

Développement Android avec Kotlin

Cours - 10 - Tâches en arrière-plan

Jordan Hiertz

Contact

hiertzjordan@gmail.com

jordan.hiertz@al-enterprise.com



Introduction : Pourquoi les tâches en arrière-plan ?

- **Évitez de bloquer le thread principal (UI)**

- L'application doit rester fluide et réactive.
- Les opérations longues ou bloquantes peuvent entraîner des ralentissements ou des ANR (Application Not Responding).

- **Quelles tâches exécuter en arrière-plan ?**

- Requêtes réseau
- Décodage d'images (bitmaps)
- Accès à l'espace de stockage
- Exécution de modèles ML

- **Objectif**

- Déléguer ces tâches à des solutions adaptées : **Threads, Coroutines, Services, WorkManager, JobScheduler...**



Sommaire

- **Services**

- Services en arrière-plan & premier plan
- Services liés

- **Tâches en arrière-plan**

- Garder l'appareil activité (WakeLock, Doze mode)
- Tâches asynchrones (Threads Java, Coroutines)
- Tâches persistantes (WorkManager, JobScheduler)



Qu'est-ce qu'un Service ?

Un composant d'application pour les tâches longue durée en arrière-plan.

- **Aucune interface utilisateur**
- Peut continuer à s'exécuter même si l'utilisateur quitte l'application (pendant un certain temps)
- Peut être **lié** à un autre composant pour interagir avec lui (**C**ommunication **I**nter-**P**rocessus)
- **Cas d'usage** : transactions réseau, streaming musical, opérations sur les fichiers etc.

⚠ Attention : Un service s'exécute dans le thread principal de son processus, il ne crée pas son propre thread et ne s'exécute pas dans un processus distinct, sauf si vous le spécifiez. Il faut exécuter les opérations bloquantes sur un thread distinct dans le service pour éviter les erreurs ANR.



Les types de services en Android

1 Service de premier plan (Foreground Service)

- Effectue une tâche **visible** pour l'utilisateur (ex : streaming audio).
- **Obligation d'afficher une notification** persistante.
- Continue de s'exécuter même si l'utilisateur quitte l'application.

2 Service en arrière-plan (Background Service)

- Effectue une tâche **non visible** par l'utilisateur (ex : nettoyage de stockage).
- Restrictions à partir d'Android 8 (API 26) lorsque l'application est inactive.

3 Service lié (Bound Service)

- Permet aux composants de l'application de **s'y lier** via `bindService()`.
- Offre une **interface client-serveur** pour envoyer des requêtes et récupérer des résultats.
- **S'arrête automatiquement** lorsque plus aucun composant n'y est lié.



Déclaration d'un Service de Premier Plan

Déclarer le service dans le manifeste

- Ajoutez un élément `<service>` dans le fichier `AndroidManifest.xml` :

```
1 <manifest xmlns:android="http://schemas.android.com/apk/res/android" ...>
2   <application ...>
3
4     <service
5       android:name=".MyMediaPlayerService"
6       android:foregroundServiceType="mediaPlayback"
7       android:exported="false">
8     </service>
9   </application>
10 </manifest>
```

```
1 android:foregroundServiceType="camera|microphone" <!-- location, dataSync, donwload, upload, health, phoneCall -->
```



Déclaration d'un Service de Premier Plan

Ajouter la permission de base (obligatoire pour tous les services de premier plan)

```
1 <manifest xmlns:android="http://schemas.android.com/apk/res/android" ...>
2
3     <uses-permission android:name="android.permission.FOREGROUND_SERVICE" />
4
5     <application ...>
6         ...
7     </application>
8 </manifest>
```

⚠ Depuis Android 14, certains types de services nécessitent des permissions dédiées :

```
1 <uses-permission android:name="android.permission.FOREGROUND_SERVICE_MEDIA_PLAYBACK" />
2 <uses-permission android:name="android.permission.FOREGROUND_SERVICE_MICROPHONE" />
3 <uses-permission android:name="android.permission.FOREGROUND_SERVICE_MEDIA_PROJECTION" />
4 <uses-permission android:name="android.permission.FOREGROUND_SERVICE_CAMERA" />
5 <uses-permission android:name="android.permission.FOREGROUND_SERVICE_LOCATION" />
```



Déclaration d'un Service de Premier Plan

```
1 class MyMediaPlayerService : Service() {
2
3     override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {
4         val notification = NotificationCompat.Builder(this, CHANNEL_ID)
5             // Créer la notification à afficher quand le service tourne
6             .build()
7
8         // Démarrer le service en tant que service de premier plan
9         if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.Q) {
10             startForeground(SERVICE_ID, notification, ServiceInfo.FOREGROUND_SERVICE_TYPE_MEDIA_PLAYBACK)
11         } else {
12             startForeground(SERVICE_ID, notification)
13         }
14         // Si on tente de démarrer le service en premier plan alors que l'application n'y est pas, une exception est levée
15         // Lancer la tâche de lecture audio ici (ex: démarrer un lecteur)
16
17         // Retourner un comportement approprié
18         return START_STICKY
19     }
20
21     override fun onBind(intent: Intent?): IBinder? {
22         return null // Ce service ne permet pas la liaison (return null)
23     }
24
25     override fun onDestroy() {
26         super.onDestroy() // Arrêter la lecture audio et libérer les ressources
27     }
28
29     companion object {
30         const val CHANNEL_ID = "music_channel"
31         const val SERVICE_ID = 1
32     }
33 }
```


Déclaration d'un Service de Premier Plan

Retour	Comportement du service après la fermeture
<code>START_STICKY</code>	Redémarre, conserve la dernière Intent (ou <code>null</code> si aucune)
<code>START_NOT_STICKY</code>	Ne redémarre pas après la fermeture
<code>START_REDELIVER_INTENT</code>	Redémarre et redélivre l'Intent à <code>onStartCommand()</code>



Démarrer un Service de Premier Plan

Dans une activité ou un autre composant :

```
1 val intent = Intent(this, MyMediaPlayBackService::class.java)
2 startForegroundService(intent)
```

Bonnes pratiques :

- **Toujours vérifier les autorisations nécessaires** avant de démarrer le service, en particulier pour Android 14+ (par exemple, permissions de la caméra, localisation, etc.).
- **Utiliser des notifications visibles** pour les services de premier plan.



Arrêter un Service de Premier Plan

Dans une activité ou un autre composant :

```
1 val intent = Intent(this, MyMediaPlayerService::class.java)
2 stopService(intent)
```

Depuis le Service :

```
1 // Retirer le service de premier plan (supprimer la notification)
2 stopForeground(STOP_FOREGROUND_REMOVE)
3
4 // Arrêter le service après avoir effectué ses opérations
5 stopSelf() // Arrête le service une fois qu'il a terminé
```



Conclusion sur les Services

Les Services permettent d'exécuter des tâches en arrière-plan indépendamment de l'interface utilisateur.

Types de Services :

- **Background Service** : Exécute des tâches en arrière-plan sans interaction avec l'utilisateur.
- **Foreground Service** : Montre une **notification persistante** pour les tâches de longue durée.
- **Bound Service** : Permet la communication avec d'autres composants via un **Binder**.



Introduction aux Tâches en Arrière-Plan

Pourquoi utiliser des tâches en arrière-plan ?

- Éviter les blocages du thread principal (UI)
- Améliorer la réactivité et l'expérience utilisateur
- Optimiser la gestion des ressources (batterie, CPU, réseau)

Choisir la bonne API est crucial !

- Mauvais choix = consommation excessive de batterie & restrictions système
- Peut affecter la présence de l'application sur le Play Store



Terminologie et Catégories des Tâches

Définition d'une application en arrière-plan :

*// Une application est en arrière-plan lorsque **aucune activité visible** n'est présente **et** qu'aucun service de premier plan n'est en cours d'exécution.*

Quand une application s'exécute en arrière-plan, le système lui impose des restrictions :

- Interdit de démarrer un service de premier plan.
- Restrictions sur les capteurs (GPS, gyroscope, etc.).
- Requêtes réseau **retardées** ou **regroupées**.

Le terme "tâche" désigne une opération effectuée en dehors du workflow principal.



Choisir la bonne option pour gérer la tâche

Dans la plupart des cas on peut déterminer l'API à utiliser en identifiant la catégorie de la tâche :

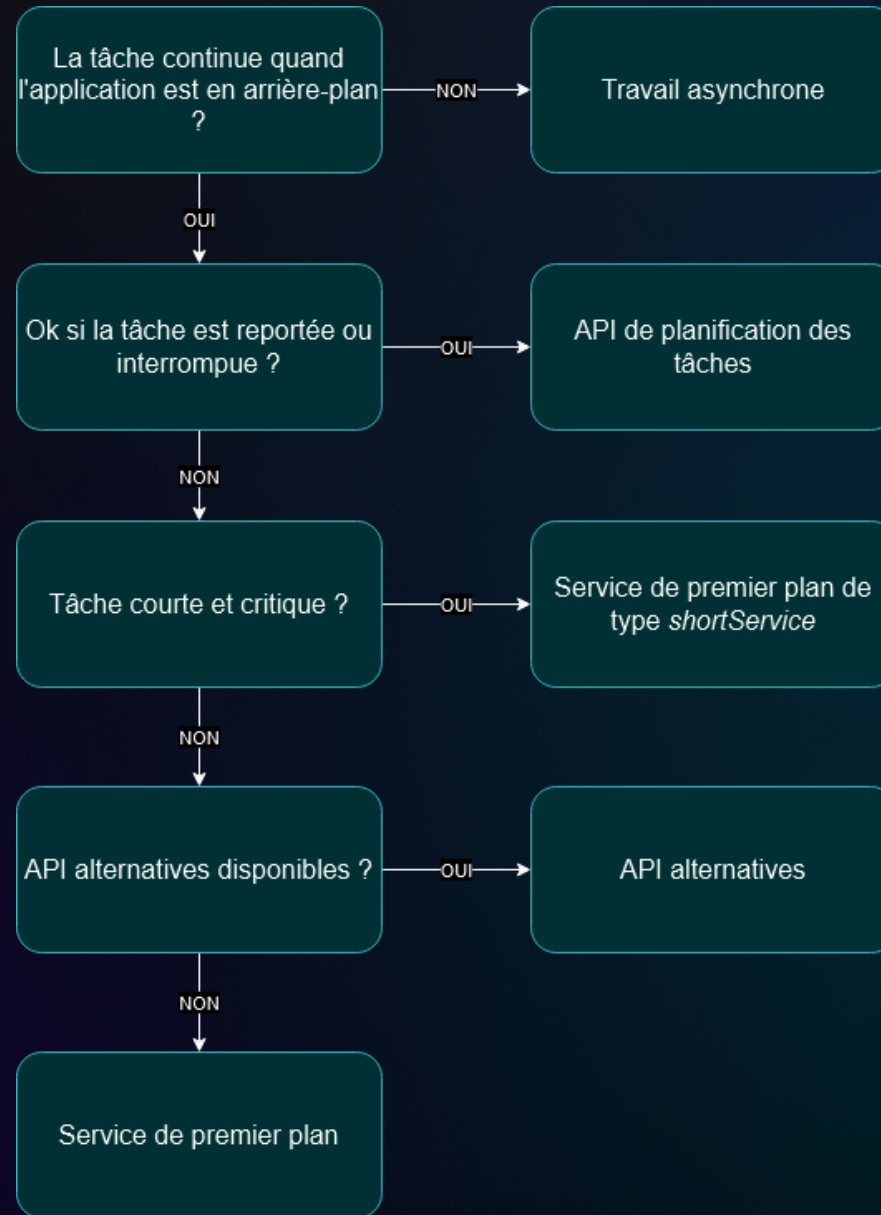
- Travail asynchrone
- API de planification des tâches
- Service de premier plan

Sinon se référer aux organigrammes fournis par Android...



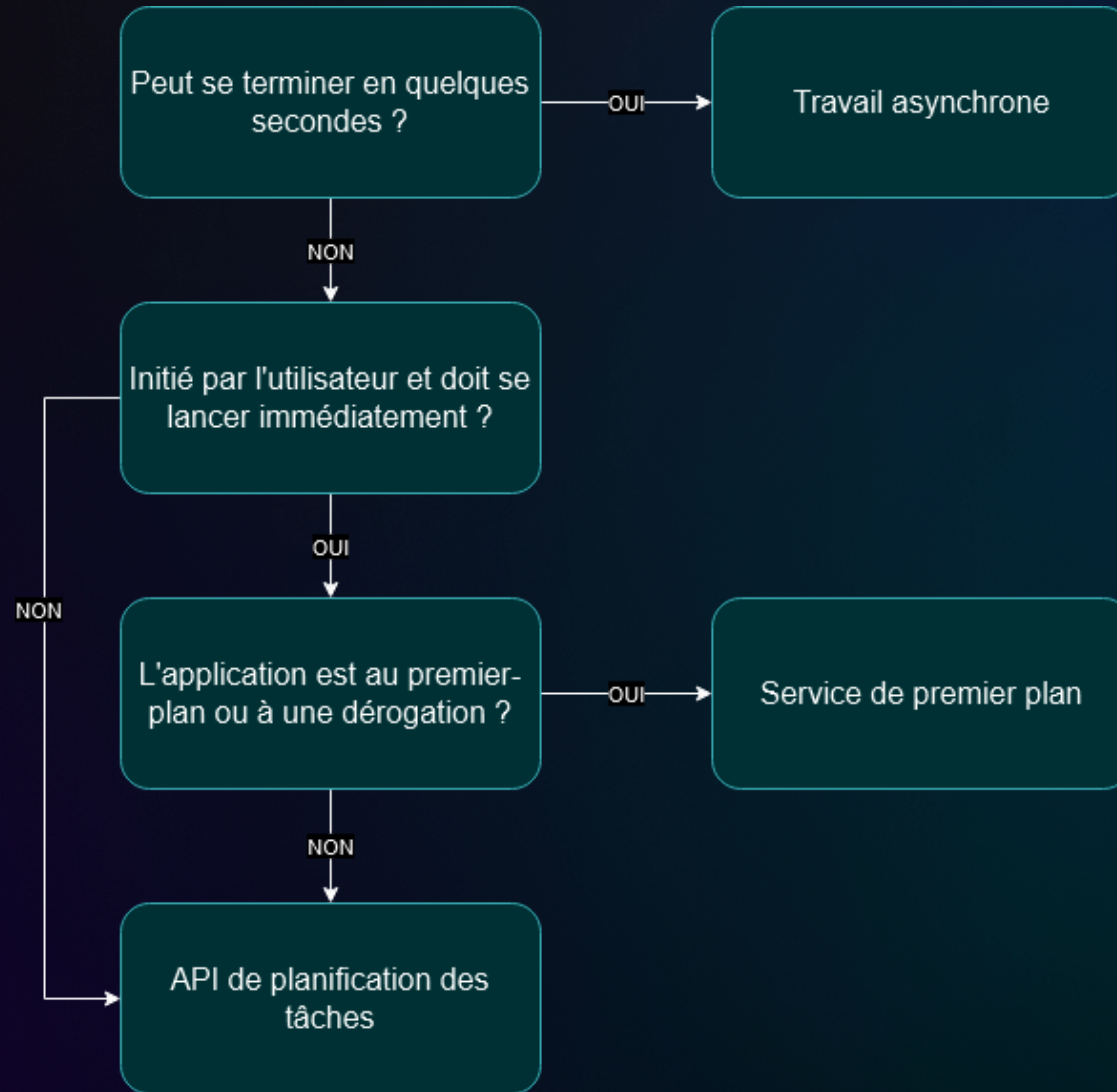
Choisir la bonne option pour gérer la tâche

Tâches déclenchées par l'utilisateur



Choisir la bonne option pour gérer la tâche

Tâches en réponse à un évènement



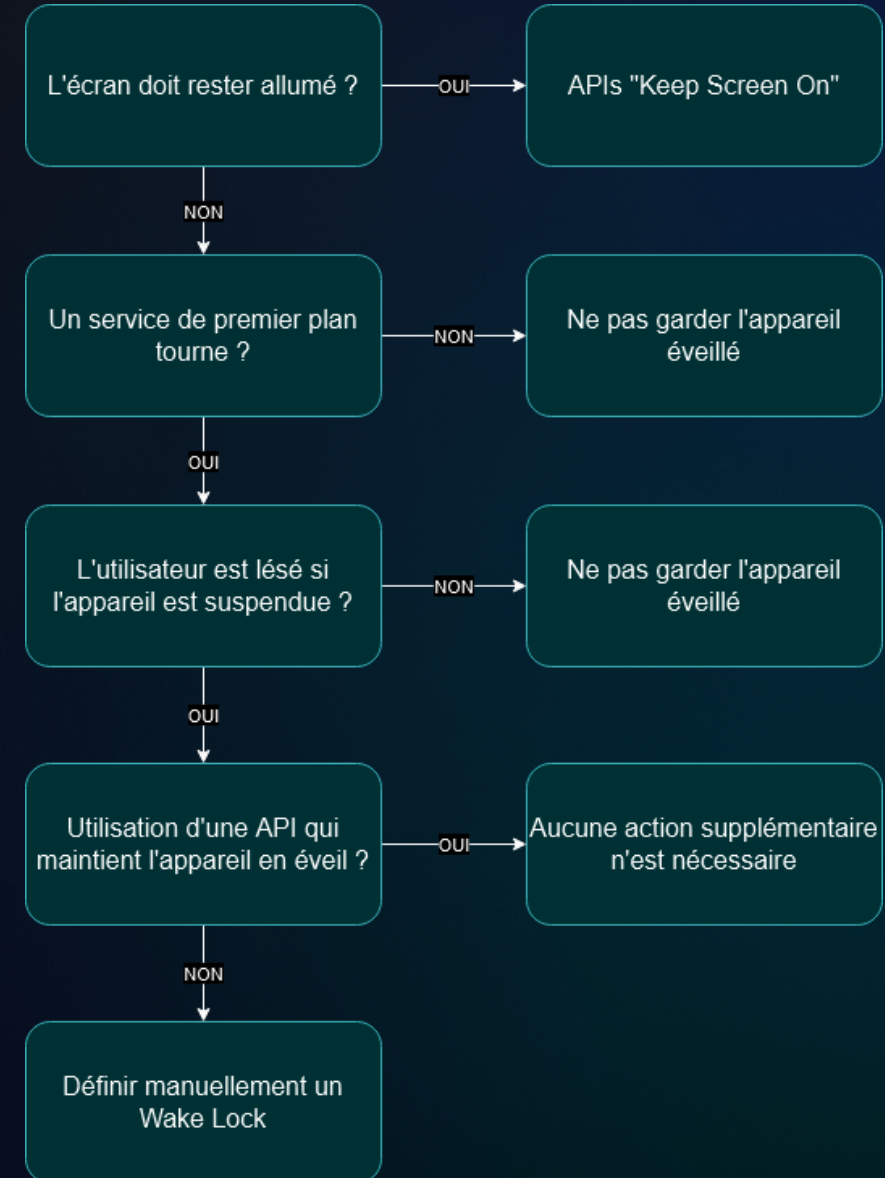
Maintenir l'appareil activé

Pourquoi empêcher l'appareil de se mettre en veille ?

- Assurer la continuité d'une tâche importante (ex : streaming vidéo, navigation GPS)
- Éviter que le processeur ne se mette en veille pendant une tâche critique

Les options disponibles :

- **Maintenir l'écran allumé** (utile pour les applications interactives)
- **Empêcher le processeur de s'endormir (Wake Lock)**



Maintenir l'écran allumé

Remarque : Laisser l'écran de l'appareil allumé peut décharger rapidement la batterie. En règle générale, vous devez laisser l'appareil éteindre l'écran si l'utilisateur n'interagit pas avec lui. Si vous devez laisser l'écran allumé, faites-le pendant le moins de temps possible.

```
1 class MainActivity : Activity() {  
2  
3     override fun onCreate(savedInstanceState: Bundle?) {  
4         super.onCreate(savedInstanceState)  
5         setContentView(R.layout.activity_main)  
6         window.addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON)  
7     }  
8 }
```

```
1 @Composable  
2 fun KeepScreenOn() {  
3     val window = LocalWindow.current  
4  
5     DisposableEffect(window) {  
6         window.addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON)  
7         onDispose {  
8             window.clearFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON)  
9         }  
10    }  
11 }
```



Définir un WakeLock pour maintenir l'appareil éveillé

Un **Wake Lock** permet d'empêcher un appareil Android de passer en veille pendant qu'une tâche critique est en cours.

⚠ Attention : Utiliser un Wake Lock peut **consommer beaucoup de batterie**. Toujours l'utiliser avec précaution et le libérer dès que possible.

```
1 <uses-permission android:name="android.permission.WAKE_LOCK" />
```

```
1 val wakeLock: PowerManager.WakeLock =  
2     (getSystemService(Context.POWER_SERVICE) as PowerManager).run {  
3         newWakeLock(PowerManager.PARTIAL_WAKE_LOCK, "MyApp::MyWakelockTag").apply {  
4             acquire(10*60*1000L /*10 minutes*/)   
5         }  
6     }  
7  
8     ...  
9  
10 wakeLock.release()
```



Restrictions système concernant les tâches en arrière-plan

Pourquoi ces restrictions ?

- Les processus en arrière-plan peuvent consommer **beaucoup de mémoire et de batterie**.
- Android impose donc **des règles strictes** pour limiter leur impact sur les performances et l'expérience utilisateur.

Si une application **abuse des ressources système**, Android peut **notifier l'utilisateur** et lui proposer de restreindre l'application.

Exemples de comportements limités :

- **Trop de wake locks** (ex : un verrou maintenu plus d'une heure écran éteint).
- **Trop de services en arrière-plan** (notamment si l'API cible < 26).

Conséquences possibles :

- Ne peut plus exécuter de Jobs ni déclencher des alarmes.
- Accès au réseau restreint.
- Sauf quand l'application est au premier-plan.





Traitement en arrière-plan asynchrone

Une tâche asynchrone est une tâche qui :

- S'exécute immédiatement
- Ne nécessite pas la persistance après le redémarrage de l'application ou de l'appareil.
- Se produit en dehors du thread principal (sinon elle bloque l'UI).

Différence avec le travail persistant (WorkManager)

 Tâche asynchrone	 Travail persistant
Exécutée immédiatement	Planifiée pour plus tard
Disparaît après exécution	Survit aux redémarrages
Ex: Requête réseau, lecture de fichier	Ex: Synchronisation de données quotidienne



Tâches asynchrones : Java vs Kotlin

En Android, le choix de la gestion des tâches asynchrones dépend du langage utilisé. Kotlin privilégie les coroutines tandis que Java repose sur les threads et les exécuteurs. Cependant, bien qu'une solution soit recommandée dans chaque langage, les deux approches fonctionnent dans les deux cas.

Langage	Solution recommandée	Remarque
Kotlin	Coroutines	Simple, concise et optimisée pour l'asynchronisme.
Java	Threads	Plus bas niveau, nécessite plus de gestion manuelle.



Tâches asynchrones - Les Threads en Android

- **Thread principal (Main Thread)** : Gère les opérations de l'interface utilisateur (UI).
- **Problème** : Les appels de longue durée (comme les requêtes réseau) ou les opérations coûteuses bloquent l'UI et peuvent provoquer des **ANR** (Application Not Responding).

Pool de Threads en Android

- Un **pool de threads** est une collection gérée de threads permettant d'exécuter des tâches en parallèle à partir d'une file d'attente.
- Les nouvelles tâches sont exécutées sur des threads existants, ce qui évite la création répétée de threads coûteux.

Pourquoi un Pool et pas un Thread à chaque travail asynchrone ?

- La création de threads est coûteuse en termes de performance.
- Utilisez un **ExecutorService** pour gérer un pool de threads et optimiser les performances de l'application.



Création d'un Pool de Threads avec ExecutorService

```
1 class MyApplication : Application() {  
2     val executorService: ExecutorService = Executors.newFixedThreadPool(4)  
3  
4     val executorService: ExecutorService = Executors.newSingleThreadExecutor()  
5  
6     val executorService: ExecutorService = Executors.newCachedThreadPool()  
7 }
```

- **FixedThreadPool** : Crée un pool avec un nombre fixe de threads (idéal pour des tâches constantes).
- **SingleThreadExecutor** : Crée un pool avec un seul thread, idéal pour des tâches séquentielles.
- **CachedThreadPool** : Crée de nouveaux threads au besoin, idéal pour des tâches courtes et ponctuelles.

Initialiser un `ExecutorService` dans la classe `Application` permet de centraliser et de rendre l'accès au pool de threads disponible tout au long du cycle de vie de l'application.



Exécuter dans un thread d'arrière-plan

```
1 // LoginRepository.kt
2 class LoginRepository(private val responseParser: LoginResponseParser) {
3
4     private val loginUrl = "https://example.com/login"
5
6     fun makeLoginRequest(jsonBody: String): Result<LoginResponse> { // méthode synchrone qui bloque le thread appelant
7         return try {
8             val url = URL(loginUrl)
9             val httpConnection = url.openConnection() as HttpURLConnection
10            httpConnection.requestMethod = "POST"
11            httpConnection.setRequestProperty("Content-Type", "application/json; charset=utf-8")
12            httpConnection.setRequestProperty("Accept", "application/json")
13            httpConnection.doOutput = true
14            httpConnection.outputStream.write(jsonBody.toByteArray(Charsets.UTF_8))
15
16            val loginResponse = responseParser.parse(httpConnection.inputStream)
17            Result.Success(loginResponse)
18        } catch (e: Exception) {
19            Result.Error<LoginResponse>(e)
20        }
21    }
22 }
```

```
1 // Result.kt
2 sealed class Result<out T> {
3     data class Success<out T>(val data: T) : Result<T>()
4     data class Error(val exception: Exception) : Result<Nothing>()
5 }
```



Exécuter dans un thread d'arrière-plan

```
1 // LoginRepository.kt
2 class LoginRepository(private val responseParser: LoginResponseParser, private val executor: Executor) {
3
4     fun makeLoginRequest(jsonBody: String) {
5         // Soumettre la tâche au pool de threads
6         executor.execute {
7             try {
8                 val url = URL(loginUrl)
9                 val httpConnection = url.openConnection() as HttpURLConnection
10                httpConnection.requestMethod = "POST"
11                httpConnection.setRequestProperty("Content-Type", "application/json; charset=utf-8")
12                httpConnection.setRequestProperty("Accept", "application/json")
13                httpConnection.doOutput = true
14                httpConnection.outputStream.write(jsonBody.toByteArray(Charsets.UTF_8))
15
16                val ignoredResponse = responseParser.parse(httpConnection.inputStream)
17            } catch (e: Exception) {
18                Result.Error<LoginResponse>(e)
19            }
20        }
21    }
22 }
```



Communiquer avec le thread principal

Dans les tâches asynchrones, une fois qu'une opération en arrière-plan est terminée (comme une requête réseau), vous devez informer le thread principal afin de mettre à jour l'interface utilisateur ou effectuer d'autres actions.

L'une des solutions les plus courantes pour cela est d'utiliser des **callbacks**. Un **callback** est une fonction qui est passée en paramètre et appelée lorsque la tâche en arrière-plan est terminée.

```
1 // Interface Callback
2 interface RepositoryCallback<T> {
3     fun onComplete(result: Result<T>)
4 }
5
6 // Repository - méthode asynchrone
7 class LoginRepository(private val executor: Executor) {
8     fun makeLoginRequest(jsonBody: String, callback: RepositoryCallback<LoginResponse>) {
9         executor.execute {
10             try {
11                 val result = makeSynchronousLoginRequest(jsonBody)
12                 callback.onComplete(result) // Appel du callback
13             } catch (e: Exception) {
14                 val errorResult = Result.Error<LoginResponse>(e)
15                 callback.onComplete(errorResult) // Appel du callback en cas d'erreur
16             }
17         }
18     }
19 }
```


Communiquer avec le thread principal

En Kotlin, au lieu d'utiliser une interface pour gérer le callback, vous pouvez passer directement une **lambda** comme paramètre. Cela simplifie le code tout en conservant les mêmes fonctionnalités asynchrones.

```
1 // Repository - méthode asynchrone avec lambda
2 class LoginRepository(private val executor: Executor) {
3     fun makeLoginRequest(jsonBody: String, callback: (Result<LoginResponse>) -> Unit) {
4         executor.execute {
5             try {
6                 val result = makeSynchronousLoginRequest(jsonBody)
7                 callback(result) // Appel du callback
8             } catch (e: Exception) {
9                 val errorResult = Result.Error<LoginResponse>(e)
10                callback(errorResult) // Appel du callback en cas d'erreur
11            }
12        }
13    }
14 }
```

Communiquer avec le thread principal

Dans un **ViewModel**, on peut facilement appeler la méthode de repository avec la lambda pour récupérer les résultats de manière asynchrone et mettre à jour l'interface utilisateur en fonction du résultat.

```
1 class LoginViewModel(private val loginRepository: LoginRepository) : ViewModel() {
2
3     fun makeLoginRequest(username: String, token: String) {
4         val jsonBody = "{ username: \"$username\", token: \"$token\" }"
5
6         loginRepository.makeLoginRequest(jsonBody) { result ->
7             when (result) {
8                 is Result.Success -> {
9                     // Traitement en cas de succès
10                    // Mettre à jour l'UI avec les données reçues
11                }
12                 is Result.Error -> {
13                     // Traitement en cas d'erreur
14                     // Afficher un message d'erreur à l'utilisateur
15                }
16            }
17        }
18    }
19 }
```

⚠ Dans cet exemple, le rappel est exécuté dans le thread appelant, qui est un thread d'arrière-plan. Cela signifie que vous ne pouvez pas modifier ni communiquer directement avec la couche d'UI jusqu'à ce que vous reveniez au thread principal.

Communiquer avec le thread principal - Utiliser un Handler

Un **Handler** permet de mettre en file d'attente des actions sur différents threads. En associant un **Looper** au thread cible, vous pouvez spécifier où exécuter l'action.

- Un **Looper** est un objet qui exécute la boucle de messages pour un thread associé.
- Utilisez la méthode `post(Runnable)` pour envoyer des tâches à un thread.

```
1 class MyApplication : Application() {  
2     val executorService: ExecutorService = Executors.newFixedThreadPool(4)  
3     val mainThreadHandler = HandlerCompat.createAsync(Looper.getMainLooper())  
4 }
```



Communiquer avec le thread principal - Utiliser un Handler

```
1 // Repository - méthode asynchrone avec lambda
2 class LoginRepository(private val executor: Executor, private val mainThreadHandler: Handler) {
3
4     fun makeLoginRequest(jsonBody: String, callback: (Result<LoginResponse>) -> Unit) {
5         executor.execute {
6             try {
7                 val result = makeSynchronousLoginRequest(jsonBody)
8
9                 mainThreadHandler.post {
10                     callback(result) // Appel du callback
11                 }
12             } catch (e: Exception) {
13                 val errorResult = Result.Error<LoginResponse>(e)
14                 mainThreadHandler.post {
15                     callback(errorResult) // Appel du callback en cas d'erreur
16                 }
17             }
18         }
19     }
20 }
21 }
```



Le "Callback Hell"

Lorsqu'on enchaîne de nombreuses opérations asynchrones qui nécessitent des callbacks imbriqués, le code devient rapidement difficile à lire, maintenir et déboguer. C'est ce qu'on appelle le **"Callback Hell"**.

```
1 repository.makeLoginRequest(username, token) { result ->
2     if (result is Result.Success) {
3         repository.fetchUserProfile(result.data.userId) { profileResult ->
4             if (profileResult is Result.Success) {
5                 repository.loadUserPreferences(profileResult.data.userId) { preferencesResult ->
6                     if (preferencesResult is Result.Success) {
7                         // Appliquer les préférences à l'UI
8                     } else {
9                         // Gérer l'erreur des préférences
10                    }
11                }
12            } else {
13                // Gérer l'erreur du profil
14            }
15        }
16    } else {
17        // Gérer l'erreur de la connexion
18    }
19 }
```



Threads, Callbacks et ExecutorService : Ce qu'il faut retenir

- Threads
 - Chaque thread exécute une tâche **indépendamment des autres threads**.
 - ⚠ **Créer trop de threads consomme beaucoup de ressources.**
- Callbacks
 - Permettent d'exécuter du code après la fin d'une tâche asynchrone.
 - ⚠ Peuvent mener au **callback hell** (imbriqué, difficile à lire).
- ExecutorService
 - Un pool de threads pour mieux gérer l'exécution des tâches.
 - Offre différentes stratégies : `FixedThreadPool`, `CachedThreadPool`, `SingleThreadExecutor`...
 - Permet de **limiter le nombre de threads** et d'optimiser les performances.

💡 Comprendre ces concepts est essentiel, même si en Android moderne, on privilégie les coroutines.



Tâches asynchrones - Les coroutines

“ Une coroutine est un modèle de conception de simultanéité que vous pouvez utiliser sur Android pour simplifier le code qui s'exécute de manière asynchrone. Coroutines ont été ajoutés à Kotlin dans la version 1.3 et sont basés sur des concepts d'autres langages.

Pourquoi utiliser les coroutines ?

- Éviter de **bloquer** le thread principal.
- Exécuter du **code asynchrone** de manière fluide.
- **Simplifier** le code et réduire l'imbrication des callbacks.
- Meilleure **gestion de la mémoire** et des erreurs.

Fonctionnalités clés

- **Légèreté** : plusieurs coroutines peuvent s'exécuter sur un même thread.
- **Moins de fuites mémoire** : grâce à la **simultanéité structurée**.
- **Résilience intégrée** : l'annulation d'une coroutine se propage automatiquement.
- **Intégration avec Jetpack** : LiveData, ViewModelScope, Room, WorkManager...



Tâches asynchrones - Les coroutines

```
1 // libs.catalog.toml
2
3 [versions]
4 kotlinxCoroutinesAndroid = "1.10.1"
5
6 [libraries]
7 kotlinx-coroutines-android = {
8     module = "org.jetbrains.kotlinx:kotlinx-coroutines-android",
9     version.ref = "kotlinxCoroutinesAndroid"
10 }
```

```
1 // build.gradle (app)
2
3 dependencies {
4
5     implementation(libs.kotlinx.coroutines.android)
6
7 }
```



Tâches asynchrones - Les coroutines

```
1 // LoginRepository.kt
2 class LoginRepository(private val responseParser: LoginResponseParser) {
3
4     private val loginUrl = "https://example.com/login"
5
6     fun makeLoginRequest(jsonBody: String): Result<LoginResponse> { // méthode synchrone qui bloque le thread appelant
7         return try {
8             val url = URL(loginUrl)
9             val httpConnection = url.openConnection() as HttpURLConnection
10            httpConnection.requestMethod = "POST"
11            httpConnection.setRequestProperty("Content-Type", "application/json; charset=utf-8")
12            httpConnection.setRequestProperty("Accept", "application/json")
13            httpConnection.doOutput = true
14            httpConnection.outputStream.write(jsonBody.toByteArray(Charsets.UTF_8))
15
16            val loginResponse = responseParser.parse(httpConnection.inputStream)
17            Result.Success(loginResponse)
18        } catch (e: Exception) {
19            Result.Error<LoginResponse>(e)
20        }
21    }
22 }
```

```
1 class LoginViewModel(
2     private val loginRepository: LoginRepository
3 ): ViewModel() {
4
5     fun login(username: String, token: String) {
6         val jsonBody = "{ username: \"$username\", token: \"$token\"}"
7         loginRepository.makeLoginRequest(jsonBody)
8     }
9 }
```

Tâches asynchrones - Les coroutines

```
1 // LoginRepository.kt
2 class LoginRepository(private val responseParser: LoginResponseParser) {
3
4     private val loginUrl = "https://example.com/login"
5
6     fun makeLoginRequest(jsonBody: String): Result<LoginResponse> { // méthode synchrone qui bloque le thread appelant
7         return makeLoginRequestSynchrone(jsonBody)
8     }
9 }
```

```
1 class LoginViewModel(
2     private val loginRepository: LoginRepository
3 ): ViewModel() {
4
5     fun login(username: String, token: String) {
6         // Create a new coroutine to move the execution off the UI thread
7         viewModelScope.launch(Dispatchers.IO) {
8             val jsonBody = "{ username: \"$username\", token: \"$token\"}"
9             loginRepository.makeLoginRequest(jsonBody)
10        }
11    }
12 }
```


Tâches asynchrones - Les coroutines

```
1 class LoginViewModel(  
2     private val loginRepository: LoginRepository  
3 ): ViewModel() {  
4  
5     fun login(username: String, token: String) {  
6         // Create a new coroutine to move the execution off the UI thread  
7         viewModelScope.launch(Dispatchers.IO) {  
8             val jsonBody = "{ username: \"$username\", token: \"$token\"}"  
9             loginRepository.makeLoginRequest(jsonBody)  
10        }  
11    }  
12 }
```

Comprendre les éléments clés :

- **viewModelScope** : un CoroutineScope intégré à ViewModel, qui annule automatiquement les coroutines lorsque le viewModel est détruit.
- **launch** : démarre une nouvelle coroutine.
- **Dispatchers.IO** : exécute la tâche sur un thread optimisé pour les opérations d'entrée/sortie (réseau, base de données, fichiers).

Déroulement de la fonction login()

- L'application appelle login() depuis l'UI (thread principal).
- launch démarre une coroutine qui exécute makeLoginRequest() sur un thread d'E/S.
- Pendant que la requête est en cours, login() continue et retourne immédiatement.



Tâches asynchrones - Les coroutines

```
1 // LoginRepository.kt
2 class LoginRepository(private val responseParser: LoginResponseParser) {
3
4     private val loginUrl = "https://example.com/login"
5
6     fun makeLoginRequest(jsonBody: String): Result<LoginResponse> { // méthode synchrone qui bloque le thread appelant
7         ...
8     }
9 }
```

⚠ Problème :

- `makeLoginRequest()` n'est **pas sécurisée**, car si elle est appelée depuis le thread principal, elle **bloque l'UI**.
- Nous devons **déplacer son exécution** vers un thread d'arrière-plan.



Tâches asynchrones - Les coroutines

```
1 // LoginRepository.kt
2 class LoginRepository(private val responseParser: LoginResponseParser) {
3
4     private val loginUrl = "https://example.com/login"
5
6     suspend fun makeLoginRequest(jsonBody: String): Result<LoginResponse> {
7         // Déplacer l'exécution vers un thread d'E/S
8         return withContext(Dispatchers.IO) {
9             // Code de requête réseau bloquant
10        }
11    }
12 }
```

- `withContext(Dispatchers.IO)` transfère l'exécution vers un thread optimisé pour les E/S, évitant ainsi tout blocage de l'UI.
- `suspend` indique que cette fonction doit être appelée depuis une coroutine.



Tâches asynchrones - Les coroutines

```
1 // LoginRepository.kt
2 class LoginRepository(private val responseParser: LoginResponseParser) {
3
4     private val loginUrl = "https://example.com/login"
5
6     suspend fun makeLoginRequest(jsonBody: String) = withContext(Dispatchers.IO) {
7         makeLoginRequestSynchron()
8     }
9 }
```

- `withContext(Dispatchers.IO)` **transfère l'exécution vers un thread optimisé pour les E/S**, évitant ainsi tout blocage de l'UI.
- `suspend` **indique que cette fonction doit être appelée depuis une coroutine.**



Tâches asynchrones - Les coroutines

```
1 class LoginViewModel(  
2     private val loginRepository: LoginRepository  
3 ): ViewModel() {  
4  
5     fun login(username: String, token: String) {  
6  
7         // Create a new coroutine on the UI thread  
8         viewModelScope.launch {  
9             val jsonBody = "{ username: \"$username\", token: \"$token\"}"  
10  
11             // Make the network call and suspend execution until it finishes  
12 -> val result = loginRepository.makeLoginRequest(jsonBody)  
13  
14             // Display result of the network request to the user  
15             when (result) {  
16                 is Result.Success<LoginResponse> -> // Happy path  
17                 else -> // Show error in UI  
18             }  
19         }  
20     }  
21 }
```

Ce qui change par rapport à avant :

- `launch` n'a plus besoin de `Dispatchers.IO`, car `withContext(Dispatchers.IO)` est déjà utilisé dans `makeLoginRequest()`.
- Le résultat de la requête réseau est maintenant traité directement pour afficher l'interface utilisateur en cas de succès ou d'échec.



Tâches asynchrones - Les coroutines

Les Dispatchers

- Les coroutines Kotlin utilisent **des coordinateurs (Dispatchers) pour gérer l'exécution sur différents threads.**
- Même sur le **thread principal**, les coroutines doivent être exécutées dans un coordinateur.
- Les coroutines **peuvent se suspendre et reprendre automatiquement**, sans bloquer le thread.

Dispatcher	Utilisation	Exemples
<code>Dispatchers.Main</code>	UI Thread (Principal)	Mise à jour de l'interface
<code>Dispatchers.IO</code>	Opérations d'entrée/sortie	Accès réseau, base de données Room, lecture/écriture de fichiers
<code>Dispatchers.Default</code>	Calcul intensif	Tri de listes, traitement JSON, algorithmes complexes



Tâches asynchrones - Les coroutines

Démarrer une coroutine

Lorsque vous travaillez avec des coroutines en Kotlin, vous avez **deux options** pour les démarrer :

- `launch` Utilisé pour les tâches "fire and forget" (exécuter sans attendre de retour).
- `async` : Exécution avec retour de résultat (`await`)

```
1 viewModelScope.launch {  
2     // Exécute du code en arrière-plan  
3     delay(1000)  
4     println("Coroutine terminée")  
5 }  
6  
7  
8 val result = viewModelScope.async {  
9     delay(1000)  
10    "Résultat obtenu"  
11 }  
12  
13 println(result.await())
```



Tâches asynchrones - Les coroutines

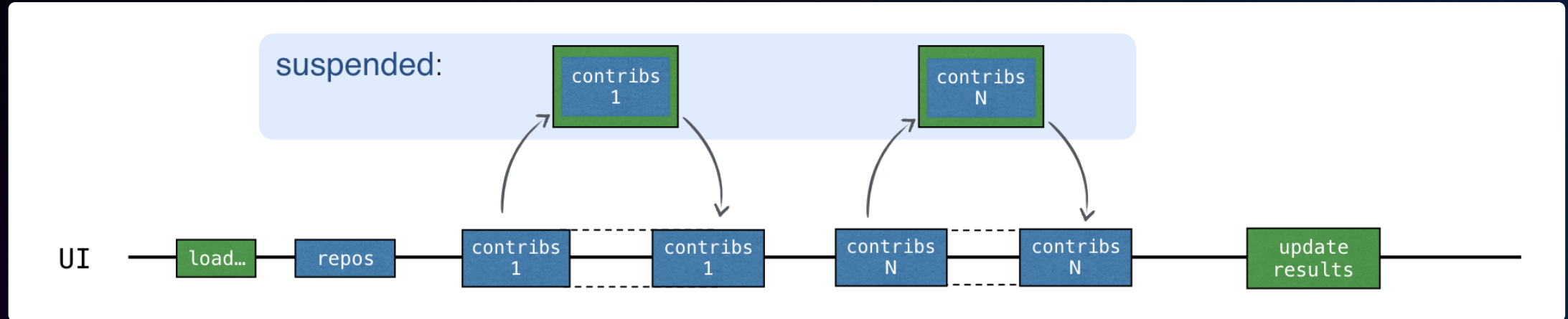
Exécution parallèle avec `async` et `awaitAll()`

```
1 suspend fun fetchTwoApis() = withContext(Dispatchers.IO) {  
2     val deferredOne = async { firstFetch() }  
3     val deferredTwo = async { secondFetch() }  
4  
5     deferredOne.await() // Attente de la première API  
6     deferredTwo.await() // Attente de la seconde API  
7 }
```

```
1 suspend fun fetchTwoApis() = withContext(Dispatchers.IO) {  
2     val deferredOne = async { firstFetch() }  
3     val deferredTwo = async { secondFetch() }  
4  
5     val deffereds = listOf(  
6         async { firstFetch() },  
7         async { secondFetch() }  
8     )  
9  
10    deffereds.awaitAll() // Attente des deux APIs  
11 }
```



Tâches asynchrones - Les coroutines



Exécution des fonctions `suspend` et `Thread UI`

1. L'interface utilisateur appelle une fonction `suspend` depuis le **thread principal**.
2. La fonction `suspend` démarre une **opération longue** en changeant de thread (`Dispatchers.IO` ou `Dispatchers.Default`).
3. Pendant l'exécution de cette tâche en arrière-plan, le **thread UI reste libre**, ce qui permet à l'application de **répondre aux interactions utilisateur**.
4. Une fois la tâche terminée, l'exécution reprend sur le **thread principal**, permettant de **mettre à jour l'UI** avec les résultats.



Coroutines : Ce qu'il faut retenir

- **Une alternative légère aux threads** pour gérer l'asynchronisme en Kotlin.
- **Suspend functions** : permettent d'écrire du code asynchrone de manière lisible et fluide.
- **Dispatchers** :
 - **Main** → UI & interactions utilisateur.
 - **IO** → Opérations disque/réseau.
 - **Default** → Calculs intensifs.
- **launch** → "fire and forget" (pas de retour de valeur)
- **async** → renvoie un résultat avec `await()`.

Pour aller plus loin 🚀

- **CoroutineScope** – Définit l'étendue d'une coroutine et permet de les annuler proprement.
- **Job** – Représente une coroutine en cours d'exécution et permet de la contrôler (annulation, hiérarchie, etc.).
- **CoroutineContext** – Définit l'environnement dans lequel une coroutine s'exécute (dispatcher, job, etc.).
- **CoroutineExceptionHandler** – Gère les erreurs non capturées dans les coroutines.



Introduction aux Flow en Kotlin

Les Flow sont une API de Kotlin permettant d'émettre et de collecter une séquence de valeurs de manière **asynchrone**. Ils sont particulièrement utiles pour représenter des flux de données qui évoluent dans le temps, comme des mises à jour en temps réel ou des événements utilisateur.

Différences entre Flow et suspend functions :

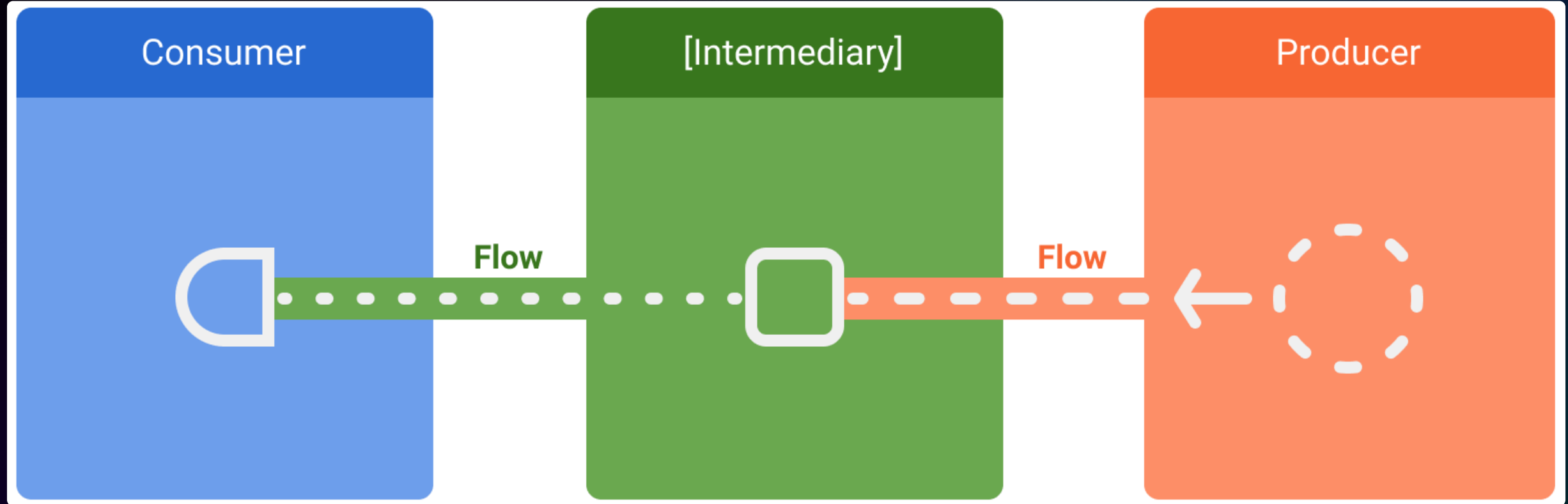
- Une **suspend function** ne retourne qu'une seule valeur.
- Un **Flow** peut émettre plusieurs valeurs au fil du temps.

Trois concepts clés des Flow :

- **Émission** (`emit`) : Un Flow peut émettre plusieurs valeurs successivement.
- **Collecte** (`collect`) : Une coroutine peut collecter les valeurs émises par un Flow.
- **Annulation** : Un Flow est annulé lorsque la coroutine qui le collecte est annulée.



Introduction aux Flow en Kotlin



Introduction aux Flow en Kotlin

StateFlow : Un Flow pour l'état

Le **StateFlow** est une variante de Flow conçue pour représenter un état observable qui évolue dans le temps. Il est particulièrement utile dans un **ViewModel** pour stocker et exposer l'état à l'UI avec Jetpack Compose.

Caractéristiques de StateFlow :

- Contient toujours une valeur initiale.
- Émet la dernière valeur stockée aux nouveaux collecteurs.

```
1 class MyViewModel : ViewModel() {
2     private val _uiState = MutableStateFlow("État initial")
3     val uiState: StateFlow<String> = _uiState
4
5     fun updateState(newValue: String) {
6         _uiState.value = newValue
7     }
8 }
9
10
11 @Composable
12 fun MyScreen(viewModel: MyViewModel) {
13     val state by viewModel.uiState.collectAsState()
14
15     Text(text = state)
16 }
```



Flow & StateFlow : Ce qu'il faut retenir

- **Flow** : une API Kotlin pour gérer des flux de données de manière asynchrone.
- **StateFlow** : un type de Flow qui conserve **un état unique** et émet ses mises à jour.
- **Particulièrement utile avec Compose** pour **réagir aux changements d'état** et mettre à jour l'UI de manière fluide.
- **Froid vs. Chaud** :
 - **Flow est "froid"** : ne démarre qu'avec un collect().
 - **StateFlow est "chaud"** : conserve et émet toujours la dernière valeur.**Géré dans ViewModel** pour exposer l'état à l'interface utilisateur sans risque de fuites mémoire.

💡 **D'autres concepts avancés existent** (buffer, conflate, collectLatest...) à explorer selon vos besoins !



Tâches persistantes avec WorkManager

WorkManager est la solution recommandée pour exécuter des tâches d'arrière-plan **persistantes**, c'est-à-dire qui doivent rester planifiées même après un redémarrage du système ou de l'application.

⚠ Mais attention :

- L'exécution **n'est pas garantie à 100 %** : elle dépend des contraintes définies, des restrictions Android (Doze Mode, optimisation de la batterie) et de la configuration de WorkManager.
- WorkManager essaie d'exécuter les tâches **le plus tôt possible** en fonction des conditions, mais peut reporter leur exécution si nécessaire.



Tâches persistantes avec WorkManager

Types de tâches gérées par WorkManager

- **Exécution immédiate**

- La tâche commence dès que possible et se termine rapidement.
- Elle peut être priorisée, mais WorkManager respecte les restrictions système.

- **Exécution longue**

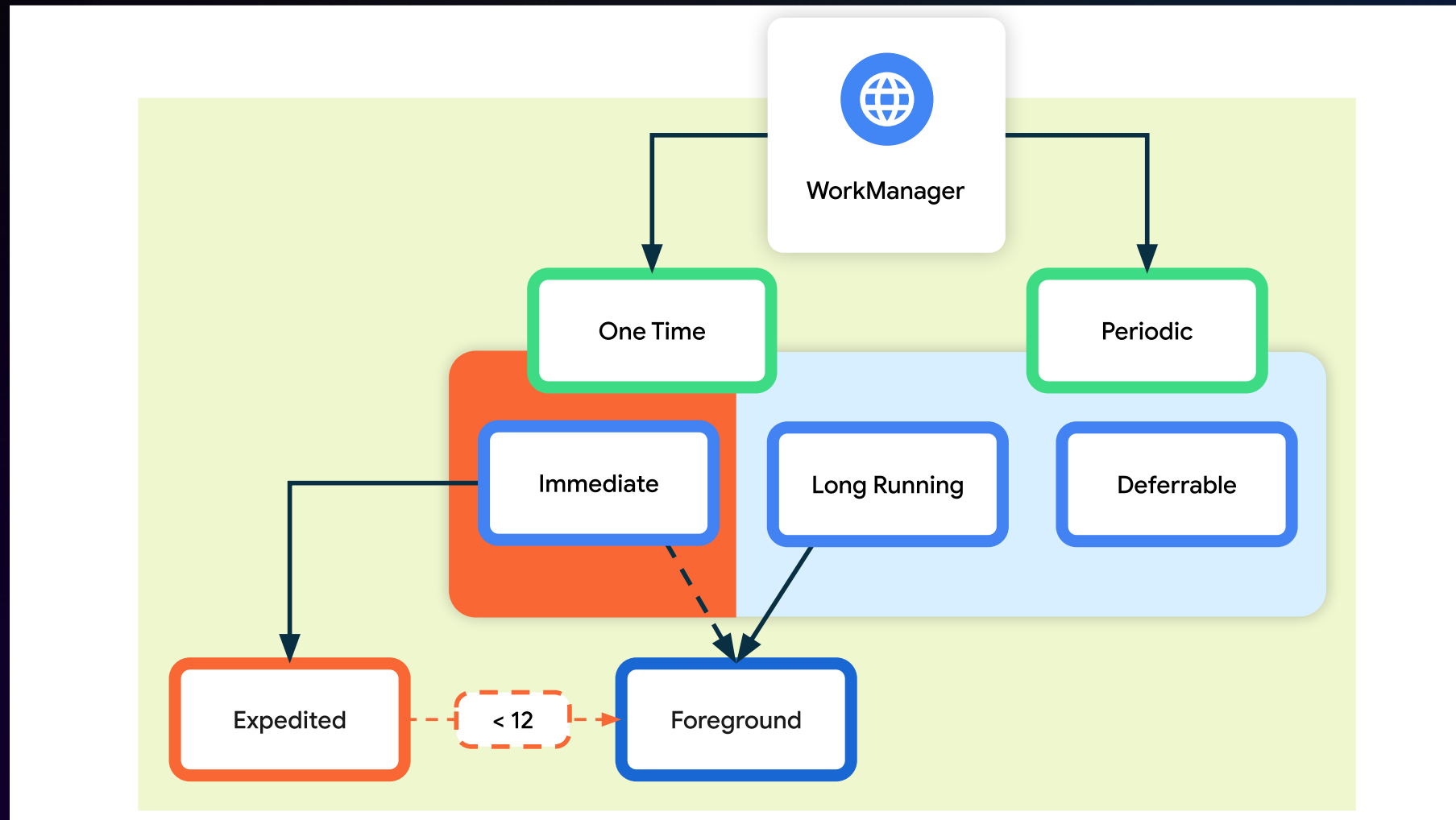
- Une tâche pouvant s'exécuter plusieurs minutes, voire plus de **10 minutes**.
- Utile pour les uploads ou traitements lourds, mais doit être bien configurée.

- **Exécution différée**

- Une tâche planifiée pour plus tard, **ponctuelle ou répétée**.
- Exemple : synchronisation des données toutes les heures.



Tâches persistantes avec WorkManager



Tâches persistantes avec WorkManager

Fonctionnalités clés de WorkManager

- **Contraintes des tâches**
 - Définir des conditions avant exécution (Wi-Fi, appareil en charge, batterie suffisante, etc.).
- **Planification efficace**
 - Exécution **ponctuelle ou répétée**, avec **plages horaires flexibles**.
 - Possibilité d'**étiqueter, surveiller et annuler** des groupes de tâches.
 - Stockage en **SQLite interne** pour assurer la persistance.
- **Tâches prioritaires**
 - Exécution immédiate en arrière-plan pour des tâches critiques de **courte durée**.



Tâches persistantes avec WorkManager

Fonctionnalités clés de WorkManager

- **Contraintes des tâches**
 - Définir des conditions avant exécution (Wi-Fi, appareil en charge, batterie suffisante, etc.).
- **Planification efficace**
 - Exécution **ponctuelle ou répétée**, avec **plages horaires flexibles**.
 - Possibilité d'**étiqueter, surveiller et annuler** des groupes de tâches.
 - Stockage en **SQLite interne** pour assurer la persistance.
- **Tâches prioritaires**
 - Exécution immédiate en arrière-plan pour des tâches critiques de **courte durée**.



Tâches persistantes avec WorkManager

```
1 // libs.catalog.toml
2
3 [versions]
4 workRuntimeKtx = "2.10.0"
5
6 [libraries]
7 androidx-work-runtime-ktx = { module = "androidx.work:work-runtime-ktx", version.ref = "workRuntimeKtx" }
```

```
1 // build.gradle (app)
2
3 dependencies {
4
5     implementation(libs.androidx.work.runtime.ktx)
6
7 }
```



Tâches persistantes avec WorkManager

Définir des tâches avec WorkManager

Les tâches sont définies en créant une **classe Worker** qui étend la classe `Worker`. La méthode `doWork()` s'exécute de manière **asynchrone** sur un thread d'arrière-plan géré par WorkManager.

```
1 class UploadWorker(appContext: Context, workerParams: WorkerParameters) : Worker(appContext, workerParams) {
2
3     override fun doWork(): Result {
4         // Effectuer le travail ici, par exemple, uploader des images
5         uploadImages()
6
7         // Indiquer si le travail a été effectué avec succès
8         return Result.success()
9     }
10
11 }
```

Types de résultats possibles dans `doWork()` :

- `Result.success()` : La tâche a été effectuée avec succès.
- `Result.failure()` : La tâche a échoué.
- `Result.retry()` : La tâche a échoué et doit être réessayée selon les règles définies.



Tâches persistantes avec WorkManager

Créer une WorkRequest

Une fois que votre tâche est définie avec un **Worker**, elle doit être planifiée en utilisant une **WorkRequest**. WorkManager offre des options flexibles pour la planification de l'exécution des tâches.

```
1 val uploadWorkRequest: WorkRequest =  
2     OneTimeWorkRequestBuilder<UploadWorker>( )  
3         .build()
```

Envoyer la **WorkRequest** à WorkManager

```
1 WorkManager  
2     .getInstance(myContext)  
3     .enqueue(uploadWorkRequest)
```

Le délai exact d'exécution dépend des **contraintes** définies dans la `WorkRequest` et des optimisations du système (comme l'économie d'énergie ou le mode veille).



Tâches persistantes avec WorkManager

Planifier une tâche périodique

WorkManager permet de planifier des tâches régulières avec `PeriodicWorkRequest`.

```
1 val saveRequest =  
2     PeriodicWorkRequestBuilder<SaveImageToFileWorker>(1, TimeUnit.HOURS)  
3     // Additional configuration  
4     .build()
```

Enregistrer la tâche périodique

```
1 WorkManager  
2     .getInstance(myContext)  
3     .enqueue(saveRequest)
```

Note : La période d'intervalle définit la durée minimale entre les répétitions. Le délai exact dépend des **contraintes** et des **optimisations du système** (économie d'énergie, etc.).



Tâches persistantes avec WorkManager

Contraintes liées aux tâches dans WorkManager

Les contraintes permettent de **différer l'exécution d'une tâche** jusqu'à ce que certaines conditions soient remplies. WorkManager propose plusieurs types de contraintes :

```
1 val constraints = Constraints.Builder()  
2     .setRequiredNetworkType(NetworkType.UNMETERED)  
3     .setRequiresCharging(true)  
4     .build()  
5  
6 val myWorkRequest: WorkRequest =  
7     OneTimeWorkRequestBuilder<MyWork>()  
8         .setConstraints(constraints)  
9         .build()
```

- Si **plusieurs contraintes** sont définies, **toutes** doivent être respectées pour que la tâche démarre.
- Si une contrainte **n'est plus remplie** pendant l'exécution, **WorkManager stoppe la tâche** et attend que les conditions soient de nouveau réunies.



WorkManager : Ce qu'il faut retenir

- **Solution recommandée** pour les tâches d'arrière-plan persistantes.
- **Gère les redémarrages** de l'application et du système.
- **Respecte les optimisations du système** (économie d'énergie, mode Sommeil).
- **Planification flexible** :
 - Exécution immédiate ou différée
 - Tâches ponctuelles ou périodiques
 - **Personnalisable avec des contraintes** (réseau, batterie, charge, etc.).
 - **Gestion des échecs** avec des règles de nouvelle tentative.

Alternative à d'autres solutions comme les services ou AlarmManager, en offrant plus de robustesse et de flexibilité.

💡 **Il existe encore d'autres fonctionnalités avancées** (chaining, input/output data...),
que vous pourrez explorer selon vos besoins !



Conclusion

La gestion des tâches en arrière-plan sur Android repose sur plusieurs outils adaptés aux différents besoins :

- **Services** : Pour exécuter des tâches de manière autonome.
- **WakeLock** : Pour garder l'appareil éveillé lorsque c'est indispensable.
- **Tâches asynchrones** : Threads, Coroutines pour éviter de bloquer l'UI.
- **WorkManager** : Pour les tâches persistantes et différables.

Il existe d'autres API adaptées à des cas spécifiques :

- **AlarmManager** : Pour exécuter des tâches à un moment précis, même après un redémarrage.
- **JobScheduler** : Pour planifier des tâches en tenant compte des contraintes système. (Remplacé par WorkManager)
- **BroadcastReceiver** : Pour réagir à des événements système (ex : connexion réseau).

