DAA - Développement d'applications Android

Communication Web

02 January 2026

Table des matières

1 Connectivité d'un smartphone

Un smartphone possède plusieurs technologies permettant de se connecter à internet, elles sont:

- Le Wi-Fi
- Les réseaux mobiles (2, 3, 4 et 5G)

Un mobile peut-être connecté à la fois en Wi-Fi et en réseau mobile. Le Wi-Fi n'est pas toujours connecté à internet.

1.1 Wi-Fi

Norme	Nom	Date	Fréquences	Débit maximum
802.11	N/A	1997	2.4 GHz	2 Mbps
802.11b	Wi-Fi 1	1999	2.4 GHz	11 Mbps
802.11a	Wi-Fi 2	1999	5 GHz	54 Mbps
802.11g	Wi-Fi 3	2003	2.4 GHz	54 Mbps
802.11n	Wi-Fi 4	2009	2.4 GHz ou 5 GHz	72 Mbps 450 Mbps
802.11ac	Wi-Fi 5	2014	5 GHz	1'000 Mbps
802.11ax	Wi-Fi 6	2019	2.4 et 5 GHz	2'400 Mbps
802.11ax	Wi-Fi 6E	2021	2.4, 5 et 6 GHz	4'800 Mbps
802.11be	Wi-Fi 7	2024	2.4, 5 et 6 GHz	30'000 Mbps
802.11bn	Wi-Fi 8	2028 ?	2.4, 5 et 6 GHz	100'000 Mbps

Fig. 1. – Capture des slides du cours – Norme Wi-Fi

La fréquence 2.4 GHz permet une meilleure portée mais des débits plus faibles. La fréquence 5 GHz permet des débits plus élevés mais une portée plus faible.

1.2 Réseaux mobiles

Génération	Année	Données (pointe)	Latence	Remarques
1G	1983	ø	ø	Système analogique
2G	1992	Kbit / s	< 1000 ms	Premier réseau numérique, initialement uniquement voix
3G	2003	Mbit / s	< 500 ms	Intégration des données dès la conception
4G	2010	Gbit / s	< 100 ms	Réseau données uniquement, intégration ultérieure de la voix
5G	2019+	Gbit / s	< 5 ms	Ouverture à l'Internet des Objets (IoT)

Fig. 2. – Capture des slides du cours – Norme Réseaux Mobiles

1.3 Connectivité

Dans une application Android, le SDK fournit des API pour gérer la connectivité réseau. La classe principale pour cela est `ConnectivityManager`. Pour accéder à cette classe, il est nécessaire d'avoir la permission `ACCESS_NETWORK_STATE` dans le fichier manifeste de l'application.

Pour que l'application puisse accéder à internet, il est également nécessaire d'ajouter la permission `INTERNET` dans le manifeste.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android" xmlns:tools="http://schemas.
android.com/tools">
    <uses-permission android:name="android.permission.INTERNET" />
</manifest>
```

> **i Info**
>
> Il ne s'agit pas d'une permission dangereuse, donc l'utilisateur n'a pas besoin de l'accepter explicitement.

1.3.1 Accès à Internet depuis une app Android

Dans le fichier manifeste, il est aussi possible de spécifier un fichier de configuration réseau via l'attribut `android:networkSecurityConfig`. Ce fichier permet de définir des règles de sécurité pour les connexions réseau, comme l'acceptation de certificats auto-signés ou la configuration des domaines autorisés.

Toutes les API réseau de Java sont également disponibles dans Android, par exemple on peut ouvrir un socket TCP/IP avec la classe `Socket` de Java.

```
thread {
  val resolutions = Inet4Address.getAllByName("www.heig-vd.ch").first()
  val socketAddress = InetSocketAddress(resolutions.hostAddress, 80)
  val socket = Socket()
  try {
    socket.connect(socketAddress)
    Log.d("MainActivity", "Host ${resolutions.hostName} is reachable")
  }
  catch (_: Exception) {}
  finally { socket.close() }
}
```

2 Services Web

2.1 Services REST

La classe `java.net.URL` permet d'interroger une API REST via HTTP. Voici un exemple d'appel GET:

```
val url = URL("https://www.fruityvice.com/api/fruit/all")
thread {
  val json = url.readText(Charsets.UTF_8)
  Log.d("MainActivity", json)
}
```

- URL effectue un appel `GET` par défaut.
- Le payload est souvent dans un format textuel comme JSON ou XML.

2.1.1 Désérialisation JSON

Pour convertir le JSON reçu en objets Kotlin, on peut utiliser une bibliothèque comme Gson. Pour cela, il faut créer des classes de données correspondant à la structure du JSON. On appelle cela des DTO (Data Transfer Object).

```
val url = URL("https://www.fruityvice.com/api/fruit/all")
thread {
  val json : String = url.readText(Charsets.UTF_8)
  val type = object : TypeToken<List<FruitDTO>>() {}.type
  val fruits = Gson().fromJson<List<FruitDTO>>(json, type)
  Log.d("MainActivity", "$fruits")
```

2.1.2 Appel POST / PUT

Il est également possible d'effectuer des appels POST ou PUT, cela permettant d'envoyer des données au serveur. Dans l'exemple précédent, nous pouvons venir ajouter un fruit en effectuant un appel PUT.

```
val url = URL("https://www.fruityvice.com/api/fruit")
thread {
  // on sérialise le nouveau fruit
  val json = Gson().toJson(newFruit)
  // on prépare la connexion
  val connection = url.openConnection() as HttpURLConnection
  connection.requestMethod = "PUT"
  connection.doOutput = true
  connection.setRequestProperty("Content-Type", "application/json")
  // on envoie le payload au service REST
  connection.outputStream.bufferedWriter(Charsets.UTF_8).use {
    it.append(json)
  }
  // on traite la réponse du service REST
  val responseCode = connection.responseCode
  Log.d("MainActivity", "responseCode: $responseCode")
  connection.inputStream.bufferedReader(Charsets.UTF_8).use {
    Log.d("MainActivity", it.readText())
  }
}
```

> **i Info**
>
> - L'utilisation de la librairie se fait de manière synchrone, certaines méthodes sont bloquantes en attendant la réponse du serveur.
> - La conenxion avec le serveur se fait lors de l'appel à `outputStream`, `inputStream` ou `responseCode`.
> - La même connexion peut-être réutilisée automatiquement pour effecteur plusieurs appels successifs.

2.2 Alternatives à `java.net.URL`

`java.net.URL` dispose d'une assez « mauvaise presse » sur Android. Plusieurs bibliothèques alternatives existent pour faciliter les appels réseau, parmi les plus populaires on trouve:

- OkHttp
- Volley
- Retrofit

Les principaux ajouts de ces librairies sont la possibilité de planifier plusieurs requêtes dans un certain ordre, d'annuler des requêtes, de géere un cache, de permettre de planifier plusieurs tentatives en cas d'échec et également de faciliter les opérations de sérialisation/désérialisation.

2.2.1 Comparaison java.net.URL vs Volley

2.2.1.1 Chargement code HTML

java.net.URL :

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

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

Volley :

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

- Pas de gestion des threads, on donne deux callbacks (succès et erreur) que la librairie appellera dans l'*UI-Thread*
- On précise vouloir traiter du texte en retour en utilisant une *StringRequest*
- Utilisation d'une queue qui va gérer l'exécution des requêtes

Fig. 3. – Capture des slides du cours – Comparaison java.net.URL vs Volley - Chargement code HTML

2.2.1.2 Chargement image

java.net.URL :

```
val url = URL("https://www.heig-vd.ch/logo.png")
thread {
  val bytes = url.readBytes()
  val bmp = BitmapFactory
        .decodeByteArray(bytes,0, bytes.size)
  runOnUiThread { imageView.setImageBitmap(bmp) }
}
```

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

Volley :

```
val url = " https://www.heig-vd.ch/logo.png "
val queue = Volley.newRequestQueue(this)
val imgRequest = ImageRequest(url,
  { bmp -> imageView.setImageBitmap(bmp) },
  1024, 1024,
  ScaleType.CENTER_INSIDE,
  Bitmap.Config.ARGB_8888,
  { error -> Log.d("MainActivity", "${error.message}") })
queue.add(imgRequest)
```

- Utilisation d'une *ImageRequest* pour traiter un payload contenant une image, la librairie va convertir l'image chargée (changement de l'encodage, redimensionnement, crop)

Fig. 4. – Capture des slides du cours – Comparaison java.net.URL vs Volley - Chargement image

2.2.1.3 Chargement JSON

java.net.URL :

```
val url = URL("https://www.fruityvice.com/api/fruit/all")
thread {
  val json = url.readText(Charsets.UTF_8)
  val type = object : TypeToken<List<FruitDTO>>() {}.type
  val fruits = Gson().fromJson<List<FruitDTO>>(json, type)
  runOnUiThread { txtView.text = fruits.toString() }
}
```

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

Volley :

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

- L'endpoint retournant un tableau, on doit utiliser une *JsonArrayRequest*
- L'objet *json* qui est passé au callback est du type ***org.json**.JSONArray* qui n'est pas directement exploitable

Fig. 5. – Capture des slides du cours – Comparaison java.net.URL vs Volley - Chargement JSON

2.2.1.4 Comparaison

- `java.net.URL` étant synchrone, on peut facilement l'utiliser sous la forme d'une coroutine.

```
suspend fun downloadHTML(urlParam : String) : String = withContext(Dispatchers.IO) {
  val url = URL(urlParam)
  url.readText(Charsets.UTF_8)
}
```

- La coroutine va retourner la dernière valeur du bloc, ici la valeur de retour de `readText`, soit le code HTML de la page sous forme d'un string, dès qu'elle est disponible.
- Volley utilisant des callbacks on peut pas directement l'utiliser dans une coroutine. Il est cependant possible de le faire en utilisant la méthode `suspendCoroutine` de Kotlin.

```
suspend fun downloadHTMLVolley(urlParam : String) : String = suspendCoroutine { cont ->
  val textRequest = StringRequest(Request.Method.GET, urlParam,
    { response -> cont.resume(response) },
    { e -> cont.resumeWithException(e) })
  queue.add(textRequest)
} // arrivé à la fin du bloc, la coroutine va se suspendre en attendant qu'un callback appelle
resume()
```

Cette méthode permet de transformer un callback en une coroutine suspendue. Lorsque la réponse est disponible, on appelle `cont.resume(response)` pour reprendre la coroutine avec la valeur de la réponse. En cas d'erreur, on appelle `cont.resumeWithException(e)` pour reprendre la coroutine avec une exception.

Si on souhaite spécifier le Dispatcher pour exécuter le callback, on peut utiliser `withContext` pour englober l'appel à `suspendCoroutine`.

```
withContext(Dispatchers.IO) {
  val htmlByVolley = downloadHTMLVolley(url)
}
```

2.3 Ktor

Ktor est un framework asynchrone pour construire des applications connectées. Il peut être utilisé à la fois côté serveur et côté client. Ktor Client permet de faire des requêtes HTTP de manière simple et efficace.

Exemple d'utilisation de Ktor Client pour faire une requête GET dans un viewModel:

```
private val ktorClient = HttpClient(Android) {
    // Configuration du moteur Android
    engine {
        connectTimeout = 5_000
        socketTimeout = 5_000
        dispatcher = Dispatchers.IO
    }

    // Plugin pour parser le JSON
    install(ContentNegotiation) {
        json(Json {
            ignoreUnknownKeys = true
            prettyPrint = true
            isLenient = true
        })
    }
}
```

```
private suspend fun requestFruits(): List<FruitDTO> {
    // 1. On lance la requête GET
    // 2. .body() convertit automatiquement le JSON en List<FruitDTO>
    return ktorClient
            .get("https://www.fruityvice.com/api/fruit/all")
            .body()
}


fun getFruits() {
    // La requête s'exécute dans le scope du ViewModel
    viewModelScope.launch {
        val allFruits = requestFruits()
        ...
    }
}
```

Fig. 6. – Capture des slides du cours – Exemple Ktor Client dans un viewModel

3 Synchronisation des données

Très souvent une application mobile va devoir être conçue pour fonctionner hors-ligne. Dans ce cas, il est nécessaire de synchroniser les données locales avec un serveur distant lorsque la connexion internet est disponible.

L'application devra travailler sur des données dans une DB locale, et effectuer la synchronisation avec le serveur distant en arrière-plan quand une connexion est disponible.

L'algorithme de synchronisation des données n'est généralement pas trivial. Dans le cas où plusieurs mobiles peuvent modifier des données en local, cela nécessite un mécanisme avancé de synchronisation et de résolution des conflits

3.1 Exemple de synchronisation

Dans un premier temps, l'application aura une base de données locale vide et devra télécharger les données depuis le serveur distant.

Fig. 7. – Capture des slides du cours – Exemple de synchronisation des données - étape 1

Le téléchargement des données peut se faire lors du premier lancement de l'application ou lorsque l'utilisateur demande une actualisation des données.

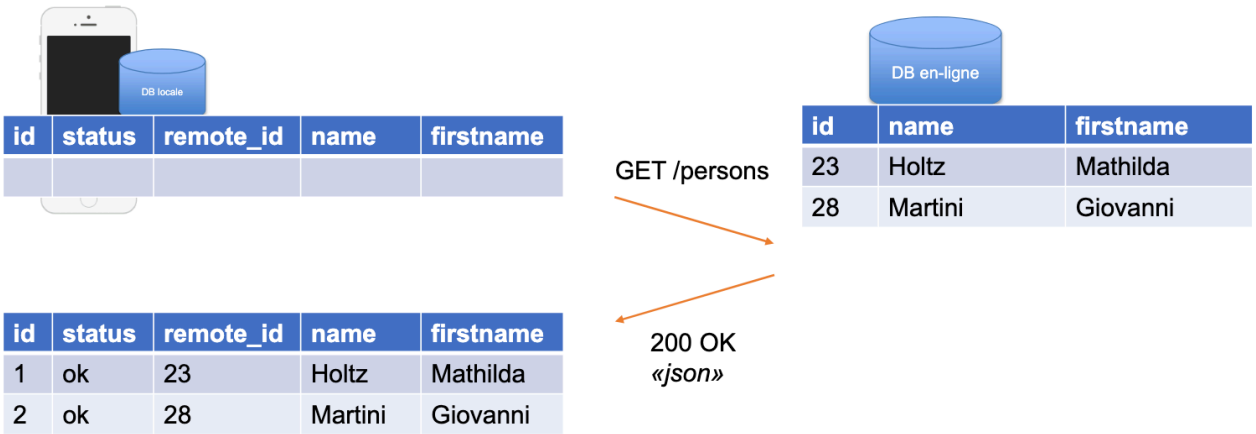

Fig. 8. – Capture des slides du cours – Exemple de synchronisation des données - étape 2

- On notera que dans la base de donnée locale, chaque enregistrement possèdera un champ `id` local ainsi qu'un champ `remoteId` correspondant à l'identifiant de l'enregistrement sur le serveur distant.
- De plus, un champ `status` permettra de savoir si l'enregistrement a été modifié localement et doit être synchronisé avec le serveur distant.

Si on crée un nouvel enregistrement localement, on lui attribue un `id` local unique, un `remoteId` à `null` et un `status` à `new`.

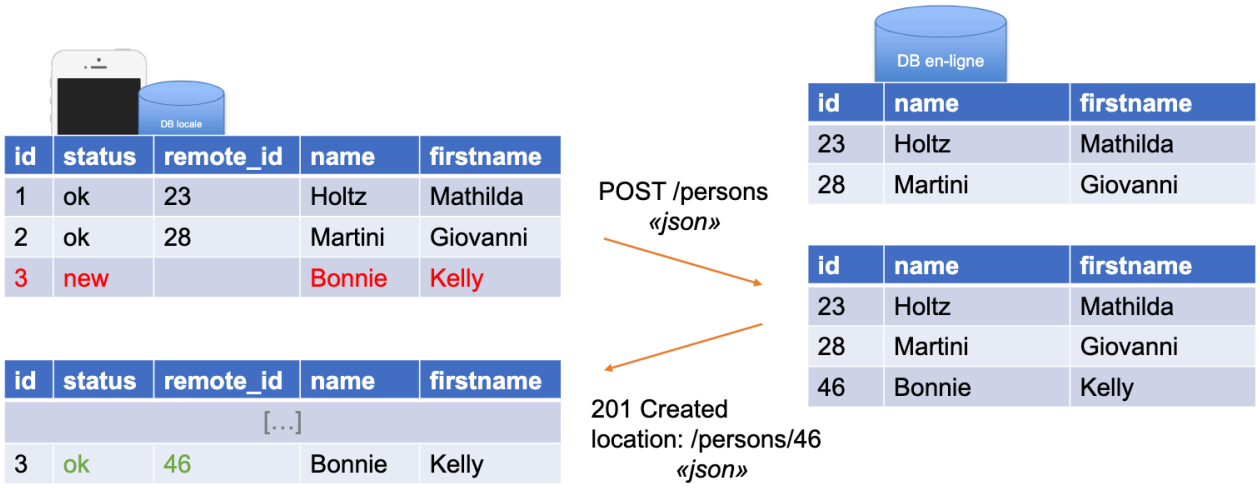

Fig. 9. – Capture des slides du cours – Exemple de synchronisation des données - étape 3

- Lors de la prochaine synchronisation, l'application détectera l'enregistrement avec le statut `new` et effectuera une requête POST vers le serveur distant pour créer l'enregistrement. Le serveur retournera un `remoteId` qui sera stocké dans la base de données locale, et le statut sera mis à jour à `ok`.

Si on modifie un enregistrement existant, on met à jour son statut à `mod`. Lors de la synchronisation, une requête PUT sera effectuée vers le serveur distant pour mettre à jour l'enregistrement.

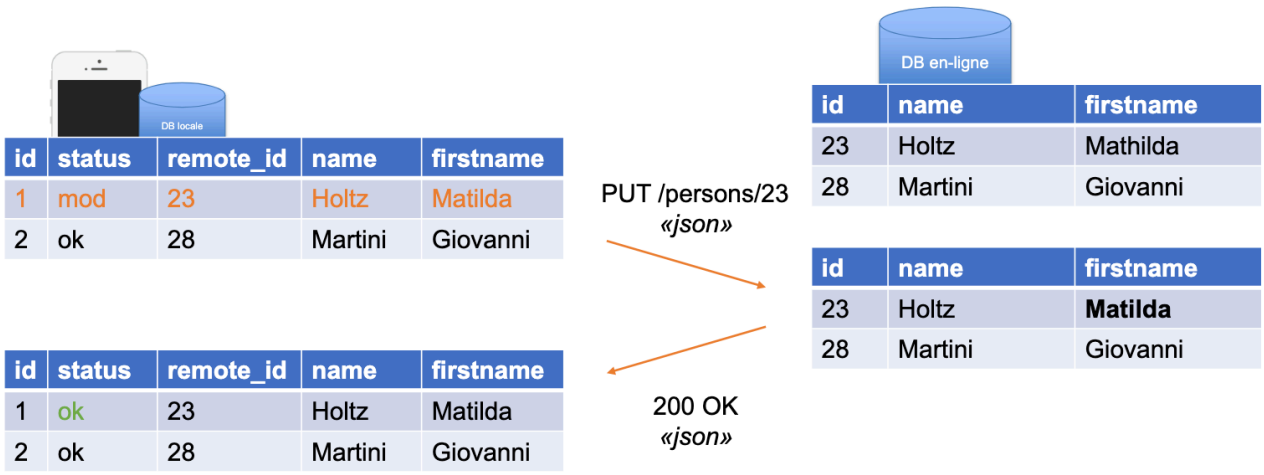

Fig. 10. – Capture des slides du cours – Exemple de synchronisation des données - étape 4

Si on supprime un enregistrement localement, on met à jour son statut à `del`. Lors de la synchronisation, une requête DELETE sera effectuée vers le serveur distant pour supprimer l'enregistrement.

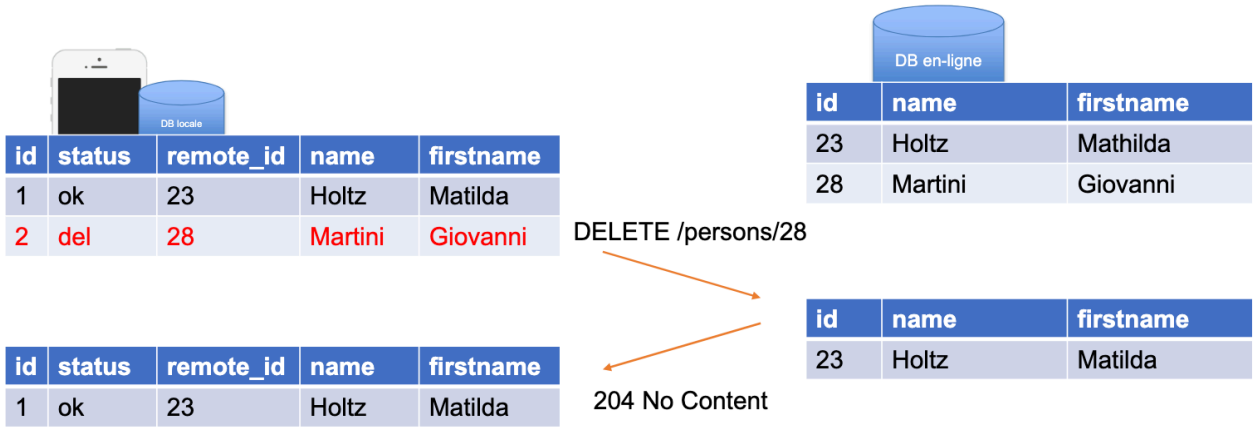

Fig. 11. – Capture des slides du cours – Exemple de synchronisation des données - étape 5

3.2 Single Source of Truth

Le principe de Single Source of Truth (SSOT) consiste à avoir une seule source de données fiable dans une application. Dans le contexte d'une application mobile avec synchronisation de données, cela signifie que la base de données locale doit toujours refléter l'état le plus récent des données, que ce soit localement ou sur le serveur distant.

Dans le cas d'une application Android, le **Repository** de l'app va être responsable de gérer cette source unique de vérité. Le **Repository** va interagir avec la base de données locale pour lire et écrire des données, et il va également gérer la synchronisation avec le serveur distant.