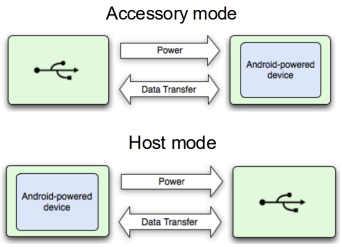
1 Protocoles de proximité

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.

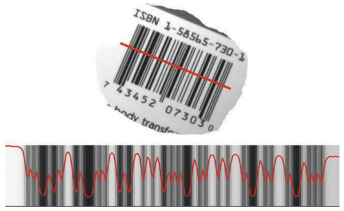
2.2 Différents types de codes-barres unidimensionnels (1D)

Codes 1D avec leurs jeux de caractères :

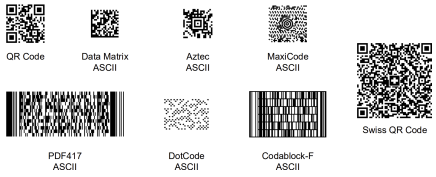
- **Code-11 :** `[0-9] + "-"` (ex: 123456)
- **Code-39 :** `[A-Z] [0-9] [ - . $ / + % ]` (ex: AB12)
- **Code-93 :** `[A-Z] [0-9] [_ - . $ / + %]` (ex: AB-12/+)
- **Code-128 :** ISO-8859-1 (ex: Eeéè)
- **EAN-13 :** `[0-9]` – longueur 13 (ex: 7601234568903)
- **UPC-A :** `[0-9]` – longueur 12 (ex: 760123456787)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

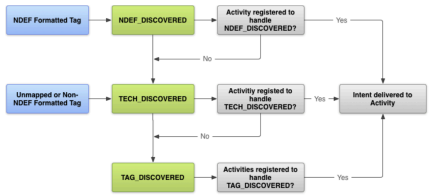
```
<uses-permission
android:name="android.permission.NFC" />
```

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)**.

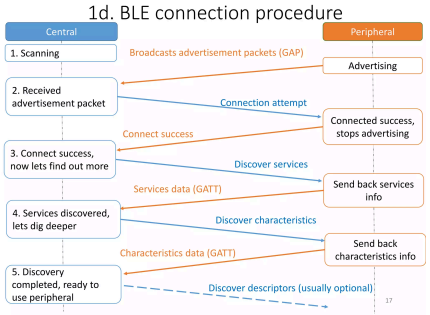
4.5 Le Bluetooth Low Energy (BLE) (Topologie Client/Serveur)

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

TODO
:3

7 je sais pas

TODO
:3

8 Codes Barres

8.1 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).

8.2 Solution technique

8.2.1 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 {...}}}
```

8.2.2 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())}
```

8.3 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

9 Lecture de tags NFC

9.1 Exercice 1 : Lecture avec activité active

9.1.1 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)
}
```

9.1.2 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

9.2 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)
            }}}}
```