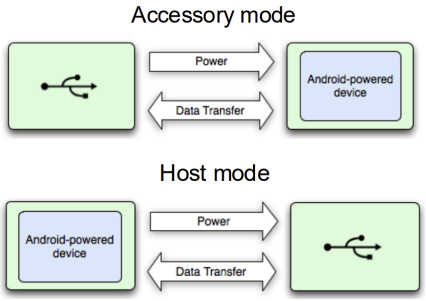
1 Protocoles de proximité

1.1 USB (Universal Serial Bus)

L'USB, présent sur **tous les mobiles récents**, est fiable pour transférer de **grosses quantités de données**. Il est directionnel (périphérique vers hôte). Initialement, smartphones = périphériques ; aujourd'hui, ils peuvent être **hôtes** (alimenter/recevoir des périphériques).



1.1.1 USB-C

Le connecteur **USB-C est réversible** (24 pins) et permet échange de données (multi-standards) et alimentation. **Attention : capacités variables** (charge, vitesse, standards : USB 2.0 à 4.0, Thunderbolt 4, puissance 60W-240W) malgré une apparence identique.

1.1.2 USB-C - Quelques exemples de standards

Normes USB (débit) :

- USB 2.0 : **480 Mbps** (2000)
- USB 3.2 Gen 1 (USB 3.0) : **4 Gbps** (2008)
- USB 3.2 Gen 2 (USB 3.1) : **10 Gbps** (2013)
- USB 3.2 Gen 2x2 : **20 Gbps** (2017)
- USB 4 Gen 3x2 : **40 Gbps** (2019)
- USB 4 Gen 4 : **80 Gbps** (2022)

USB Power Delivery (PD) permet une charge jusqu'à **240W** ($48V \cdot 5A$), si supporté.

Thunderbolt utilise aussi USB-C.

1.1.3 USB - Android

Périphériques USB communs (stockage, clavier, souris, Ethernet, manette) sont **souvent supportés par défaut** sur Android (compatibilité variable). Le SDK Android permet d'intégrer des **pilotes USB spécifiques** (ex: caméra).

1.1.4 USB - iOS

Sur iOS, accessoires non reconnus nativement nécessitent le **programme MFi** *(Made For iPhone)*. Ce programme Apple certifie et donne accès à des ressources (API Bluetooth, CarPlay, Find My). Périphériques peuvent nécessiter une **puce MFi**. Lightning = USB 2.0 (480 Mbps). iPhone 15 marque la transition vers **USB 3.2 Gen 2 (10 Gbps)**.

1.1.5 USB - iOS - Alternatives

Le port jack (supprimé en 2016) était une parade MFi ; adaptateurs Lightning/jack requièrent aussi MFi. Cartes Ethernet supportées nativement. Protocoles réseau possibles (ex: webcam via boîtier RTP).

2 Les codes-barres

2.1 Qu'est-ce qu'un code-barres ? - Origines

Code-barres : données (num/alpha) en barres claires/foncées (épaisseur/espacement), lisibles par machines. 1961 : wagons. 1974 : 1er produit en caisse.

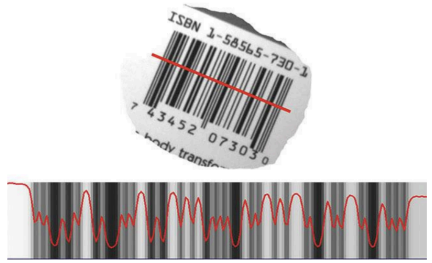
2.2 Différents types de codes-barres unidimensionnels (1D)

Codes 1D avec leurs jeux de caractères :

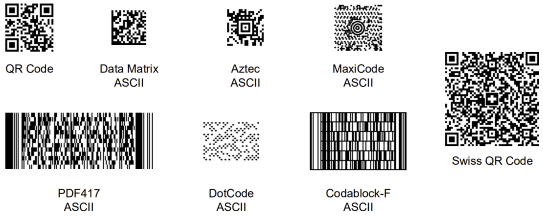
- **Code-11 :** `[0-9] + "-"` (ex: 123456)
- **Code-39 :** `[A-Z] [0-9] [ - . $ / + % ]` (ex: AB12)
- **Code-93 :** `[A-Z] [0-9] [_ - . $ / + %]` (ex: AB-12/+)
- **Code-128 :** ISO-8859-1 (ex: Eeéè)
- **EAN-13 :** `[0-9]` – longueur 13 (ex: 7601234568903)
- **UPC-A :** `[0-9]` – longueur 12 (ex: 760123456787)
- **DAFT :** `[DAFT]`

2.3 Lecture d'un code-barres 1D

Historiquement : **faisceau laser**, analyse lumière réfléchie. Efficace pour 1D. **Faible densité d'info** (identifiants) ; applications récentes demandent plus.



2.4 Différents types de codes-barres bidimensionnels (2D)



2.5 Lecture d'un code-barres 2D

Lecture via **analyse photo**. QR Code : motifs de position, alignement, format. Image parfaite difficile (reflets). **QR Codes intègrent code correcteur d'erreurs** :

4 niveaux disponibles L(7%), M(15%), Q(25%), H(30%). Permet lecture même si partiellement endommagé.

2.6 Différentes tailles de Codes QR

QR Codes : **longueur variable**, 4 modes (max. 7'089 chiffres, 4'296 alphanum, 2'953 ISO-8859-1 et 1'817 Kanji). Taille (Version) de 1 (21·21, 152 bits L) à 40 (177·177, 23'648 bits L).

2.7 Différents types de Codes QR (contenu structuré)

Formats standardisés : `tel:`, `https:`, `mailto:`, `geo:`, `WIFI:`, `VEVENT` (calendrier), `VCARD` (carte de visite).

```
BEGIN:VEVENT
SUMMARY:Festival
DTSTART:20230728T160000Z
DTEND:20230728T213000Z
LOCATION:Bord de plage
END:VEVENT
```

2.8 Différents types de Codes QR - Texte libre

Mode texte libre pour besoins spécifiques (ex: JSON, données certif. Covid encodées Base45).

2.9 Différents types de Codes QR - Exemple GS1

D'ici 2027, QR Code GS1 remplacera code barres EAN. Contiendra URI produit (digital link : num lot/série, infos nutri, marketing), identifiant GS1(déjà contenu dans EAN), métadonnées (série, lot, date exp.). Utilisable sur tags NFC.

2.10 Différents types de Codes QR - Dynamiques

QR Codes dynamiques : URL redirigeant vers contenu réel. Permet **màj contenu, redirection selon matériel du client, URL courtes, obtention de stats d'utilisation**. Confiance fournisseur cruciale (Disponibilité, sécurité, confidentialité).

2.11 Codes QR - Lecture sur mobile

Souvent via app appareil photo native. Apps tierces existent (parfois publicitaires). Android : librairies **`zxing`** (Java, maintenance) et **`ML Kit`** (Google, ML) pour intégration.

2.12 Codes QR - Utilisations sur smartphone

Smartphone = lecteur. Librairie **`zxing`** peut aussi générer/afficher QR Codes.

3 Le NFC (Near Field Communication)

3.1 NFC - Near Field Communication (Généralités)

NFC : sous-classe RFID. Tag passif alimenté par lecteur. **Très courte portée** (max 3-4cm), **communication bidirectionnelle**. Majorité smartphones Android (SDK complet) et iOS (iPhones/Watch) équipés. iOS : accès API progressif (lecture NDEF 2017, écriture 2019). UE accuse Apple de limiter accès NFC (paiements).

3.2 NFC - Near Field Communication (Modes de fonctionnement)

Trois modes NFC :

- **Mode émulation de carte (HCE) :** Mobile = carte sans contact (ex: paiement).
- **Lecture / Ecriture :** Mobile lit/écrit tags passifs (stockage, actions (ouvrir url, etc)).
- **Mode peer-to-peer :** Échange direct entre deux appareils NFC.

3.3 NFC - Lecture / Ecriture de tags (Technologies et Types)

Technologies NFC : NFC-A/B/F/V, MifareClassic/Ultralight. Types de tags 1 à 5 (capacités variables : ex NTAG210µ 48B, NTAG216 888B).

3.4 NDEF - NFC Data Exchange Format

NDEF : format standardisé pour messages sur tags NFC / entre smartphones. **Well-Knowns** : URI (http:, tel:, mailto:, etc.), TEXT (premier byte encodage), SMARTPOSTER(URI + metadata (titre, logo)). API Android : bas et haut niveau. iOS : plus abstrait (nouvelles API spécifiques pour devs/pays sélectionnés car apple pue).

3.5 Utilisation de NFC sur Android (Permissions et Intent-filter)

Permission

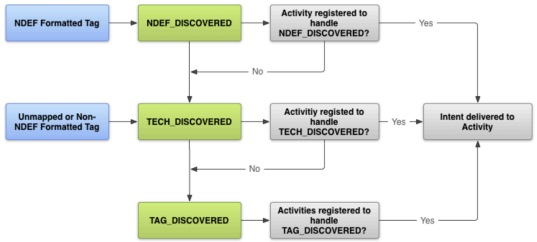
```
<uses-permission android:name="android.permission.NFC" />
```

Inscription via `intent-filter` (Manifest ou `ForegroundDispatch` runtime) pour notification lecture tag.

```
<intent-filter>
  <action android:name="android.nfc.action.NDEF_DISCOVERED" />
  <category android:name="android.intent.category.DEFAULT" />
  […]
</intent-filter>
```

3.6 Utilisation de NFC sur Android (Niveaux d'abstraction)

`Intent` informe activité. 3 niveaux : `ACTION_NDEF_DISCOVERED` (payload NDEF), `ACTION_TECH_DISCOVERED` (technos A,B,F,V), `ACTION_TAG_DISCOVERED` (tout tag NFC sera lu).



4 Le Bluetooth

4.1 Le Bluetooth « Classique » (Historique et Évolution)

Développé en fin 1990, alternative sans-fil USB (périphériques vers hôte, connexion bi-directionnelle chiffré après appairage). Portée ~10m, peu mobile et trop gourmant en énergie.

Évolution : 1.0 (1999, ~721 kbits), 2.0 EDR (Enhanced Data Rate) (2006 ~2,1 Mbits), 3.0 HS (High Speed) (2009), **4.0 (2010, intro BLE)**, 5.x (nouveautés BLE), 6.0 (prévu 2024, Channel Sounding (localisation)).

4.2 Le Bluetooth « Classique » (Profils)

Profils : A2DP (audio), HSP/HFP (mains-libres), PBAP (contacts), AVRCP (multimédia), PAN (internet), HID (clavier/souris).

4.3 Le Bluetooth « Classique » (Support OS et custom)

Support natif varie. iOS gère HFP, A2DP, etc. Périphériques "custom" possible : ajout profils sur Android (complexe) ; Uniquement via **MFi Program sur iOS**.

4.4 Le Bluetooth Low Energy (BLE) (Généralités)

BLE : techno cousine, indépendante (non compatible) du Classique. Puces souvent `dual-mode` (Classique et BLE).

Vise **faible conso** : portée 5-100m, débit ~1Mbits (~100kbits utile), petits périph. sur pile. V5.0 (2016) : **double portée *ou* débit, réseaux maillés (GRE tu sais)**.

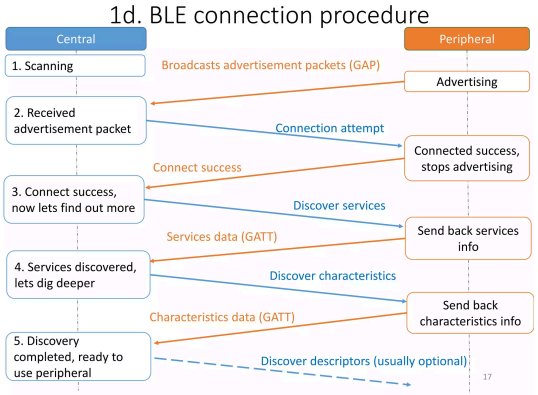
4.5 Le Bluetooth Low Energy (BLE) (Topologie Client/Serveur)

Topologie **client/serveur** : `Central device` (client : tel/tablette) vs `Peripheral devices` (serveurs : capteurs/montres).

4.6 Le Bluetooth Low Energy (BLE) (Phases GAP et GATT)

Deux phases BLE :

- **GAP (Generic Access Profile) :** Avant connexion, diffusion infos périph (nom, services, poss de conn?).
- **GATT (Generic Attribute Profile) :** Après connexion, structure échange données (services, caractéristiques, descripteurs).



4.7 Le Bluetooth Low Energy (BLE) (Services et UUIDs)

`Service` BLE = ensemble de `Characteristics` (variables) pour une fonctionnalité.

Services standards (ex: Battery 0x180F) avec UUIDs 16 bits (implicite 128).

`0x1805` → `00001805-0000-1000-8000-00805f9b34fb` Services proprios : UUIDs 128 bits.

4.8 Le Bluetooth Low Energy (BLE) (Modes de communication)

Modes : **connecté** (clair, aka périphérique publie ses services et tt le monde peut s'y connecter) ou **appairé/bondé (chiffré après échange des clés)**. Appairage : `Just Works™` (vulnérable MITM), `Out of Band` (NFC/Wi-Fi), `Passkey` (PIN, il faut avoir un clavier et un écran), `Numeric comparison` (BLE 4.2+ et 2 écrans nécessaires).

4.9 Le Bluetooth Low Energy (BLE) (Service et Operations de Characteristic)

`Service` contient une ou plusieurs `Characteristics`. Chaque `Characteristic` expose opérations (Read, Write, Notify, Indicate), obligatoires/optionnelles/interdites.

Permissions typiques :

- **Read :** Lire valeur actuelle
- **Write :** Modifier paramètres/commandes
- **Notify :** Recevoir mises à jour automatiques (unidirectionnel)
- **Indicate :** Comme Notify mais avec accusé réception

4.10 Le Bluetooth Low Energy (BLE) (Détails d'une Characteristic)

`Characteristic` : valeur (int, float, string, binaire), défaut **20Bytes (max 512B)**. Opérations : Lecture, Ecriture, Indication/Notification (Le Central doit s'y abonner). Peut avoir `Descriptors` (métadonnées).

4.11 Le Bluetooth Low Energy (BLE) (Structure GATT - Exemple)

GATT : Services > Characteristics (avec propriétés : Read, Notify) > Descriptors.

4.12 Le Bluetooth Low Energy (BLE) - Exemple (Current Time Characteristic)

`Characteristic Current Time` (10B) : [b0,b1 UInt16 LE] Année, [b2, UInt8] Mois, [b3, UInt8] Jour, [b4, UInt8] Heure, [b5, UInt8] Min, [b6, UInt8] Sec, [b7, UInt8] Jour sem, [b8, UInt8] Fractions256, [b9, UInt8] Raison ajust.

4.13 Le Bluetooth Low Energy (BLE) - Evolutions

- **5.0 (2016) :** Débit `2M PHY` / portée `Coded PHY` augmentés, **réseaux maillés**.
- **5.1 (2019) :** Angle arrivée/départ (**localisation**).
- **5.2 (2020) :** Profil audio BLE (**`LE Audio`**) remplace A2DP.
- **5.3 (2021) :** Optimisations.
- **5.4 (2023) :** Chiffrement annonces.

5 Autres que BLE

- **Wi-Fi direct :** Connexions p2p Wi-Fi pour gros échanges de données.
- **Google Nearby :** Librairie Android pour **com. p2p avancée** (BT, Wi-Fi, audio/ultrasons).
- **AirDrop :** Apple (MacOS, iOS, ipadOS) partage fichiers (BT, Wi-Fi).

6 Les capteurs et les wearables

6.1 Capteurs

Tendance à augmenter le nombre de capteurs plus on avance.

Capteurs simples = luminosité [lx], pression [hPa], proximité [cm] (en général binaire on/off), humidité [%], température [°C].

Capteurs mouvements attention aux système d'axe, pour natel : x, y sur l'écran et z parrallèle à celui-ci (+z face à l'écran, -z dos du natel) :

- **Accéléromètre** = indique sur chaque axe [m * s^(-2)], au repos accélération de la gravité, utile pour mouvement peu précis (secouement, immobilité, marche/course).
- **Magnétomètre** = indique pour chaque axe l'intensité du champ magnétique [*micro* T], utile pour positionnement par rapport au nord magnétique mais facilement perturbé (aimant, masse métallique).
- **Gyroscope** = meilleur que l'accéléromètre mais moins répandu et plus consommateur d'énergie, indique la vitesse de rotation sur les 3 axes [rad * s^(-1)], utile pour la VR ou AR.

6.1.1 Accès capteurs Kotlin

Comme *LocationManager* on accède aux capteurs par le *SensorManager*, exemple inscription à accéléromètre :

```
val sensorManager = getSystemService(SENSOR_SERVICE) as
SensorManager
val accelerometer = sensorManager.getDefaultSensor(
  Sensor.TYPE_ACCELEROMETER)
// inscription au capteur
// ici listener est un SensorEventListener, on peut
utiliser "this" si on est dans une activité p.ex.
sensorManager.registerListener(listener, accelerometer,
SensorManager.SENSOR_DELAY_NORMAL)
// […]
// désinscription
sensorManager.unregisterListener(listener)
```

On peut donner une **précision temporelle** : `SENSOR_DELAY_NORMAL`, `SENSOR_DELAY_UI`, `SENSOR_DELAY_GAME`, `SENSOR_DELAY_FASTEST` (Android 12 si fréquence > 200Hz alors permissions particulières).

Attention => toujours se **désinscrire** d'un capteur si plus nécessaires car résultats dans le Thread-UI donc pas ralentir inutilement.

Le *SensorManager* donne résultats à un *SensorEventListener* (p.ex. Activité) qui doit avoir :

```
// appelée à chaque nouvelle valeur
override fun onSensorChanged(event: SensorEvent) {
  when(event.sensor.type) {
    // event.values est un tableau de float de données
brutes, sa taille dépend du capteur
    Sensor.TYPE_ACCELEROMETER -> {
      event.values // float[3]
    }
    Sensor.TYPE_LIGHT -> {
      event.values // float[1]
    }
  }
}
override fun onAccuracyChanged(sensor: Sensor?, accuracy:
Int) {}
```

Attention => comme nous somme sur une **JVM** nous avons donc un *garbage collector*, si l'on doit créer beaucoup d'instance alors celui-ci peut prendre, dans des cas extrêmes, plus de ressources que l'application. Cela peut être le cas avec les capteurs car on peut devoir reporter un nouvel *SensorEvent* plusieurs centaines de fois par secondes. Dans ce cas alors on aura un **pool d'instances** qui sera géré par le *SensorManager* qui modifiera des *SensorEvent* existant pour éviter d'en recréer. Donc il ne faut **pas garder de références** sur les *SensorEvent* car ils pourront être modifier par la suite (copie, traitement immédiat).

Les données brutes récupérées ne sont pas toujour utiles seules donc Le *SensorManager* met à disposition des méthodes pour l'interprétation de ces données. Exemple avec la pression pour en déduire l'altitude :

```
// p0 pression au niveau de la mer, p pression actuelle
SensorManager.getAltitude(float p0, float p)
```

Pour gyro, accel ou magnéto les données brutes sont une matrice de rotation 3D. La position de référence est l'axe -z aligné sur le centre de gravité de la terre et +y aligné sur le nord magnétique.

```
SensorManager.getRotationMatrix( float[] R, // matrice de
rotation
  float[] I, // matrice d'inclinaison
  float[] gravity, // Accéléromètre
  float[] geomagnetic) // Magnétomètre
```

Matrice de transformations 3D :

6.1.2 Capteurs composites

Des capteurs virtuel qui fusionnent et traite les données de capteurs physiques :

- **Step counter/detector (podomètre)** = basé en général sur accéléromètre
- **Game rotation vector** = gyro + accél + magnéto
- **Linear Acceleration** = accél + gyro ou magnéto
- …

6.1.3 Analyse avancée capteurs

On peut reconnaître des activités humaines grâce aux capteur (si fitness, transport, se déplace, …).

4 groupes d'attribut :

- **Environnemental** = temp, bruit, etc.
- **Mouvement** = accél, gyro, magnéto
- **Localistion** = GPS, etc.
- **Physiologique** = temp corporelle, rythme cardiaque, etc.

1 seul capteur ne suffira pas en général pour déterminer avec précision l'activité.

On doit avoir la **permission** `ACTIVITY_RECOGNITION` depuis Android 10 (2019) pour le podomètre et autres librairies. Mais l'accès aux données brutes (< 200 Hz) est toujours possible.

Utilisation détournée = écoute de discussions grâce à l'accél à partir des vibrations provoquée par le micro, détérminer appui de touche du clavier à partir de rotation, etc.

6.2 Wearables

Donc vêtements ou accessoire avec éléments informatiques et éléctroniques (dans IoT). Toujours disponible et fonctionnalités en tout temps.

3 catégories :

- **Tracking, Measuring, Sensing** = capteurs corporels, interface limitée, faible consommation, données analysée et affichées.
- **Réalité améliorée** = aide vie courante, navigation, rappels, etc.
- **Second écran** = extension du natel pour tâche simples

Beacoup d'application possibles : médical (suivi taux de glucose, …), fitness (montres, ceinture, …), etc.

Ils sont en général reliés au travers de leur propre réseau **BAN** (Body Area Network) et en **BLE**.

Ils sont très peu interopérable (application propriétaire). Dans le domaine médical il existe la norme **Continua** pour essayer de promouvoir l'interopérabilité.

Pour le médical, depuis 2021 on a la législation suisse **ODim** pour préciser et renforcer les procédures nécessaire à la mise sur le marché d'une application de ce type.

- Exemple ODim : une application mobile accompagnant une aide auditive et permettant de régler son volume sonore est considérée comme une app médicale.

Les montres connéctées sont la catégorie la plus répendue de wearable (Android Wear, Apple Watch).

6.2.1 Wear OS (anciennement Android Wear)

Une montre Wear OS est appairée en BLE avec un natel.

En 1.0 si la montre devait passer par la connexion du natel pour accéder à internet (proxy BLE) mais depuis 2.0 elle peut y accéder directement en Wi-Fi ou LTE (mais privilégie le proxy BLE pour l'économie d'énergie, utilisation Wi-Fi/LTE si accès internet haut débit demandé ou en absence de natel).

On a plusieur approche de développement : **Notifications**, **Complications**, **Tuiles**, **Application**.

On pourra choisir la surface suivant la priorité des fonctionnalités/informations auxquelles on donne accès.

Complication
- Météo actuelle

Notification
- Alertes météo

Tuile
- Météo actuelle
- Prévisions du jour

Application
- Météo actuelle
- Prévisions et évolution du jour
- Prévisions de la semaine
- Paramètres

Le **développement est similaire que pour Android** : Activités, ViewModels, Librairies Jetpack, etc.

Attention toutefois à la réalisation de l'interface graphique qui est primordial (défilement vertical à privilégier).

Pour synchroniser les données entre une application montre et natel -> *Wearable Data Layer API* qui fait parties des **Google Play Services**.

Cet API permet :
- ***Message Client* d'envoyer des messages** = appel RPC avec payload limités.
- ***Channel Client* transférer des données** = streaming audio ou vidéo.
- D'annoncer des fonctionnalités prises en charges.
- ***Data Client* synchroniser les données** = espace de stockage privé, chaque noeud (natel, montre, cloud) peut lire et écrire les données et on notifie les autre noeuds des changements, la synchro est automatique.

	Data Client	Message Client	Channel Client
Data size greater than 100 kb	Yes	No	Yes
Requires a network connection, such as saving data to the cloud	Yes Evtl. via Bluetooth	Uses Bluetooth	Uses Bluetooth
Can send messages to nodes that aren't currently connected	Yes	No	No
Can send from one device to another device, such as from wear to mobile	No, from the cloud to all the nodes	Yes	Yes, for both one-way requests and bidirectional requests
Supports fanning out data to all the devices	Yes	No	No
Reliability	Yes	Yes, when Bluetooth connection is established	Yes

7 NDK

Le NDK permet d'utiliser du code C/C++ dans une application Android. Le code est compilé pour différente architecture sous forme de librairie partagée (.so). Le NDK ne remplace pas Kotlin, mais est utilisé dans des cas particuliers où les performances sont critiques.

7.1 Contenu

- CLANG pour compilation et LLD pour linkgage
- La STL (C++14 par défaut, C++17 ou partiellemnet C++20)
- APIs android(log, sensor, asset_manager, etc.)

7.2 Architectures

Le code du NDK est compilé pour 4 architectures de processeur. Cela permet de pouvoir faire tourner l'application sur un maximum de téléphones.

Name	arch	ABI	triple
32-bit ARMv7	arm	armeabi-v7a	arm-linux-androideabi
64-bit ARMv8	aarch64	aarch64-v8a	aarch64-linux-android
32-bit Intel	x86	x86	i686-linux-android
64-bit Intel	x86_64	x86_64	x86_64-linux-android

7.3 Integration

Deux approches:
- JNI (Java Native Interface) : appelle des méthode c++ depuis Kotlin
- native-activity: Application entièrement en C++ et utilise OpenGL ES pour l'UI.

Utilisation d'un makefile Android pour lister les fichiers source mais l'utilisation de CMake est recommendée.

7.4 Types

Le NDK propose des types équivalents aux types primitifs de Java/Kotlin.

Java	JNI	Description	Kotlin
boolean	jboolean	1 byte, unsigned	Boolean
int	jint	4 bytes, signed	Int
long	jlong	8 bytes, signed	Long
short	jshort	2 bytes, signed	Short
byte	jbyte	1 byte, signed	Byte
char	jchar	2 bytes, unsigned	Char
float	jfloat	4 bytes	Float
double	jdouble	8 bytes	Double
void	void	N/A	Unit

La gestion de types plus complexes comme les objets demande plus de précaution et sont généralement transformé en type c++

Java	JNI	Kotlin
Object	jobject	Any
String	jstring	String
Class	jclass	Class<>
int[]	jintArray	IntArray
double[]	jdoubleArray	DoubleArray

7.5 Exemple

7.5.1 CMake

Ajout dans le fichier gradle:

```
android {
    ...
    externalNativeBuild {
        cmake {
            path "src/main/cpp/CMakeLists.txt"
            version "3.31.16"
        }
    }
}
```

Contenu CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.22.1)

# Options de compilation
set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_FLAGS_ RELEASE "${CMAKE_CXX_FLAGS_ RELEASE} -O3")

add_library(
  native-lib  # Nom de notre bibliothèque

  SHARED

  native-lib.cpp) # Fichier(s) source

# Pour trouver une autre bibliothèque, on utilise un autre find_library
find_library(
  log-lib # Nom variable pour lier la bibliothèque
  log) # Nom de la bibliothèque à lier

target_link_libraries(
  native-lib # Link les différents composants

  ${log-lib})
```

7.5.2 C++

```
#include <jni.h>
extern "C"
JNIEXPORT jint JNICALL
Java_ch_heigvd_iict_dma_mynativeapplication_MainActivity_nativeSum(
  JNIEnv *env,
  jobject thiz,
  jint v1,
  jint v2) {
    return v1 + v2;
}
```

7.5.3 Kotlin

```
class MainActivity : AppCompatActivity() {
  companion object {
    init {
      System.loadLibrary("native-lib")
    }
  }

  private external fun nativeSum(v1: Int, v2: Int) : Int

  override fun onCreate(savedInstanceState: Bundle?) {
    val un = 1
    val deux = 2
    val resultat = nativeSum(un, deux)
  }
}
```

7.6 Remarques

- Il est plutôt conseiller de ne pas utiliser appeler les méthodes c++ depuis l'UI thread
- Niveau d'optimisation, -0z est conseillé (taille + vitesse)
- Il y a une copie de notre libaririe partagée pour chaque architecture, donc la taille de l'application augmente.

8 Instant App et App Clips

Les Instant Apps et App Clips sont des applications Android qui peuvent être utilisées sans installation complète. Elles permettent aux utilisateurs d'accéder rapidement à des fonctionnalités spécifiques d'une application sans avoir à la télécharger entièrement.
- Instant App -> Android, depuis octobre 2017
- App Clip -> iOS, depuis septembre 2020 (iOS 14)

Offre des fonctions avancées car natives mais plus restraintes qu'une app normale; limitées à 15Mo; Peux proposer de télécharger l'application complète et transférer les données de l'Instant App vers l'application complète.

8.1 Utilisation

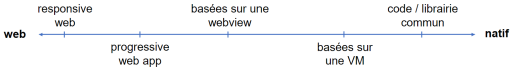
- Permettre de tester une application avant de l'installer
- Apperçu de jeux mobile

9 Cross-platefrormes

9.1 Pourquoi?

Outils différents entre Android et iOS, philosophie différente, besoin de deux équipes de développement, etc.

Pour répondre à ce besoins, différente approches ont émergées, allant de la simple webview à des solutions plus complexes. Les prmières solutions était proches des technologies web afin de permettre aux dev. web de facilement les utiliser, mais les dernières solutions sont plus proches des applications natives afin d'avoir accès à plus de fonctionnalités spécifiques des mobiles.



9.2 App web

Les applications web sont des sites web qui se comportent comme des applications. Elles sont accessibles via un navigateur et peuvent être installées sur l'écran d'accueil du téléphone. Elles utilisent des technologies web standard (HTML, CSS, JavaScript) et sont exécutées dans un navigateur. Malheuresement, pas tous les navigateurs supportent toutes les librairies JavaScript.

Librairie	Chrome *Android*	Firefox *Android*	WebView Android	Safari *iOS*	WebView *iOS*
Storage	✓	✓	✓	✓	✓
Bluetooth	✓	✗	✗	✗	✗
Geolocation	✓	✓	✓	✓	✓
Sensor	✓	✗	✓	✗	✗
NDEFReader	✓	✗	✓	✗	✗

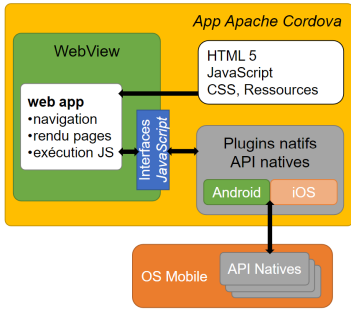
9.3 Progressive Web App (PWA)

Application web pouvant être "installée" sur le téléphone. Une web app est considérée comme une PWA si elle peut fonctionner en l'absence de réseau et qu'un raccourci peut être ajouté à l'écran d'accueil. Pour cela, elles doivent avoir un fichier manifest.json décrivant l'application et un service worker pour gérer le cache et les notifications push.

Sur iOS, les PWA sont limitées par le navigateur Safari et ne peuvent pas accéder à toutes les fonctionnalités du téléphone.

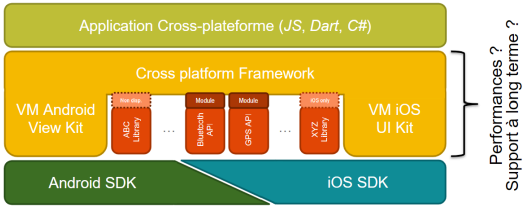
9.4 Apache Cordova

Apache Cordova est un framework qui permet de créer des applications mobiles en utilisant des technologies web (HTML, CSS, JavaScript). La web app est empaquetée dans une application native qui utilise une webview pour afficher le contenu. Cordova fournit des plugins pour accéder aux fonctionnalités natives du téléphone (caméra, GPS, etc.).



9.5 Solutions basée sur une VM

Application contenant une vm executant le code de l'application, avec un moteur de rendu graphique ainsi que des modules permettant d'accéder aux fonctionnalités natives du téléphone. Permt d'avoir un seul code pour android et iOS mais également d'autre plateforme (Windows, Mac, etc.).



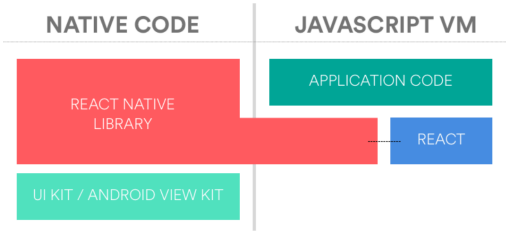
Il existe principalement trois solutions:

- **Flutter**, Google, Dart, Liscence BSD
- **React Native**, Meta (Facebook), JavaScript, License MIT
- .NET MAUI (successeur de Xamarin), Microsoft, C#, License MIT

Les moteurs de jeux comme unity permettent également des applications cross-plateforme.

9.5.1 React Native

React Native permet de créer une UI qui sera ensuite mappée vers les widgets natifs de la plateforme. Le code JavaScript sera interprété par une VM JavaScript.



La VM est stockée sous forme de librairies partagées (fichiers .so), ce qui a pour conséquence d'avoir une VM pour chaque architecture. Il existe des packages pour accéder aux fonctionnalités natives du téléphone de manière unifiée.

Comme le code JS est interprété, il est possible de mettre à jour l'application sans passer par le store, mais il est interdit d'ajouter des fonctionnalités ou de changer significativement l'app.

9.5.2 Flutter

Initialement développé pour les mobiles, aujourd'hui il permet également le développment pour d'autres plateformes comme Linux, MacOS, Windows, Fuchsia ou le web. Principaux composants:

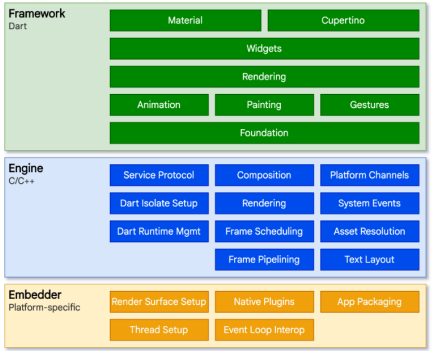
- Dart: langage à la Java, compilation just-in-time (JIT) pour le développement et sur Desktop, et compilation ahead-of-time (AOT) pour la production.
- Foundation Library: Ensemble d'API pour construire app
- Skia: Moteur graphique
- Widgets: 2 collections (Material et Cupertino) pour ressembler aux applications Android et iOS.
- Development Tools: Outils pour le développement, le débogage et la compilation des applications.

Il existe 4 manières d'exécuter du code Dart:

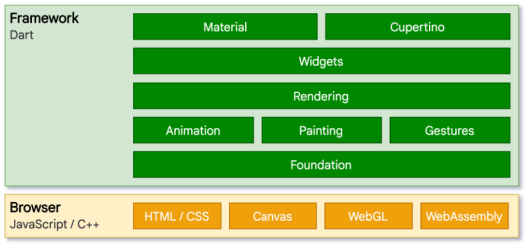
- Natif (code compilé pour une plateforme spécifique)
- Autonome (utilisation de la VM Dart)
- Compilation AOT (approche utilisée par Flutter pour Android et iOS)
- Transpilation en JS (utilisée pour le web)

Mise à disposition par la commmunauté de package pour accéder aux fonctionnalités natives du téléphone.

Cross-plateformes – *Flutter* – *Mobile/Desktop*



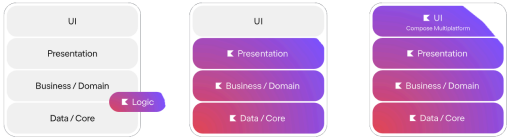
Cross-plateformes – *Flutter* – *web*



9.6 Kotlin Multiplatform Mobile (KMM)

Nouvelle approche en beta proposée par JetBrains. Permet de partager du code entre Android et iOS en utilisant Kotlin. Le code partagé est compilé en code natif pour chaque plateforme. KMM permet de partager la logique métier, les modèles de données et les tests, tout en permettant d'utiliser les API natives de chaque plateforme.

Possibilité de développer qu'une partie des composants en kmm et le reste en taif pour Android et iOS selon les besoins. Est actuellement en train d'être complété par Compose Multiplateforme qui permettra de mettre en commun également l'UI.



10 Codes Barres

10.1 Le problème qu'il y avait

Résolution insuffisante : CameraX analyse par défaut en **640×480 pixels**, insuffisant pour décoder les QR Codes complexes (versions 25, 40).

10.2 Solution technique

10.2.1 Augmentation de la résolution d'analyse

MLKit nécessite **minimum 2 pixels par "pixel" du QR Code** (idéal), en réalité **4-8 pixels** requis.

Compromis trouvé : Résolution de **5 mégapixels (2592×1920)** au lieu de 640×480.

```
private val barcodesUseCase by lazy {
    ImageAnalysis.Builder()
        // Par défaut: 640x480 (insuffisant)
        .setTargetResolution(Size(2592, 1920)) // 5MP
        .setBackpressureStrategy(
          ImageAnalysis.STRATEGY_KEEP_ONLY_LATEST)
        .build().apply {...}}}
```

10.2.2 Optimisation du scanner

Limitation aux QR Codes uniquement pour améliorer performances car defacto, on scanne tout, code-barres, etc.:

```
private val barcodeClient by lazy {
    BarcodeScanning.getClient(BarcodeScannerOptions
        .Builder()
        // Limite aux QR Codes
        .setBarcodeFormats(Barcode.FORMAT_QR_CODE)
        .build())}
```

10.3 Limitations pratiques

- **QR Code v40 (177·177) :** Techniquement lisible mais **pas adapté grand public** (taille physique, temps de traitement)
- **Recommandation Google :** 2MP pour usage standard, 5MP pour cas extrêmes
- **Latence vs qualité :** 20MP natif = trop lent, 5MP = bon compromis

11 Lecture de tags NFC

11.1 Exercice 1 : Lecture avec activité active

11.1.1 Décodage NDEF - Format RTD_URI

Byte 0 : Préfixe URI

- `0x01` = `http://www.`
- `0x02` = `https://www.`
- `0x03` = `http://`
- `0x04` = `https://`

Bytes suivants : Reste URI en UTF-8

Donc ce qu'il fallait faire :

Pattern match le premier byte pour déterminer le préfixe, puis concatène le reste de l'URI en décodant en UTF-8.

```
private fun decodeUriRecord(record: NdefRecord): String {
    val payload = record.payload
    val prefix = when (payload[0]) {
        0x01.toByte() -> "http://www."
        0x02.toByte() -> "https://www."
        0x03.toByte() -> "http://"
        0x04.toByte() -> "https://"
        else -> ""
    }
    return prefix + String(payload, 1, payload.size - 1,
Charsets.UTF_8)
}
```

11.1.2 Décodage NDEF - Format RTD_TEXT

Byte 0 : Flags

- Bit 7 : Encodage (0=UTF-8, 1=UTF-16)
- Bits 5-0 : Longueur code langue

Code langue : ASCII (ex: "fr", "fr-CH") **Contenu** : Texte en UTF-8/UTF-16 selon flag

11.2 Exercice 2 : Lecture app inactive

Modifier pour ouverture automatique activité lors scan NFC. **Éviter instances multiples**.

Ce qu'il fallait faire :

Inidiquer dans le manifest l'activité à ouvrir lors du scan NFC, et gérer l'intent dans l'activité pour afficher le contenu.

```
 <activity
            <...>
            <intent-filter>
                <action
android:name="android.nfc.action.NDEF_DISCOVERED" />
                <category
android:name="android.intent.category.DEFAULT" />
                <data android:mimeType="text/plain" />
            </intent-filter>
        </activity>
```

Et dans MainActivity.kt :

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        // ... setup UI

        // Traiter le tag si app lancée par NFC
        handleNfcIntent(intent)
    }

    override fun onNewIntent(intent: Intent) {
        super.onNewIntent(intent)
        // Nouveau tag scanné (app déjà active)
        handleNfcIntent(intent)
    }

    private fun handleNfcIntent(intent: Intent) {
        if (intent.action ==
NfcAdapter.ACTION_NDEF_DISCOVERED) {
            val rawMessages = intent.
```

```
getParcelableArrayExtra(NfcAdapter.EXTRA_NDEF_MESSAGES)
            rawMessages?.let { messages ->
                val ndefMessages = messages.map { it as
NdefMessage }
                // Traiter les messages NDEF...
                processNdefMessages(ndefMessages)
            }}}}
```

12 Boussole 3D

12.1 Objectif

Utiliser les capteurs du téléphone doonc accéléromètre et magnétomètre pour faire une boussole en 3D

12.2 Manip 1

On va utiliser les résultats des deux capteurs pour générer la matrice de rotations que l'on pourra donner au renderer 3D OpenGL afin de faire bouger notre flèche 3D.

Donc on va utiliser enregister les 2 capteur sur notre activité puis on va récupérer les données de ceux-ci appeler `getRotationMatrix()` pour avoir notre matrice de rotation et la donner en appelant `openGLRenderer.swapRotMatrix()`.

```
override fun onCreate(savedInstanceState: Bundle?) {
  // [...]
  sensorManager = getSystemService(SENSOR_SERVICE) as
SensorManager
  accelerometer =
sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER)
  magnetometer  =
sensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD)
}

override fun onResume() {
  super.onResume()
  SensorManager.registerListener(this, accelerometer,
sensorManager.SENSOR_DELAY_GAME)
  SensorManager.registerListener(this, magnetometer,
sensorManager.SENSOR_DELAY_GAME) // pareil que plus haut
}

// Ne pas oublier
override fun onPause() {
  super.onPause()
  sensorManager.unregisterListener(this)
}
```

12.3 Manip 2

On applique un low-pass filter sur les données de l'accéléromètre et du magnétomètre pour les rendre moins bruyantes car sans cela la flèche est tremblante même si on ne bouge pas.

13 Exercice 4 : Wearables

13.1 Ce qu'il fallait faire (Ex 1)

Dans la partie notificationActivity, ce qu'il fallait faire était de dupliquer les notifications générales et de les envoyer à la montre à l'aide du `WearableExtender` :

```
notif.extend(NotificationCompat.WearableExtender()
    .addAction(action1)
    .addAction(action2)
    .addAction(action3))
```

Sinon le reste du code était classique.

13.2 Ce qu'il fallait faire (Ex 2)

13.2.1 Objectif

Créer deux applications synchronisées (smartphone + montre) qui partagent des données en temps réel via la **Wearable Data Layer API**. L'exemple concret : synchroniser une couleur RGB entre les deux appareils via des sliders.

13.2.2 Architecture MVVM partagée

- **Même ViewModel** dans le module `common` (code réutilisé)
- **Instances séparées** du ViewModel (pas de partage d'instance)
- **Communication via Data Layer API** uniquement

13.2.3 Problèmes du code original

Le code initial était incomplet :

- ❌ Pas de gestion des interactions utilisateur (sliders)
- ❌ Pas d'observation des changements de couleur
- ❌ Pas d'envoi de données vers la Data Layer API

13.2.4 Solutions implémentées

13.2.4.1 A. Gestion des interactions utilisateur

```
class DataSyncActivity : AppCompatActivity(),
                         OnSeekBarChangeListener {

    override fun onCreate(savedInstanceState: Bundle?) {
        // Configuration des listeners pour les sliders RGB
        binding.red.setOnSeekBarChangeListener(this)
        binding.green.setOnSeekBarChangeListener(this)
        binding.blue.setOnSeekBarChangeListener(this)
    }
}
```

13.2.4.2 B. Envoi des données (smartphone → montre)

```
override fun onStopTrackingTouch(seekBar: SeekBar) {
    val r = binding.red.progress
    val g = binding.green.progress
    val b = binding.blue.progress

    // Envoi vers Data Layer API
    dataClient.putDataItem(Tools.createColorUpdate(r,g,b))
}
```

Mécanisme :

1. Utilisateur relâche un slider (`onStopTrackingTouch`)
2. Récupération des valeurs RGB actuelles
3. Création d'un `DataItem` via `Tools.createColorUpdate()`
4. Envoi via `dataClient.putDataItem()`

13.2.4.3 C. Réception et mise à jour de l'interface

```
override fun onCreate(savedInstanceState: Bundle?) {
    // Observation des changements de couleur
    colorViewModel.color.observe(this) { newColor ->
        binding.root.setBackgroundColor(newColor)
        binding.red.progress = Color.red(newColor)
        binding.green.progress = Color.green(newColor)
        binding.blue.progress = Color.blue(newColor)
    }
}
```

13.2.4.4 D. Gestion du cycle de vie

```
override fun onResume() {
    super.onResume()
    // Récupération des données existantes
dataClient.dataItems.addOnSuccessListener(colorViewModel)
    // Écoute des changements en temps réel
    dataClient.addListener(colorViewModel)
}

override fun onPanelClosed(featureId: Int, menu: Menu) {
    super.onPanelClosed(featureId, menu)
    // Arrêt de l'écoute pour éviter les fuites mémoire
    dataClient.removeListener(colorViewModel)
}
```

13.2.5 Flux de synchronisation complet

1. **Smartphone** : User bouge slider rouge
2. **Smartphone** : `onStopTrackingTouch` → `putDataItem(R,G,B)`
3. **Data Layer API** : Synchronisation automatique
4. **Montre** : `ColorViewModel.onDataChanged()` appelé
5. **Montre** : `_color.value` mis à jour
6. **Montre** : Interface observe `color` → background + sliders mis à jour

14 Exercice 8 : NDK (Native Development Kit) Android

14.1 Objectif

Comparer les performances entre **Kotlin** et **C++** pour le tri d'un million d'entiers via le NDK Android.

14.2 Architecture

- **MainActivity** : Interface avec boutons "Tri Kotlin" et "Tri C++"
- **IntListViewModel** : Logique métier avec `updateList()` (Kotlin) et `updateListNDK()` (C++)
- **native-lib.cpp** : Implémentation native du tri

14.3 Implémentation clé

14.3.1 Côté Kotlin

```
// Tri Kotlin standard
fun updateList(nbr: Int = defaultNbrOfInt) {
    val time = measureTimeMillis {
        val sortedArray = rawArray.sorted()
    }
}

// Tri natif C++
fun updateListNDK(nbr: Int = defaultNbrOfInt) {
    val time = measureTimeMillis {
        val sortedArray = nativeSort(rawArray)
    }
}

private external fun nativeSort(numbers: IntArray):
IntArray
```

14.3.2 Côté C++

```
extern "C"
JNIEXPORT jintArray
Java_..._nativeSort(JNIEnv *env, jobject thiz, jintArray
numbers) {
    // 1. Conversion JNI → std::vector
    std::vector<int> v = convert(env, numbers);

    // 2. Tri natif
    std::sort(v.begin(), v.end());

    // 3. Conversion std::vector → JNI
    return convert(env, v);
}
```

14.4 Mesures de performance

L'implémentation mesure **3 opérations** :

1. **Conversion JNI → C++** : Marshalling des données
2. **Tri natif** : `std::sort()` pur
3. **Conversion C++ → JNI** : Marshalling de retour

14.5 Points clés

14.5.1 Optimisations CMake

```
set(CMAKE_CXX_FLAGS_RELEASE "${CMAKE_CXX_FLAGS_RELEASE} -
O3")
set(CMAKE_CXX_FLAGS_DEBUG "${CMAKE_CXX_FLAGS_DEBUG} -O3")
```

14.5.2 Structure APK

```
lib/
├── arm64-v8a/libnative-lib.so    # ARM 64-bit
├── armeabi-v7a/libnative-lib.so  # ARM 32-bit
└── x86_64/libnative-lib.so       # Intel/Émulateurs
```

14.6 Avantages vs Inconvénients

14.6.1 ✅ Avantages

- Performance pour calculs intensifs
- Réutilisation de code C++ existant

14.6.2 ❌ Inconvénients

- Complexité JNI
- Overhead des conversions
- Debugging plus difficile

14.7 Résultats attendus

1. Tri C++ plus rapide que Kotlin
2. Conversions JNI ajoutent un overhead
3. Gain global dépend du ratio calcul/conversion
4. Optimisation compiler cruciale pour les performances