Développement Android avec Kotlin

Cours - 12 - Requêtes réseau en Android

Jordan Hiertz

Contact

hiertzjordan@gmail.com

jordan.hiertz@al-enterprise.com

Requêtes réseau en Android

Dans une application mobile, il est souvent nécessaire de communiquer avec un serveur pour récupérer ou envoyer des données. Cela permet d'afficher des informations dynamiques, d'interagir avec des services en ligne ou encore de synchroniser des données entre appareils.

- Permissions et accès réseau
- Détection de la connectivité
- Requêtes avec `HttpURLConnection`
- Utilisation de `OkHttp`
- Simplification avec `Retrofit` et `KotlinX Serialization`

Permissions et accès réseau

Avant de pouvoir effectuer des requêtes réseau, il est essentiel de demander l'autorisation d'utiliser internet et de vérifier la connectivité du réseau.

- Demande de permission :

  Ajouter la permission `INTERNET` dans le fichier `AndroidManifest.xml`.

```xml
<manifest xmlns:android="http://schemas.android.com/apk/res/android" ...>

  <uses-permission android:name="android.permission.INTERNET" />

  <application ...></application>
</manifest>
```

Permissions et accès réseau

Avant de pouvoir effectuer des requêtes réseau, il est essentiel de demander l'autorisation d'utiliser internet et de vérifier la connectivité du réseau.

- Vérifier la connectivité :

  Utiliser `ConnectivityManager` pour vérifier l'état de la connexion réseau.

```
1  private const val DEBUG_TAG = "NetworkStatusExample"
2  ...
3  val connMgr = getSystemService(Context.CONNECTIVITY_SERVICE) as ConnectivityManager
4  var isWifiConn: Boolean = false
5  var isMobileConn: Boolean = false
6  connMgr.allNetworks.forEach { network ->
7      connMgr.getNetworkInfo(network).apply {
8          if (type == ConnectivityManager.TYPE_WIFI) {
9              isWifiConn = isWifiConn or isConnected
10         }
11         if (type == ConnectivityManager.TYPE_MOBILE) {
12             isMobileConn = isMobileConn or isConnected
13         }
14     }
15 }
```

Permissions et accès réseau

Avant de pouvoir effectuer des requêtes réseau, il est essentiel de demander l'autorisation d'utiliser internet et de vérifier la connectivité du réseau.

- Vérifier la connectivité :

  Utiliser `ConnectivityManager` pour vérifier l'état de la connexion réseau.

```kotlin
fun isOnline(): Boolean {
    val connMgr = getSystemService(Context.CONNECTIVITY_SERVICE) as ConnectivityManager
    val networkInfo: NetworkInfo? = connMgr.activeNetworkInfo
    return networkInfo?.isConnected == true
}
```

Détecter les modifications de connexion

Pour détecter les changements de connexion réseau (Wi-Fi ou mobile), on utilise un `BroadcastReceiver`. Cela permet d'adapter l'affichage de l'application en fonction de l'état de la connexion.

- Créer un `NetworkReceiver` :

  Le `BroadcastReceiver` intercepte l'action `CONNECTIVITY_ACTION` et met à jour l'état de la connexion.

```
class NetworkReceiver : BroadcastReceiver() {
    override fun onReceive(context: Context, intent: Intent) {
        val conn = context.getSystemService(Context.CONNECTIVITY_SERVICE) as ConnectivityManager
        val networkInfo = conn.activeNetworkInfo

        // Vérification et mise à jour de l'affichage en fonction de la connexion
        if (networkInfo?.type == ConnectivityManager.TYPE_WIFI) {
            refreshDisplay = true
            Toast.makeText(context, "Wi-Fi connecté", Toast.LENGTH_SHORT).show()
        } else {
            refreshDisplay = false
            Toast.makeText(context, "Perte de connexion", Toast.LENGTH_SHORT).show()
        }
    }
}
```

Détecter les modifications de connexion

Pour détecter les changements de connexion réseau (Wi-Fi ou mobile), on utilise un `BroadcastReceiver`. Cela permet d'adapter l'affichage de l'application en fonction de l'état de la connexion.

- Créer un `NetworkReceiver` :

  Le `BroadcastReceiver` est enregistré dans `onCreate()` et annulé dans `onDestroy()` pour éviter une consommation excessive des ressources.

```
override fun onCreate() {
    super.onCreate()
    val filter = IntentFilter(ConnectivityManager.CONNECTIVITY_ACTION)
    registerReceiver(networkReceiver, filter)
}

override fun onDestroy() {
    super.onDestroy()
    unregisterReceiver(networkReceiver)
}
```

Détecter les modifications de connexion - NetworkCallback

En plus de l'utilisation d'un `BroadcastReceiver`, une autre manière moderne et efficace de détecter les changements de réseau est d'utiliser `NetworkCallback`.

- Utiliser `NetworkCallback` avec `ConnectivityManager` :

  `NetworkCallback` permet de réagir en temps réel aux changements de réseau

```
1  val connectivityManager = getSystemService(Context.CONNECTIVITY_SERVICE) as ConnectivityManager
2
3  connectivityManager.registerDefaultNetworkCallback(object : ConnectivityManager.NetworkCallback() {
4      override fun onAvailable(network : Network) {
5          Log.e(TAG, "The default network is now: " + network)
6      }
7
8      override fun onLost(network : Network) {
9          Log.e(TAG, "The application no longer has a default network. The last default network was " + network)
10     }
11
12     override fun onCapabilitiesChanged(network : Network, networkCapabilities : NetworkCapabilities) {
13         Log.e(TAG, "The default network changed capabilities: " + networkCapabilities)
14     }
15
16     override fun onLinkPropertiesChanged(network : Network, linkProperties : LinkProperties) {
17         Log.e(TAG, "The default network changed link properties: " + linkProperties)
18     }
19 })
```

Détecter les modifications de connexion - NetworkCallback

En plus de l'utilisation d'un `BroadcastReceiver`, une autre manière moderne et efficace de détecter les changements de réseau est d'utiliser `NetworkCallback`.

- Utiliser `NetworkCallback` avec `ConnectivityManager` :

  De la même façon, le `NetworkCallback` devrait être enlevé dans `onDestroy` ou une autre méthode du cycle de vie

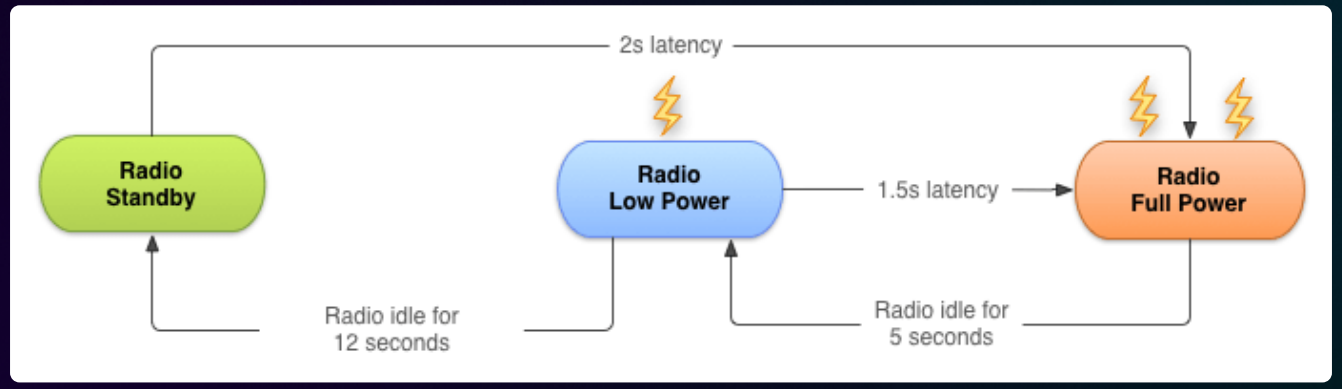
```
override fun onDestroy() {
    super.onDestroy()
    // Désenregistrement du NetworkCallback lorsque l'activité est détruite
    connectivityManager.unregisterNetworkCallback(networkCallback)
}
```

Optimiser l'accès au réseau et la consommation de batterie

Machine à États de la radio sans fil

La radio sans fil de l'appareil dispose de trois états d'énergie principaux pour optimiser la consommation de batterie :

- **Pleine puissance** : Connexion active, débit maximal.
- **Énergie faible** : État intermédiaire, réduit la consommation d'énergie d'environ 50%.
- **Veille** : Consommation minimale, aucune connexion active.

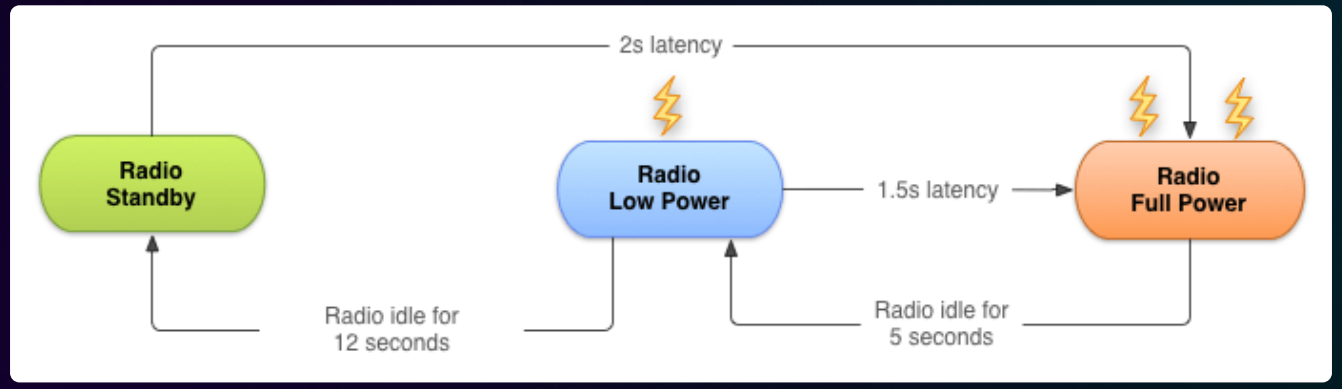


Optimiser l'accès au réseau et la consommation de batterie

Problèmes de latence

Les transitions entre ces états ont des latences qui impactent les requêtes réseau :

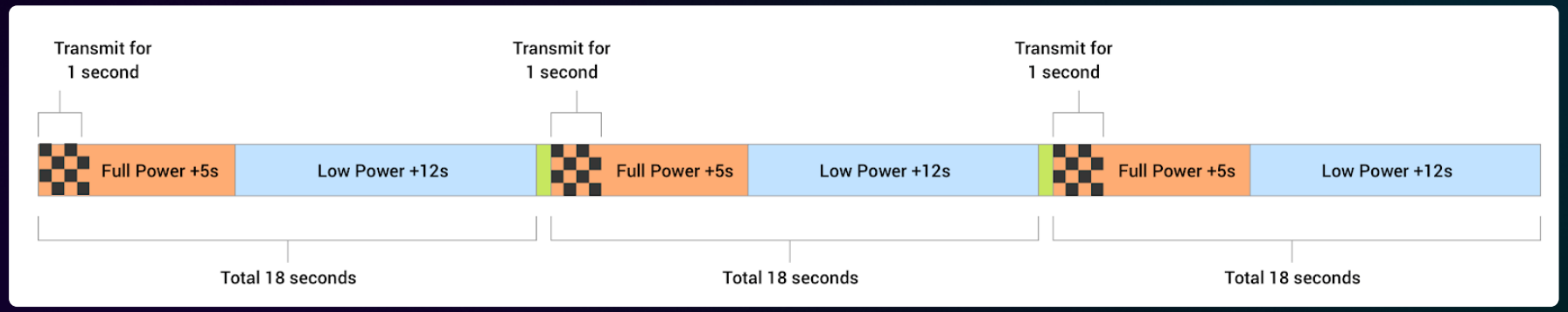
- **Pleine puissance -> Énergie faible** : Latence d'environ 1,5 seconde.
- **Veille -> Pleine puissance** : Latence pouvant dépasser 2 secondes.



Optimiser l'accès au réseau et la consommation de batterie

Conseils d'optimisation

- **Réduire les appels fréquents** aux ressources réseau pour éviter les transitions fréquentes entre états.
- **Regrouper les requêtes réseau** : Effectuer plusieurs requêtes en même temps pour minimiser les périodes de transition.
- **Utiliser les API comme WorkManager** : Pour les tâches longues ou périodiques, en s'assurant qu'elles s'exécutent pendant que le réseau est optimal.



Effectuer une requête API avec HttpURLConnection

- Étape 1 : Créer une connexion HTTP

  Pour effectuer une requête réseau avec `HttpURLConnection`, nous devons d'abord créer une instance de `URL` et ouvrir une connexion avec `openConnection()`.

```
1 val url = URL("https://jsonplaceholder.typicode.com/users")
2 val urlConnection = url.openConnection() as HttpURLConnection
```

Effectuer une requête API avec HttpURLConnection

- Étape 2 : Configurer la connexion

  Ici, on utilise une méthode GET pour récupérer des données. On définit aussi les timeouts.

```
1 urlConnection.requestMethod = "GET"  // Utilisation de GET pour récupérer des données
2 urlConnection.connectTimeout = 5000  // Timeout de connexion
3 urlConnection.readTimeout = 5000    // Timeout de lecture
```

Effectuer une requête API avec HttpURLConnection

- Étape 3 : Lire la réponse

  On récupère la réponse sous forme de texte, en utilisant un `BufferedReader`.

```
1 val inputStream = urlConnection.inputStream
2 val response = inputStream.bufferedReader().use { it.readText() }
```

Effectuer une requête API avec HttpURLConnection

- Étape 4 : Traiter la réponse

  On suppose que la réponse est en JSON. Voici comment la parser et accéder aux informations.

```
val jsonResponse = JSONArray(response)
for (i in 0 until jsonResponse.length()) {
    val user = jsonResponse.getJSONObject(i)
    val name = user.getString("name")
    val email = user.getString("email")
    println("Name: $name, Email: $email")
}
```

Effectuer une requête API avec HttpURLConnection

- Étape 5 : Fermer la Connexion

  Enfin, on ferme la connexion pour libérer les ressources.

```
1 urlConnection.disconnect()
```

Introduction à OkHttp

- **OkHttp** est une bibliothèque HTTP développée par **Square**, une entreprise spécialisée dans la création d'outils et de bibliothèques pour améliorer le développement Android et Java.

- **Square** est également derrière d'autres bibliothèques populaires comme :
  - **Retrofit** : Un client HTTP de plus haut niveau qui fonctionne bien avec OkHttp.
  - **Moshi** : Une bibliothèque de parsing JSON qui fonctionne très bien avec Kotlin et OkHttp.
  - **Picasso** : Une bibliothèque pour le chargement et l'affichage d'images.

- **Pourquoi utiliser OkHttp ?**
  - Simplifie les requêtes HTTP.
  - Gère les erreurs de façon fluide.
  - Prise en charge du caching, des cookies, et des connexions persistantes.

Intégration d'OkHttp

- Ajouter la dépendance à `build.gradle` :

```
1 implementation("com.squareup.okhttp3:okhttp:4.10.0")
```

Requête GET avec OkHttp

- Exemple de requête avec OkHttp

```kotlin
val client = OkHttpClient()

val request = Request.Builder()
    .url("https://jsonplaceholder.typicode.com/users")
    .get() // default
    .build()

client.newCall(request).execute().use { response ->
    if (!response.isSuccessful) throw IOException("Unexpected code $response")
    val responseBody = response.body?.string()
    Log.d("API Response", responseBody ?: "No data")
}
```

- `OkHttpClient` gère la requête HTTP.
- `Request.Builder()` construit la requête.
- `newCall().execute()` envoie la requête et récupère la réponse.

Personnalisation du client HTTP avec OkHttp

```
val client = OkHttpClient.Builder()
    .addInterceptor { chain ->
        val request = chain.request().newBuilder()
            .addHeader("Authorization", "Bearer <token>")
            .build()
        chain.proceed(request)
    }
    .addInterceptor(LoggingInterceptor())  // Exemple d'intercepteur pour logs
    .connectTimeout(10, TimeUnit.SECONDS)  // Timeout de connexion
    .readTimeout(30, TimeUnit.SECONDS)     // Timeout de lecture
    .build()
```

```
// Utilisation du client personnalisé
val request = Request.Builder()
    .url("https://jsonplaceholder.typicode.com/users")
    .build()

client.newCall(request).execute().use { response ->
    if (!response.isSuccessful) throw IOException("Unexpected code $response")
    val responseBody = response.body?.string()
    Log.d("API Response", responseBody ?: "No data")
}
```

Gestion des erreurs avec OkHttp

- OkHttp permet de gérer les erreurs de façon simple grâce à son API :
  - `response.isSuccessful` : Vérifie si la réponse est réussie (code HTTP 2xx).
  - **Gestion des exceptions** : Si la requête échoue, OkHttp lèvera une exception que l'on peut capturer.

```
try {
    val response = client.newCall(request).execute()
    if (!response.isSuccessful) {
        throw IOException("Erreur lors de la requête: ${response.code()}")
    }
    // Traiter la réponse
} catch (e: Exception) {
    // Gérer l'erreur
    Toast.makeText(context, "Erreur de connexion", Toast.LENGTH_SHORT).show()
}
```

Requête Asynchrone avec OkHttp

- Avec `OkHttp`, il est facile d'effectuer des requêtes de manière asynchrone pour éviter de bloquer le thread principal.

```
val request = Request.Builder().url(url).build()
client.newCall(request).enqueue(object : Callback {
    override fun onFailure(call: Call, e: IOException) {
        runOnUiThread {
            Toast.makeText(applicationContext, "Erreur de connexion", Toast.LENGTH_SHORT).show()
        }
    }

    override fun onResponse(call: Call, response: Response) {
        if (response.isSuccessful) {
            val responseBody = response.body()?.string()
            runOnUiThread {
                // Traiter le succès
            }
        }
    }
})
```

Utiliser Retrofit avec KotlinX Serialization

Retrofit simplifie la gestion des requêtes HTTP en transformant les appels réseau en objets Kotlin. Associé à **KotlinX Serialization**, il automatise la conversion des données JSON en objets Kotlin et vice versa.

Exemple d'implémentation avec `Retrofit` + `KotlinX Serialization`

- Ajoutez les dépendances

```
[versions]
kotlin = "2.1.10"
retrofit = "2.11.0"
retrofitKotlinxSerializationConverter = "1.0.0"

[libraries]
retrofit = { module = "com.squareup.retrofit2:retrofit", version.ref = "retrofit" }
retrofit-kotlinx-serialization-converter = {
    module = "com.jakewharton.retrofit:retrofit2-kotlinx-serialization-converter",
    version.ref = "retrofit2KotlinxSerializationConverter"
}

[plugins]
serialization = { id = "org.jetbrains.kotlin.plugin.serialization", version.ref = "kotlin" }
```

Utiliser Retrofit avec KotlinX Serialization

Retrofit simplifie la gestion des requêtes HTTP en transformant les appels réseau en objets Kotlin. Associé à **KotlinX Serialization**, il automatise la conversion des données JSON en objets Kotlin et vice versa.

Exemple d'implémentation avec `Retrofit` + `KotlinX Serialization`

- Ajoutez les dépendances

```
plugins {
    ...
    alias(libs.plugins.serialization)
}

dependencies {
    ...
    implementation(libs.retrofit.kotlinx.serialization.converter)
    implementation(libs.retrofit)
}
```

Configurer le client OkHttp avec KotlinX Serialization

Afin de configurer un client OkHttp pour l'utilisation avec Retrofit et KotlinX Serialization, nous devons d'abord créer un **OkHttpClient** et configurer le **converter** pour utiliser **KotlinX Serialization**.

```kotlin
// Créer une instance de Json pour la sérialisation
val json = Json { ignoreUnknownKeys = true }  // Ignorer les clés inconnues

// Créer un OkHttpClient
val client = OkHttpClient.Builder()
    .callTimeout(DEFAULT_TIMEOUT_NS, TimeUnit.NANOSECONDS)
    .connectTimeout(0, TimeUnit.NANOSECONDS)
    .build()

// Créer Retrofit avec le converter pour KotlinX Serialization
val retrofit = Retrofit.Builder()
    .baseUrl("https://jsonplaceholder.typicode.com/") // URL de base
    .client(client)  // Utiliser le client OkHttp
    .addConverterFactory(json.asConverterFactory("application/json".toMediaType()))  // Ajouter le converter
    .build()


// Créer le service API
val apiService = retrofit.create(ApiService::class.java)
```

Définir l'interface ApiService

L'interface **`ApiService`** définit les appels API que vous voulez effectuer. Chaque méthode correspond à une requête HTTP, et les annotations Retrofit (comme `@GET`, `@POST`, etc.) permettent de lier les méthodes aux endpoints spécifiques.

```
interface ApiService {

    @GET("users")
    suspend fun getUsers(): List<User> // Récupérer la liste des utilisateurs

    @GET("users/{id}")
    suspend fun getUserById(@Path("id") id: Int): User // Récupérer un utilisateur par ID

    @POST("posts")
    suspend fun createPost(@Body post: Post): Post // Créer un nouveau post

    @PUT("posts/{id}")
    suspend fun updatePost(@Path("id") id: Int, @Body post: Post): Post // Mettre à jour un post avec PUT

    @DELETE("posts/{id}")
    suspend fun deletePost(@Path("id") id: Int) // Supprimer un post
}
```

Définir les classes de données sérialisables

Objectif : Sérialiser et désérialiser les réponses JSON avec KotlinX Serialization

```kotlin
@Serializable
data class Post(
    val userId: Int,
    val id: Int,
    val title: String,
    val body: String
)
```

- **@Serializable** : Cette annotation indique que la classe peut être sérialisée et désérialisée par KotlinX Serialization.
- Les propriétés de la classe **Post** correspondent directement aux champs JSON renvoyés par l'API de **jsonplaceholder**.
- Chaque propriété doit avoir le même nom que dans la réponse JSON pour que la sérialisation/désérialisation fonctionne automatiquement. Si nécessaire, des annotations supplémentaires comme **@SerialName** peuvent être utilisées pour spécifier des noms différents.

Utiliser l'ApiService pour effectuer une requête API

Objectif : Effectuer un appel API avec Retrofit et récupérer les données

```
class PostRepository(private val apiService: ApiService) {

    suspend fun getPosts(): List<Post> {
        val response = apiService.getPosts()

        // Vérification de la réponse et gestion des erreurs
        return if (response.isSuccessful) {
            response.body() ?: emptyList()  // Retourner les posts si la réponse est valide
        } else {
            // Gestion d'erreur, par exemple log ou lancer une exception
            emptyList()
        }
    }

    suspend fun getPostById(id: Int): Post? {
        val response = apiService.getPostById(id)

        return if (response.isSuccessful) {
            response.body()  // Retourner le post si la réponse est valide
        } else {
            null  // Retourner null si l'appel échoue
        }
    }
}
```

Conclusion

- **Compréhension des bases** :
  - Utilisation de `HttpURLConnection` pour effectuer des requêtes réseau manuelles.
  - Simplification avec **OkHttp**, plus moderne et flexible pour gérer les connexions réseau.
- **Outils modernes** :
  - **Retrofit** : Une bibliothèque pour automatiser les requêtes réseau et gérer la conversion JSON, facilitant ainsi l'intégration avec les APIs.
- **Avantages de Retrofit et OkHttp** :
  - Simplicité, flexibilité et personnalisation des requêtes réseau.
  - Prise en charge facile des formats de données modernes comme JSON, avec moins de code.