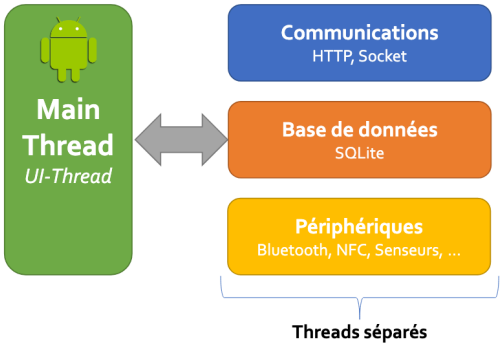
Guillaume Trüeb
Résumé DAA TE2
06-01-2026

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 `suspended` permet d'écrire du **code asynchrone** de manière **séquentielle**, facilitant la lecture et la maintenance.

> ⚠ **Warning**
>
> Suspend = sucre syntaxique, compilateur convertit en callbacks (Continuation).

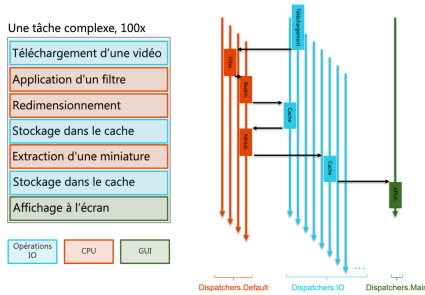
Exemple :

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

Dispatchers (contextes d'exécution) Associe une tâche à un thread ou pool de threads spécifique.

- **`Dispatchers.Main`** : UI thread, interactions UI
- **`Dispatchers.IO`** : I/O (réseau, fichiers, DB) - max 64 threads dynamiques
- **`Dispatchers.Default`** : calculs CPU intensifs - nbr threads = nbr coeurs

> ⚠ **Warning**
>
> Toujours utiliser `withContext(Dispatchers.IO)` pour I/O, jamais sur Main.




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

Le téléchargement de l'image dans le dispatcher *IO*

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

Le décodage de l'image dans le dispatcher "*CPU*"

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

L'affichage de l'image dans l'*UI-Thread*

Dispatcher custom

```
val myDispatcher = Executors
  .newFixedThreadPool(4)
  .asCoroutineDispatcher()
```

Pour des besoins spécifiques (ex: limiter la concurrence).

withContext

```
suspend fun downloadImage(url: URL): Bitmap? =
  withContext(Dispatchers.IO) {
    BitmapFactory.decodeStream(
      url.openConnection().getInputStream()
    )
  }
```

`withContext` change le dispatcher uniquement pour le bloc de code.

Scopes Les scopes permettent de limiter la durée de vie des coroutines à un contexte spécifique.

- **`GlobalScope`** : scope application (éviter, memory leaks)
- **`lifecycleScope`** : lié Activity/Fragment, auto-stop
- **`viewModelScope`** : lié ViewModel

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

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

Permet de convertir des APIs basées sur callbacks en fonctions suspensives.

```
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)
  }
```

Fonctionnement :
- Suspend la coroutine
- Fournit `Continuation` pour reprendre
- `cont.resume(value)` : reprend avec résultat
- `cont.resumeWithException(e)` : reprend avec erreur

> **⚠ Warning**
>
> Ne jamais appeler `resume()` ou `resumeWithException()` plusieurs fois.

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** :

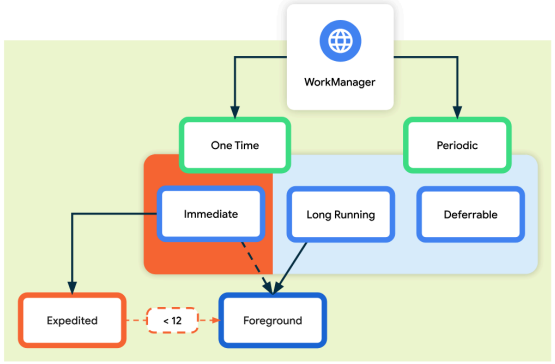
```
while(value > 0 && isActive) {
  // travail
}
```

3. **yield() pour coopération** :

```
while(value > 0) {
  yield()  // donne la main et vérifie annulation
  // travail
}
```

WorkManager

Usage
- Tâches longues/périodiques garanties
- Survit fermeture app et redémarrage
- Contraintes : réseau, batterie, stockage



Mode Doze
- Économie batterie : tâches différées
- Fenêtres activité contrôlées par système
- Classification apps : Active, Working set, Frequent, Rare, Restricted
- Sortie périodique pour sync données (fréquence variable)

Contraintes WorkManager

```
val constraints = Constraints.Builder()
  .setRequiredNetworkType(NetworkType.CONNECTED)
  .setRequiresBatteryNotLow(true)
  .setRequiresCharging(false)
  .build()

val workRequest = PeriodicWorkRequestBuilder<MyWork>(15,
TimeUnit.MINUTES)
  .setConstraints(constraints)
  .build()
```

Implémentation

```
class MyWork(appContext: Context, workerParams: WorkerParameters) :
Worker(appContext, workerParams) {
  override fun doWork(): Result {
    // tâche
    return Result.success()  // ou Result.failure() / Result.retry()
  }
}

val workManager = WorkManager.getInstance(applicationContext)
val myWorkRequest = OneTimeWorkRequestBuilder<MyWork>().build()
workManager.enqueue(myWorkRequest)
```

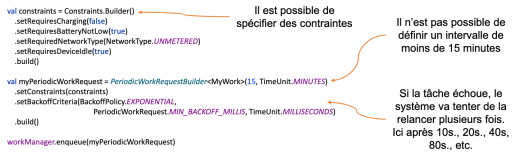
Types de tâches
- **Immédiates** : exécution ASAP (peut être différée par système)
- **Longue durée** : > 10 minutes (notification obligatoire)
- **Différables** : programmées/périodiques (intervalle min 15min)

> **⚠ Warning**
>
> Fréquence minimale pour PeriodicWorkRequest : 15 minutes.

App Standby Buckets (API 28+) Classification selon utilisation :
- **Active** : utilisée actuellement
- **Working set** : utilisée régulièrement
- **Frequent** : utilisée fréquemment mais pas quotidien
- **Rare** : rarement utilisée
- **Restricted** : consomme beaucoup de ressources

Plus une app est utilisée, moins elle est restreinte.

App Hibernation (API 30+) Après plusieurs mois sans interaction :
- Révocation permissions runtime
- Arrêt tâches programmées
- Arrêt notifications push (FCM)
- Vidage caches



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

- Gestion «manuelle» des threads
- On utilise *readText* car on s'attend à recevoir du texte
- Pas de gestion des erreurs

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

- 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,0, bytes.size)
  runOnUiThread { imageView.setImageBitmap(bmp) }
}
```

- On utilise *readBytes* pour obtenir le payload brut
- L'image est décodée puis affichée

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)
```

- 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>>() {}.type
  val fruits = Gson().fromJson<List<FruitDTO>>(json, type)
  runOnUiThread { txtView.text = fruits.toString() }
}
```

- On utilise *readText* pour obtenir le *json* brut, puis *Gson* pour le désérialiser

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

- L'endpoint retournant un tableau, on doit utiliser une ***JsonArrayRequest***
- 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)

```kotlin
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(Json {
            ignoreUnknownKeys = true
            prettyPrint = true
            isLenient = true
        })
    }
}
```

```kotlin
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()
        ...
    }
}
```

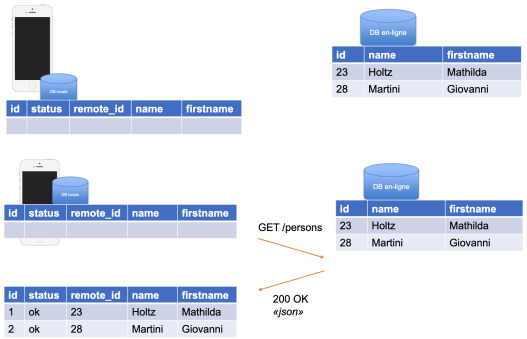
Comparaison des approches
- **java.net.URL** : bas niveau, verbeux, synchrone
- **Volley** : callbacks, cache, retry auto, queue requêtes
- **Ktor** : coroutines natives, type-safe, moderne, recommandé

Synchronisation données

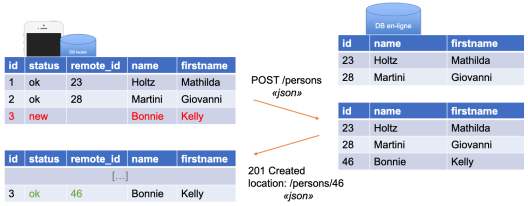
Contexte
- App fonctionne hors-ligne
- DB locale synchronisée avec serveur
- Algorithme sync complexe (conflits multiples devices)

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

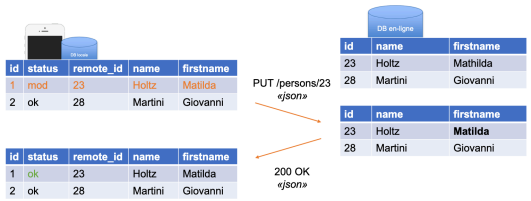
Cycle de synchronisation
1. Fetch : télécharger nouvelles données serveur
2. Apply : appliquer changements locaux vers serveur
3. Update : mettre à jour status et remoteId



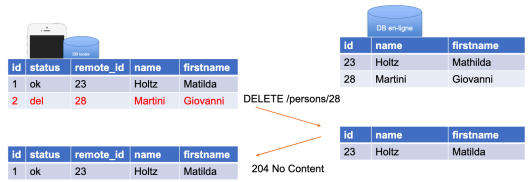
Création locale
```
status = new, remoteId = null
→ Sync : POST → remoteId reçu, status = ok
```



Modification
```
status = mod
→ Sync : PUT
```



Suppression
```
status = del
→ Sync : DELETE
```



Single Source of Truth (SSOT)
- DB locale = source unique de vérité
- Repository gère sync avec serveur
- UI observe DB uniquement (jamais réseau direct)
- Pattern offline-first (app fonctionne sans réseau)

Flux de données
```
UI ← observe ← ViewModel ← Repository ← DB locale
                                ↓ sync ↑
                            Serveur distant
```

Résolution conflits
- Mécanisme avancé si multi-devices modifient données
- Timestamps, versions, merge strategies
- Dépend du contexte métier

Exemple Repository
```kotlin
class Repository(private val dao: PersonDAO) {
  fun insert(person: Person) {
    person.status = "new"
    dao.insert(person)
    lifecycleScope.launch {
      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
```kotlin
@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

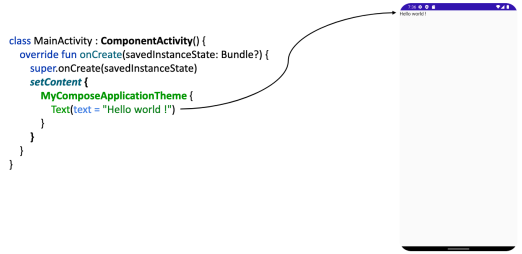
Solutions avancées :
- Timestamps + dernière modification gagne
- Versioning (ETags HTTP)
- Résolution manuelle utilisateur
- Operational Transform (Google Docs)
- CRDTs (Conflict-free Replicated Data Types)

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
- Idempotente (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")
  }
}
```

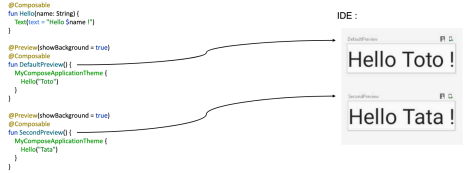
Différence stateful vs stateless

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

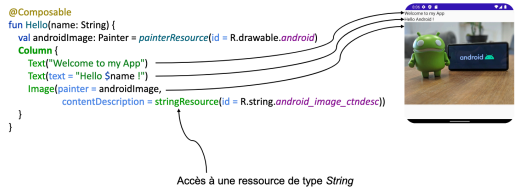
```
@Preview
@Composable
fun Hello(name: String = "World") {
  Text(text = "Hello $name!")
}
```



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 FrameLayout).

ConstraintLayout

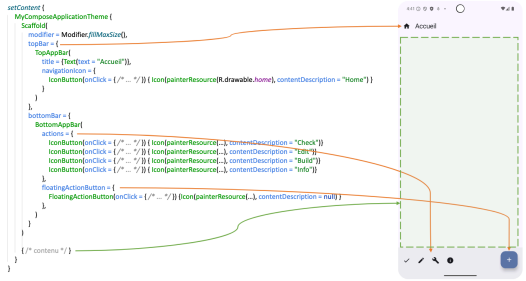
Nécessite définir identifiants pour références :

```
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

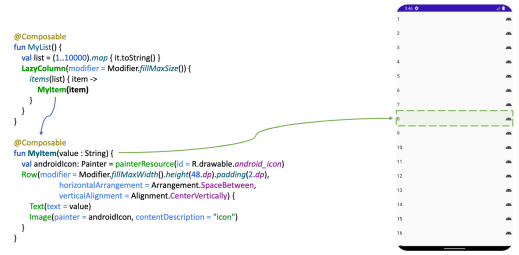
Scaffold

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



Listes paresseuses
- `LazyColumn` : liste verticale scrollable
- `LazyRow` : liste horizontale scrollable
- `LazyVerticalGrid` : grille verticale scrollable

Différence avec RecyclerView
- Affichent uniquement éléments visibles (performances)
- Pas de recyclage : recomposition systématique
- Plus simple à implémenter que RecyclerView



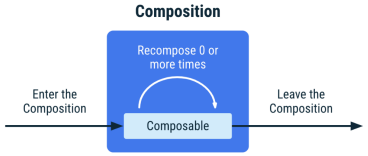
Gestion événements

```
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.



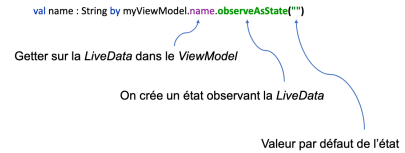
Compteur

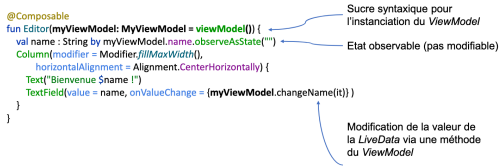
```
@Composable
fun Counter() {
  var counter by remember { mutableStateOf(0) }
  Button(onClick = { ++counter }) {
    Text("+")
  }
  Text("$counter")
}
```

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

Le *ViewModel* et sa *Factory* :

```
class MyViewModel(initialName: String) : ViewModel() {
  private val _name = MutableLiveData(initialName)

  val name : LiveData<String> get() = _name

  fun changeName(newName : String) {
    _name.value = newName
  }
}

class MyViewModelFactory(private val initialName: String) : ViewModelProvider.Factory {
  override fun <T : ViewModel> create(modelClass: Class<T>): T {
    if(modelClass.isAssignableFrom(MyViewModel::class.java))
      return MyViewModel(initialName) as T
    throw IllegalArgumentException("Unknown ViewModel class")
  }
}
```

La fonction composable utilisant la *Factory* :

```
@Composable
fun Editor(myViewModel: MyViewModel = viewModel(factory = MyViewModelFactory("Toto"))) {
  val name : String by myViewModel.name.observeAsState("")
  Column(modifier = Modifier.fillMaxWidth(),
      horizontalAlignment = Alignment.CenterHorizontally) {
    Text("Bienvenue $name !")
    TextField(value = name, onValueChange = {myViewModel.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

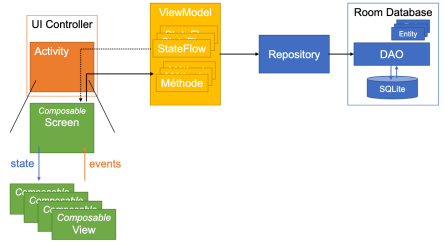
```
class PersonViewModel : ViewModel() {
  private val _person = MutableStateFlow(Person("", ""))
  val person: StateFlow<Person> = _person.asStateFlow()

  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()
```

SharingStarted.WhileSubscribed

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

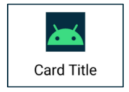


Layout adaptatif

BoxWithConstraints

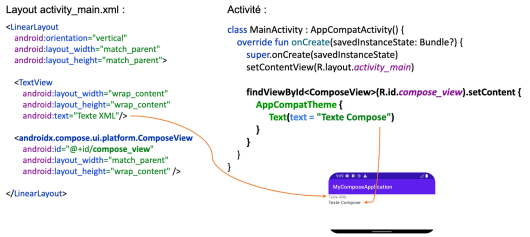
```
@Composable
fun Card() {
  BoxWithConstraints {
    if (maxWidth < 400.dp) {
      Column { /* Layout mobile */ }
    } else {
      Row { /* Layout tablette */ }
    }
  }
}
```

```
val windowSizeClass = calculateWindowSizeClass(activity = this)
windowSizeClass.widthSizeClass
```



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)

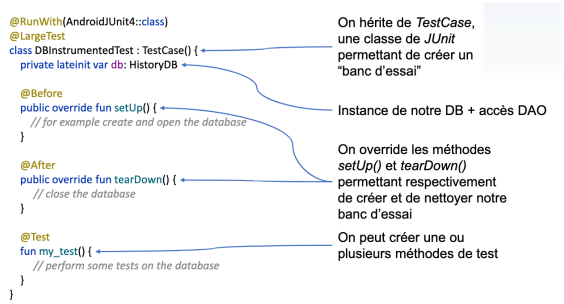
Tests instrumentalisés

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