if(name != null) p.name = name
    if(firstName != null) p.firstName = firstName
    _person.value = p // setter direct (pas
    postValue pour sync)
}
```

StateFlow

Avantages sur LiveData avec Compose

- Meilleure intégration avec coroutines Kotlin
- Gestion états plus fine
- Meilleures performances
- API flux réactif

Recommandé à la place de LiveData avec Compose.

ViewModel

```
class PersonViewModel : ViewModel() {
    private val _person =
MutableStateFlow<Person>()
    val person: StateFlow<Person> =
_person.asStateFlow()
```

```
fun changePerson(name: String? = null,
firstName: String? = null) {
    _person.update { currentPerson ->
currentPerson.copy(
        name = name ?: currentPerson.name,
        firstName = firstName ?:
currentPerson.firstName
    )
}
```

```
}
}

Room avec StateFlow
// DAO
@Query("SELECT * FROM Contact")
fun getAllContacts(): Flow<List<Contact>>

// Repository
val allContacts: Flow<List<Contact>> =
contactsDao.getAllContacts()

// ViewModel
val allContacts: StateFlow<List<Contact>> =
repository.allContacts
.stateIn(
    scope = viewModelScope,
    started =
SharingStarted.WhileSubscribed(5000L),
    initialValue = emptyList()
)
```

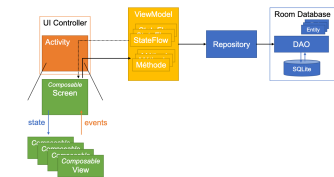
```
// Compose
val contacts by
contactViewModel.allContacts.collectAsStateWithLifecycle()
```

```
// ViewModel
val allContacts: StateFlow<List<Contact>> =
repository.allContacts
.stateIn(
    scope = viewModelScope,
    started =
SharingStarted.WhileSubscribed(5000L),
    initialValue = emptyList()
)
```

```
// Compose
val contacts by
contactViewModel.allContacts.collectAsStateWithLifecycle()
```

SharingStarted.WhileSubscribed

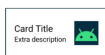
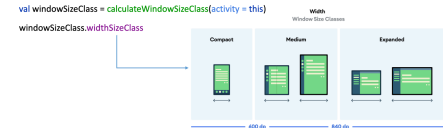
- Démarre Flow quand collecteurs actifs
- Arrête après 5000ms sans collecteurs
- Économie ressources



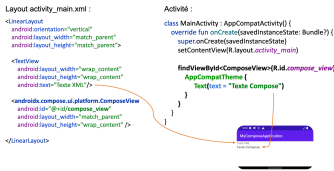
Layout adaptatif

BoxWithConstraints

```
@Composable
fun Card() {
    BoxWithConstraints {
        if (maxWidth < 400.dp) {
            Column { /* Layout mobile */ }
        } else { /* Layout tablette */ }
    }
}
```



- UI Hybride
- Possible mais non recommandé
 - ComposeView : Compose dans XML
 - AndroidView : XML dans Compose
 - Usage : migration progressive



Tests automatisés

3 types de tests

- **Tests unitaires** : classes/méthodes individuelles (rapides)
- **Tests d'intégration** : interactions composants (moyens)
- **Tests bout en bout/système** : comportement utilisateur (lents)

Pyramide des tests

- Beaucoup de tests unitaires (base)
- Moyennement de tests d'intégration (milieu)
- Peu de tests E2E (sommet)

Tests unitaires (JUnit)

Structure de base

```
import org.junit.Test
import org.junit.Assert.*
import org.junit.Before
import org.junit.After
```

```
class CalculatorTest {
    private lateinit var calculator: Calculator

    @Before
    fun setUp() {
        calculator = Calculator()
        // Initialisation avant chaque test
    }

    @After
    fun tearDown() {
        // Nettoyage après chaque test
    }

    @Test
    fun testPow() {
        assertEquals(8.0, calculator.pow(2.0, 3.0),
0.001)
        assertEquals(1.0, calculator.pow(5.0, 0.0),
0.001)
    }

    @Test
    fun testFactorial() {
        assertEquals(120, calculator.factorial(5))
        assertEquals(1, calculator.factorial(0))
    }
}
```

Annotations principales

- @Test : marque une méthode de test
- @Before : exécuté avant chaque test (setUp)
- @After : exécuté après chaque test (tearDown)

Tests instrumentalisés

Configuration de base

```
@RunWith(AndroidJUnit4::class)
class MyInstrumentedTest : TestCase() {
```

```
    @Before
    public override fun setUp() {
        // Initialisation
    }

    @After
    public override fun tearDown() {
        // Nettoyage
    }
}
```

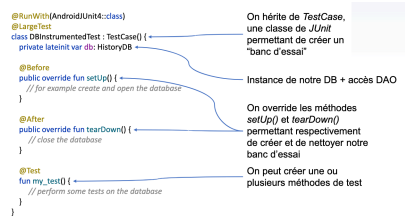
Éléments clés

- @RunWith(AndroidJUnit4::class) : runner Android
- Héritage de TestCase : structure banc d'essai
- @Before setUp() : init
- @After tearDown() : nettoyage

Tests Room

Entity exemple :

```
@Entity(tableName = "history", indices =
[Index("date")])
data class History(
    @PrimaryKey(autoGenerate = true) var id: Long?
= null,
    var expression: String,
    var result: Long? = null,
    var date: Date? = Date()
)
```



```
@RunWith(AndroidJUnit4::class)
class HistoryDAOTest : TestCase() {
    private lateinit var db: HistoryDB
    private lateinit var dao: HistoryDAO
```

```
    @Before
    public override fun setUp() {
        val context =
ApplicationProvider.getApplicationContext<Context>()
        db = Room.inMemoryDatabaseBuilder(
            context,
            HistoryDB::class.java
        ).build()
        dao = db.historyDao()
    }

    @After
    public override fun tearDown() {
        db.close()
    }

    @Test
    fun insertAndRetrieve() {
        val history = History(expression = "2+2",
result = 4)
        dao.insert(history)
        val all = dao.fullHistory().getOrAwaitValue()
        assertEquals(1, all.size)
        assertEquals("2+2", all[0].expression)
    }
}
```

Room.inMemoryDatabaseBuilder crée DB en mémoire pour tests (pas de persistance).

Tests Compose

Configuration

```
@get:Rule
val composeTestRule = createComposeRule()
```

Tests basés texte (limité)

```
@Test
fun editorScreenTest() {
    composeTestRule.setContent {
        EditorScreen(emptyTestPerson)
    }
    composeTestRule.onNodeWithText("Name").performTextInput(name)
    composeTestRule.onNodeWithText("Bienvenue",
substring = true)
        .assertTextEquals("Bienvenue $fname $name !")
}
```

Limitations texte

- Sensible refactoring
- Problèmes traduction
- Ambiguïté éléments multiples

Solution : testTag

```
// Composable
Text(
    modifier = Modifier.testTag("welcome-msg"),
    text = "Bienvenue"
)

// Test
composeTestRule.onNodeWithTag("welcome-msg")
    .assertTextEquals("Bienvenue Jean Neige !")
```

Cheat-sheet : <https://developer.android.com/jetpack/compose/testing-cheatsheet>

CI/CD

Prérequis

- SDK Android
- Émulateur pour tests instrumentalisés

Docker

1. Image openjdk:11-jdk (base Java)
2. Télécharger SDK Android
3. Accepter licences CLI (sdkmanager --licenses)
4. Télécharger versions nécessaires (platforms, build-tools)
5. Gradle : app:lintDebug, app:assembleDebug, app:testDebug

Tutoriel : <https://howtodoandroid.com/setup-gitlab-ci-android/>

Configuration émulateur

- Installation CLI automatisable
- Accélération matérielle requise (config hôte)
- Options CI : --no-audio --no-windows
- Démarrage : plusieurs minutes (monitoring avec adb)
- Désactiver animations avant tests (requis)

Commandes

```
# Lancer tests sur émulateur
gradlew app:connectedAndroidTest
```

```
# Désactiver animations
adb shell settings put global
window_animation_scale 0
adb shell settings put global
transition_animation_scale 0
adb shell settings put global
animator_duration_scale 0
```

⚠ Warning

Ne pas oublier de stopper l'émulateur après tests pour libérer ressources.

Tests supplémentaires

Monkey Testing (Play Store)

Installation + interactions aléatoires

- Identifiants pour connexion
- Tests multi-appareils
- Vérification accessibilité
- Détection failles sécurité

Firebase Robo Tests

- Cartographie automatique app
- Captures écran/vidéos
- Logs détaillés
- Profilage
- Vérification niveaux API
- Version gratuite limitée
- Intégration CI/CD possible