Labo 5 - Tâches asynchrones et coroutines

DAA

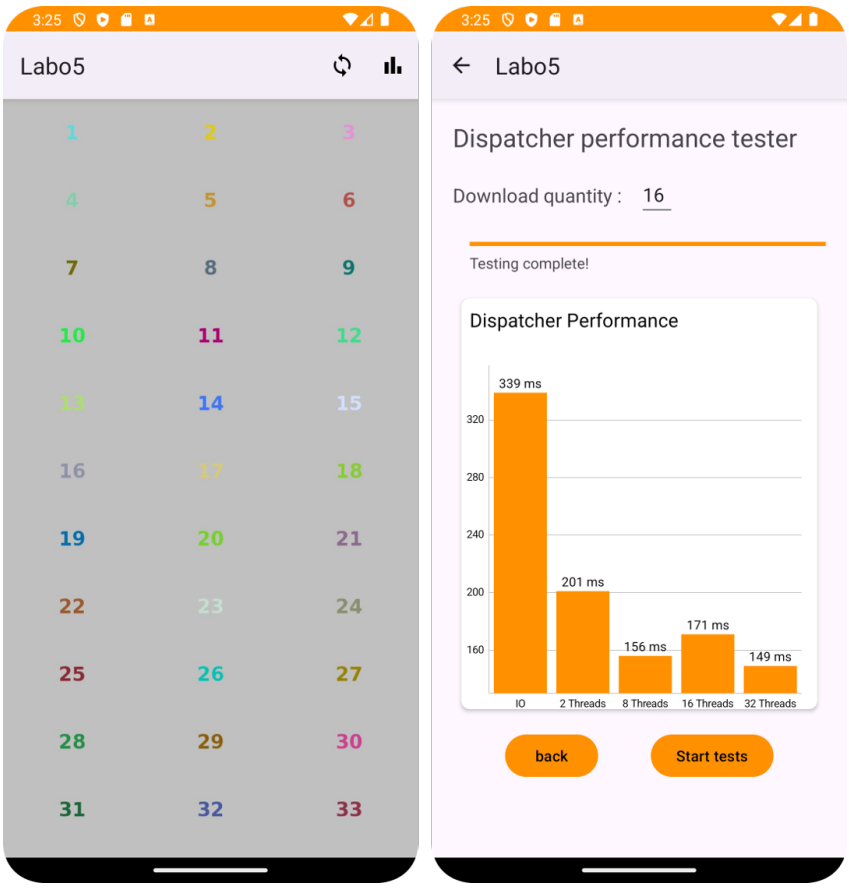
Timothée Van Hove, Léo Zmoos

21 décembre 2023

Table des matières

Introduction

Ce laboratoire consiste à concevoir une application Android, se présentant sous la forme d'une galerie d'images dynamique. À travers la mise en pratique des concepts de programmation asynchrone et de la gestion des ressources locales, ce projet vise à illustrer l'application des Coroutines Kotlin et du WorkManager pour des opérations de téléchargement et de mise en cache des images récupérées depuis le réseau. La gestion du réseau et l'efficience des opérations en arrière-plan sont au cœur des enjeux abordés, mettant en lumière l'importance d'une architecture d'application bien conçue.



Choix d'implémentation

Gestion de la connectivité

La connectivité à Internet est essentielle pour le fonctionnement de notre application. Nous avons donc mis en place une méthode dédiée pour vérifier la disponibilité du réseau :

```
object Network {
    fun isNetworkAvailable(context: Context): Boolean {
        val manager = context.getSystemService(Context.CONNECTIVITY_SERVICE) as ConnectivityManager
        val capabilities = manager.activeNetwork ?: return false
        val actNw = manager.getNetworkCapabilities(capabilities) ?: return false

        return when {
            actNw.hasTransport(NetworkCapabilities.TRANSPORT_WIFI) -> true
            actNw.hasTransport(NetworkCapabilities.TRANSPORT_CELLULAR) -> true
            actNw.hasTransport(NetworkCapabilities.TRANSPORT_ETHERNET) -> true
            else -> false
        }
    }
}
```

Cette méthode nous permet de gérer les cas où une connexion Internet est absente ou perdue, en particulier lors du démarrage initial de l'application et lors des tentatives de rafraîchissement de la `RecyclerView`. Pour informer l'utilisateur de l'absence de connectivité, un dialogue d'erreur est affiché :

```
object Dialogs {
    fun showNoConnectionDialog(context: Context, inflater: LayoutInflater) {
        val dialogView = inflater.inflate(R.layout.dialog_no_connection, null)
        val dialog = AlertDialog.Builder(context)
            .setView(dialogView).setNegativeButton("Ok") { dialog, _ -> dialog.dismiss()}.create()

        dialog.show()
    }
}
```

Ce dialogue est déclenché dans la `MainActivity` via la méthode `tryLoadImages`, qui tente de charger les images si une connexion est disponible ou affiche le dialogue d'erreur dans le cas contraire :

```
class MainActivity : AppCompatActivity(), OnItemClickListener {
    // ( ... )
    private fun tryLoadImages(): Boolean {
        return if (isNetworkAvailable(this)) {
            adapter.updateItems(items)
            true
        } else {
            showNoConnectionDialog(this, layoutInflater)
            false
        }
    }
    // ( ... )
}
```

Swipe to refresh

Nous avons intégré la fonctionnalité “Swipe to refresh” dans notre application pour permettre une actualisation facile des données affichées. Pour implémenter cette feature, nous avons enveloppé notre `RecyclerView` dans un `SwipeRefreshLayout`. Ce conteneur permet à l'utilisateur de rafraîchir le contenu en effectuant un geste de glissement vers le bas. Voici comment nous avons structuré notre mise en page XML pour intégrer le `SwipeRefreshLayout` :

```
<androidx.swiperefreshlayout.widget.SwipeRefreshLayout
    android:id="@+id/swipe_refresh_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/recycler_view"
    android:layout_width="match_parent"
    android:layout_height="0dp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

</androidx.swiperefreshlayout.widget.SwipeRefreshLayout>
```

Dans l'activité hôte, nous avons configuré le `SwipeRefreshLayout` en implémentant `setOnRefreshListener`. Dans ce cas, nous déclenchons le processus de nettoyage manuel du cache et le rechargement des images:

```
binding.swipeRefreshLayout.setOnRefreshListener {
    manualClearCache()
    binding.swipeRefreshLayout.isRefreshing = false
}
```

Implémentation de tests de performance

Nous avons profité de la rallonge du délai de rendu de ce laboratoire pour aller plus loin que la consigne en implémentant une activité dont la tâche est de tester la performance de différents dispatchers et d'afficher les résultats dans un graphe en barres.

Activité hôte (`TestActivity`)

Cette activité permet de lancer les tests de performance en cliquant sur le 2e bouton du menu. L'exécution des tests se fait de manière asynchrone pour ne pas bloquer l'interface utilisateur, tout en fournissant des retours en temps réel via une barre de progression et des messages d'état.

```
private fun launchTests() {
    //( ... )
    // Prepare the list of URLs for the images to be downloaded
    val items = List(nbDownloads) { URL("${ENDPOINT}${it + 1}${FILE_EXT}") }

    // Initialize the progress bar
    progressBar.apply { max = PerformanceTester.dispatcherPairs.size; progress = 0 }

    // Start the test in a coroutine
    lifecycleScope.launch {
        val testResults = PerformanceTester.testDispatcherPerformance(
            items,
            this,
            lifecycleScope,
            updateUI = { status -> binding.textViewStatus.text = status },
            updateProgress = { progress -> progressBar.progress = progress }
        )
        //( ... )
    }
}
```

Une fois que nous avons récupéré les résultats des tests, nous les affichons dans le graphique grâce à la méthode `setupBarChart`:

```
private fun setupBarChart(testResults: List<TestResult>) {
    val entries = ArrayList<BarEntry>() // List to hold bar entries
    val labels = ArrayList<String>() // List to hold axis labels

    testResults.forEachIndexed { index, result ->
        entries.add(BarEntry(index.toFloat(), result.duration.toFloat()))
        labels.add(result.dispatcherName)
    }
    // ( ... )
    barChart.invalidate() // Display the chart
}
```

Implémentation des tests (`PerformanceTester`)

Cette classe utilise la logique de téléchargement d'images déjà établie dans notre application, mais l'applique à différents `CoroutineDispatchers` pour comparer leur efficacité.

```
private suspend fun downloadImage(url: URL): Bitmap? {
    return try {
        with(ImageDownloader()){decodeImage(downloadImage(url)!!)}
    } catch (e: Exception) {
        Log.e("ImageDownload", "Error downloading image from $url", e)
        null
    }
}
```

Création des `Dispatchers`

Nous avons défini une liste de `CoroutineDispatcher` comprenant le dispatcher par défaut "IO" et plusieurs dispatchers personnalisés avec un nombre fixe de threads. Cette diversité permet d'évaluer l'impact du nombre de threads sur les performances de téléchargement.

```
val dispatcherPairs = listOf(
    "IO" to Dispatchers.IO,
    "2 Threads" to Executors.newFixedThreadPool(2).asCoroutineDispatcher(),
    "8 Threads" to Executors.newFixedThreadPool(8).asCoroutineDispatcher(),
    "16 Threads" to Executors.newFixedThreadPool(16).asCoroutineDispatcher(),
    "32 Threads" to Executors.newFixedThreadPool(32).asCoroutineDispatcher()
)
```

Test des performances

La méthode`testDispatcherPerformance` prend en charge l'itération sur chaque dispatcher, exécutant les tâches de téléchargement simultanément pour un ensemble donné d'URLs et mesurant le temps nécessaire pour achever ces tâches. Les résultats sont ensuite retournés sous forme de liste, prêts à être affichés dans le graphique.

```
import kotlinx.coroutines.CoroutineDispatcher as CD
import kotlinx.coroutines.CoroutineScope as CS

suspend fun testDispatcherPerformance(
    items: List<URL>, testScope: CS, uiScope: CS, updateUI: (String) -> Unit,updateProgress: (Int)
    ↪  -> Unit
    ): List<TestResult> {
    val results = mutableListOf<TestResult>()

    testScope.async {
            dispatcherPairs.forEachIndexed { index, (name, dispatcher) ->
                uiScope.launch { updateUI("Testing $name dispatcher...") }.join()
                results.add(TestResult(name, getDuration(items, dispatcher, testScope)))
                uiScope.launch { updateUI("Testing complete!"); updateProgress(index + 1) }.join()
            }
        }.await()
        return results
}

private suspend fun getDuration(items: List<URL>, disp: CD, scope: CS): Long {
    val startTime = System.currentTimeMillis()
    scope.launch(disp) { items.map { url -> launch { downloadImage(url) } }.joinAll() }.join()
    return System.currentTimeMillis() - startTime
}
```

Question 3.1

Veuillez expliquer comment votre solution s'assure qu'une éventuelle Couroutine associée à une vue (item) de la RecyclerView soit correctement stoppée lorsque l'utilisateur scrolle dans la galerie et que la vue est recyclée.

Chaque élément de la RecyclerView est géré par une classe ViewHolder, qui est responsable du lancement et de la gestion de la coroutine pour le téléchargement et l'affichage de l'image associée à cet élément. Voici comment la gestion des coroutines est implémentée :

Dans la classe ViewHolder, une coroutine est démarrée dans la méthode `bind(url: URL)`. Cette coroutine gère le téléchargement et l'affichage de l'image. La coroutine est lancée dans un `LifecycleCoroutineScope` . Ce `CoroutineScope` est utilisé pour lier les coroutines au cycle de vie de l'activité ou du fragment hôte, ce qui garantit que les coroutines sont annulées lorsque le cycle de vie est détruit.

Pour garantir que chaque élément a une coroutine distincte, une variable `currentUrl` est utilisée. Lorsqu'une nouvelle URL est liée à un `ViewHolder`, il vérifie si cette URL est différente de l'actuelle. Si elle est différente, cela indique que l'élément est sur le point d'afficher une nouvelle image, nécessitant l'annulation de toute coroutine en cours associée à l'image précédente:

```
fun bind(url: URL) {
    // Check if the new URL is different from the current one
    if (currentUrl != url.toString()) {
        // Cancel any existing download job for the previous URL
        downloadJob?.cancel()

        // Reset the visibility of the ProgressBar and ImageView
        progressBar.visibility = View.VISIBLE
        image.visibility = View.INVISIBLE

        // Update the current URL
        currentUrl = url.toString()

        // Start a new coroutine for downloading and displaying the image
        downloadJob = scope.launch {
            val cachedBitmap =
                getBitmap(url.path.substring(url.path.lastIndexOf('/') + 1), url)
                updateImageView(cachedBitmap)
        }
    }
}
```

Le `RecyclerView` recycle les vues lorsqu'elles défilent hors de la zone visible. Pour gérer ça, la méthode `onViewRecycled` de l'adaptateur est surchargée. Cette méthode est appelée automatiquement lorsqu'un élément sort de la zone visible, indiquant que la vue est sur le point d'être recyclée. Dans cette méthode, la méthode `unbind()` du `ViewHolder` est appelée. Cette méthode est chargée d'annuler toute coroutine active associée au ViewHolder. L'annulation est obtenue en appelant `downloadJob?.cancel()`, qui annule en toute sécurité la coroutine si elle est actuellement active. Cela garantit que tout téléchargement ou traitement d'image en cours est arrêté, libérant ainsi des ressources.

```
override fun onViewRecycled(holder: ViewHolder) {
    holder.unbind()
}
```

Finalement, dans la méthode `unbind()`, la visibilité de ProgressBar et d'ImageView est réinitialisée. Cela prépare le ViewHolder à la réutilisation avec un nouvel élément, garantissant ainsi un état cohérent.

```
fun unbind() {
    // Cancel any ongoing download job to prevent memory leaks and unnecessary work
    downloadJob?.cancel()

    // Reset the visibility of the ProgressBar and ImageView
    progressBar.visibility = View.VISIBLE
    image.visibility = View.INVISIBLE

    // Clear the current URL since the view is being recycled
    currentUrl = null
}
```

Grâce à cette implémentation, nous garantissons que toute coroutine associée à un élément de la RecyclerView est gérée correctement. Lorsqu'un élément est recyclé, sa coroutine associée est annulée, empêchant toute opération en arrière-plan qui n'est plus nécessaire.

Question 3.2

> *Comment pouvons-nous nous assurer que toutes les Coroutines soient correctement stoppées lorsque l'utilisateur quitte l'Activité ? Veuillez expliquer la solution que vous avez mise en œuvre, est-ce la plus adaptée ?*

Pour gérer efficacement les coroutines, nous avons utilisé le `lifecycleScope`, un `CoroutineScope` intégré lié au cycle de vie de l'activité. Ce scope assure que les coroutines lancées en son sein soient automatiquement annulées lorsque l'activité atteint sa phase de destruction (`onDestroy()`).

Dans notre `MainActivity`, nous utilisons `lifecycleScope` pour lancer des coroutines liées aux opérations d'interface utilisateur. `lifecycleScope` s'aligne sur le cycle de vie de l'activité, garantissant ainsi que les coroutines ne continuent pas à s'exécuter si l'activité est détruite, évitant les opérations redondantes.

Bien que `lifecycleScope` gère automatiquement l'annulation des coroutines, nous avons renforcé ce comportement dans la méthode `onDestroy()` de notre activité :

```
override fun onDestroy() {
    super.onDestroy()
    if (isFinishing) {
        // This ensures coroutines are cancelled only when the activity is truly finishing
        lifecycleScope.coroutineContext.cancelChildren()
    }
}
```

Dans ce code, nous appelons `cancelChildren()` sur le contexte de coroutines de `lifecycleScope`. Cette méthode annule toutes les coroutines en cours dans ce scope. L'utilisation de `isFinishing` permet de distinguer entre une destruction temporaire (comme une rotation d'écran) et une fermeture définitive de l'activité.

Est-ce que cette solution est la plus adaptée? N'étant pas experts Android, il est difficile de répondre à cette question, cependant nous pouvons apporter quelques éléments de réponse:

- Si nos coroutines doivent se terminer quel que soit le cycle de vie de l'activité (par exemple, la synchronisation des données en arrière-plan qui doit se poursuivre pendant les modifications de configuration), une portée différente comme `ViewModelScope` ou un `CoroutineScope` personnalisé lié au cycle de vie de l'application peut être plus approprié.
- L'utilisation de `isFinishing` dans `onDestroy()` est cruciale pour faire la différence entre la destruction d'activité due aux changements de configuration et la destruction finale. Si notre application gère les modifications de configuration (comme les rotations d'écran) sans recréer l'activité, notre approche fonctionne bien.
- Pour les tâches ou opérations de longue durée qui doivent survivre à l'activité, nous pouvons envisager d'utiliser un `WorkManager` ou des services.

Question 3.3

Est-ce que l'utilisation du Dispatchers.IO est la plus adaptée pour des tâches de téléchargement ? Ou faut-il plutôt utiliser un autre Dispatcher, si oui lequel ? Veuillez illustrer votre réponse en effectuant quelques tests.

Comme mentionné dans la partie Implémentation de tests de performance, nous avons mis en place une activité permettant de faire des tests de performance concernant plusieurs dispatchers. Voici les résultats de nos tests:

- Avec 16 images téléchargées en parallèle, nous voyons clairement que le dispatcher IO est le plus lent
- Avec 32 images téléchargées en parallèle, le dispatcher custom de 32 threads est le plus rapide
- Avec 64 images téléchargées en parallèle, le dispatcher custom de 16 threads est le plus rapide
- Avec 128 images téléchargées en parallèle le dispatchers custom de 16 threads est le plus rapide

`Dispatchers.IO` est optimisé pour les opérations d'entrée/sortie qui bloquent les threads, comme la lecture et l'écriture de fichiers ou les opérations de réseau. Il utilise un pool de threads partagé qui est dimensionné automatiquement en fonction des besoins de l'application. Cependant, si une tâche de téléchargement implique des opérations CPU-intensives, comme le **décodage d'images**, l'utilisation d'un `Dispatcher` personnalisé avec un pool de threads dédié peut offrir de meilleures performances, car cela permet de mieux contrôler la concurrence et la parallélisation des tâches. Dans notre cas n'importe lequel des dispatchers custom avec un thread pool de 2, 8, 16 ou 32 threads est systématiquement plus rapide.

Question 3.4

> *Nous souhaitons que l'utilisateur puisse cliquer sur une des images de la galerie afin de pouvoir, par exemple, l'ouvrir en plein écran. Comment peut-on mettre en place cette fonctionnalité avec une RecyclerView? Comment faire en sorte que l'utilisateur obtienne un feedback visuel lui indiquant que son clic a bien été effectué, sur la bonne vue.*

Nous avons déjà implémenté ce genre de feature dans le labo précédent (cliquer pour éditer une note dans une recyclerView), et la procédure est très similaire.

Afficher l'image en plein écran:

Nous allons créer une interface qui nous permettra de propager le clickListener vers la MainActivity:

```
interface OnItemClickListener {
    fun onItemClick(position: Int, items: List<URL>)
}
```

Ensuite dans notre MainActivity, nous allons implémenter le clickListener de notre interface. Dans ce click listener, nous allons récupérer l'url de l'image et lancer une nouvelle activité. Le fait de récupérer l'url, nous permettrait (en fonction de l'API) d'afficher une version haute résolution de l'image.

```
class MainActivity : AppCompatActivity(), OnItemClickListener{
// ( ... )
    override fun onItemClick(position: Int, items: List<URL>) {
        val imageUrl = items[position].toString()
        val intent = Intent(this, FullScreenImageActivity::class.java)
        intent.putExtra("IMAGE_URL", imageUrl)
        startActivity(intent)
    }
// ( ... )
}
```

Maintenant, créons notre activité d'affichage de l'image, avec son layout. Dans cette activité nous avons utilisé glide, une librairie qui nous permet de charger et afficher des images depuis différentes sources. `glide` prends en charge le caching, la gestion de la mémoire et le décodage de l'image.

```
class FullScreenImageActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_full_screen_image)

        val imageUrl = intent.getStringExtra("IMAGE_URL")

        // Use Glide to load the image
        Glide.with(this).load(imageUrl).into(findViewById(R.id.fullScreenImageView))
    }
}
```

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <ImageView
        android:id="@+id/fullScreenImageView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:scaleType="centerCrop"/>
</FrameLayout>
```

finalement, dans notre adapter, nous allons implémenter le `ClickListener` des éléments de la RecyclerView pour récupérer la position de l'élément sur lequel nous avons cliqué:

```
class ImageRecyclerAdapter(urls: List<URL> = listOf(), private val scope: LifecycleCoroutineScope,
↪   private val itemClickListener: OnItemClickListener) :
    RecyclerView.Adapter<ImageRecyclerAdapter.ViewHolder>() {
    // ( ... )

    inner class ViewHolder(view: View) : RecyclerView.ViewHolder(view) {
        init {
            view.setOnClickListener {
                val position = adapterPosition
                if (position != RecyclerView.NO_POSITION) {
                    itemClickListener.onItemClick(position, items)
                }
            }
        }
    //( ... )
}
```

Ajouter un feedback

Nous avons d'abord essayé d'utiliser un ripple effect, mais cette méthode n'a pas fonctionné. Nous avons alors opté pour une animation que nous avons implémenté sur le ClickListener de chaque item de la RecyclerView dans le block `init` du `ViewHolder`:

```
init {
    view.setOnClickListener {
    // Scale animation for visual feedback
        view.animate().scaleX(0.9f).scaleY(0.9f).setDuration(200).withEndAction {
            view.animate().scaleX(1f).scaleY(1f).setDuration(200).start()
        }
        // ( ... )
    }
}
```

Question 4.1

Lors du lancement de la tâche ponctuelle, comment pouvons-nous faire en sorte que la galerie soit rafraîchie ?

Il suffit d'appeler `adapter.notifyDataSetChanged()` pour rafraichir la galerie. Pour faire cela, la méthode `manualClearCache` appelle la méthode `tryLoadImages`, qu elle même va déclencher le rafraîchissement en appelant `updateItems` qui est dans l'adapter

```
private fun manualClearCache() {
    val clearCacheWork = OneTimeWorkRequestBuilder<ClearCacheWorker>().build()
    WorkManager.getInstance(this).enqueue(clearCacheWork)

    // After clearing the cache, reload the images
    if (tryLoadImages())
        Toast.makeText(this, "Cache cleared and images reloaded", Toast.LENGTH_LONG).show()
}

private fun tryLoadImages(): Boolean {
    return if(isNetworkAvailable(this)){
        adapter.updateItems(items)
        true
    } else{
        showNoConnectionDialog(this, layoutInflater)
        false
    }
}
```

Dans l'adapter:

```
fun updateItems(newItems: List<URL>) {
    items = newItems
    notifyDataSetChanged()
}
```

Question 4.2

Comment pouvons-nous nous assurer que la tâche périodique ne soit pas enregistrée plusieurs fois ? Vous expliquerez comment la librairie WorkManager procède pour enregistrer les différentes tâches périodiques et en particulier comment celles-ci sont ré-enregistrées lorsque le téléphone est redémarré.

`WorkManager` est conçu pour garantir que les tâches asynchrones soient gérées de manière robuste, même à travers les redémarrages de l'appareil ou les arrêts de l'application. Pour les tâches périodiques, il est crucial d'assurer qu'une tâche spécifique ne soit pas enregistrée plusieurs fois afin d'éviter des exécutions redondantes ou inattendues. Pour prévenir l'enregistrement multiple de la même tâche périodique, `WorkManager` fournit la méthode `enqueueUniquePeriodicWork`. Cette méthode prend un nom unique pour la work request, ce qui permet de s'assurer qu'il n'y a qu'une seule instance de cette tâche qui soit programmée à un moment donné.

Avec `enqueueUniquePeriodicWork`, nous utilisons également une `ExistingPeriodicWorkPolicy` qui définit la conduite à tenir si une tâche avec le même nom unique est déjà enregistrée :

- `KEEP` : Si une tâche avec le même nom existe déjà, la nouvelle tâche ne sera pas enregistrée. Cela garantit que la tâche originale continue de s'exécuter selon son intervalle défini sans interruption.
- `REPLACE` : La tâche existante sera remplacée par la nouvelle tâche. Cette option peut être utilisée pour mettre à jour ou redéfinir une tâche avec des paramètres ou des contraintes mis à jour.

Persistance et redémarrage

`WorkManager` utilise une base de données interne pour suivre les tâches. Lorsqu'un travail est planifié, il est persisté dans cette base de données. En cas de redémarrage de l'appareil, `WorkManager` écoute l'intention `BOOT_COMPLETED` et reprogramme automatiquement les travaux enregistrés en se basant sur les informations stockées.

Implémentation dans l'application

Dans notre application, nous utilisons la méthode `enqueueUniquePeriodicWork` avec la politique `ExistingPeriodicWorkPolicy.KEEP` pour planifier la tâche de nettoyage du cache :

```
class MainActivity : AppCompatActivity(), OnItemClickListener {

    companion object {
        // ( ... )
        const val uniqueWorkName = "ClearCachePeriodicWorkLabo5"
        val WM_POLICY = ExistingPeriodicWorkPolicy.KEEP
        val UNIT = TimeUnit.MINUTES
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        // ( ... )
        setPeriodicCacheClear()
        // ( ... )
    }

    // ( ... )

    private fun setPeriodicCacheClear() {
        // Bind the periodic cache clear
        val clearCacheRequest = PeriodicWorkRequestBuilder<ClearCacheWorker>(INTERVAL, UNIT).build()
        val wm = WorkManager.getInstance(this)
        wm.enqueueUniquePeriodicWork(uniqueWorkName, WM_POLICY, clearCacheRequest)
    }

    // ( ... )
}
```

Avec cette mise en œuvre, nous garantissons que notre tâche de nettoyage du cache ne sera programmée qu'une seule fois. Si l'application tente de planifier à nouveau la tâche avec le même `uniqueWorkName`, la politique `KEEP` s'assurera que la première instance de la tâche continue de fonctionner comme prévu sans duplication.

Conclusion

Ce laboratoire nous a permis de nous plonger dans les méandres de la programmation asynchrone et de la gestion des ressources réseau sur Android. L'opportunité de dépasser les exigences de base en développant une fonctionnalité de tests de performance a non seulement été fun mais a également renforcé notre compréhension pratique des dispatchers et de la concurrence en Kotlin.

La gestion de la connectivité réseau nous a obligé à réfléchir à la robustesse de l'application concernant les changements d'état de réseau. C'est pertinent car les utilisateurs s'attendent à une fiabilité continue, même dans des conditions réseau fluctuantes.

L'utilisation de `WorkManager` pour des tâches périodiques nous a permis d'apprendre l'implémentation de la planification d'une tâche persistante utilisée pour le nettoyage du cache. Cette compréhension sera très utile pour la conception d'applications à l'avenir.

Finalement, ce projet a été une opportunité d'appliquer les bonnes pratiques en matière de développement Android, de réfléchir à l'expérience utilisateur lors de l'ajout de nouvelles fonctionnalités et d'utiliser une bonne approche pour l'optimisation des performances. Les compétences et les connaissances acquises au cours de ce laboratoire nous sera très utile pour nos futurs projets de développement mobile.