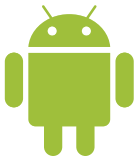


ANDROID

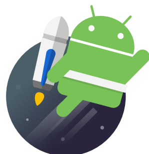
Laurent Provot <laurent.provot@uca.fr>

Janvier 2023

Android Jetpack

Jetpack

- API Android grandissante, fragmentée => on s'y perd
- Architecture logicielle répétitive, fatiguante à mettre en place (boilerplate code)
- Demande des développeurs de Best Practices



- Proposition d'Android Jetpack mi 2018
  - Ce n'est pas une bibliothèque supplémentaire :)
  - C'est un ensemble de composants et de guides pour tendre vers de la qualité sous Android
  - cf. <https://developer.android.com/jetpack/>

Objectifs 2e partie de module

- Développer une petite application « classique » en suivant les recommandations Jetpack
  - le thème est libre
  - on se limitera en cours à la partie « Architecture »
- Vous devez me montrer à travers ça que vous avez compris les best practices préconisées dans Jetpack
- Vous serez notés principalement sur la qualité de code
- Contraintes :
  - utiliser un max de AAC (Android Architecture Components)
  - l'appli devra faire des requêtes réseaux
  - certains éléments devront être modifiables, avec persistance
  - utiliser une `RecyclerView`
  - avoir des notifications
  - pas le droit d'utiliser de bibliothèque externe au framework Android (exceptions... à voir avec moi, *Retrofit* OK)

Architecture désirée

- MVVM... what else ;-)

Activity / Fragment

ViewModel LiveData

Local LiveData

Repository LiveData

Remote

Données
SQLite

WebService

Architecture désirée

- MVVM... what else ;-)

Activity / Fragment

ViewModel LiveData

Local LiveData

Repository LiveData

Remote

Données SQLite

WebService

En P1 nous avons tout fait sur le thread principal : pas bien !!!

L'asynchrone sous Android

Le modèle de thread

- Juste quelques mots
- GUI mono-threadé (`UiThread` ou `MainThread`)
- Si bloqué « application not responding » (ANR) dialog
- Règles :
  1. **NE PAS** bloquer le thread UI
  2. **NE PAS** accéder aux éléments du GUI en dehors de l'UI thread
- Utilisation des `Service` (pas threadé)
- Utilisation des `IntentService` (threadé)
- Bas niveau : `Thread`, `Handler`, ...
- Haut niveau : `AsyncTask` (*deprecated* depuis API 30)
- Chargement des données : `Loader` (*deprecated* depuis API 28)
- Java : `ExecutorService`
- Kotlin : les coroutines

Les threads de calculs

```
fun onClick(v: View) {
    Thread(Runnable {
        val b: Bitmap = loadImageFromNetwork("http://example.com/image.png")
        imageView.setImageBitmap(b)
    }).start();
}
```

- Pas bien, viole la deuxième règle

Les threads de calculs

- Méthodes helper
  - `Activity.runOnUiThread(Runnable)`
  - `View.post(Runnable)`
  - `View.postDelayed(Runnable, Long)`

```
fun onClick(v: View) {
    Thread {
        val b: Bitmap = loadImageFromNetwork("http://example.com/image.png")
        imageView.post { imageView.setImageBitmap(b) }
    }.start()
}
```

Les tâches asynchrones

- Pour les tâches courtes
- `AsyncTask<Params, Progress, Result>`

```
class DownloadFilesTask : AsyncTask<URL, Int, Long>() {
    override fun doInBackground(vararg urls: URL): Long {
        val count = urls.size
        var totalSize: Long = 0
        for (i in 0 until count) {
            totalSize += Downloader.downloadFile(urls[i]);
            publishProgress((i / count.toFloat() * 100).toInt())
            if (isCancelled) break
        }
        return totalSize
    }
    override fun onProgressUpdate(vararg progress: Int) {
        setProgressPercent(progress[0]);
    }
    override fun onPostExecute(result: Long) {
        showDialog("Downloaded " + result + " bytes");
    }
}
```

Les classes de concurrence en Java

- On utilise les primitives haut niveau du package `java.util.concurrent`
- `Executor`, `ExecutorServices`, `Executors`, `Future`, ...

Par exemple :

- `Executors.newSingleThreadExecutor()` : factory qui crée un `ExecutorService` ayant un thread unique qui sera chargé d'exécuter un besoin donné.
- `Executors.newFixedThreadPool(nbThreads: Int)` : factory qui crée un `ExecutorService` ayant un pool de `nbThreads` threads qui s'exécuteront en parallèle de façon à traiter plusieurs besoins simultanément.

Le lancement d'une tâche sur le `ServiceExecutor` se fait grâce à sa méthode `execute(task: Runnable)`

Intégration au `Repository`

```kotlin
/**
 *  Un thread dédié à l'exécution asynchrone des méthodes
 *  de mise à jour des données dans la BdD.
 */
val IO_EXECUTOR: ExecutorService = Executors.newSingleThreadExecutor()


/**
 *  La source de vérité de l'information. Ici c'est juste un wrapper
 *  à la DAO qui permet d'exécuter en asynchrone les opérations qui
 *  le nécessitent.
 *  Dans la vraie vie on y fait aussi le choix de piocher l'information
 *  au bon endroit (BdD, fichier, source en remote, …)
 */
class DogRepository(private val dogDao: DogDao) {
    fun insert(dog: Dog) = IO_EXECUTOR.execute { dogDao.insert(dog) }
    fun delete(dog: Dog) = IO_EXECUTOR.execute { dogDao.delete(dog) }
    fun update(dog: Dog) = IO_EXECUTOR.execute { dogDao.update(dog) }

    fun findById(dogId: Long) = dogDao.findById(dogId)
    fun getAll() = dogDao.getAll()
}
```