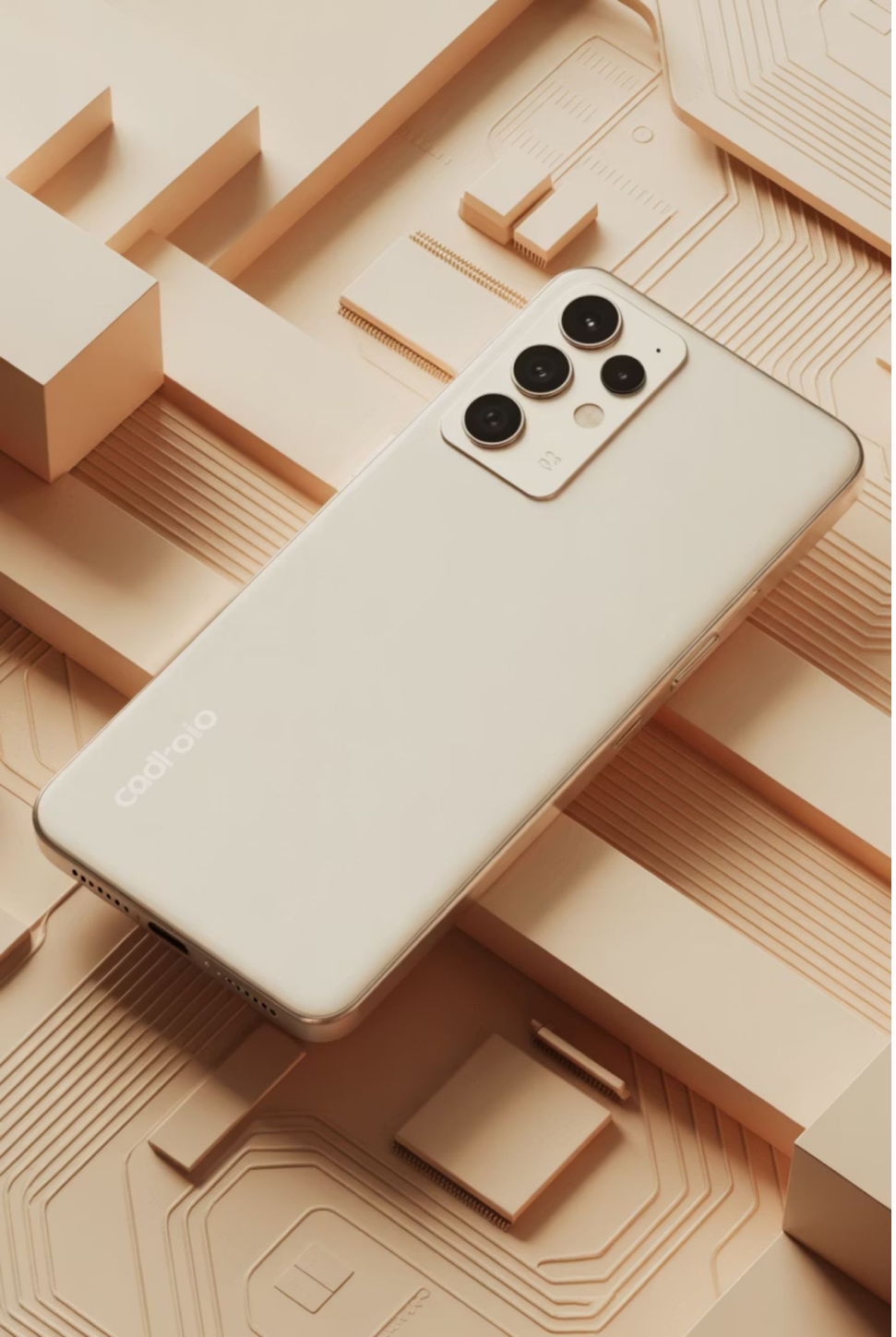


Chapitre 7 : API Android

Maîtriser l'interaction avec les fonctionnalités natives du téléphone pour créer des applications puissantes et sécurisées

Plan du Chapitre

01

Informations de base

Accès aux contacts et numéros de téléphone via Content Providers

02

Téléphonie

Gestion des appels, SMS et MMS avec les APIs système

03

Géolocalisation

Intégration GPS et services de cartographie

04

Caméra

Capture de photos et vidéos avec l'appareil

05

Multimédia

Sonneries, vibrations et feedback utilisateur

06

Interactions tactiles

Gestion avancée des gestes et événements

07

Réseau

Wi-Fi, Bluetooth, NFC et connectivité

Objectif du Chapitre

Comprendre les API natives Android

Ce chapitre vous permettra de maîtriser comment une application Android peut **interagir avec les fonctionnalités natives du téléphone** comme la téléphonie, les contacts, le GPS, la caméra et le réseau.

Vous apprendrez à exploiter ces APIs tout en respectant le **modèle de permissions et de sécurité Android**, essentiel pour protéger la vie privée des utilisateurs et garantir la conformité de vos applications.

Chaque API sera étudiée avec ses cas d'usage réels, ses bonnes pratiques et ses limitations techniques.

Fonctionnalités

Accès aux capacités matérielles du device

Sécurité

Gestion des permissions runtime

Pratique

Cas d'usage concrets et patterns

Content Providers et Accès aux Contacts

Architecture des données partagées Android

Les **Content Providers** constituent le mécanisme fondamental d'Android pour partager des données entre applications. Ils agissent comme une interface standardisée permettant à votre application d'accéder aux informations système de manière sécurisée et contrôlée.

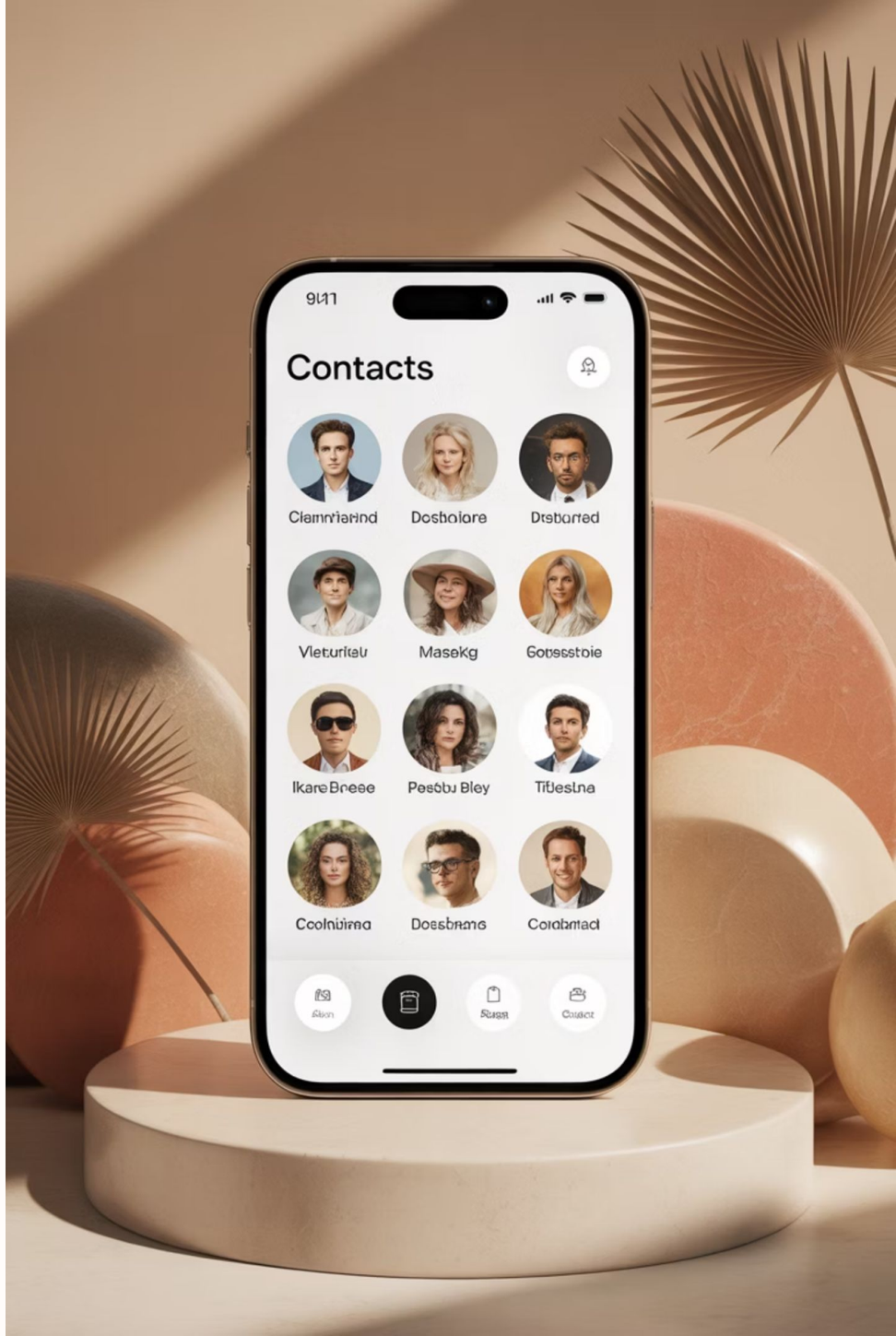
Données privées

Informations stockées exclusivement dans votre application, inaccessibles aux autres apps

Données partagées

Contacts, médias, calendrier - accessibles via Content Providers avec permissions appropriées

Le **ContactsContract** est le Content Provider dédié à la gestion des contacts. Il organise les données en plusieurs tables liées : contacts bruts, contacts agrégés, numéros de téléphone, emails et adresses.



Lecture des Contacts

Permissions nécessaires

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

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

Utilisation du Cursor

Le **Cursor** est l'objet qui permet de parcourir les résultats d'une requête sur un Content Provider. Il fonctionne comme un pointeur sur les lignes de données retournées.
Vous devez toujours fermer le Cursor après utilisation pour éviter les fuites mémoire.

Données accessibles

- **Nom complet** du contact
- **Numéro(s) de téléphone** avec type (mobile, fixe, travail)
- **Adresse(s) email** avec catégorie
- Photo de profil
- Informations supplémentaires selon la source

Important : Depuis Android 10+, l'accès au numéro de téléphone du device lui-même est fortement limité pour protéger la vie privée. Concentrez-vous sur les contacts du répertoire.

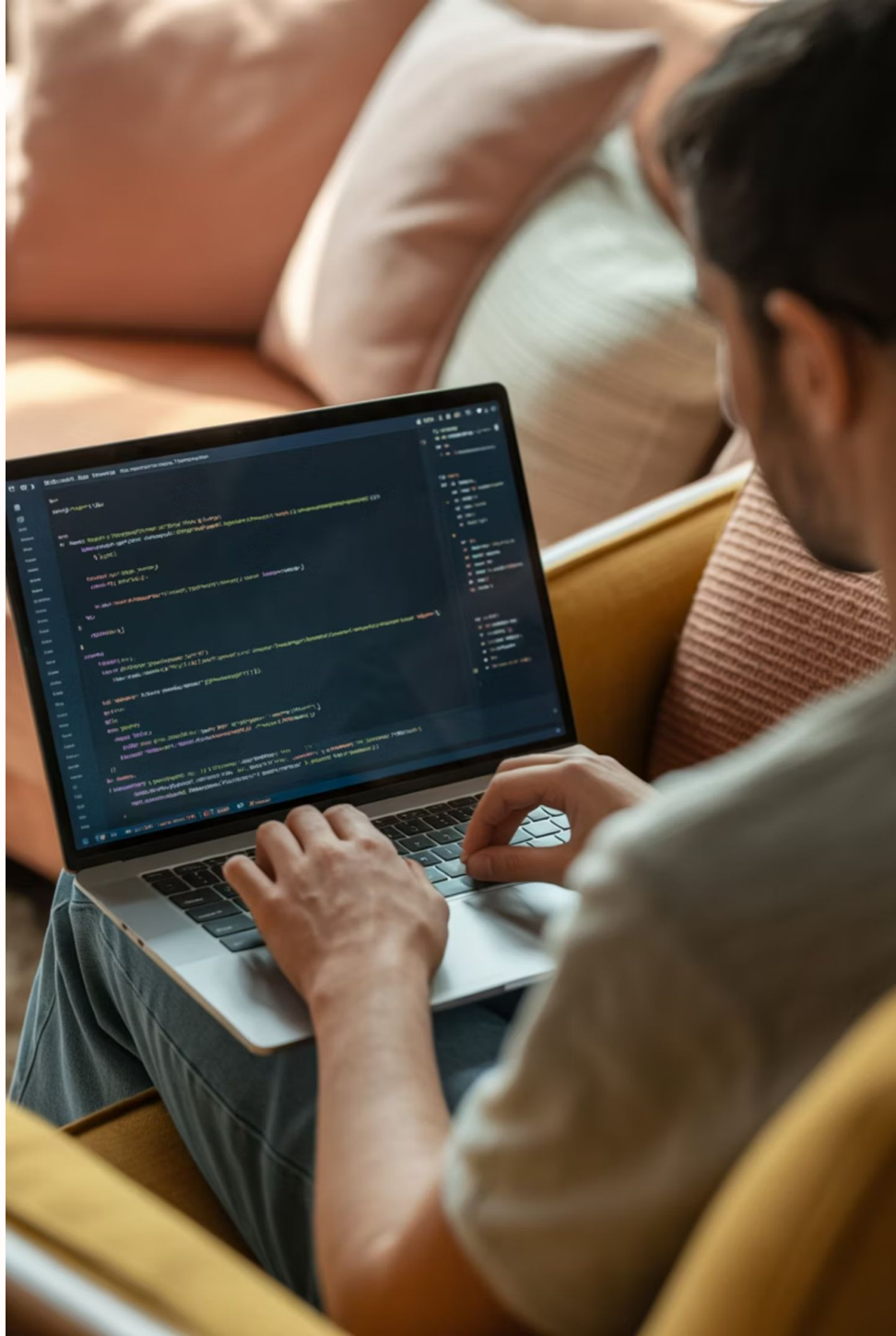
Exemple pratique : lire les contacts

```
Cursor cursor = getContentResolver().query(
    ContactsContract.Contacts.CONTENT_URI,
    null, null, null, null
);

while (cursor.moveToNext()) {
    String name = cursor.getString(
        cursor.getColumnIndex(
            ContactsContract.Contacts.DISPLAY_NAME
        )
    );
    Log.d("CONTACT", name);
}
```

`cursor.close(); // Important : libérer les ressources`

Ce code interroge le **ContentResolver** pour obtenir un **Cursor** contenant tous les contacts. La méthode `moveToNext()` parcourt chaque ligne, et `getString()` extrait le nom d'affichage. N'oubliez jamais de fermer le Cursor pour éviter les fuites mémoire.



Cas d'Usage et Bonnes Pratiques

Application de messagerie

Sélection de contacts pour envoyer des messages, affichage des conversations avec noms et photos

CRM mobile

Synchronisation des contacts clients, gestion des interactions et historique des communications

Synchronisation

Sauvegarde et restauration des contacts, synchronisation multi-devices avec cloud

Bonnes pratiques essentielles

Demande de permission runtime

Depuis Android 6.0, vous devez demander les permissions dangereuses au moment de l'exécution, avec une explication claire de l'utilité

Gestion du refus utilisateur

Votre application doit continuer à fonctionner même si l'utilisateur refuse l'accès aux contacts. Prévoyez des alternatives ou dégradez gracieusement les fonctionnalités

Fonctions de Téléphonie

Appels téléphoniques

Android offre deux approches pour initier des appels : l'**appel direct** qui compose automatiquement le numéro, et l'**appel via l'application Téléphone** qui ouvre le dialer avec le numéro pré-rempli, laissant l'utilisateur confirmer l'appel.

ACTION_DIAL

Ouvre le dialer sans permission - l'utilisateur contrôle l'appel

ACTION_CALL

Lance l'appel directement - nécessite CALL_PHONE permission

Permissions téléphonie

CALL_PHONE

Permet d'initier des appels directement sans interaction utilisateur

SEND_SMS

Autorise l'envoi de SMS de manière programmatique

RECEIVE_SMS

Permet de recevoir et lire les SMS entrants

Gestion des SMS et MMS

Via Intent

Ouvre l'app de messagerie par défaut avec le texte pré-rempli. Approche recommandée car elle respecte le choix de l'utilisateur

Via SmsManager

Envoi direct depuis votre code. Nécessite SEND_SMS permission et gestion des erreurs d'envoi

Réception de SMS

Pour recevoir des SMS, vous devez implémenter un **BroadcastReceiver** qui écoute l'action SMS_RECEIVED. Cependant, depuis Android 4.4, seule l'application de messagerie par défaut peut modifier les SMS. Les autres apps peuvent uniquement les lire.

MMS (Multimedia Messaging Service) : Plus complexe que les SMS car ils contiennent images, vidéos ou audio. La gestion des MMS passe généralement par l'application de messagerie par défaut du système.

Cas d'usage pratiques

- **Bouton "Appeler"** dans une app e-commerce pour contacter le service client
- **Confirmation par SMS** pour vérification d'identité ou codes OTP
- **Notifications critiques** envoyées par SMS en complément des push notifications

Exemples pratiques

Initier un appel

Utilisez `ACTION_DIAL` pour ouvrir le composeur avec un numéro pré-rempli, ou `ACTION_CALL` pour appeler directement (nécessite CALL_PHONE).

Envoyer un SMS

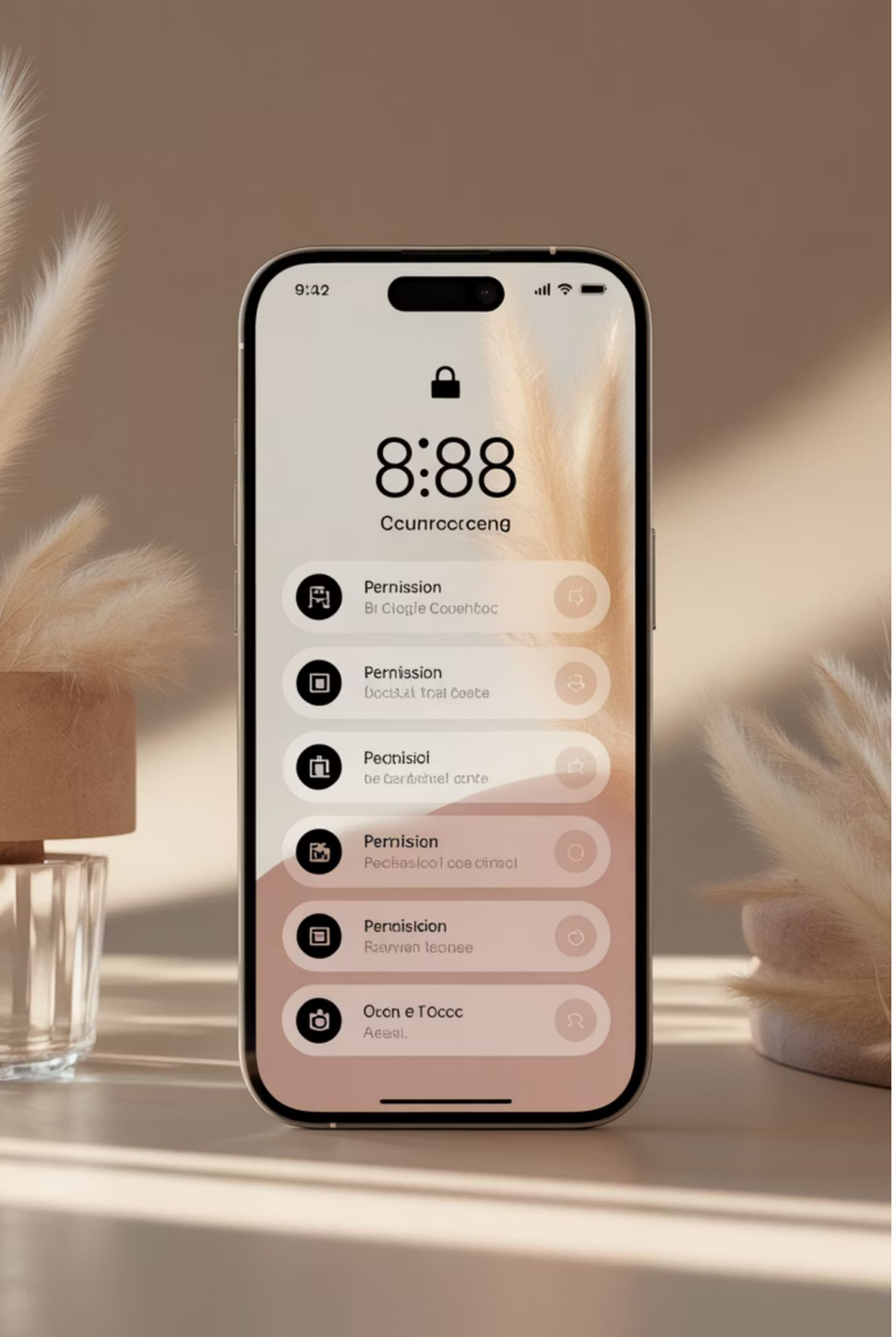
La classe `SmsManager` permet d'envoyer des SMS programmatiquement via la méthode `sendTextMessage()`.

Exemple : Appel téléphonique

```
Intent intent = new Intent(
    Intent.ACTION_DIAL
);
intent.setData(
    Uri.parse("tel:0612345678")
);
startActivity(intent);
```

Exemple : Envoi de SMS

```
SmsManager sms =
 SmsManager.getDefault();
sms.sendTextMessage(
 "0612345678",
 null,
 "Bonjour depuis Android",
 null, null
);
```



Permissions et bonnes pratiques

CALL_PHONE

Nécessaire pour initier un appel directement sans passer par le composeur. Permission dangereuse qui nécessite l'accord explicite de l'utilisateur.

SEND_SMS

Requise pour envoyer des SMS programmatiquement. Peut engendrer des coûts pour l'utilisateur, d'où l'importance de la transparence.

Privilégier les apps système

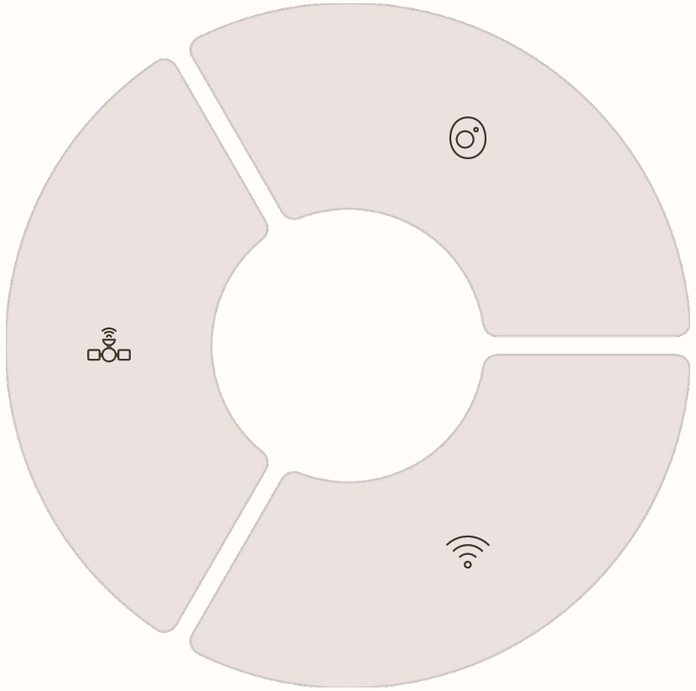
Android recommande d'utiliser les Intents pour déléguer aux applications système (Téléphone, Messages). Cela offre une meilleure expérience utilisateur et respecte ses préférences.

Géolocalisation avec Android

Sources de localisation

GPS

Précision élevée (5-10m) mais consommation batterie importante



Réseau mobile

Précision moyenne (100-500m), consommation modérée

Wi-Fi

Bonne précision en zone urbaine (20-50m), faible consommation

APIs de localisation

LocationManager

API historique d'Android, accès direct aux fournisseurs de localisation. Plus complexe à utiliser mais offre un contrôle fin.

FusedLocationProvider

API moderne recommandée par Google. Fusionne automatiquement les sources pour optimiser précision et batterie.

Permissions et Cartographie

Permissions de localisation

1

ACCESS_COARSE_LOCATION

Localisation approximative via réseau et Wi-Fi. Suffisante pour beaucoup d'apps

2

ACCESS_FINE_LOCATION

Localisation précise via GPS. Nécessaire pour navigation et tracking précis

3

Background Location

Depuis Android 10, permission séparée pour localisation en arrière-plan. Justification obligatoire

Intégration de cartes

Les APIs de cartographie comme **Google Maps** permettent d'afficher des cartes interactives dans vos applications. Le composant **MapView** s'intègre facilement dans vos layouts.

Marqueurs

Positionnez des points d'intérêt avec icônes personnalisées et info-bulles

Position actuelle

Affichez la position de l'utilisateur avec cercle de précision

Navigation

Zoom, déplacement, rotation et inclinaison de la carte

Exemple : récupérer la position

```
LocationManager lm = (LocationManager)
    getSystemService(LOCATION_SERVICE);

Location location = lm.getLastKnownLocation(
    LocationManager.GPS_PROVIDER
);

if (location != null) {
    double latitude = location.getLatitude();
    double longitude = location.getLongitude();

    Log.d("POSITION", "Lat: " + latitude +
          ", Lng: " + longitude);
}
```

⚡ Optimisation batterie

Utilisez `getLastKnownLocation()` pour obtenir rapidement une position sans activer le GPS. Pour un suivi continu, privilégiez les mises à jour espacées.

🔒 Permissions dynamiques

Demandez la permission au moment où l'utilisateur déclenche une action nécessitant la localisation. Expliquez toujours le contexte d'utilisation.

Caméra et Appareil Photo

Deux approches d'accès caméra

Via Intent

Délègue à l'application Caméra système. Simple et recommandé pour captures basiques. Retourne Bitmap ou URI du fichier créé.

Via Camera2 API / CameraX

Contrôle avancé de la caméra (exposition, focus, format). Nécessaire pour apps photo professionnelles ou AR.

Permissions et stockage

Permissions requises

- **CAMERA** : accès à la caméra physique
- **WRITE_EXTERNAL_STORAGE** : enregistrement des photos (selon version Android)

Depuis Android 10, l'accès au stockage externe utilise **Scoped Storage** qui limite l'accès aux fichiers.

Stockage des images

Stockage interne : fichiers privés à l'app, supprimés à la désinstallation

Stockage externe : dossiers publics (DCIM, Pictures), accessibles par d'autres apps et galerie

Cas d'usage courants

- **Scan de documents**
  OCR pour extraire texte de factures, cartes d'identité
- **Photo de profil**
  Capture et upload d'avatar utilisateur
- **QR Code / Code-barres**
  Scan pour paiements, authentification, inventaire

Exemples pratiques

Capturer une image via Intent

Android permet d'utiliser l'application Caméra système via un **Intent implicite**. Cette approche est recommandée car elle respecte les préférences utilisateur et ne nécessite pas d'implémenter toute la logique de capture.

Déclenchement de la caméra

```
Intent intent = new Intent(
    MediaStore.ACTION_IMAGE_CAPTURE
);
startActivityForResult(intent, 100);
```

Le code `100` est un identifiant de requête qui vous permettra de récupérer le résultat.

Récupération de la photo

```
@Override
protected void onActivityResult(
    int requestCode,
    int resultCode,
    Intent data
) {
    if (requestCode == 100) {
        Bitmap photo = (Bitmap)
            data.getExtras().get("data");
        imageView.setImageBitmap(photo);
    }
}
```

Permission CAMERA : Obligatoire dans le manifeste. Note : l'image retournée est une miniature. Pour une qualité complète, utilisez un fichier de destination avec `EXTRA_OUTPUT`.

Cas d'usage de la caméra

Photo de profil

Permettre aux utilisateurs de personnaliser leur compte avec une photo capturée ou sélectionnée depuis la galerie.

Scan de documents

Numériser des cartes d'identité, factures ou codes QR pour automatiser la saisie d'informations.

Réalité augmentée

Superposer des éléments virtuels sur le flux caméra en temps réel pour le shopping ou les jeux.

Vibrations, Sons et Feedback Tactile

API Vibrator

L'**API Vibrator** permet de déclencher des vibrations pour fournir un retour haptique à l'utilisateur. La permission **VIBRATE** doit être déclarée dans le manifest.

1

Vibration courte

Confirmation d'action (50-100ms)

2

Vibration longue

Alerte ou notification (300-500ms)

3

Motif personnalisé

Séquence on/off pour feedback unique

Gestion des sons système

Android distingue plusieurs **types de sons** avec des volumes contrôlables indépendamment :

- **Sonnerie** : appels entrants
- **Notification** : alertes et messages
- **Alarme** : réveils et rappels
- **Média** : musique et vidéos

Utilisez **MediaPlayer** pour des fichiers audio longs (musique, podcasts) et **SoundPool** pour des sons courts répétitifs (effets sonores, feedback).

Respect de l'UX : Vérifiez toujours le mode silencieux avant de jouer un son. N'abusez pas des vibrations qui peuvent irriter l'utilisateur et consommer de la batterie.

Exemples pratiques

Feedback haptique avec Vibrator

Le service **Vibrator** permet de créer des retours tactiles pour améliorer l'expérience utilisateur. La vibration doit être courte et pertinente pour ne pas devenir intrusive.

Vibration simple

```
Vibrator v = (Vibrator)
    getSystemService(VIBRATOR_SERVICE);
v.vibrate(500); // 500ms
```

Permission requise

`VIBRATE` dans le manifeste AndroidManifest.xml

Sons système avec MediaPlayer

Pour jouer un son personnalisé, placez votre fichier audio dans `res/raw/` et utilisez MediaPlayer.

```
MediaPlayer mp = MediaPlayer.create(
    this,
    R.raw.notification
);
mp.start();
```

Bonne pratique : Respectez toujours le mode silencieux ou Ne pas déranger de l'appareil avec `AudioManager`.

Ces feedbacks sont essentiels dans les notifications, confirmations d'action, ou interfaces de jeu pour renforcer l'engagement utilisateur.

Écrans Tactiles et Gestes

Événements tactiles MotionEvent

ACTION_DOWN

Doigt touche l'écran - début d'interaction

1

ACTION_MOVE

Doigt se déplace sur l'écran - drag en cours

2

ACTION_UP

Doigt quitte l'écran - fin d'interaction

3

Gestes avancés avec GestureDetector

Le **GestureDetector** simplifie la détection de gestes complexes en analysant les séquences de MotionEvent. Il identifie automatiquement les patterns courants.

Scroll

Défilement vertical ou horizontal du contenu

Swipe / Fling

Balayage rapide dans une direction

Pinch

Pincement pour zoom avec deux doigts

Long Press

Appui prolongé pour menu contextuel

Double Tap

Double clic rapide pour zoom ou action

Exemple pratique

Événements tactiles et gestures

L'écran tactile est l'interface principale d'interaction sur Android. La gestion des événements **MotionEvent** permet de créer des expériences utilisateur riches et fluides.

ACTION_DOWN

Le doigt touche l'écran. Point de départ de toute interaction tactile.

ACTION_MOVE

Le doigt se déplace sur l'écran. Utilisé pour le glissement, le dessin, ou le drag & drop.

ACTION_UP

Le doigt quitte l'écran. Finalise l'action et déclenche l'événement (clic, swipe, etc.).

Exemple d'implémentation

```
view.setOnTouchListener((v, event) -> {
 switch (event.getAction()) {
 case MotionEvent.ACTION_DOWN:
 Log.d("TOUCH", "Écran touché");
 break;
 case MotionEvent.ACTION_MOVE:
 float x = event.getX();
 float y = event.getY();
 // Logique de déplacement
 break;
 case MotionEvent.ACTION_UP:
 Log.d("TOUCH", "Doigt levé");
 break;
 }
 return true;
});
```

APIs Réseau et Connectivité

État et types de réseau

Votre application doit vérifier l'**état de connexion** avant d'effectuer des opérations réseau. Les permissions **INTERNET** et **ACCESS_NETWORK_STATE** sont essentielles.

Wi-Fi

Activation, scan des réseaux disponibles, connexion programmatique (limité depuis Android 10)

Bluetooth

Bluetooth classique pour audio et transferts, BLE pour objets connectés basse consommation

NFC

Communication courte distance pour paiements sans contact, badges, et échange rapide de données

Wi-Fi Direct

Connexion peer-to-peer sans routeur pour partage de fichiers et streaming local

Sécurité des communications

- [ ] **Utilisez HTTPS systématiquement**
  Chiffrement des échanges obligatoire depuis Android 9. Protège contre l'interception et la modification des données

- [ ] **Validez les certificats SSL**
  Ne désactivez jamais la vérification des certificats, même en développement. Utilisez des certificats de test valides

APIs Réseau et Connectivité

Permission INTERNET

Déclaration obligatoire dans le manifeste pour toute communication réseau :

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

Vérifier la connexion

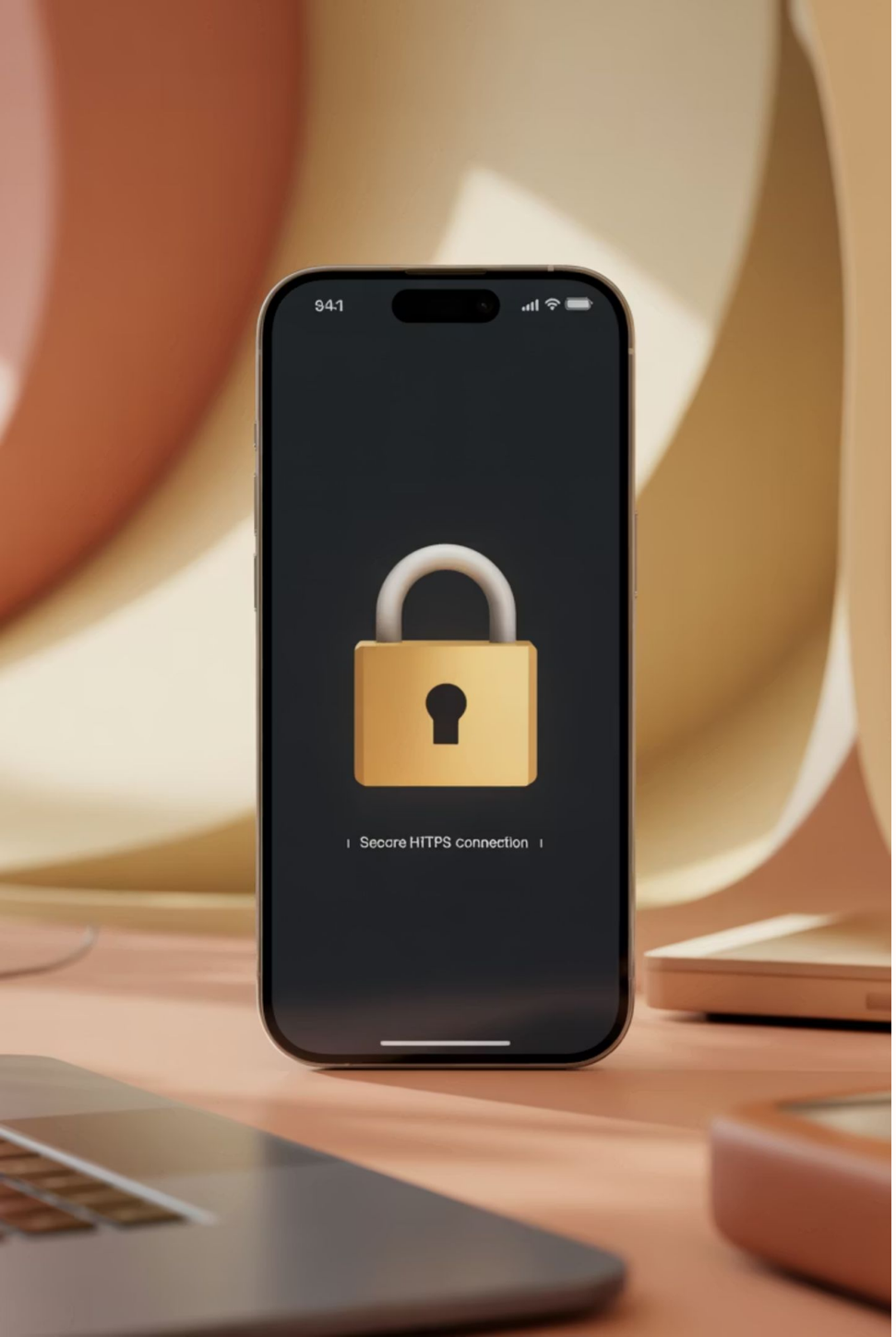
Avant toute requête, vérifiez l'état du réseau avec `ConnectivityManager` pour éviter les erreurs et améliorer l'UX.

Requête HTTP simple

```
URL url = new URL(
    "https://api.example.com/data"
);
HttpURLConnection conn =
    (HttpURLConnection)
    url.openConnection();
conn.setRequestMethod("GET");
```

Vérification réseau

```
ConnectivityManager cm =
 (ConnectivityManager)
 getSystemService(
 CONNECTIVITY_SERVICE
 );
NetworkInfo netInfo =
 cm.getActiveNetworkInfo();
if (netInfo != null &&
 netInfo.isConnected()) {
 // Connexion OK
}
```



Sécurité et bonnes pratiques réseau

Internet ≠ Wi-Fi

Une connexion Internet peut être Wi-Fi, données mobiles, ou Ethernet. Ne présumez jamais du type de connexion et adaptez votre usage (téléchargement lourd sur Wi-Fi uniquement).

Toujours utiliser HTTPS

Depuis Android 9, le trafic HTTP non chiffré est bloqué par défaut. Utilisez exclusivement HTTPS pour protéger les données de vos utilisateurs contre les interceptions.

Librairies modernes recommandées

Préférez **Retrofit** ou **OkHttp** à HttpURLConnection pour simplifier les appels API REST, gérer le JSON automatiquement, et bénéficier d'une meilleure gestion des erreurs.

Cette base réseau prépare directement l'intégration d'APIs REST complexes et la construction d'applications connectées professionnelles.

Conclusion : Responsabilité et Bonnes Pratiques

Principes fondamentaux

Les APIs Android offrent un **accès puissant aux capacités matérielles** du device, mais cette puissance s'accompagne d'une grande responsabilité envers les utilisateurs.

Permissions explicites

Demandez uniquement les permissions nécessaires et expliquez clairement pourquoi vous en avez besoin

Respect de la vie privée

Ne collectez et ne stockez que les données essentielles. Anonymisez quand possible. Conformez-vous au RGPD

Gestion d'erreurs robuste

Anticipez les refus de permissions, absences de matériel, erreurs réseau. Dégradez gracieusement les fonctionnalités

Vers les applications connectées

La maîtrise de ces APIs natives constitue le **fondement pour créer des applications modernes** qui interagissent avec des services web, consomment des APIs REST, et s'intègrent dans des écosystèmes IoT.

Dans les prochains chapitres, vous découvrirez comment connecter vos applications à des backends cloud, gérer l'authentification, et construire des architectures distribuées performantes et sécurisées.