Elaborer une application Android sécurisée

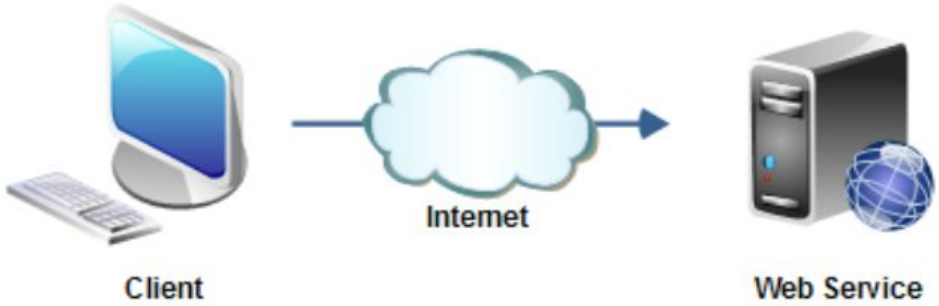
Support du cours développement mobile

ETTAHERI Nizar
ettaheri.nizar@gmail.com

A.1 – Découvrir le protocole HTTP sous Android

- # Introduction aux Web services

Les **Web Services** permettent aux applications de communiquer entre elles via Internet. Ils sont utilisés pour échanger des données et exécuter des opérations entre différentes plateformes et technologies.



• Définition d’un Web Service

Un **web service** est un programme s'exécutant sur un serveur accessible depuis Internet et fournissant un service, Par exemple :

● Google est un service web qui vous permet de rechercher des sites web.

● Une application météo communique avec un service web qui fournit la météo.

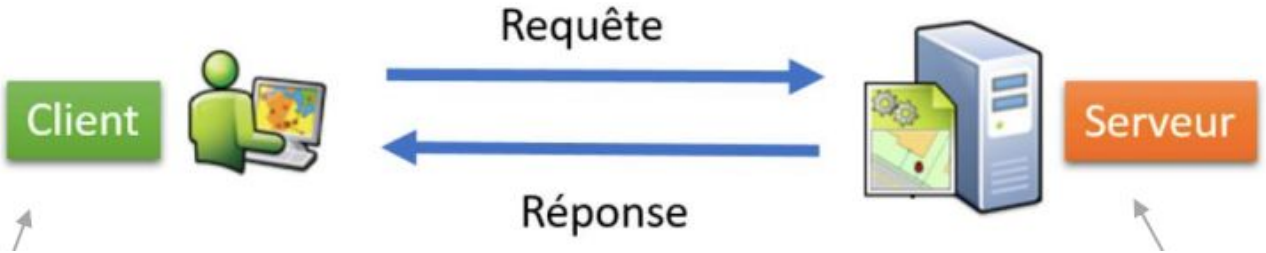
● Un réseau social est un service web qui permet de retrouver ses amis et de communiquer avec.

• Présentation de protocole REST

REST (Representational State Transfer) est un style d'architecture qui définit des contraintes pour la conception d'API Web. Il repose sur le protocole HTTP et permet aux applications d'échanger des données en utilisant des requêtes simples et efficaces.

Exemples d'utilisation :

- Une application mobile récupère la météo via une API REST.
- Un site e-commerce interagit avec une API bancaire pour traiter les paiements.
- Un service web fournit des données sur les films et séries.



- # Méthodes HTTP utilisées dans REST

REST utilise les méthodes HTTP pour définir les opérations sur les ressources :

Méthode HTTP	Action	Exemple
GET	Lire des données	GET /users → Récupérer la liste des utilisateurs
POST	Ajouter une donnée	POST /users + JSON → Créer un utilisateur
PUT	Modifier une ressource	PUT /users/1 + JSON → Modifier un utilisateur
DELETE	Supprimer une donnée	DELETE /users/1 → Supprimer un utilisateur

- # Introduction aux Web services

Postman est un outil utilisé pour tester des API (Application Programming Interface). Il permet d'envoyer des requêtes HTTP, de visualiser les réponses, d'automatiser des tests et d'analyser les performances des API.

Tester facilement les API sans écrire de code.

✔ Envoyer des requêtes **GET, POST, PUT, DELETE** et voir les réponses.

✔ Ajouter des **en-têtes HTTP, tokens d'authentification et corps de requêtes**.

✔ Sauvegarder et organiser des requêtes dans des **collections**.

✔ Automatiser des tests avec **Postman Tests (JavaScript)**.

✔ Générer du code pour appeler une API dans différents langages (**Python, Java, cURL, etc.**).

POSTMAN

TP

- Installez **Postman**
- Utilisez **Postman** pour exécuter ces commandes:

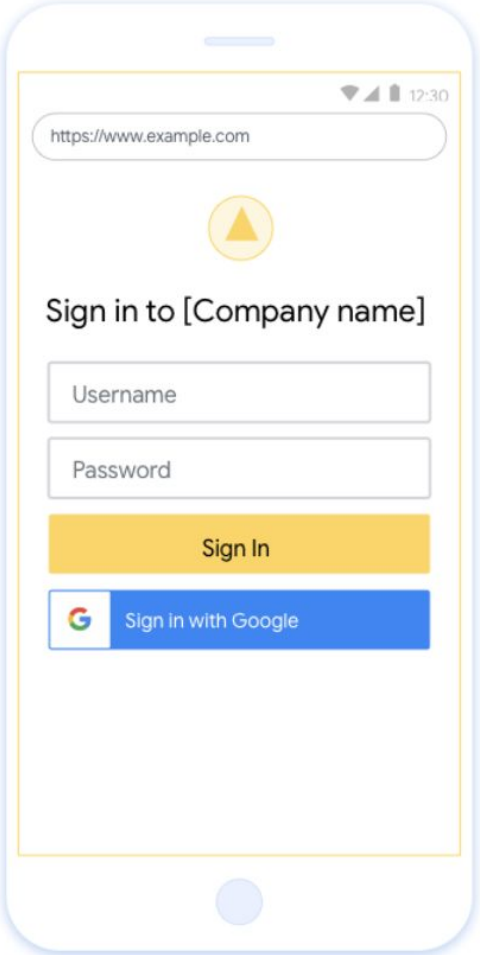
  https://jsonplaceholder.typicode.com/posts

  1. GET /posts
  2. GET /posts/1
  3. GET /posts/1/comments
  4. GET /comments?postId=1
  5. POST /posts
  6. PUT /posts/1
  7. PATCH /posts/1
  8. DELETE /posts/1

A.2 – Identifier les fondamentaux de sécurité d’une application mobile

- Utilisation de OAUTH

OAuth (Open Authorization) est un protocole d'autorisation standardisé permettant à une application d'accéder à des ressources protégées sur un autre service sans exposer les identifiants de l'utilisateur. Dans le contexte d'une application mobile, OAuth permet à un utilisateur de se connecter à un service (par exemple, Google, Facebook) en utilisant son compte existant, tout en garantissant la sécurité des données.



• Qu'est-ce que OAuth ?

OAuth permet à une application de demander un **jeton d'accès** (access token) à un service externe pour interagir avec une ressource protégée au nom de l'utilisateur, sans que ce dernier ait à partager ses identifiants. Ce jeton est utilisé dans les requêtes pour prouver l'autorisation de l'utilisateur.

Principaux acteurs dans OAuth :

- **Le client** : L'application mobile qui demande un accès à une ressource.
- **Le serveur d'autorisation** : Gère l'authentification de l'utilisateur et la délivrance des jetons d'accès.
- **Le serveur de ressources** : L'API ou le service auquel l'application souhaite accéder (par exemple, une API de météo, Google Drive).
- **L'utilisateur** : L'entité qui autorise l'accès à ses données.

- Intégration de OAuth dans une App Mobile (Exemple Google)

Étapes d'implémentation :

1. Créer un projet dans la **Google Developer Console**
2. Configurer les autorisations OAuth dans l'application mobile
3. Utiliser le jeton d'accès pour accéder aux ressources
4. Renouveler le jeton d'accès (si nécessaire)

- Intégration de OAuth dans une App Mobile (Exemple Google)

1. **Créer un projet dans la Google Developer Console**

   - Accédez à **Google Developer Console**.
   - Créez un projet et activez l'API Google que vous souhaitez utiliser (par exemple, Google Drive, Gmail).
   - Obtenez les identifiants client OAuth 2.0 (ID client et clé secrète).

- Intégration de OAuth dans une App Mobile (Exemple Google)

2. Configurer les autorisations OAuth dans l'application mobile

Pour Android, vous pouvez utiliser la Google **Sign-In API**.

Voici les étapes générales pour l'intégrer dans une application mobile :

- Ajoutez la bibliothèque Google Sign-In dans le fichier build.gradle

  implementation("com.google.android.gms:play-services-auth:21.1.1")

- Configurez le client OAuth dans l'activité de connexion
- Ajoutez une fonction pour gérer la connexion
- Gérer la réponse dans onActivityResult
- Récupérer l'ID et le jeton d'accès

- Intégration de OAuth dans une App Mobile (Exemple Google)

3. Utiliser le jeton d'accès pour accéder aux ressources

Une fois que l'utilisateur est authentifié et que vous avez récupéré un jeton d'accès, vous pouvez l'utiliser pour effectuer des requêtes authentifiées vers les API Google, par exemple :

```
val request = Request.Builder()
.url("https://www.googleapis.com/drive/v3/files")
.addHeader("Authorization", "Bearer $accessToken")
.build()

val response = client.newCall(request).execute()
```

- Intégration de OAuth dans une App Mobile (Exemple Google)

4. Renouveler le jeton d'accès (si nécessaire)

Si votre jeton d'accès expire, vous pouvez utiliser le **jeton de rafraîchissement** pour obtenir un nouveau jeton d'accès sans que l'utilisateur ait besoin de se reconnecter.

Sécuriser OAuth dans une Application Mobile

Il est crucial de mettre en place certaines pratiques pour garantir la sécurité dans une application mobile :

- **Ne jamais stocker de clés secrètes dans le code source** : Utilisez des moyens sécurisés comme le **Keystore** sur Android pour stocker des informations sensibles.
- **Limiter la portée des autorisations** : Ne demandez que les autorisations nécessaires à votre application (par exemple, uniquement l'accès à l'email et pas à toute la gestion de l'utilisateur).
- **Utiliser des connexions HTTPS** pour sécuriser les échanges de jetons.
- **Vérifier la validité des jetons** : Assurez-vous que le jeton d'accès est valide avant d'effectuer des requêtes.

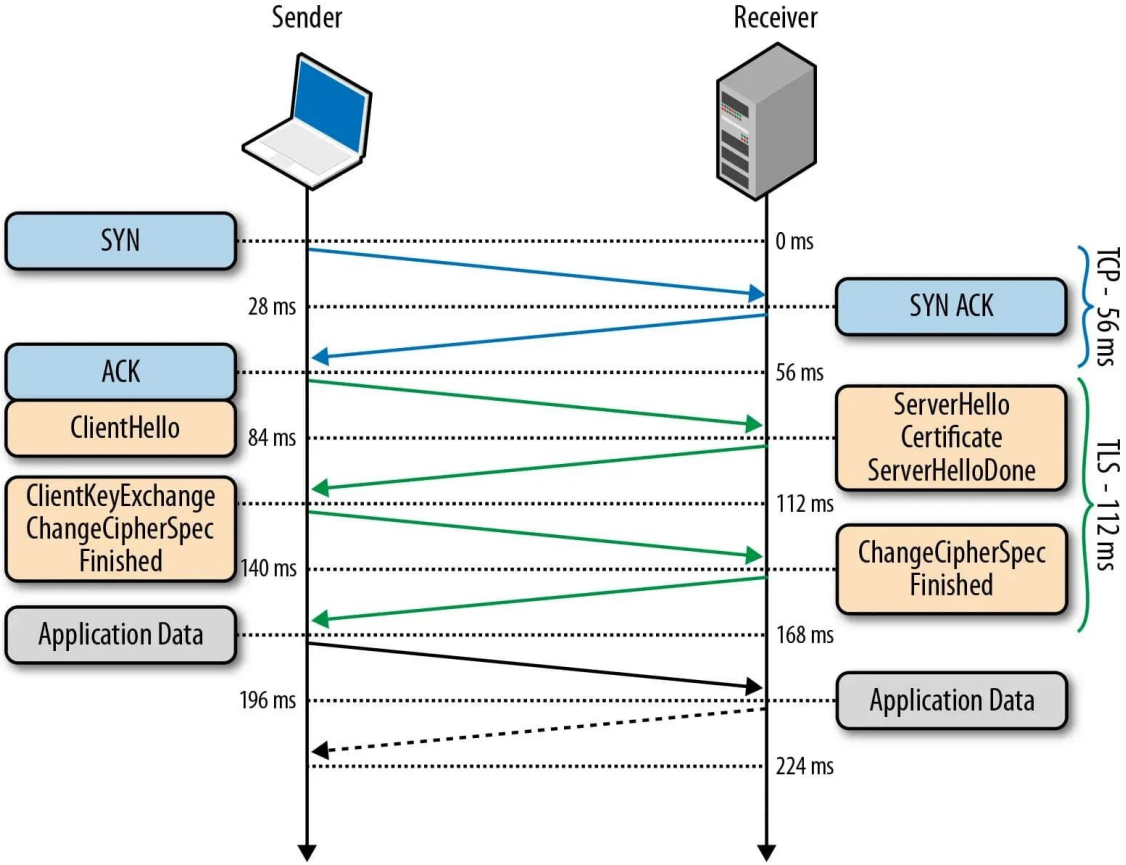
Exemple

https://github.com/NizarETH/OAuthApp

- # Intégration de SSL

L'intégration de SSL (Secure Sockets Layer), ou plus récemment TLS (Transport Layer Security), est essentielle pour garantir la sécurité des communications entre une application mobile et les serveurs distants. SSL/TLS chiffre les données échangées pour protéger la confidentialité et l'intégrité de l'information en transit. Cela est particulièrement important lors de l'envoi d'informations sensibles telles que les identifiants de connexion, les informations personnelles, etc.

- Intégration de SSL



Qu'est-ce que SSL/TLS ?

SSL/TLS est un protocole de sécurité permettant de chiffrer la communication entre deux entités (par exemple, une application mobile et un serveur) pour empêcher les attaquants d'intercepter ou de modifier les données échangées.

- **SSL (Secure Sockets Layer)** était la version initiale, mais a été progressivement remplacée par **TLS (Transport Layer Security)**, qui est plus sûr.

Fonctionnement de SSL/TLS :

- **Chiffrement** : Les données envoyées et reçues sont chiffrées pour éviter qu'elles ne soient lues par des tiers.
- **Authentification** : Le serveur prouve son identité grâce à un certificat SSL, garantissant à l'utilisateur qu'il communique bien avec le serveur authentique.
- **Intégrité** : Le protocole garantit que les données n'ont pas été altérées en cours de transmission.

Pourquoi utiliser SSL/TLS dans une Application Mobile ?

1. Sécuriser les données
2. Authentification du serveur
3. Conformité et confiance

Pourquoi utiliser SSL/TLS dans une Application Mobile ?

1. **Sécuriser les données**

Lorsque vous échangez des informations sensibles, comme des informations de paiement ou des identifiants, SSL/TLS assure que les données restent confidentielles et ne peuvent pas être interceptées par des attaquants.

Pourquoi utiliser SSL/TLS dans une Application Mobile ?

2. Authentification du serveur

SSL/TLS permet à l'application mobile de vérifier que le serveur auquel elle se connecte est bien celui auquel elle veut se connecter, ce qui empêche les attaques de type man-in-the-middle.

Pourquoi utiliser SSL/TLS dans une Application Mobile ?

3. Conformité et confiance

Les utilisateurs ont confiance dans les applications sécurisées. SSL/TLS est également requis pour des applications conformes aux normes de sécurité, comme PCI-DSS (pour les paiements) ou HIPAA (pour les données médicales).

Comment intégrer SSL/TLS dans une Application Mobile ?

A. Obtention et installation d'un certificat SSL/TLS sur le serveur

B. Forcer les connexions HTTPS dans l'application mobile

C. Vérification de la connexion HTTPS dans l'application mobile

Comment intégrer SSL/TLS dans une Application Mobile ?

A. Obtention et installation d'un certificat SSL/TLS sur le serveur

Avant d'intégrer SSL dans votre application, le serveur doit disposer d'un certificat SSL valide. Voici les étapes pour l'obtenir et l'installer :

1. **Obtenir un certificat SSL** : Acheter un certificat SSL auprès d'une autorité de certification (CA) comme Let's Encrypt (gratuit), DigiCert, ou GlobalSign.
2. **Installer le certificat SSL sur le serveur** : Une fois le certificat obtenu, installez-le sur le serveur web (Apache, Nginx, etc.) pour qu'il puisse chiffrer les connexions avec les clients.

Comment intégrer SSL/TLS dans une Application Mobile ?

B. Forcer les connexions HTTPS dans l'application mobile

Dans une application mobile, vous devez vous assurer que toutes les connexions HTTP sont redirigées vers **HTTPS** (HTTP sécurisé). Voici les étapes générales pour forcer cette sécurité :

1. Configurer HTTPS pour les API dans l'application mobile

- Vous pouvez forcer les connexions HTTPS en utilisant **HttpsURLConnection** ou des bibliothèques comme **OkHttp** ou **Retrofit**.

2. Ajouter un certificat SSL spécifique dans l'application (si nécessaire)

Si votre application doit se connecter à un serveur qui utilise un certificat SSL spécifique ou auto-signé, vous devrez peut-être intégrer ce certificat dans l'application mobile pour vérifier l'authenticité du serveur.

- Inclure le certificat dans le dossier **res/raw/**.
- Utiliser un **TrustManager** personnalisé pour accepter ce certificat.

Comment intégrer SSL/TLS dans une Application Mobile ?

C. Vérification de la connexion HTTPS dans l'application mobile

Dans votre application mobile, vous devez vous assurer que la connexion HTTPS est bien établie et que la communication est sécurisée. Voici les étapes :

Utiliser un certificat de confiance : L'application Android vérifie les certificats SSL/TLS du serveur lors de la connexion HTTPS

Vérification des erreurs de certificat : Assurez-vous que l'application gère correctement les erreurs SSL, comme les certificats expirés, les erreurs de chaîne de certification, ou les attaques de type Man-in-the-Middle (MITM).

Tester la sécurité de la connexion : Utilisez des outils comme Wireshark ou Burp Suite pour tester la sécurité des connexions et vérifier que les données sont bien chiffrées.

- # Protection avec Proguard

1. Qu'est-ce que ProGuard ?

ProGuard est un outil de minification, obfuscation et optimisation du code source dans les applications Java (et Android). Il permet :

Minification : Réduit la taille de l'application en supprimant le code inutilisé.

Obfuscation : Rend le code difficile à comprendre pour un attaquant en modifiant les noms de variables, méthodes et classes.

Optimisation : Améliore les performances de l'application en supprimant les parties du code qui ne sont pas utilisées.

Cela rend plus difficile pour un attaquant de comprendre la logique de l'application si elle est décompilée, et réduit les informations disponibles pour l'ingénierie inverse.

Activer ProGuard dans un projet Android

Activer la minification dans build.gradle (niveau module) :

Dans votre fichier build.gradle de votre application, vous devez activer ProGuard en modifiant la configuration de build dans la section buildTypes. Par défaut, la minification est désactivée.

Exemple pour activer ProGuard dans la version de release de votre application :

```
android {
    buildTypes {
        release {
            // Active la minification et l'obfuscation
            minifyEnabled true

// Indique quel fichier de configuration ProGuard utiliser
            proguardFiles
getDefaultProguardFile('proguard-android-optimize.txt'
), 'proguard-rules.pro'
        }
    }
}
```

Activer ProGuard dans un projet Android

Configurer les règles ProGuard :

Dans votre projet Android, vous pouvez définir des règles de configuration de ProGuard dans un fichier proguard-rules.pro à la racine de votre projet. Ce fichier vous permet de spécifier quelles classes, méthodes ou attributs ne doivent pas être obfusqués, minifiés ou supprimés.

Exemple d'une règle basique dans proguard-rules.pro :

```
# Conserver certaines classes et méthodes spécifiques
-keep class com.example.myapp.** { *; }
-dontwarn com.example.myapp.**

# Empêcher l'obfuscation des annotations
-keepattributes *Annotation*
# Ne pas supprimer les méthodes utilisées par la reflection
-keepclassmembers class * {
    public static void main(java.lang.String[]);
}
# Préserver les classes de sécurité ou liées aux cryptages
-keep class javax.crypto.** { *; }
-keep class org.apache.commons.** { *; }
```

Activer ProGuard dans un projet Android

Effectuer un build de release :

Une fois ProGuard activé et les règles définies, vous pouvez générer une version "release" de votre application :

```
./gradlew assembleRelease
```

Cela obfusquera le code, minimisera la taille de l'APK et appliquera les optimisations définies.

Obfuscation du code avec ProGuard

L’obfuscation consiste à renommer les classes, méthodes et variables en utilisant des noms courts et non significatifs, afin que le code devient difficile à comprendre.

```
class UserManager {
    var username;
    var password;

    fun login(user : String, pass : String) {
        // login logic
    }
}
```

Avant obfuscation

```
class a {
    var a;
    var b;

    fun a(a : String, b : String) {
        // login logic
    }
}
```

Après obfuscation

Protéger des parties sensibles de l'application

Il existe des parties spécifiques du code que vous pouvez **exclure** de l'obfuscation ou de la minification, comme les clés API, les algorithmes de cryptage ou les méthodes utilisées par la réflexion.

- Exclure les classes sensibles de l'obfuscation
- Exclure les méthodes utilisées par la réflexion

La réflexion en Kotlin fonctionne **au moment de l'exécution**. Elle permet d'examiner ou manipuler les classes, propriétés et fonctions **dynamiquement**, sans les connaître à la compilation.

Protéger des parties sensibles de l'application

Exclure les classes sensibles de l'obfuscation :

Si vous avez des classes sensibles que vous ne voulez pas obfusquer (comme des classes de cryptographie), vous pouvez utiliser l'option -keep dans le fichier proguard-rules.pro pour les protéger.

Exemple pour exclure une classe de l'obfuscation :

```
-keep class com.example.security.EncryptionUtils { *; }
```

Protéger des parties sensibles de l'application

Exclure les méthodes utilisées par la réflexion :

ProGuard pourrait supprimer ou renommer des méthodes utilisées par la réflexion si vous ne les spécifiez pas explicitement dans les règles.

Pour éviter cela, vous pouvez ajouter des règles comme :

```
-keepclassmembers class * {
    public <methods>;
}
```

Éviter la décompilation avec des outils

En plus de l'utilisation de ProGuard, il est recommandé d'ajouter des mécanismes de sécurité supplémentaires pour renforcer la protection contre la décompilation et l'ingénierie inverse :

- **Utiliser des outils de "Code Obfuscation" avancés** comme **R8**, qui remplace ProGuard dans les versions récentes d'Android Studio. R8 offre des optimisations plus efficaces et plus sécurisées.
- **Intégration de techniques anti-debugging et anti-tampering** : Implémentez des vérifications pour détecter si l'application est déboguée ou modifiée, ce qui rend plus difficile l'injection de code malveillant.
- **Crypter les clés API et autres données sensibles** dans l'application pour éviter qu'elles ne soient extraites lors de la décompilation. Utilisez des outils de sécurité comme **Encyptor** ou **Keychain** pour stocker ces données de manière sécurisée.

Tester et valider l'efficacité de la protection

Après avoir appliqué ProGuard, vous devez tester l'application pour vous assurer qu'elle fonctionne correctement et qu'aucun code essentiel n'a été supprimé ou obfusqué de manière incorrecte.

1. **Testez la décompilation** : Utilisez des outils comme **JADX** ou **JD-GUI** pour décompiler l'APK et voir si la logique de l'application est facilement compréhensible.
2. **Testez les fonctionnalités de l'application** : Vérifiez que les fonctionnalités de votre application (surtout celles qui utilisent la réflexion ou des API sensibles) fonctionnent toujours correctement après la minification et l'obfuscation.

```
jadx -d output_folder mon_application.apk

jadx-gui mon_application.apk
```

R8 et ProGuard : Différences et Utilisation en Android

R8 et ProGuard sont des outils d’optimisation et de minification du code utilisés dans les applications Android.

ProGuard

Ancien outil utilisé pour réduire la taille du code en supprimant les classes inutilisées, en renommant les noms des classes et méthodes, et en optimisant le bytecode.

✅ **Avantages :**

- Minification du code
- Obfuscation (rend le code plus difficile à comprendre)
- Optimisation du bytecode

❌ **Inconvénients :**

- Lent
- Moins efficace que R8

R8

Depuis **Android Studio 3.4**, **R8 a remplacé ProGuard** par défaut. Il offre les mêmes fonctionnalités mais est plus performant.

✅ **Avantages :**

- **Plus rapide** que ProGuard
- **Meilleure optimisation** du code
- **Fusionne** les étapes de compilation et d’optimisation

Configuration de R8 dans une application Android

R8 utilise les mêmes fichiers de configuration que ProGuard : proguard-rules.pro

Exemple de règles ProGuard/R8 :

```
# Ne pas obfusquer les modèles Room
-keep class com.example.app.data.model.** { *; }

# Ne pas obfusquer les classes annotées avec @Keep
-keep @androidx.annotation.Keep class *

# Ne pas supprimer les méthodes natives utilisées par JNI
-keepclasseswithmembernames class * {
    native <methods>;
}
```

Activer/Désactiver R8

Dans gradle.properties :

```
android.enableR8=true   # R8 activé (par défaut)

android.enableR8=false  # Désactiver et utiliser ProGuard
```

R8 est aujourd'hui le standard et offre de meilleures performances que ProGuard.
Il est conseillé de l'utiliser sauf en cas de problème spécifique.

TP 1

A.3 – Consommer des API sous Android

- # Introduction à FireBase

Firebase est une **plateforme Backend-as-a-Service (BaaS)** proposée par Google. Elle permet de gérer facilement des fonctionnalités comme l'authentification, la base de données, le stockage, les notifications push, et bien plus, sans avoir à développer un backend complet.

Principales Fonctionnalités de Firebase

A. Authentification

Gère les connexions avec **Google, Facebook, email/password, etc.**

Exemple d'authentification avec email/password :

```
val auth = FirebaseAuth.getInstance()

auth.signInWithEmailAndPassword("email@example.com",
"password123")
    .addOnCompleteListener { task ->
        if (task.isSuccessful) {
            Log.d("FirebaseAuth", "Connexion réussie")
        } else {
            Log.e("FirebaseAuth", "Erreur : ${task.exception?.message}")
        }
    }
```

Principales Fonctionnalités de Firebase

B. Firestore (Base de données NoSQL en temps réel)

Stocke et synchronise des données entre plusieurs appareils.

Exemple d’ajout de données :

```
val db = FirebaseFirestore.getInstance()
val user = hashMapOf("name" to "Alice", "age" to 25)

db.collection("users").document("alice")
    .set(user)
    .addOnSuccessListener { Log.d("Firestore", "Données ajoutées") }
    .addOnFailureListener { e -> Log.e("Firestore", "Erreur : $e") }
```

Principales Fonctionnalités de Firebase

C. Firebase Storage

Stocke des fichiers comme **des images, vidéos, PDF**.

Exemple d’upload d’image :

```
val storageRef = FirebaseStorage.getInstance().reference
val fileUri = Uri.fromFile(File("path/to/image.jpg"))
val fileRef = storageRef.child("images/photo.jpg")

fileRef.putFile(fileUri)
    .addOnSuccessListener { Log.d("FirebaseStorage", "Upload réussi")
}
    .addOnFailureListener { e -> Log.e("FirebaseStorage", "Erreur : $e")
}
```

Principales Fonctionnalités de Firebase

D. Firebase Cloud Messaging (Notifications Push)

Permet d’envoyer des **notifications push** aux utilisateurs.

Exemple pour récupérer le token FCM :

```
FirebaseMessaging.getInstance().token
    .addOnCompleteListener { task ->
        if (task.isSuccessful) {
            val token = task.result
            Log.d("FCM", "Token reçu : $token")
        }
    }
```

Principales Fonctionnalités de Firebase

E. Firebase Analytics

Permet de **suivre le comportement des utilisateurs** dans l'application.

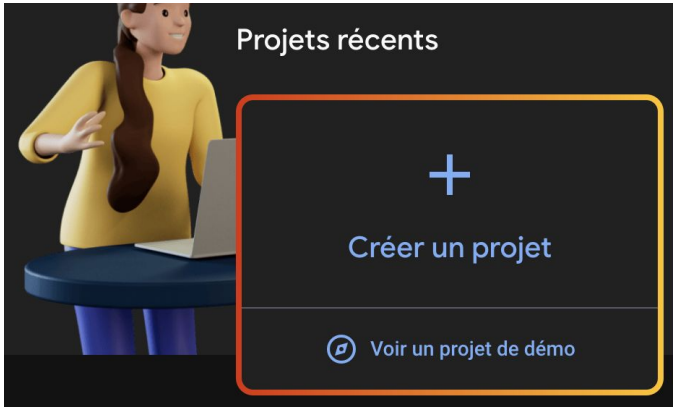
Exemple d'événement personnalisé :

```
val analytics = Firebase.analytics
val bundle = Bundle().apply {
    putString("screen_name", "home")
}
analytics.logEvent("screen_view", bundle)
```

2. Installation de Firebase dans un projet Android

A. Ajouter Firebase à un projet

1. Aller sur **Firebase Console**
2. Créer un projet Firebase
3. Ajouter une application Android (renseigner le **package name**)
4. Télécharger le fichier **google-services.json** et le placer dans **app/**



2. Installation de Firebase dans un projet Android

B. Ajouter Firebase dans build.gradle

Dans **build.gradle (Project)** :

```
dependencies {
    classpath 'com.google.gms:google-services:4.3.15'
    // Assurez-vous d'avoir la dernière version
}
```

Dans **build.gradle (Module: app)** :

```
plugins {
    id 'com.android.application'
    id 'com.google.gms.google-services'
    // Ajout du plugin Firebase
}
```

Puis ajouter les dépendances nécessaires (Firestore et Auth) :

```
dependencies {
    implementation 'com.google.firebase:firebase-auth-ktx:22.0.0'
    implementation 'com.google.firebase:firebase-firestore-ktx:24.8.1'
}
```

3. Sécurité et Bonnes Pratiques

1. Utiliser Firebase Authentication pour sécuriser les accès.
2. Mettre à jour les règles de sécurité Firestore (Firestore Database > Rules) :

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /users/{userId} {
      allow read, write: if request.auth != null && request.auth.uid == userId;
    }
  }
}
```

3. Éviter d’exposer des clés API sensibles en les stockant dans un fichier sécurisé.

TP 2

• Manipulation de REST API

Une **API REST (Representational State Transfer)** permet à une application de communiquer avec un serveur via HTTP. Elle est souvent utilisée pour récupérer, envoyer, modifier ou supprimer des données.

Une API REST utilise généralement les méthodes HTTP suivantes :

- **GET** → Récupérer des données
- **POST** → Envoyer des données
- **PUT** → Mettre à jour des données
- **DELETE** → Supprimer des données

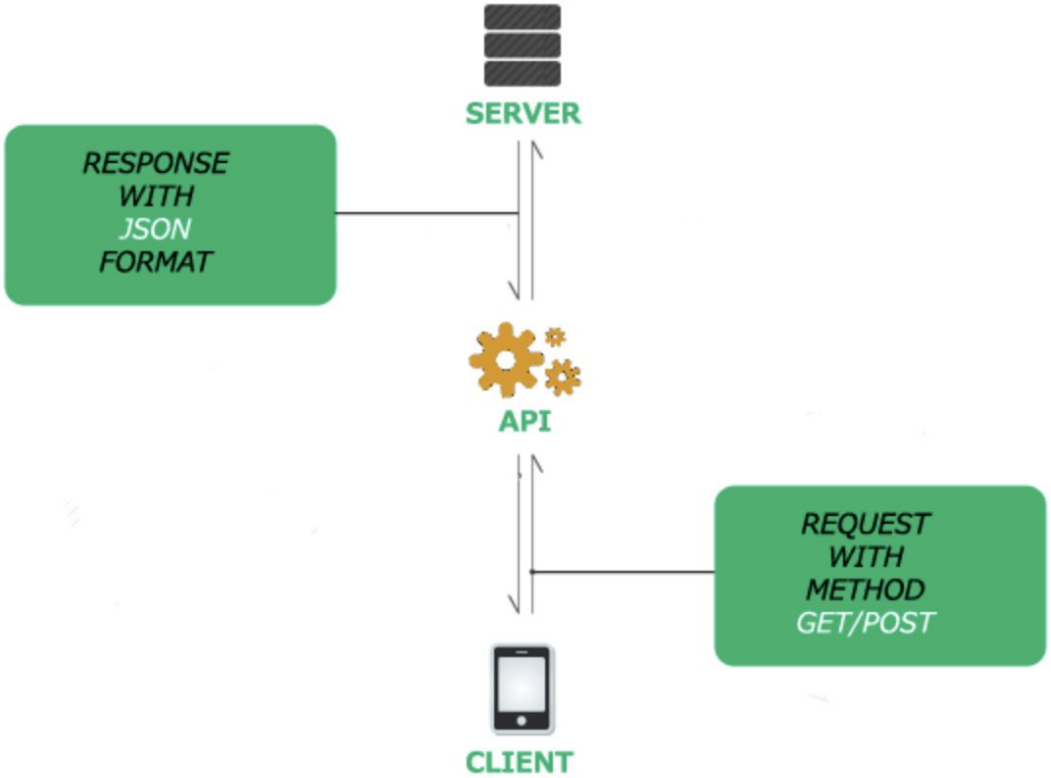
Exemple d'API publique : https://jsonplaceholder.typicode.com/posts (retourne une liste d'articles).

Définition de Retrofit

Retrofit est une bibliothèque **open-source** développée par **Square** qui facilite la communication entre une application **Android (Kotlin/Java)** et une **API REST**.

Pourquoi utiliser Retrofit ?

- Simplifie l'envoi et la réception des requêtes HTTP
- Convertit automatiquement les réponses JSON en objets Kotlin
- Gère les requêtes **synchrones et asynchrones**
- Supporte **les requêtes GET, POST, PUT, DELETE**
- Compatible avec **Gson, Moshi, Scalars, RxJava, Coroutines**



Comment fonctionne Retrofit ?

Retrofit suit 3 étapes principales :

1. **Définir une interface API** avec les endpoints (ex: @GET("posts"))
2. **Configurer Retrofit** en spécifiant l'URL de base et le convertisseur (ex: Gson)
3. **Faire des appels réseau** et récupérer les données

Exemple d'utilisation

Étape 1 : Ajouter Retrofit au projet (`build.gradle`)

```
dependencies {
    implementation 'com.squareup.retrofit2:retrofit:2.9.0'
    implementation 'com.squareup.retrofit2:converter-gson:2.9.0'
}
```

Exemple d'utilisation

Étape 2 : Définir une interface API

```
import retrofit2.Call
import retrofit2.http.GET

interface ApiService {
    @GET("posts")
    fun getPosts(): Call<List<Post>>
}
```

Exemple d'utilisation

Étape 3 : Configurer Retrofit

```
import retrofit2.Retrofit
import retrofit2.converter.gson.GsonConverterFactory

object RetrofitInstance {
    private const val BASE_URL = "https://jsonplaceholder.typicode.com/"

    val api: ApiService by lazy {
        Retrofit.Builder()
            .baseUrl(BASE_URL)
            .addConverterFactory(GsonConverterFactory.create())
            .build()
            .create(ApiService::class.java)
    }
}
```

Exemple d'utilisation

Étape 4 : Faire un appel API

```
RetrofitInstance.api.getPosts().enqueue(object : Callback<List<Post>> {
    override fun onResponse(call: Call<List<Post>>, response:
Response<List<Post>>) {
        if (response.isSuccessful) {
            val posts = response.body()
            println("Données reçues : $posts")
        }
    }

    override fun onFailure(call: Call<List<Post>>, t: Throwable) {
        println("Erreur : ${t.message}")
    }
})
```

TP 3

- # Gestion des images avec Glide

Glide est une bibliothèque d'Android pour le chargement d'images, particulièrement efficace pour gérer des images à partir d'URL, de fichiers locaux ou de ressources internes. Elle offre des fonctionnalités avancées permettant d'optimiser la gestion de l'affichage d'images, en réduisant les problèmes liés à la mémoire et au traitement d'images lourdes.

Utilisation de Glide

1. **Chargement Asynchrone** : Glide charge les images de manière asynchrone en arrière-plan, ce qui signifie qu'il ne bloque pas l'interface utilisateur. Cela permet de maintenir une expérience utilisateur fluide.

2. **Gestion de la Mémoire** : Glide optimise l'utilisation de la mémoire avec un cache sur disque et en mémoire. Il ajuste dynamiquement la taille des images pour correspondre à la résolution de l'écran et la taille de la vue.

Utilisation de Glide

3. **Transformation d'Images** : Glide prend en charge des transformations d'images puissantes et flexibles via les API comme BitmapTransformation ou DrawableTransformation. Vous pouvez appliquer des filtres, recadrer des images ou appliquer des arrondis sur les coins d'images.

Exemple pour arrondir une image :

```
Glide.with(context)
    .load(imageUrl)
    .transform(RoundedCorners(20)) // Applique un arrondi de 20px
    .into(imageView)
```

Utilisation de Glide

4. **Chargement d'Images Animées** : Glide prend en charge les animations GIF et les vidéos WebM. Vous pouvez charger des GIF animés directement dans vos vues.

Exemple pour charger un GIF :

```
Glide.with(context).asGif().load(gifUrl).into(imageView)
```

Utilisation de Glide

5. **Placeholders et Erreurs** : Vous pouvez définir des images par défaut à afficher pendant le chargement ou en cas d'erreur. Cela améliore l'expérience visuelle en attendant que l'image se charge.

```
Glide.with(context)
    .load(imageUrl)
    .placeholder(R.drawable.loading) // Image affichée pendant le chargement
    .error(R.drawable.error) // Image affichée en cas d'échec
    .into(imageView)
```

Utilisation de Glide

6. **Téléchargement d'Image avec des Options de Cache** : Vous pouvez définir des stratégies de cache pour contrôler la manière dont Glide gère la mise en cache des images, ce qui est utile pour réduire l'utilisation des données et accélérer les chargements d'images.

Exemple pour forcer le cache uniquement sur disque :

```
Glide.with(context)
    .load(imageUrl)
    .diskCacheStrategy(DiskCacheStrategy.ALL) // Toutes les images mises en cache
    .into(imageView)
```

Utilisation de Glide

7. **Préchargement des Images** : Glide vous permet de précharger des images avant qu'elles ne soient réellement affichées. Cela peut être utile pour améliorer la performance lorsque vous prévoyez d'afficher une image dans une autre activité ou fragment.

Exemple de préchargement :

```
Glide.with(context)
    .load(imageUrl)
    .preload() // Précharge l'image sans l'afficher
```

Utilisation de Glide

7. **Sélection de Taille d'Image** : Glide vous permet de spécifier des dimensions exactes pour une image, ce qui est utile pour économiser la mémoire et réduire le temps de chargement, en particulier sur les images haute résolution.

Exemple :

```
Glide.with(context)
    .load(imageUrl)
    .override(500, 500) // Réduit la taille de l'image à 500x500px
    .into(imageView)
```

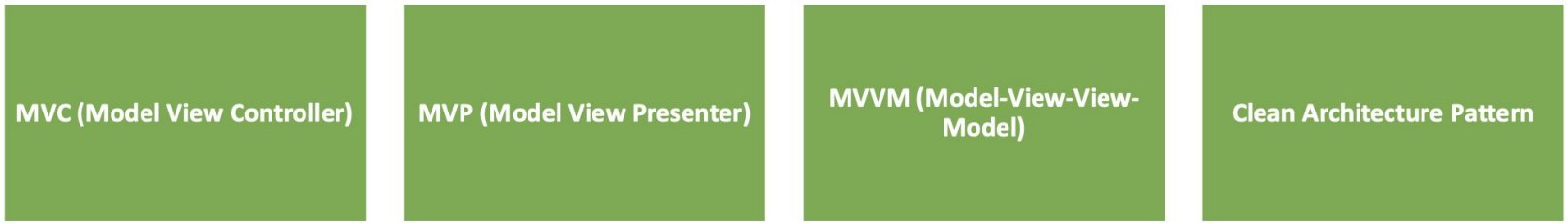
Glide est une bibliothèque puissante et flexible pour le chargement d'images dans Android, avec de nombreuses options pour optimiser les performances et gérer les images efficacement. Vous pouvez l'utiliser pour des besoins de base ou des cas plus complexes, comme les transformations et l'optimisation de la mémoire.

TP 4

Maîtriser les architectures d’une application mobile Android

Les modèles MVC, MVP, MVVM et Clean Arch.

Un modèle de conception est une forme réutilisable d'une solution à un problème de conception. C'est un style de codage par lequel nous pouvons gérer les différents composants du système que nous réalisons. Nous allons ici discuter de quatre de ces modèles de conception : MVC, MVP, MVVM et Clean Architecture.



Maîtriser les architectures d’une application mobile Android

Le modèle MVC (Model View Controller)

Il s'agit de l'une des approches les plus utilisées dans le développement de logiciels. Le MVC se compose de trois éléments principaux :

• **Model** : Le modèle représente l'objet dans l'application. Il contient la logique de l'endroit où les données doivent être récupérées. Il peut également contenir la logique par laquelle le contrôleur peut mettre à jour la vue. Dans Android, le modèle est principalement représenté par des classes d'objets.

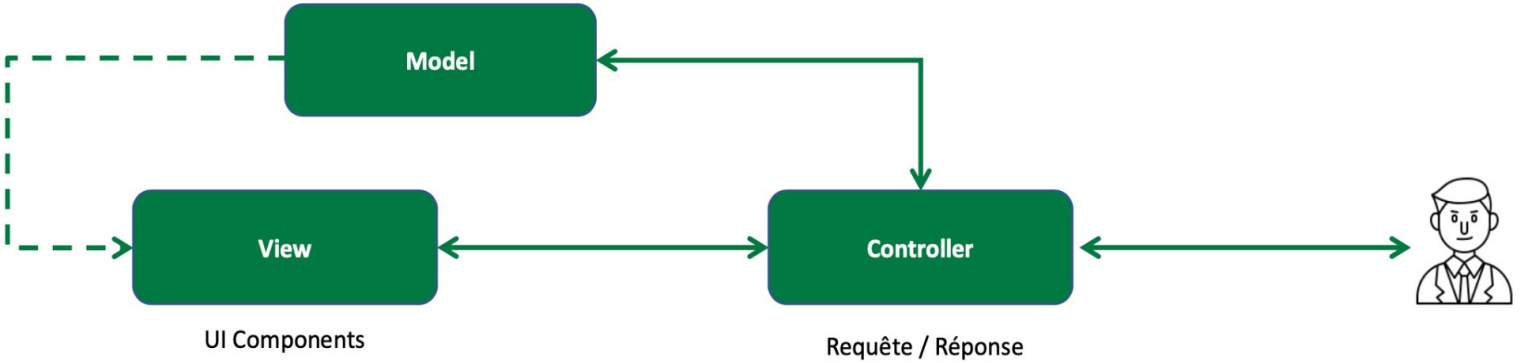
• **View** : La vue est constituée des composants qui peuvent interagir avec l'utilisateur et est responsable de la façon dont le modèle est affiché dans l'application. Dans Android, la vue est principalement représentée par le XML où les mises en page peuvent être conçues.

• **Controller** : Le contrôleur agit comme un médiateur entre le modèle et la vue. Il contrôle le flux de données dans l'objet modèle et met à jour la vue lorsque les données changent. Dans Android, le contrôleur est principalement représenté par les activités et les Fragments.

Maîtriser les architectures d’une application mobile Android

Le modèle MVC (Model View Controller)

Android n'est pas capable de suivre complètement l'architecture MVC, car Activity/Fragment peut agir à la fois comme contrôleur et comme vue, ce qui fait que tous les codes sont regroupés au même endroit. L'Activity/Fragment peut être utilisé pour dessiner plusieurs vues pour un seul écran dans une application, ainsi les différents appels de données et les vues sont peuplés au même endroit. Par conséquent, pour résoudre ce problème, nous pouvons utiliser différents modèles de conception ou mettre en œuvre MVC soigneusement en prenant soin des conventions et en suivant les directives de programmation appropriées.



Maîtriser les architectures d’une application mobile Android

Le modèle MVP (Model View Presenter)

Model View Presenter (MVP) est dérivé du modèle MVC. Le MVP est utilisé pour minimiser la forte dépendance de la vue, ce qui est le cas dans le MVC. Il sépare la vue et le modèle en utilisant le présentateur. Le présentateur décide de ce qui doit être affiché sur la vue :

• **Model** : Le modèle représente les objets de l'application. Il contient la logique de l'endroit où les données doivent être récupérées.

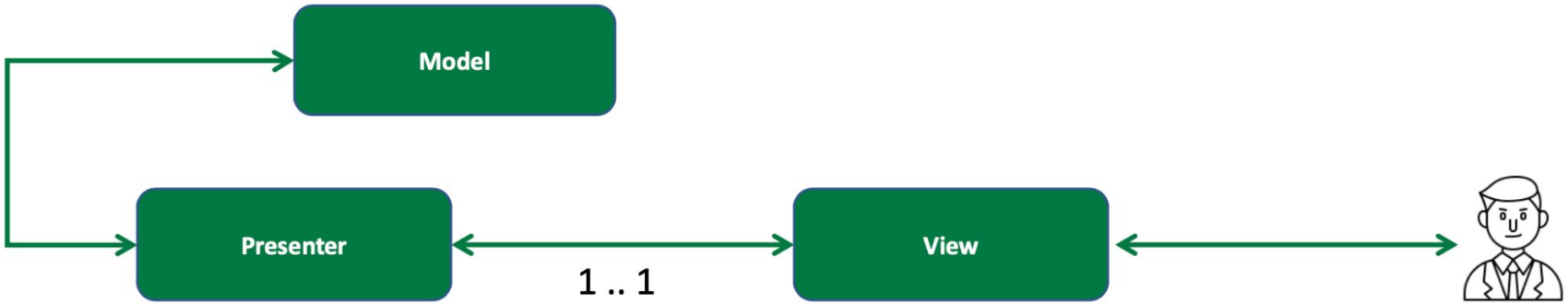
• **View** : La vue rend les informations aux utilisateurs et contient un composant d'interface utilisateur (fichiers XML), une activité, des Fragments et un dialogue sous la couche de vue. Elle n'a pas d'autre logique implémentée.

• **Presenter** : La couche de présentateur exécute la tâche du contrôleur et sert de médiateur entre la vue et le modèle. Mais contrairement au contrôleur, elle ne dépend pas de la vue. La vue interagit avec le présentateur pour que les données soient affichées, et le présentateur prend ensuite les données du modèle et les renvoie à la vue dans un format présentable. Le présentateur ne contient aucun composant d'interface utilisateur ; il se contente de manipuler les données du modèle et de les afficher sur la vue.

Maîtriser les architectures d’une application mobile Android

Le modèle MVP (Model View Presenter)

Dans la MVP, le présentateur communique avec la vue par le biais d'interfaces. Les interfaces sont définies dans la classe du présentateur, à laquelle il transmet les données requises. L'activité/Fragment ou tout autre composant de la vue implémente les interfaces et rend les données de la manière qu'il souhaite. La connexion entre le présentateur et la vue est un à un.



Maîtriser les architectures d’une application mobile Android

Le modèle MVVM (Model View View Model)

MVVM (Model-View-View-Model) : Il est similaire au modèle MVC, à la seule différence qu'il possède une liaison de données bidirectionnelle avec la vue et le modèle de vue. Les modifications de la vue sont propagées par le modèle de vue, qui utilise un modèle d'observateur pour communiquer entre le modèle de vue et lemodèle. Dans ce cas, la vue est complètement isolée du modèle.

MVVM comporte les éléments suivants :

• **Model** : Le modèle représente les objets de l'application. Il contient la logique de l'endroit où les données doivent être récupérées.

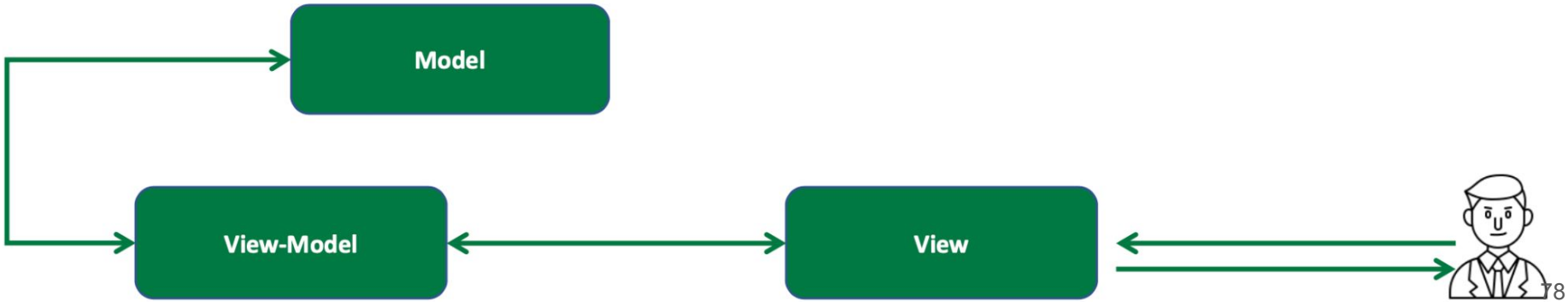
• **View** : La vue est similaire à celle du modèle MVC, qui rend les informations aux utilisateurs et contient un fichier .xml de composant d'interface utilisateur, une activité, des Fragments et un dialogue sous la couche de vue. Elle n'a pas d'autre logique mise en œuvre.

• **View-model** : Le modèle de vue aide à maintenir l'état de la vue et apporte des modifications au modèle en fonction des données obtenues de la vue.

Maîtriser les architectures d’une application mobile Android

Le modèle MVVM (Model View View Model)

De nombreuses vues peuvent être liées à un modèle de vue, ce qui crée une relation de plusieurs à un entre la vue et un modèle de vue. En outre, une vue possède des informations sur le modèle de vue, mais le modèle de vue ne possède aucune information sur la vue. La vue n'est pas responsable de l'état des informations ; celles-ci sont gérées par le modèle de vue, et la vue et le modèle ne sont reflétés que par les modifications apportées au modèle de vue.



Maîtriser les architectures d'une application mobile Android

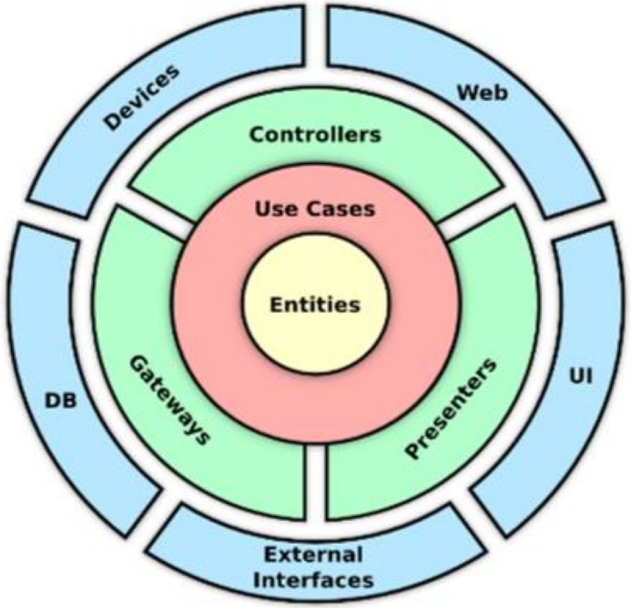
Comparaison des trois architectures

	MVC	MVP	MVVM
Maintenance	Difficile à maintenir	Facile à maintenir	Facile à maintenir
Difficulté	Facile à apprendre	Facile à apprendre	Plus difficile à apprendre en raison des fonctions supplémentaires
Type de relation	Relation plusieurs à un entre le contrôleur et la vue	Relation 1 à 1 entre le présentateur et la vue	Relation plusieurs à un entre la vue et le modèle de vue
Tests unitaires	En raison d'un couplage étroit, il est difficile à tester avec des tests unitaires	Bonnes performances	Excellentes performances
Point d'accès	Contrôleur	Vue	Vue
Références	La vue n'a pas de référence au contrôleur	La vue fait référence au presenter	La vue fait référence au modèle de vue

Maîtriser les architectures d’une application mobile Android

Modèle d'architecture propre (Clean Architecture Pattern)

Le modèle d'architecture propre, dans ses termes les plus simples, signifie écrire un code propre, en le séparant en couches, la couche externe étant vos implémentations et la couche interne étant la logique commerciale. Une interface relie ces deux couches, contrôlant la façon dont les couches externes utilisent les couches internes.
Ce type de modèle d'architecture de code est également connu sous le nom d'architecture en Onion en raison de ses différentes couches, comme le montre la figure suivante.



Les couches internes n'ont aucune idée des couches externes.
Les couches externes utilisent les composants des couches internes en fonction de leurs besoins. Ce qui signifie que les couches externes dépendent des implémentations de la logique métier des couches internes. La dépendance est donc orientée vers l'intérieur.

Maîtriser les architectures d’une application mobile Android

Modèle d'architecture propre (Clean Architecture Pattern)

• **Entities (couche 0)** : Les entités sont l'une des deux couches de domaine du cercle intérieur qui représentent la logique métier et de domaine. Il s'agit de règles métier à l'échelle de l'entreprise, ou de données qui seront toujours vraies ou statiques pour cette application. Les entités peuvent être des objets avec des méthodes ou simplement une collection de structures de données et de fonctions. Elles encapsulent les éléments les plus généraux ou de plus haut niveau et sont donc les moins susceptibles de changer suite à une modification d'une couche externe.Par exemple, les fonctionnalités de base d'une application ne seraient pas modifiées par le passage de framework frontal React à Angular.

• **Use cases (couche 1)** : Les cas d'utilisation constituent la deuxième couche du domaine. Elles définissent les règles de gestion spécifiques à l'application. Elles encapsulent et mettent en œuvre tous les cas d'utilisation approuvés pour l'application. Les cas d'utilisation contrôlent le flux vers et depuis les entités et peuvent demander aux entités d'utiliser leurs règles à l'échelle de l'entreprise pour accomplir certaines tâches utilisateur. Les changements dans cette couche n'affecteront pas les entités ou les couches plus externes. Toutefois, cette couche devra être modifiée si des couches externes sontmodifiées.

Maîtriser les architectures d’une application mobile Android

Modèle d'architecture propre (Clean Architecture Pattern)

• **Adaptateurs d'interface (couche 2) :** Cette couche adapte l'entrée dans une forme qui est plus utilisable par les couches de cas d'utilisation et d'entités. Elle formate également la sortie des entités ou des cas d'utilisation dans une forme qui convient le mieux aux canaux externes. La couche des adaptateurs est la limite effective entre les couches du cercle intérieur et extérieur. Aucun code à l'intérieur de cette couche ne doit connaître ou référencer quoi que ce soit de plus externe que cette couche. Les adaptateurs d'interface peuvent être considérés comme un convertisseur qui convertit et relaie les informations de la manière la plus utilisable par les couches internes et externes respectivement.

• **Framework et Drivers (couche 3) :** Framework et Drivers est la couche de présentation qui est généralement composée de frameworks et d'outils tels que des bases de données, des frameworks web, etc. Cette couche ne comporte pas beaucoup de code mais contient plutôt toutes les références concrètes nécessaires à des détails spécifiques tels que des opérations spécifiques à la base de données ou des commandes spécifiques au framework actuel. Par exemple, elle contient entièrement l'architecture MVC d'une interface graphique.

Maîtriser les architectures d’une application mobile Android

Avantages d'architecture propre (Clean Architecture Pattern)

• **Hautement testable :** L'architecture propre est construite de A à Z pour être testée. Il est possible de créer des cas de test pour chaque couche afin de déterminer où les erreurs se produisent dans le cercle.

• **Indépendante du framework** : L'architecture propre ne s'appuie pas sur les outils d'un framework spécifique et n'utilise pas le framework comme dépendance dans le code.

• **Indépendante de la base de données** : La majorité des applications ne connaissent pas ou n'ont pas besoin de connaître la base de données dont elles sont issues. Cela permet d'adopter une nouvelle base de données sans modifier la majorité du code source.

• **Indépendante de l’IU** : Les frameworks UI existent sur la couche la plus externe et ne sont donc qu'un présentateur pour les données transmises par les couches internes. Ainsi, il est possible de changer l'interface utilisateur à tout moment.

Quizz

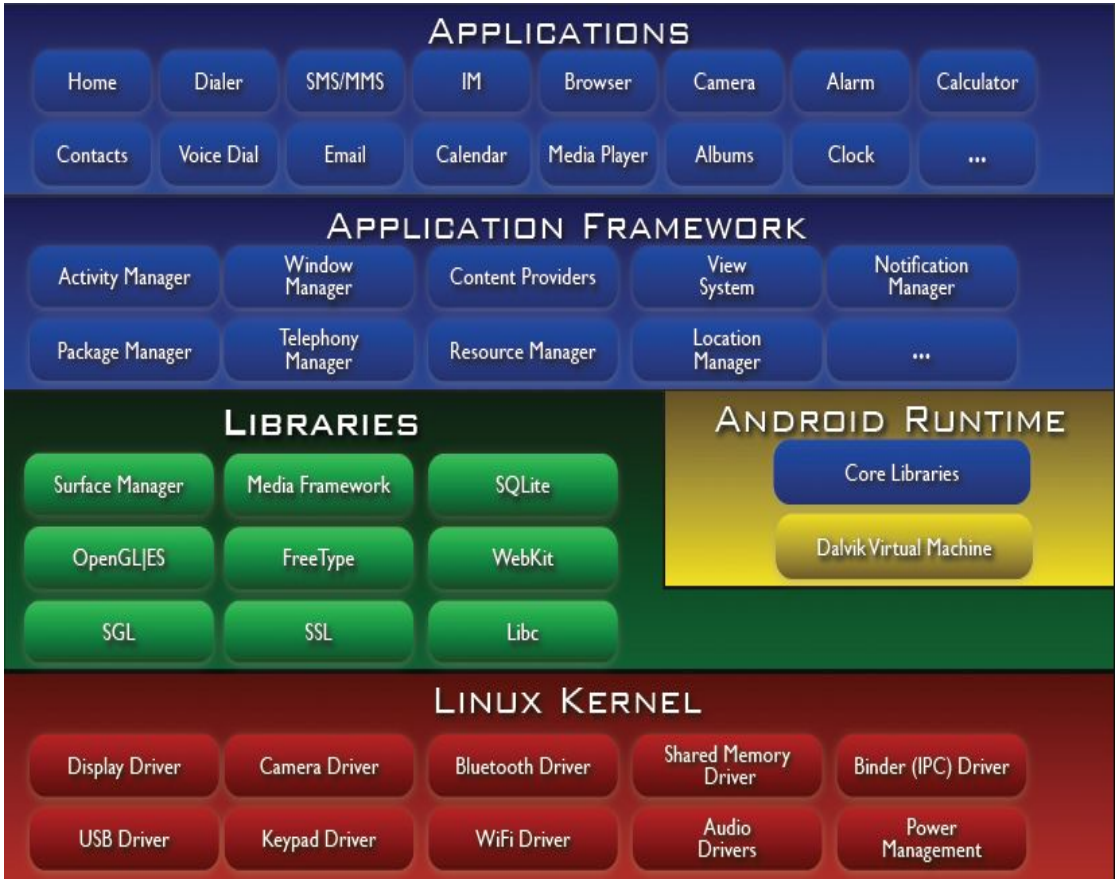
Android

Android est basé sur un kernel linux 2.6.xx, au dessus du kernel il y a "**l'hardware abstraction layer**" qui permet de séparer la plateforme logique du matériel.

Au dessus de cette couche d'abstraction on retrouve les librairies C/C++ utilisées par un certain nombre de composants du système Android.

Au dessus des librairies on retrouve l'Android Runtime, cette couche contient les librairies cœurs du Framework ainsi que la machine virtuelle exécutant les applications.

Au dessus la couche "**Android Runtime**" et des librairies cœurs on retrouve le Framework permettant au développeur de créer des applications. Enfin au dessus du Framework il y a les applications.



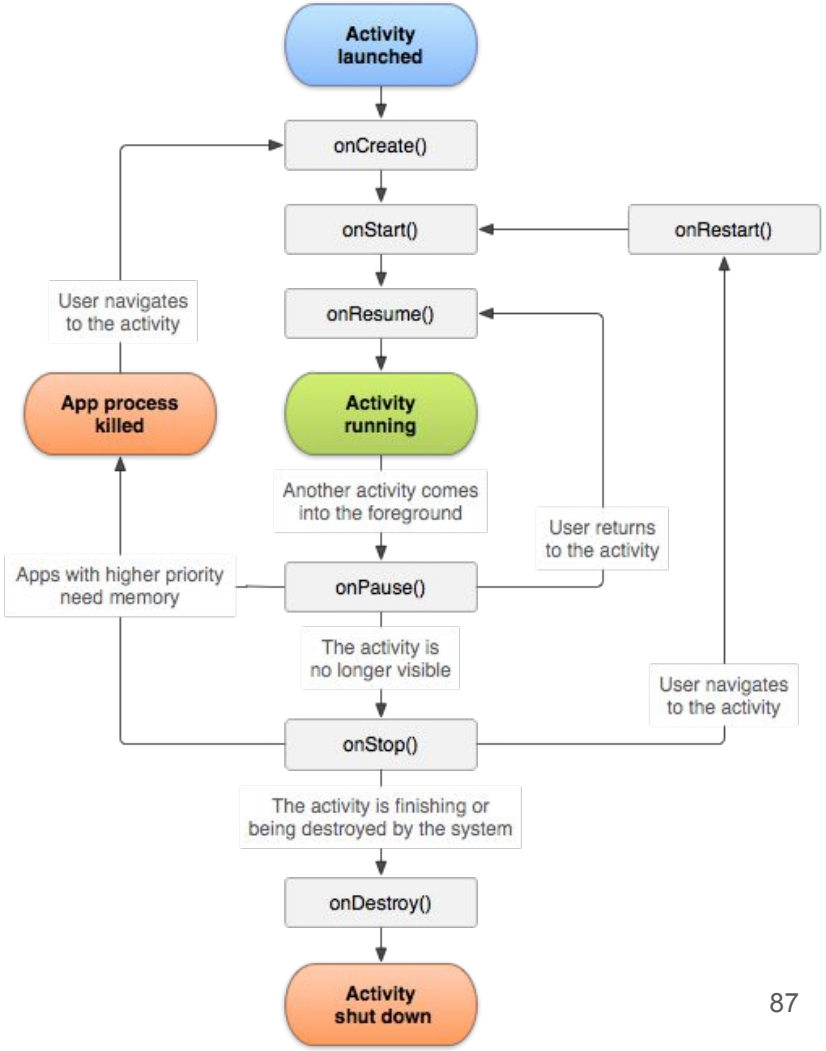
Connaître le cycle de vie des composants applicatifs

Activity

Une activité (activity) représente un écran dans une application Android, l’emploi des layouts permet de garantir un rendu adapté selon le type d’appareil et les dimensions de l’écran.

Connaître le cycle de vie des composants applicatifs

Cycle de vie d'une Activity



Connaître le cycle de vie des composants applicatifs

États d'activité et visibilité des applications

- Créé (Created) = pas encore visible
- Démarré (Started) = visible
- Repris (Resume) = visible
- Mis en pause (Paused) = partiellement invisible
- Stoppé (Stopped) = caché
- Détruit (Destroyed) = disparu de la mémoire

Les changements d'état sont déclenchés par l'action de l'utilisateur.

Les changements de configuration tels que la rotation du dispositif ou l'action du système.

Connaître le cycle de vie des composants applicatifs

Implémentation et remplacement des callbacks :

• Seul onCreate() est nécessaire

• Remplacer les autres rappels pour modifier le comportement par défaut

• **onCreate()–> Created :**

  • Appelé lorsque l'activité est créée pour la première fois, par exemple lorsque l'utilisateur touche l'icône du lanceur

  • Effectue toute la configuration statique : création des vues, liaison des données aux listes...

  • Appelé une seule fois pendant la durée de vie d'une activité

  • Prend un Bundle avec l'état précédemment gelé de l'activité, s'il y en a un

  • L'état créé est toujours suivi par onStart().

Connaître le cycle de vie des composants applicatifs

onStart() –> Started

• Appelé lorsque l'activité devient visible pour l'utilisateur

• Peut être appelé plusieurs fois au cours du cycle de vie

• Suivi de onResume() si l'activité passe au premier plan, ou de onStop() si elle devient cachée

Connaître le cycle de vie des composants applicatifs

onRestart() –> Started

- Appelé après l'arrêt d'une activité, immédiatement avant son redémarrage ;
- État transitoire ;
- Toujours suivi de onStart().

Connaître le cycle de vie des composants applicatifs

onResume() –> Resumed/Running

- Appelé lorsque l'activité commence à interagir avec l'utilisateur
- L'activité est passée au sommet de la pile d'activités
- Commence à accepter les entrées de l'utilisateur
- État d'exécution
- Toujours suivi de onPause().

Connaître le cycle de vie des composants applicatifs

onPause() –> Paused

• Appelé lorsque le système est sur le point de reprendre une activité précédente

• L'activité est partiellement visible mais l'utilisateur quitte l'activité

• Généralement utilisé pour valider les modifications non sauvegardées des données persistantes, arrêter les animations et tout ce qui consomme des ressources

• Les implémentations doivent être rapides car l'activité suivante n'est pas reprise avant le retour de cette méthode

• Suivi par onResume() si l'activité revient à l'avant, ou onStop() si elle devient invisible pour l'utilisateur

Connaître le cycle de vie des composants applicatifs

onStop() –> Stopped

• Appelé lorsque l'activité n'est plus visible pour l'utilisateur

• Une nouvelle activité est lancée, une activité existante est placée devant celle-ci ou celle-ci est détruite

• Opérations qui étaient trop lourdes pour onPause()

• Suivie par onRestart() si l'activité revient pour interagir avec l'utilisateur, ou onDestroy() si l'activité disparaît

Connaître le cycle de vie des composants applicatifs

onDestroy() –> Destroyed

• Dernier appel avant la destruction de l'activité

• L'utilisateur revient à l'activité précédente ou modifie la configuration

• L'activité se termine ou le système la détruit pour gagner de l'espace

• Appeler la méthode isFinishing() pour vérifier

• Le système peut détruire l'activité sans l'appeler, alors utiliser onPause() ou onStop() pour sauvegarder les données ou l'état

Connaître le cycle de vie des composants applicatifs

Quand la configuration change-t-elle ?

Les changements de configuration invalident la disposition actuelle ou d'autres ressources de l'activité lorsque l'utilisateur :

• Fait pivoter l'appareil

• Choisit une langue système différente, ce qui entraîne une modification des paramètres locaux

• Entre en mode multifenêtre (à partir d'Android 7)

Connaître le cycle de vie des composants applicatifs

Que se passe-t-il en cas de changement de configuration ?

Au changement de configuration, Android :

1. Arrête l'activité en appelant :

• onPause()

• onStop()

• onDestroy()

2. Redémarre l'activité en appelant :

• onCreate()

• onStart()

• onResume()

Connaître le cycle de vie des composants applicatifs

Une modification de la configuration entraîne l'appel de onDestroy()

La rotation d'écran est un type de modification de la configuration qui entraîne l'arrêt et le redémarrage de l'activité. Pour simuler cette modification de configuration et examiner ses effets, procédez comme suit :

```
MainActivity          com.example.dessertclicker          D  onCreate Called
MainActivity          com.example.dessertclicker          D  onStart Called
MainActivity          com.example.dessertclicker          D  onResume Called
MainActivity          com.example.dessertclicker          D  onPause Called
MainActivity          com.example.dessertclicker          D  onStop Called
MainActivity          com.example.dessertclicker          D  onDestroy Called
MainActivity          com.example.dessertclicker          D  onCreate Called
MainActivity          com.example.dessertclicker          D  onStart Called
MainActivity          com.example.dessertclicker          D  onResume Called
```

Connaître le cycle de vie des composants applicatifs

État de l'instance d'activité :

• Les informations sur l'état sont créées pendant que l'activité est en cours, comme un compteur, un texte utilisateur, une progression de l'animation;

• L'état est détruit lorsque :

- L'appareil est tourné
- La langue change
- L'on appuie sur le bouton arrière
- Le système efface la mémoire

Connaître le cycle de vie des composants applicatifs

Sauvegarde et restauration de l'état de l'activité

- Le système sauvegarde uniquement :
  - L'état des vues avec un ID unique (android:id), comme le texte saisi dans EditText
  - L'intention qui a lancé l'activité et les données dans ses extras
- L'utilisateur est responsable de la sauvegarde des autres activités et des données de progression de l'utilisateur
- Implémenter onSaveInstanceState() dans l'activité :
  - Appelée par le runtime Android lorsqu'il y a une possibilité que l'activité soit détruite
  - Sauvegarde les données uniquement pour cette instance de l'activité pendant la session en cours

```
@Override
public void onSaveInstanceState(Bundle outState) {
super.onSaveInstanceState(outState);

outState.putString("count",
String.valueOf(mShowCount.getText()));
}
```

- un Bundle est une classe qui permet de stocker et de transmettre des paires clé-valeur de données entre les composants d'une application, comme les Activity, Fragment, ou même lors de la sauvegarde de l'état de l'application.

Connaître le cycle de vie des composants applicatifs

Sauvegarde et restauration de l'état de l'activité

• Deux façons de récupérer le Bundle sauvegardé :
  • Dans la méthode préférée **onCreate**(Bundle mySavedState), afin de garantir que l'interface utilisateur, y compris tout état sauvegardé, soit de nouveau opérationnelle aussi rapidement que possible.
  • Implémentation d'un **callback** (appelé après onStart()) onRestoreInstanceState(Bundle mySavedState).

```
@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
mShowCount = findViewById(R.id.show_count);

if (savedInstanceState != null) {
        String count =
        savedInstanceState.getString("count");
if (mShowCount != null)
        mShowCount.setText(count);
        }
}
```

Connaître le cycle de vie des composants applicatifs

Sauvegarde et restauration de l'état de l'activité

- onRestoreInstanceState(Bundle state) :

```
@Override
public void onRestoreInstanceState (Bundle
mySavedState) {
super.onRestoreInstanceState(mySavedState);
  if (mySavedState != null) {
      String count =
                  mySavedState.getString("count");
   if (count != null)
        mShowCount.setText(count);
}
}
```

Connaître le cycle de vie des composants applicatifs

État de l'instance et redémarrage de l'application

• Lors de l'arrêt et du redémarrage d'une nouvelle session d'application, les états de l'instance d'activité sont perdus et les activités reprennent leur apparence par défaut.

• Lorsque des données utilisateur doivent être sauvegardées entre deux sessions d'application, utiliser des préférences partagées ou une base de données.

Quiz

TP 1

Intent

un Intent est un objet qui permet à différents composants (comme les activités, les services, et les récepteurs de diffusion) de communiquer entre eux. Les intents sont principalement utilisés pour :

- Démarrer des activités
- Transmettre des données
- Lancer des services
- Envoyer des diffusions (Broadcast)

Démarrer des activités

Vous pouvez utiliser un intent pour lancer une nouvelle activité ou passer d'une activité à une autre.

Par exemple, ouvrir une nouvelle page dans votre application.

```
val intent = Intent(this, SecondeActivite::class.java)
startActivity(intent)
```

Transmettre des données

Un intent peut transporter des informations d'une activité à une autre via des extras (clé-valeur). Par exemple, passer une chaîne de caractères d'une activité à une autre :

```
val intent = Intent(this,
SecondeActivite::class.java)
intent.putExtra("cle", "valeur")
startActivity(intent)
```

Lancer des services

Un intent peut également démarrer ou interagir avec un service en arrière-plan.

```
val intent = Intent(this, MonService::class.java)
startService(intent)
```

Envoyer des diffusions (Broadcast)

Les intents peuvent également servir à diffuser des événements ou des notifications au sein du système ou à d'autres applications.

Les Types d'intents

Il existe deux types d'intents :

- Intent explicite : Celui-ci spécifie directement le composant à démarrer, comme dans l'exemple ci-dessus où l'activité ou le service est explicitement mentionné.
- Intent implicite : Celui-ci ne spécifie pas le composant exact mais déclare une action à effectuer, permettant ainsi à d'autres applications ou composants capables d'accomplir cette action de répondre à l'intent.

  Exemple d'intent implicite pour ouvrir une page web :

```
val intent = Intent(Intent.ACTION_VIEW)
intent.data = Uri.parse("http://www.example.com")
 startActivity(intent)
```

Bundle

un **Bundle** est un objet utilisé pour stocker et transmettre des paires clé-valeur entre différentes composantes d'une application, comme entre deux activités ou fragments.

Transmettre des données entre activités

Lors du démarrage d'une nouvelle activité, vous pouvez ajouter un Bundle à un Intent pour passer des informations.

```
val intent = Intent(this,SecondeActivite::class.java)
val bundle = Bundle()
bundle.putString("cle", "valeur")
intent.putExtras(bundle)
startActivity(intent)
```

Sauvegarder l'état de l'activité

Lors d'événements comme la rotation de l'écran ou lorsqu'une activité est mise en pause, un Bundle peut être utilisé pour sauvegarder l'état de l'interface utilisateur. Cela permet de restaurer l'état de l'application lorsque l'activité est recréée.

```
override fun onSaveInstanceState(outState: Bundle) {
        super.onSaveInstanceState(outState)
        outState.putString("cle", "valeur") }

override fun onRestoreInstanceState(savedInstanceState: Bundle) {
        super.onRestoreInstanceState(savedInstanceState)
        val valeur = savedInstanceState.getString("cle")
}
```

Fragment

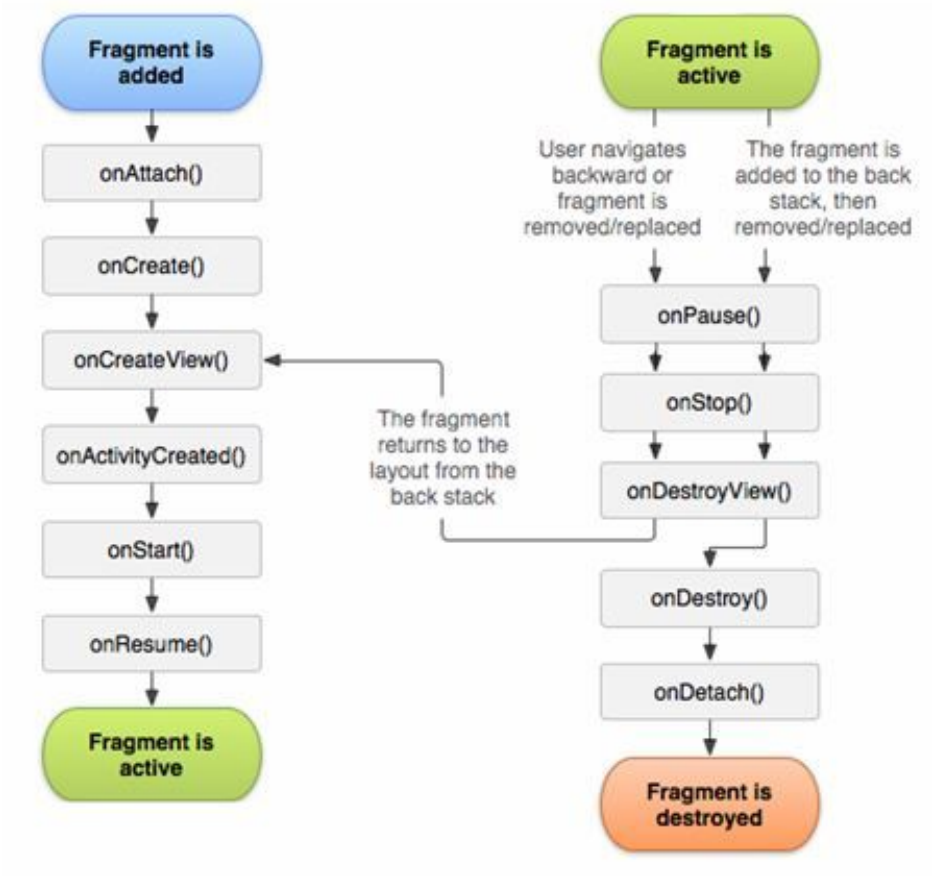
Un **Fragment** est une composante d'interface utilisateur réutilisable qui représente une partie d'une activité. Les fragments sont utilisés pour créer des interfaces plus flexibles et dynamiques, en particulier sur des écrans plus grands comme les tablettes. Ils peuvent être combinés ou remplacés dynamiquement à l'intérieur d'une activité, permettant ainsi une gestion plus modulaire de l'interface.

Caractéristiques principales des Fragments

Cycle de vie : Le cycle de vie d'un fragment est étroitement lié à celui de l'activité dans laquelle il est intégré, mais il a ses propres méthodes de cycle de vie comme :

- onCreate()
- onCreateView()
- onViewCreated()
- onStart()
- onResume()
- onPause()
- onDestroyView()

Cycle de vie


Fragment is added
onAttach()
onCreate()
onCreateView()
onActivityCreated()
onStart()
onResume()
Fragment is active
Fragment is active
User navigates backward or fragment is removed/replaced
The fragment is added to the back stack, then removed/replaced
onPause()
onStop()
onDestroyView()
onDestroy()
onDetach()
Fragment is destroyed
The fragment returns to the layout from the back stack

Exemple minimaliste de cycle de vie d'un fragment

```
class MonFragment : Fragment() {
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
        ): View? {
        return inflater.inflate(R.layout.fragment_layout, container, false)
    }
}
```

Ajout de Fragments à une activité

Il existe deux façons d'ajouter un fragment à une activité :

- **Statique** : Déclaré directement dans le fichier XML de l'activité.

```
<fragment android:id="@+id/monFragment"
android:name="com.exemple.MonFragment"
android:layout_width="match_parent"
android:layout_height="match_parent" />
```

- **Dynamique** : Ajouté ou remplacé à l'exécution via du code Java/Kotlin à l'aide du FragmentManager et du FragmentTransaction.

```
val fragment = MonFragment()
supportFragmentManager.beginTransaction()
.replace(R.id.fragment_container, fragment) .commit()
```

Communication entre activité et fragment

Un fragment peut communiquer avec l'activité dans laquelle il est hébergé ou avec d'autres fragments via des interfaces, Bundle ou des ViewModel partagés.

Exemple d'une interface pour transmettre des données de l'activité au fragment :

```
class MainActivity : AppCompatActivity(), MonFragmentListener {
    override fun onFragmentInteraction(data: String) {
        // Gérer l'interaction depuis le fragment
    }
}

interface MonFragmentListener {
    fun onFragmentInteraction(data: String)
}
```

Passage de données à un Fragment

Lorsqu'on crée un fragment, il est courant d'utiliser un Bundle pour transmettre des données.

```
val fragment = MonFragment()
val bundle = Bundle()
bundle.putString("cle", "valeur")
fragment.arguments = bundle
```

Dans le fragment, vous pouvez récupérer ces données dans la méthode onCreate() ou onViewCreated().

```
val valeur = arguments?.getString("cle")
```

Communication entre fragment et Activity

1. Définir une interface dans le Fragment.
2. Faire en sorte que l'Activity implémente cette interface.
3. Passer les données via l'interface.

1. Créer une interface dans le Fragment

Créer une interface à l'intérieur du Fragment pour envoyer les données à l'Activity.

```
class ExampleFragment : Fragment() {

    private lateinit var dataPasser: OnDataPass

    override fun onAttach(context: Context) {
        super.onAttach(context)
            dataPasser = context as OnDataPass }

override fun onViewCreated(view: View, savedInstanceState: Bundle?)
{
        super.onViewCreated(view, savedInstanceState)

        val button: Button = view.findViewById(R.id.my_button)
        button.setOnClickListener {
            dataPasser.myData("Hello Activity")
        }
    }
}
```

```
interface OnDataPass {
    fun myData(data: String)
}
```

2. Implémenter l'interface dans l'Activity

Dans l'Activity, implémente l'interface pour recevoir les données du Fragment.

```
class ExampleActivity : AppCompatActivity(), ExampleFragment.OnDataPass {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_example)
    }
    override fun myData(data: String) {
        Log.d("ExampleActivity", "Données reçues : $data")
        Toast.makeText(this, data, Toast.LENGTH_SHORT).show()
    }
}
```

3. Mettre en place l'interface lors de l'attachement du fragment

Quand le Fragment est attaché à l'Activity, il vérifie si l'Activity implémente l'interface et utilise cette méthode pour passer les données.

```
override fun onAttach(context: Context) {
        super.onAttach(context)
            dataPasser = context as OnDataPass
}
```

Fragments imbriqués

Vous pouvez également imbriquer des fragments au sein d'autres fragments pour des interfaces plus complexes, en utilisant childFragmentManager (Exemple d'utilisation de ViewPager) au lieu de supportFragmentManager.

Avantages des Fragments

- **Réutilisabilité** : Un fragment peut être réutilisé dans plusieurs activités, ce qui permet de simplifier et d'optimiser le développement d'interfaces complexes.
- **Modularité** : Les fragments permettent une interface utilisateur modulaire, où différentes parties de l'interface peuvent être affichées ou masquées selon la taille ou l'orientation de l'écran.
- **Gestion dynamique** : Ils peuvent être ajoutés, remplacés ou retirés pendant l'exécution, permettant des interfaces flexibles et interactives.

Adapter

un Adapter est une classe intermédiaire qui sert de pont entre une source de données et un composant d'interface utilisateur (UI) qui affiche ces données. Les adapters sont largement utilisés dans les composants UI tels que les ListView, RecyclerView, Spinner, et autres. Ils permettent d'adapter et d'afficher les données provenant de différentes sources (comme des tableaux, des listes, des bases de données, etc.) dans un format visuel approprié.

Types d'adapters en Android

- ArrayAdapter
- BaseAdapter
- RecyclerView.Adapter
- SimpleAdapter

ArrayAdapter

Utilisé pour lier un tableau ou une liste de données à un composant UI comme un ListView ou un Spinner.

Chaque élément de la liste est une simple vue de texte par défaut, mais vous pouvez personnaliser la disposition des éléments.

```
val listView: ListView = findViewById(R.id.listView)
val data = arrayOf("Item 1", "Item 2", "Item 3")

val adapter = ArrayAdapter(this, android.R.layout.simple_list_item_1, data)
listView.adapter = adapter
```

BaseAdapter

C'est une classe abstraite qui sert de base pour créer des adapters personnalisés lorsque les besoins de personnalisation sont plus importants que ce que l'ArrayAdapter ou d'autres classes fournies offrent.

Vous devez implémenter des méthodes comme getCount(), getItem(), getItemId(), et getView() pour définir comment les données sont affichées.

```
val listView: ListView = findViewById(R.id.listView)
val adapter = CustomAdapter(this, listOf("Item 1", "Item 2", "Item 3"))
listView.adapter = adapter
```

BaseAdapter

```
class CustomAdapter(private val context: Context, private val data: List<String>) : BaseAdapter() {
    override fun getCount(): Int = data.size
    override fun getItem(position: Int): String = data[position]
    override fun getItemId(position: Int): Long = position.toLong()
    override fun getView(position: Int, convertView: View?, parent: ViewGroup?): View {
        val view = convertView ?: LayoutInflater.from(context).inflate(R.layout.list_item, parent, false)
        val textView: TextView = view.findViewById(R.id.textView)
        textView.text = getItem(position)
        return view  }
}
```

RecyclerView.Adapter

Utilisé spécifiquement avec le composant RecyclerView, qui est plus performant que ListView pour des grandes listes de données.

Le RecyclerView.Adapter fonctionne en tandem avec un ViewHolder, ce qui permet de réutiliser les vues de manière efficace (réduisant ainsi le coût en termes de performance).

```
val recyclerView: RecyclerView = findViewById(R.id.recyclerView)
recyclerView.layoutManager = LinearLayoutManager(this)
recyclerView.adapter = MyAdapter(listOf("Item 1", "Item 2", "Item 3"))
```

RecyclerView.Adapter

```
class MyAdapter(private val items: List<String>) : RecyclerView.Adapter<MyAdapter.MyViewHolder>() {
    class MyViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
        val textView: TextView = itemView.findViewById(R.id.textView)    }
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): MyViewHolder {
        val view = LayoutInflater.from(parent.context).inflate(R.layout.list_item, parent, false)
        return MyViewHolder(view)     }
    override fun onBindViewHolder(holder: MyViewHolder, position: Int) {
        holder.textView.text = items[position]    }
    override fun getItemCount(): Int = items.size
}
```

TP 2

LiveData

LiveData en Kotlin est une classe observable et consciente du cycle de vie, utilisée pour stocker et observer des données qui peuvent être affichées dans les composants de l'interface utilisateur, comme les activités ou les fragments. Elle fait partie des Architecture Components d'Android et est souvent utilisée pour gérer les données liées à l'interface utilisateur en respectant le cycle de vie de celle-ci.

Principales caractéristiques de LiveData :

Consciente du cycle de vie : LiveData ne notifie les observateurs que lorsque ceux-ci sont dans un état de cycle de vie actif (comme STARTED ou RESUMED). Cela évite les fuites de mémoire et les mises à jour inutiles lorsque l'activité ou le fragment est inactif.

Synchronisation automatique de l'UI : Lorsque les données contenues dans LiveData changent, les composants d'interface utilisateur observateurs (activités/fragments) sont automatiquement mis à jour.

Pas de fuites de mémoire : Les observateurs sont automatiquement retirés lorsqu'ils ne sont plus en état d'observer (comme lorsqu'une activité est détruite).

Gestion des configurations : En cas de changement de configuration (comme la rotation de l'écran), LiveData conserve les données, ce qui évite de devoir recharger des données après de tels événements.

LiveData et MutableLiveData

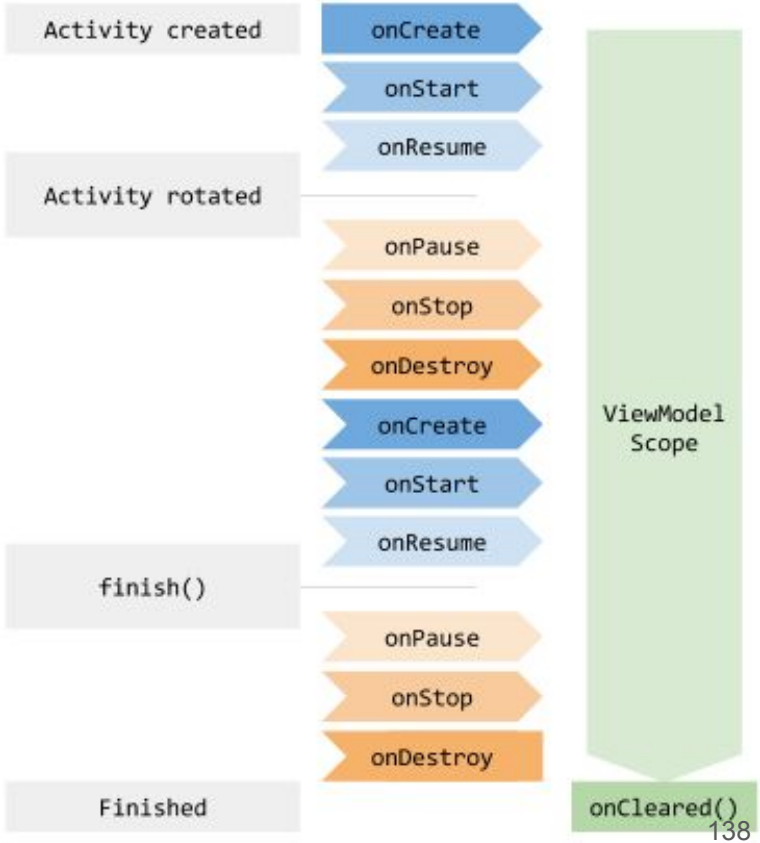
LiveData permet de gérer les données de manière sécurisée et efficace en tenant compte du cycle de vie des composants.

MutableLiveData est utilisé pour modifier les données, tandis que LiveData est utilisé pour l'exposer aux observateurs.

Les composants d'UI (activités, fragments) observent la LiveData et réagissent aux changements de données, ce qui permet une synchronisation automatique.

ViewModel

un **ViewModel** fait partie des composants d'architecture d'Android et est conçu pour stocker et gérer les données liées à l'interface utilisateur tout en étant conscient du cycle de vie des composants. Cela permet aux données de survivre aux changements de configuration, comme la rotation de l'écran, qui entraînent souvent la recréation des activités ou des fragments.



Principaux avantages du ViewModel

Conscience du cycle de vie : Le ViewModel est conscient du cycle de vie des composants de l'interface utilisateur comme les activités et les fragments. Cela signifie que les données sont conservées même si ces composants sont recréés.

Séparation des préoccupations : Le ViewModel sépare la logique des données de l'interface utilisateur, ce qui facilite la gestion du code et le rend plus maintenable.

Survie aux changements de configuration : Contrairement aux activités et fragments, qui sont détruits et recréés lors des changements de configuration (comme la rotation de l'écran), les ViewModels restent en mémoire jusqu'à ce que l'activité soit définitivement fermée.

Comment utiliser un ViewModel :

Ajouter la dépendance dans le fichier build.gradle :

```
implementation 'androidx.lifecycle:lifecycle-viewmodel-ktx:2.6.0'
```

En utilisant un ViewModel, tu peux éviter les pertes de données lors des changements de configuration et organiser ton code de manière plus claire en séparant la logique métier des composants de l'interface utilisateur.

Exemple de ViewModel

```
class SimpleViewModel : ViewModel() {

    val text = MutableLiveData<String>()

    init {

        text.value = "Bonjour, ViewModel !"

    }

    fun updateText(newText: String) {

        text.value = newText

    }

}
```

```
class MainActivity : AppCompatActivity() {

private val viewModel: SimpleViewModel by viewModels()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        viewModel.text.observe(this, Observer { newText ->

            textView.text = newText
        })

      button.setOnClickListener {
            viewModel.updateText("Le texte a changé !")
        }
    }
}
```

Afficher une image locale avec ImageView :

Pour afficher une image à partir des ressources locales (comme le dossier res/drawable), tu peux utiliser un **ImageView**.

```
val imageView: ImageView = findViewById(R.id.imageView)
imageView.setImageResource(R.drawable.your_image)
```

```
<ImageView
    android:id="@+id/imageView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/your_image" />
```

Charger une image depuis une URL avec Glide

Pour charger des images à partir d'une URL (par exemple depuis une API ou un site web), il est recommandé d'utiliser une bibliothèque comme **Glide**.

Glide est plus moderne et optimisé pour la gestion d'images en mémoire.

```
dependencies {
  implementation 'com.github.bumptech.glide:glide:4.16.0'
}
```

```
val imageView: ImageView = findViewById(R.id.imageView)
val url = "https://example.com/image.jpg"
Glide.with(this) .load(url) .into(imageView)
```

Afficher une image depuis le stockage de l'appareil

Si tu veux afficher une image stockée sur l'appareil, tu peux utiliser le chemin du fichier.

```
val imageView: ImageView = findViewById(R.id.imageView)
val imgFile = File("/storage/emulated/0/Download/sample_image.jpg")

if (imgFile.exists()) {
    val bitmap = BitmapFactory.decodeFile(imgFile.absolutePath)
    imageView.setImageBitmap(bitmap)
}
```

TP 3

BroadcastReceiver

un BroadcastReceiver est un composant permettant de recevoir et de réagir à des messages émis par le système ou par d'autres applications sous forme de "diffusions" (broadcasts). Les BroadcastReceiver sont souvent utilisés pour surveiller des événements système, tels que la connexion ou la déconnexion d'un réseau, le niveau de la batterie, ou l'installation et la suppression d'applications.

Création d’un BroadcastReceiver

Pour créer un BroadcastReceiver, il faut :

1. **Définir une classe qui hérite de BroadcastReceiver**.
2. **Surcharger la méthode onReceive** pour traiter l'action spécifique lorsque le message est reçu.

Exemples d’utilisations courantes

Les BroadcastReceiver sont utilisés pour :

- Réagir aux changements de réseau (connexion, déconnexion).
- Détecter la fin du chargement de l'appareil.
- Surveiller le niveau de batterie.
- Suivre l'état du Wi-Fi ou du Bluetooth.

Exemple

```
class NetworkChangeReceiver : BroadcastReceiver() {

    override fun onReceive(context: Context, intent: Intent) {

        val connectivityManager = context.getSystemService(Context.CONNECTIVITY_SERVICE) as
ConnectivityManager

        val networkInfo: NetworkInfo? = connectivityManager.activeNetworkInfo

            if (networkInfo != null && networkInfo.isConnected) {

            Toast.makeText(context, "Connected to the internet", Toast.LENGTH_SHORT).show()

        } else {

            Toast.makeText(context, "Disconnected from the internet", Toast.LENGTH_SHORT).show()

         } }

}
```

Enregistrer le BroadcastReceiver

Il y a deux façons d'enregistrer un BroadcastReceiver :

1. **Déclaration statique** dans le fichier **AndroidManifest.xml** : idéal pour les événements système globaux comme le démarrage de l'appareil.

```
<receiver android:name=".NetworkChangeReceiver">
    <intent-filter>
        <action
android:name="android.net.conn.CONNECTIVITY_CHANGE"/>
    </intent-filter>
</receiver>
```

2. **Enregistrement dynamique** dans le code, par exemple dans une `Activity` ou un `Service` :

```
val networkChangeReceiver = NetworkChangeReceiver()
val intentFilter = IntentFilter("android.net.conn.CONNECTIVITY_CHANGE")
registerReceiver(networkChangeReceiver, intentFilter)
```

Pour **désenregistrer** un BroadcastReceiver enregistré dynamiquement, utilisez :

```
unregisterReceiver(networkChangeReceiver)
```

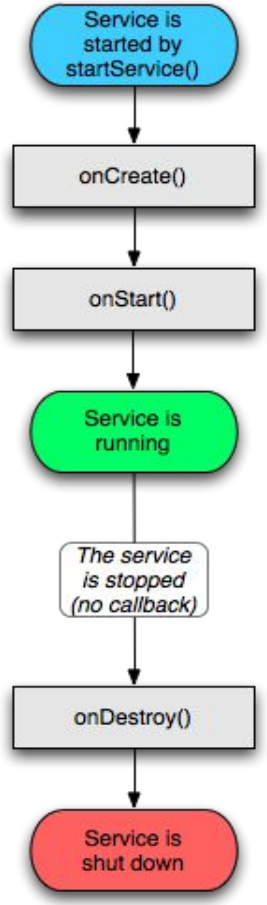
Déclaration au fichier Manifest

il faut obligatoirement déclarer le Broadcast dans le fichier Manifest :

```
<receiver android:name=".NetworkChangeReceiver" />
```

Service

un **Service** est un composant qui permet d'exécuter des tâches en arrière-plan sans interface utilisateur. Les services sont utiles pour les opérations de longue durée, comme la lecture de musique, les téléchargements, ou la synchronisation de données, même si l'application n'est pas visible.



Cycle de vie de Service

Types de Services en Android

Started Service : Un service est "starté" lorsque l'application l'appelle avec startService(). Une fois démarré, il fonctionne indépendamment et continue même si l'utilisateur quitte l'application.

Bound Service : Un service est "lié" lorsque l'application appelle bindService(). Les composants peuvent interagir avec ce service, envoyer des demandes et recevoir des réponses. Il est utile pour les tâches nécessitant des interactions continues.

Foreground Service : Un service en premier plan est visible pour l'utilisateur, souvent utilisé pour des tâches critiques, comme le GPS ou le streaming audio. Un service en premier plan doit afficher une notification pour indiquer qu'il est actif.

Exemples d'Utilisation des Services

- Lecture de musique en arrière-plan
- Synchronisation des données avec un serveur distant
- Téléchargements de fichiers
- Tracking GPS pour des applications de navigation

Implémentation d'un Service

Pour créer un service, on étend la classe Service et on surcharge les méthodes **onCreate()**, **onStartCommand()**, et **onDestroy()**.

Implémentation de Service

```
class MyService : Service() {
    override fun onBind(intent: Intent?): IBinder? {
        return null // Pour un service démarré et non lié }
    override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {
        Toast.makeText(this, "Service Started", Toast.LENGTH_SHORT).show()
        return START_STICKY // Redémarre automatiquement le service si le système le tue }
    override fun onDestroy() {
        super.onDestroy()
        Toast.makeText(this, "Service Destroyed", Toast.LENGTH_SHORT).show() }
}
```

Lancer et Arrêter un Service

Pour lancer le service, on utilise startService() :

```
val intent = Intent(this, MyService::class.java)
startService(intent)
```

Pour arrêter le service, on utilise stopService() :

```
stopService(intent)
```

Avant d'utiliser un Service

Les services doivent être déclarés dans le fichier AndroidManifest.xml.

Les Foreground Services nécessitent des autorisations supplémentaires, surtout sur les versions récentes d'Android.

Déclaration au fichier Manifest

il faut obligatoirement déclarer le service dans le fichier Manifest :

```
<service android:name=".MyService" />
```

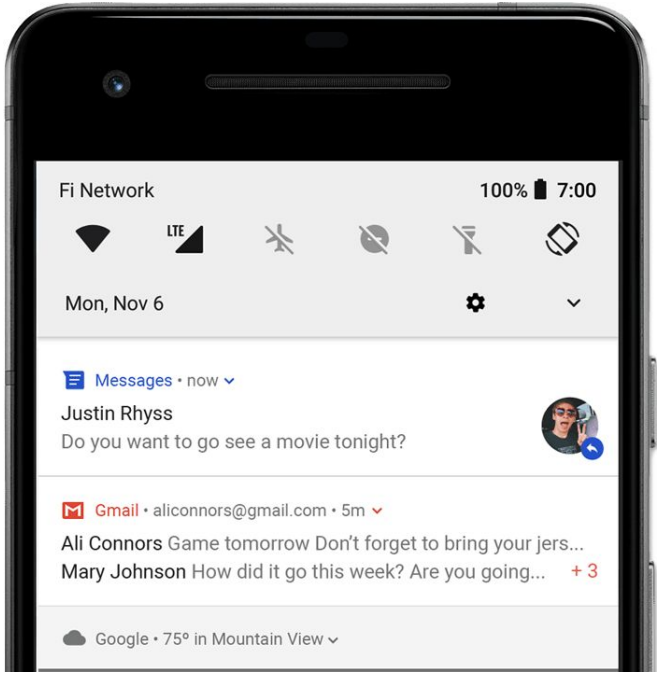
Notification

une **notification** est un message affiché à l'utilisateur en dehors de l'interface de l'application pour l'informer d'événements importants. Elle apparaît généralement sous forme de message dans la barre d'état (ou notification bar) en haut de l'écran.

Les notifications permettent de garder l'utilisateur engagé en lui fournissant des informations même lorsque l'application est en **arrière-plan** ou **fermée**.

Les notifications peuvent inclure des éléments comme :

- Un titre,
- Un texte de message,
- Une icône,
- Des actions spécifiques (par exemple, ouvrir l'application, répondre à un message).



Créer un Canal de Notification

Depuis Android 8.0 (API 26), les notifications nécessitent un canal de notification pour être affichées.

```
fun createNotificationChannel(context: Context) {
    if (Build.VERSION.SDK_INT >=
Build.VERSION_CODES.O) {
        val channelId = "your_channel_id"
        val channelName = "Your Channel Name"
        val importance =
NotificationManager.IMPORTANCE_DEFAULT
        val channel = NotificationChannel(channelId,
channelName, importance).apply {
            description = "Description of your channel"
        }

    val notificationManager: NotificationManager =

context.getSystemService(Context.NOTIFICATION_SERVIC
E) as NotificationManager
        notificationManager.createNotificationChannel(channel)
    }
}
```

Créer et Afficher la Notification

Une fois le canal créé, vous pouvez créer et afficher une notification.

```
fun showNotification(context: Context) {
    val channelId = "your_channel_id"
    val intent = Intent(context, YourActivity::class.java)
    val pendingIntent =
PendingIntent.getActivity(context, 0, intent,
PendingIntent.FLAG_IMMUTABLE)

    val notification =
NotificationCompat.Builder(context, channelId)
        .setSmallIcon(R.drawable.ic_notification_icon)
        .setContentTitle("Titre de la notification")
        .setContentText("Le contenu de notification.")

.setPriority(NotificationCompat.PRIORITY_DEFAULT
)  // Priorité de la notification
        .setContentIntent(pendingIntent)
        .setAutoCancel(true).build()

    with(NotificationManagerCompat.from(context)) {
        notify(1, notification)      }
}
```

Appel de notification

L'utilisation d'une notification dans une Activity

```
class MainActivity : AppCompatActivity() {

 override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        createNotificationChannel(this)

        // Afficher une notification
        showNotification(this)
    }
}
```

Dans Manifest, il faut ajouter la permission :

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

TP 5

C.1 – Manipuler les données sous Android

- # Manipulation des fichiers

La manipulation des fichiers dans Android se fait principalement via les API fournies par Android Storage.

1. Stockage interne (Internal Storage)
2. Stockage externe (External Storage)
3. Stockage avec MediaStore (Scoped Storage - Android 10+)
4. Stockage avec Jetpack DataStore (Remplaçant de SharedPreferences)

1. Stockage interne (Internal Storage)

Les fichiers sont stockés dans l'espace privé de l'application.

Ils sont supprimés lorsque l'application est désinstallée.

Écrire dans un fichier intern

```
val filename = "mon_fichier.txt"
val fileContents = "Hello Android!"
openFileOutput(filename,
Context.MODE_PRIVATE).use { output ->
    output.write(fileContents.toByteArray())
}
```

Lire un fichier interne

```
val filename = "mon_fichier.txt"
val fileContent =
openFileInput(filename).bufferedReader().use {
it.readText() }
Log.d("FileContent", fileContent)
```

2. Stockage externe (External Storage)

- Utilisé pour stocker des fichiers accessibles par d'autres applications.
- Peut être privé (dans un dossier réservé à l'application) ou public (visible par d'autres apps).
- Depuis Android 10 (API 29), l'accès au stockage est restreint (Scoped Storage).

Écrire dans le stockage externe privé

```
val file = File(getExternalFilesDir(null),
"mon_fichier_externe.txt")
file.writeText("Contenu du fichier externe")
```

Lire depuis le stockage externe privé

```
val file = File(getExternalFilesDir(null),
"mon_fichier_externe.txt")
if (file.exists()) {
    val content = file.readText()
    Log.d("FileContent", content) }
```

Permissions nécessaires

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

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
    requestPermissions(arrayOf(Manifest.permission.WRITE_EXTERNAL_STORAGE), 1)
}
```

3. Stockage avec MediaStore

Depuis Android 10, l’accès direct au stockage externe est restreint. Il faut utiliser MediaStore pour gérer les fichiers multimédias.

Sauvegarder une image dans Pictures/

```
val contentValues = ContentValues().apply {
    put(MediaStore.Images.Media.DISPLAY_NAME, "image_test.jpg")
    put(MediaStore.Images.Media.MIME_TYPE, "image/jpeg")
    put(MediaStore.Images.Media.RELATIVE_PATH, Environment.DIRECTORY_PICTURES) }

val uri = contentResolver.insert(MediaStore.Images.Media.EXTERNAL_CONTENT_URI, contentValues)
uri?.let { outputStream ->
    contentResolver.openOutputStream(it)?.use { output ->
        val bitmap = BitmapFactory.decodeResource(resources, R.drawable.mon_image)
        bitmap.compress(Bitmap.CompressFormat.JPEG, 100, output) }
}
```

3. Stockage avec MediaStore

Lire une image depuis Pictures/

```
val projection = arrayOf(MediaStore.Images.Media._ID, MediaStore.Images.Media.DISPLAY_NAME)
val cursor = contentResolver.query(
    MediaStore.Images.Media.EXTERNAL_CONTENT_URI, projection, null, null, null )

cursor?.use {
    val idColumn = it.getColumnIndexOrThrow(MediaStore.Images.Media._ID)
    val nameColumn = it.getColumnIndexOrThrow(MediaStore.Images.Media.DISPLAY_NAME)

    while (it.moveToNext()) {
        val id = it.getLong(idColumn)
        val name = it.getString(nameColumn)
        Log.d("Image", "ID: $id, Name: $name")
    }
}
```

4. Stockage avec Jetpack DataStore

Si vous stockez des petites données structurées, utilisez DataStore (recommandé par Google).

Créer et écrire dans un DataStore

```
val Context.dataStore: DataStore<Preferences>
by preferencesDataStore(name = "settings")

suspend fun saveSetting(context: Context, key:
String, value: String) {
    val dataStoreKey = stringPreferencesKey(key)
    context.dataStore.edit { settings ->
        settings[dataStoreKey] = value
    }
}
```

Lire depuis un DataStore

```
val dataStoreKey =
stringPreferencesKey("username")
val usernameFlow: Flow<String?> =
context.dataStore.data
    .map { preferences -> preferences[dataStoreKey] }

lifecycleScope.launch {
    usernameFlow.collect { username ->
        Log.d("DataStore", "Username: $username")
    }
}
```

Recommandation pour la manipulation des fichiers

Préférer Internal Storage pour des fichiers sensibles.

Utiliser MediaStore pour interagir avec le stockage externe.

Adopter DataStore pour stocker des préférences utilisateur.

Toujours vérifier les permissions et demander les accès nécessaires uniquement si besoin.

TP : 4

C.1 – Utilisation des SharePreferences

Utilisation des SharePreferences

SharedPreferences est une solution simple pour stocker de petites données de manière persistante, comme des préférences utilisateur, des flags, ou des jetons. C’est une sorte de fichier clé-valeur.

Utilisation des SharePreferences

Types de données que tu peux stocker

- **String**
- **Int**
- **Boolean**
- **Float**
- **Long**

Utilisation des SharePreferences

1. Créer/accéder à un fichier SharedPreferences

```
val sharedPref = getSharedPreferences("mon_pref", Context.MODE_PRIVATE)
```

Context.MODE_PRIVATE signifie que le fichier est accessible **seulement** par ton appli.

Utilisation des SharePreferences

1. Créer/accéder à un fichier SharedPreferences

Les modes de `SharedPreferences`

Mode	Description
Context.MODE_PRIVATE	(Valeur par défaut) Le fichier est **privé** à ton application.
Context.MODE_WORLD_READABLE	Permettait à d'autres applis de **lire** le fichier. **Déprécié**.
Context.MODE_WORLD_WRITEABLE	Permettait à d'autres applis de **modifier** le fichier. **Déprécié**.
Context.MODE_MULTI_PROCESS	Permet l'accès multi-processus (Android < 6.0). Pas fiable. Déconseillé.

Utilisation des SharePreferences

2. Enregistrer une valeur

```
val sharedPref = getSharedPreferences("mon_pref", Context.MODE_PRIVATE)
with (sharedPref.edit()) {
    putString("cle_nom", "Alex")
    putInt("cle_age", 28)
    apply() // ou commit()
}
```

apply() est asynchrone (recommandé).

commit() est synchrone (renvoie un booléen de succès).

Utilisation des SharePreferences

3. Lire une valeur

```
val sharedPref = getSharedPreferences("mon_pref", Context.MODE_PRIVATE)
val nom = sharedPref.getString("cle_nom", "Nom inconnu")
val age = sharedPref.getInt("cle_age", 0)
```

Utilisation des SharePreferences

4. Supprimer une valeur ou tout vider

```
// Supprimer une clé
sharedPref.edit().remove("cle_nom").apply()

// Vider toutes les données
sharedPref.edit().clear().apply()
```

Utilisation des SharePreferences

5. Classe utilitaire **SharedPrefsManager**

```
object SharedPrefsManager {

    private const val PREF_NAME = "my_app_prefs"

    fun saveToken(context: Context, token: String) {
        val prefs = context.getSharedPreferences(PREF_NAME, Context.MODE_PRIVATE)
        prefs.edit().putString("auth_token", token).apply()
    }
    fun getToken(context: Context): String? {
        val prefs = context.getSharedPreferences(PREF_NAME, Context.MODE_PRIVATE)
        return prefs.getString("auth_token", null)
    }
    fun clearToken(context: Context) {
        val prefs = context.getSharedPreferences(PREF_NAME, Context.MODE_PRIVATE)
        prefs.edit().remove("auth_token").apply()
    }
}
```

Utilisation des SharePreferences

SharePreferences est Idéal pour stocker :

- Nom de l’utilisateur
- Jeton de session (Token)
- Préférences (dark mode, langue, etc.)

Pas fait pour stocker des listes complexes ou des objets volumineux (utiliser Room ou une DB).

TP : 5

EncryptedSharedPreferences

EncryptedSharedPreferences est une solution sécurisée fournie par Android pour stocker des données sensibles de manière chiffrée dans les préférences partagées (SharedPreferences). Elle utilise le chiffrement AES pour protéger les clés et les valeurs.

Avantages :

- Chiffrement automatique des clés et valeurs
- Prise en charge native par Android Jetpack Security
- Simple à implémenter

EncryptedSharedPreferences

Dépendance à ajouter

Dans ton fichier build.gradle (niveau module) :

```
implementation "androidx.security:security-crypto:1.1.0-alpha06"
```

Exemple

```
fun getSecurePrefs(context: Context): SharedPreferences {
    val masterKey = MasterKey.Builder(context)
        .setKeyScheme(MasterKey.KeyScheme.AES256_GCM) .build()
   return EncryptedSharedPreferences.create( context,
        "secure_prefs_file",
        masterKey,
        EncryptedSharedPreferences.PrefKeyEncryptionScheme.AES256_SIV,
        EncryptedSharedPreferences.PrefValueEncryptionScheme.AES256_GCM
    )
}
```

Recommandations

- Utilise EncryptedSharedPreferences uniquement pour **de petites données sensibles** (tokens, mots de passe, etc.)

- Pour les **fichiers ou bases de données sensibles**, regarde du côté de SQLCipher ou chiffrement manuel

C’est quoi SQLite ?

SQLite est une base de données locale embarquée. Elle est légère, ne nécessite pas de serveur, et est parfaite pour stocker des données sur l’appareil (comme des contacts, des notes, ou un historique).

Mais écrire des requêtes SQL brutes peut vite devenir pénible. C’est là que Room entre en jeu.

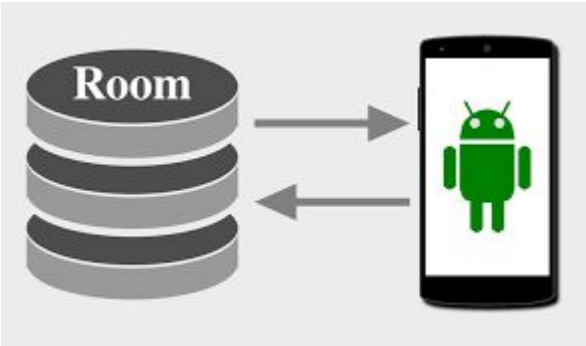
Pourquoi utiliser Room ?

Room est une abstraction de SQLite.

Il utilise des annotations Kotlin pour gérer la base, ce qui évite les erreurs SQL.

Il est compatible avec les coroutines (asynchrone).

Il intègre facilement la vérification à la compilation des requêtes SQL.



Étapes pour utiliser Room dans un projet Kotlin Android

Étape 1 : Ajouter les dépendances

Ouvre le fichier build.gradle (Module: app) et ajoute :

```
def room_version = "2.6.1"

implementation "androidx.room:room-runtime:$room_version"
kapt "androidx.room:room-compiler:$room_version"
implementation "androidx.room:room-ktx:$room_version" // pour les coroutines
```

```
apply plugin: 'kotlin-kapt'
```

Étapes pour utiliser Room dans un projet Kotlin Android

Étape 2 : Créer une Entity (une table)

```
@Entity(tableName = "users")
data class User(
    @PrimaryKey(autoGenerate = true)
    val id: Int = 0,
    val name: String,
    val age: Int
)
```

- @Entity indique que cette classe représente une table.
- tableName = "users" donne le nom de la table.
- @PrimaryKey(autoGenerate = true) : la clé primaire s'incrémente automatiquement.

Étapes pour utiliser Room dans un projet Kotlin Android

Étape 3 : Créer le DAO (Data Access Object)

```
@Dao
interface UserDao {

    @Insert
    suspend fun insert(user: User)

    @Query("SELECT * FROM users")
    suspend fun getAllUsers(): List<User>

    @Update
    suspend fun updateUser(user: User)

    @Delete
    suspend fun deleteUser(user: User)
}
```

@Dao : désigne une interface d'accès à la base.

@Insert, @Update, @Delete sont des **annotations** qui génèrent automatiquement le SQL correspondant.

@Query(...) : permet d'écrire ses propres requêtes SQL.

Étapes pour utiliser Room dans un projet Kotlin Android

Étape 4 : Créer la base de données

```
import androidx.room.Database
import androidx.room.RoomDatabase

@Database(entities = [User::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
}
```

- `@Database` indique que c'est la classe de base Room.
- `entities = [...]` liste les tables.
- `version = 1` est la version de la base (utilisé lors des migrations).

Étapes pour utiliser Room dans un projet Kotlin Android

Étape 5 : Initialiser Room dans l'application

- Singleton de la base de données

```
@Database(entities = [User::class], version = 1)
abstract class AppDatabase : RoomDatabase() {

    abstract fun userDao(): UserDao

    companion object {
        @Volatile
        private var INSTANCE: AppDatabase? = null
```

```
fun getInstance(context: Context): AppDatabase {
            return INSTANCE ?: synchronized(this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    AppDatabase::class.java,
                    "user_database"
                ).build()
                INSTANCE = instance
                instance
            }
        }
    }
}
```

@Volatile + synchronized(this) : pour éviter les bugs en multithreading (création double).

Étapes pour utiliser Room dans un projet Kotlin Android

Création du Repository

Le **Repository** agit comme **intermédiaire** entre la **base de données (Room DAO)** et le **ViewModel** (ou tout autre composant de l'application).

```
class UserRepository(private val userDao: UserDao) {

    suspend fun insertUser(user: User) {
        userDao.insert(user)
    }

    suspend fun getAllUsers(): List<User> {
        return userDao.getAllUsers()
    }

    suspend fun updateUser(user: User) {
        userDao.updateUser(user)
    }

    suspend fun deleteUser(user: User) {
        userDao.deleteUser(user)
    }
}
```

Exemple d'utilisation dans un ViewModel

```
class UserViewModel(application: Application) :
AndroidViewModel(application) {

 private val db = AppDatabase.getInstance(application)

  private val repository = UserRepository(db.userDao())

    fun addUser(user: User) {

        viewModelScope.launch {

            repository.insertUser(user)

        }

    }
```

```
fun fetchUsers(onResult: (List<User>) -> Unit)
{

        viewModelScope.launch {

            val users = repository.getAllUsers()

            onResult(users)

        }

    }

}
```

Utilisation dans une MainActivity

```
class MainActivity : AppCompatActivity() {

    private lateinit var viewModel: UserViewModel

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        viewModel = ViewModelProvider(this)[UserViewModel::class.java]

        val user = User(name = "Laura", age = 30)

        viewModel.addUser(user)

        viewModel.fetchUsers { users ->
            users.forEach {
                Log.d("User", "Nom: ${it.name}, Âge: ${it.age}")
            }
        }
    } }
```

Résumé architecture

1. AppDatabase (singleton)
2. UserDao (interface)
3. UserRepository (logique métier)
4. UserViewModel (lié à l'UI)
5. Activity / Fragment

Étapes de migration Room

1. Modifier la version dans AppDatabase

```
@Database(entities = [User::class], version = 2)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
}
```

Étapes de migration Room

2. Créer la stratégie de migration

```
val MIGRATION_1_2 = object : Migration(1, 2) {
    override fun migrate(database: SupportSQLiteDatabase) {
        // Exécuter une requête SQL pour ajouter la colonne
        database.execSQL("ALTER TABLE users ADD COLUMN email TEXT NOT NULL DEFAULT """)
    }
}
```

Étapes de migration Room

3. Appliquer la migration au moment de la création de la base

```
val db = Room.databaseBuilder(
    context,
    AppDatabase::class.java,
    "my-database"
).addMigrations(MIGRATION_1_2)
 .build()
```

Astuces migration

Cas	Solution
Ajouter une colonne	ALTER TABLE
Supprimer une colonne/table	Room ne supporte pas directement → il faut créer une **nouvelle table**, copier les données, supprimer l'ancienne
Renommer une table	Même chose, créer + copier + supprimer
Migrer plusieurs versions	Tu peux chaîner plusieurs .addMigrations(MIGRATION_1_2, MIGRATION_2_3)

TP

C.2 – Partager des données entre applications

Protection contre les entrées malveillantes

Protéger une application Android Kotlin contre les entrées malveillantes est fondamental pour éviter des failles comme :

- L'injection SQL,
- Les crashs,
- La prise de contrôle des flux,
- Des attaques XSS si tu affiches du contenu Web.

Protection contre les entrées malveillantes

1. **Validation systématique des entrées utilisateur**

Toujours vérifier et filtrer ce que l'utilisateur saisit, avant de l'utiliser.

- **Longueur minimale/maximale** (ex : nom entre 2 et 50 caractères)
- **Caractères autorisés** (ex : uniquement lettres et chiffres pour un pseudo)
- **Format spécifique** (ex : email, téléphone)

```
// Exemple de vérification d'un email
fun isValidEmail(email: String): Boolean {
    return android.util.Patterns.EMAIL_ADDRESS.matcher(email).matches()
}

// Exemple de vérification d'un champ de mot de passe
fun isValidPassword(password: String): Boolean {
    return password.length in 8..20 && password.none { it.isWhitespace() }
}
```

Protection contre les entrées malveillantes

2. **Utiliser Room avec des requêtes paramétrées**

Ne JAMAIS concaténer des données utilisateur directement dans une requête SQL.

Avec Room, tu es déjà protégé si tu utilises **@Query** avec des **paramètres** :

```
@Query("SELECT * FROM users WHERE email = :email")
fun findUserByEmail(email: String): User?
```

:email est **sécurisé automatiquement** par Room.

Protection contre les entrées malveillantes

3. **Échapper les entrées dans WebView**

Si tu affiches du texte utilisateur dans une **WebView**, utilise **Html.escapeHtml()** :

```
val safeHtml = Html.escapeHtml(userInput)

webView.loadData(safeHtml, "text/html", "UTF-8")
```

Sinon, un utilisateur pourrait injecter du JavaScript malicieux (attaque XSS).

Protection contre les entrées malveillantes

4. **Limiter les permissions et sécuriser les API**

Si tu récupères ou envoies des données vers un serveur, utilise HTTPS et authentifie bien l’utilisateur.

Ne fais jamais confiance aux données reçues, même du serveur.

Protection contre les entrées malveillantes

5. Gérer les Exceptions proprement

Ne jamais laisser une application planter sans contrôle :

```
try {
    val number = userInput.toInt()
} catch (e: NumberFormatException) {
    showToast("Veuillez entrer un nombre valide.")
}
```

Sinon, un mauvais input peut faire crasher ton app.

Protection contre les entrées malveillantes

6. Limiter l'accès aux composants Android

Dans ton **AndroidManifest.xml**, évite d'exposer des **Activity**, **Service**, **BroadcastReceiver**, etc., sans besoin.

```
<activity
    android:name=".ui.LoginActivity"
    android:exported="false" />
```

Protection contre les entrées malveillantes

Risque	**Protection Kotlin/Android**
SQL Injection	Room avec paramètres
XSS WebView	**Html.escapeHtml()**
Crash par mauvais input	Validation et **try-catch**
Vol de données via API	HTTPS + validation
Accès non autorisé à composants	**android:exported="false"**

TP 7

Utilisation du ContentProvider

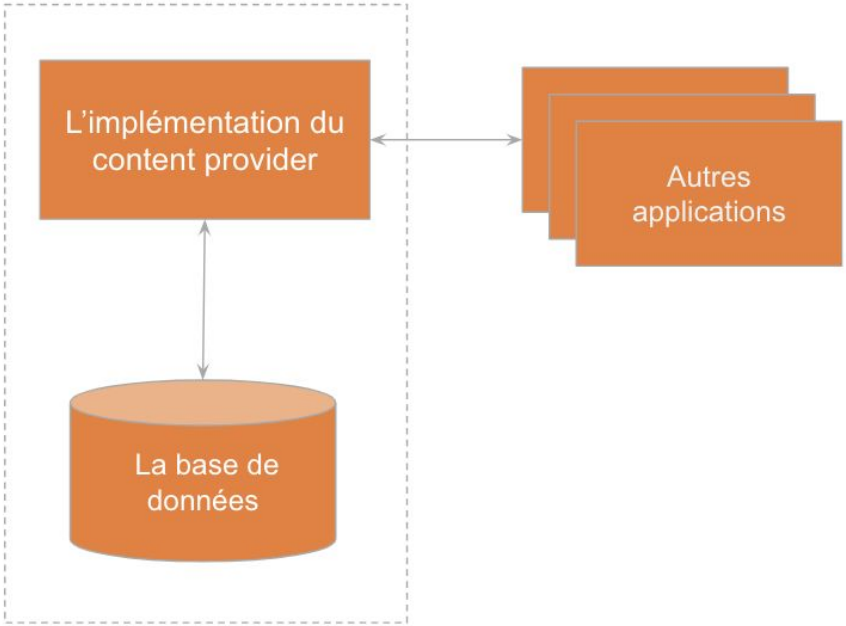
Un **ContentProvider** est une classe Android spéciale qui permet d'accéder à des données (locales ou distantes) via une interface standardisée.

Il sert principalement :

- Partager des données entre applications,
- Structurer l'accès aux données internes de ton app

ContentProvider du système Android :

- Contacts (ContactsContract)
- Images (MediaStore)
- SMS (Telephony.Sms)



Utilisation du ContentProvider

Structure d'un ContentProvider :

Un ContentProvider expose **des données** sous forme d'URI (ex : content://com.exemple.app.provider/table_name).

Un ContentProvider **implémente** généralement 6 fonctions principales :

Fonction	Rôle
query()	Lire des données
insert()	Insérer des données
update()	Modifier des données
delete()	Supprimer des données
getType()	Retourner le type MIME d'une URI
onCreate()	Initialiser les ressources

Utilisation du ContentProvider

```
class UserProvider : ContentProvider() {
    override fun onCreate(): Boolean {
        return true
    }
    override fun query(
        uri: Uri, projection: Array<out String>?, selection: String?,
        selectionArgs: Array<out String>?, sortOrder: String?
    ): Cursor? {
        return null  }
    override fun insert(uri: Uri, values: ContentValues?): Uri? {
        return null }
```

```
    override fun update(uri: Uri, values:
ContentValues?, selection: String?, selectionArgs:
Array<out String>?): Int {
        return 0 }
override fun delete(uri: Uri, selection: String?,
selectionArgs: Array<out String>?): Int {
        return 0 }
    override fun getType(uri: Uri): String? {
        return null  }
}
```

Déclarer dans AndroidManifest.xml :

```
<provider
    android:name=".UserProvider"
android:authorities="com.exemple.app.provider"
    android:exported="false" />
```

Utilisation du ContentProvider

Sécurité dans un ContentProvider

Par défaut :

- Si android:exported="true", **toutes les applications** peuvent interagir avec ton provider.
- Si android:exported="false", **seule votre app peut** y accéder.

Problèmes classiques à éviter :

Danger	Solution
Fuite de données	Ne jamais exposer inutilement exported=true
Injection SQL via selection ou selectionArgs	Toujours utiliser selectionArgs pour éviter l'injection
Mauvais contrôle d'accès	Ajouter des vérifications manuelles dans ton provider (ex : check d'authentification)

Utilisation du ContentProvider

Si ton ContentProvider est uniquement interne à ton application, utilise plutôt :

- **Room**
- ou une simple **SQLiteOpenHelper**

Le ContentProvider est surtout **utile** **quand il y a des échanges entre applications**.

Élément	À retenir
Quand l'utiliser ?	Partage de données entre apps ou accès structuré interne
Attention à	exported + contrôle d'accès + protection contre injection
Sécuriser	selectionArgs, permissions, validation dans le code
URI	Point d'entrée (content://authority/path)

Utilisation du ContentProvider

On ajoute un **ContentProvider** dans le projet de manipulation d'API Rest pour partager les données avec une autre application.

```
override fun onCreate(savedInstanceState: Bundle?)
{
   super.onCreate(savedInstanceState)
   setContentView(R.layout.activity_main)
val url = "products/3"
val call = apiService.getProduct(url)
call.enqueue(object :Callback<JSONObject> {
    override fun onResponse(call:

Call<JSONObject>, response:Response<JSONObject>){
        if (response.isSuccessful) {
            val product = response.body()
            handleProduct(product)
        }
    }
    override fun onFailure(call: Call<JSONObject>,
t: Throwable) {
        Log.i("TAG", "Error: ${t.message}")
    }
})
```

```
private fun handleProduct(response: JSONObject) {
  val productDAO =
ProductDatabase.getDatabase(application).productDAO
()

  CoroutineScope(Dispatchers.IO).launch {
  val parsedProduct =
Product(response.get("id").toString().toInt()
            response.get("title").toString()
        )
        productDAO.insertProduct(parsedProduct)
        val product = productDAO.getAll()?.get(0)
        val values = ContentValues()

values.put(MyContentProvider.name,
product?.title.toString())  // la valeur à partager
contentResolver.insert(MyContentProvider.CONTENT_UR
I, values)
   }.start()
}
```

Utilisation du ContentProvider

```
class MyContentProvider : ContentProvider() {
    companion object {
        const val PROVIDER_NAME =
"com.example.myapplication.provider"
        const val URL =
"content://$PROVIDER_NAME/products"
        val CONTENT_URI = Uri.parse(URL)
        const val id = "id"
        const val title = "title"
        const val uriCode = 1
        var uriMatcher: UriMatcher? = null
    }

    private lateinit var db: NotesDatabase

    init {
        uriMatcher =
UriMatcher(UriMatcher.NO_MATCH)
        uriMatcher!!.addURI(PROVIDER_NAME,
"products", uriCode)
        uriMatcher!!.addURI(PROVIDER_NAME,
"products/*", uriCode)
    }
```

```
override fun getType(uri: Uri): String? {
        return when (uriMatcher!!.match(uri)) {
            uriCode ->
"vnd.android.cursor.dir/products"
            else -> throw
IllegalArgumentException("Unsupported URI:
$uri")
        }
    }

    override fun onCreate(): Boolean {
        val context = context
        db = Room.databaseBuilder(
            context!!.applicationContext,
            NotesDatabase::class.java,
"products-db"
        ).build()
        return true
    }
```

Utilisation du ContentProvider

```
override fun query(
    uri: Uri, projection: Array<String>?, selection: String?,
    selectionArgs: Array<String>?, sortOrder: String?
): Cursor? {
    var sortOrder = sortOrder

    // Accès à la base de données Room
    val context = context!!
    val products = db.productDao().getAllProducts()
    // Accéder aux produits via Room

    // Créer un Cursor à partir des données de Room
    val matrixCursor = MatrixCursor(arrayOf("id", "name"))

    for (product in products) {
        matrixCursor.addRow(arrayOf(product.id, product.name))
    }

    // Définir l'URI de notification pour l'observateur
    matrixCursor.setNotificationUri(context.contentResolver, uri)

    return matrixCursor
}
```

Utilisation du ContentProvider

```
override fun insert(uri: Uri, values:
ContentValues?): Uri? {
        val name = values?.getAsString("title") ?:
return null
        val product = Product(title = title) //
Création d'un objet Product via Room

        // Insertion via Room
        val rowID =
db.productDao().insertProduct(product)

        if (rowID > 0) {
            val _uri =
ContentUris.withAppendedId(CONTENT_URI, rowID)

context!!.contentResolver.notifyChange(_uri, null)
            return _uri
        }
        throw SQLiteException("Failed to add a
record into $uri")
    }
```

```
override fun update(uri: Uri, values:
ContentValues?, selection: String?,
        selectionArgs: Array<String>?
    ): Int {
 var count = 0
count = when(uriMatcher!!.match(uri)){uriCode -> {
   val product = Product(id =
    selectionArgs?.get(0)?.toInt() ?: 0, title =

     values?.getAsString("title") ?: "")
db.productDao().updateProduct(product)
 1 // On suppose que 1 produit est mis à jour
       }
else -> throw IllegalArgumentException("Unknown
URI $uri")}

context!!.contentResolver.notifyChange(uri, null)
        return count }
```

Utilisation du ContentProvider

```
override fun delete(
        uri: Uri,
        selection: String?,
        selectionArgs: Array<String>?
    ): Int {
        var count = 0
        count = when (uriMatcher!!.match(uri)) { uriCode -> {
                val product = Product(id = selectionArgs?.get(0)?.toInt() ?: 0)
                db.productDao().deleteProduct(product)
                1 // On suppose que 1 produit est supprimé
            }
            else -> throw IllegalArgumentException("Unknown URI $uri")
        }
        context!!.contentResolver.notifyChange(uri, null)
        return count
    }
}
```

Utilisation du ContentProvider

Pour utiliser le ContentProvider il faut l'ajouter dans le fichier AndroidManifest aussi.
Le projet actuel partage la valeur de **title** avec une deuxième application.

```
<provider
    android:name="com.example.myapplication.MyContentProvider"
    android:authorities="com.example.myapplication.provider"
    android:enabled="true"
    android:exported="true"/>
```

```
I/System.out: title : Samsung Universe 9
```

TP 8

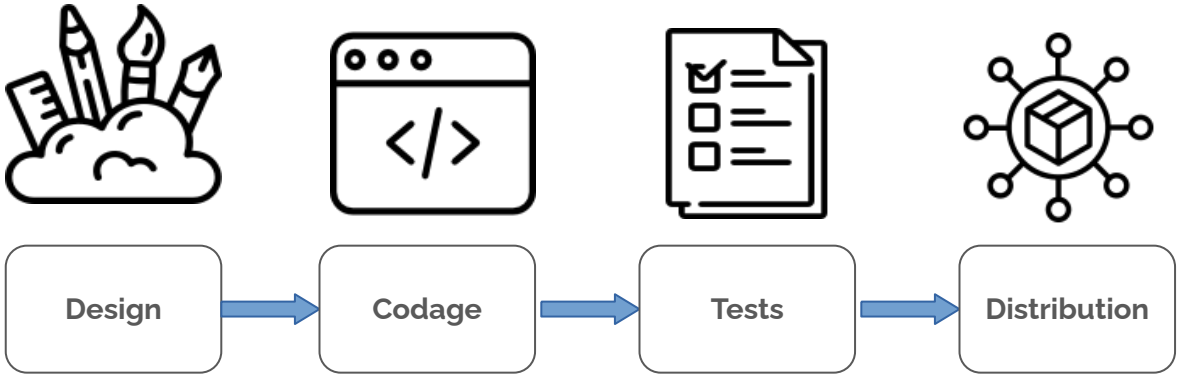
D.1 - Gérer les signatures et versions

Configuration de l’application pour la release

- **Publiez votre application Android sur le PlayStore**

Introduction

- La distribution de l'application est la dernière étape d'une série de processus :



- **Mettre de l'ordre dans votre application.**

  Tests, nettoyage, version, nom, icône, licence, etc.

- **Signature de l’application.**

  Indispensable pour pouvoir la publier.

- **Teste de l’application comme un utilisateur final.**

  Se mettre à la place de l'utilisateur final

- **Publication de l’application.**

  Déployez votre application en ligne.

Mettre de l'ordre dans votre application

- Nettoyez le projet : il faut effacer toutes les traces et les fichiers ou dossiers utilisés pour le débogage et lors des tests.

  Il faut examiner par exemple le contenu des répertoires « **res** » et « **assets** ».

- Arrêtez la journalisation : supprimer tout ce qui est en rapport avec « **Logcat** ». On peut par exemple baliser leurs utilisations comme suit :

```
if(BuildConfig.DEBUG) {
        //si on se trouve en version debug, alors on affiche
        //des messages dans le Logcat
        Log.d(...);
    }
```

- ou utiliser **Timber** : https://github.com/JakeWharton/timber

- **Désactivez le débogage** : soit la retirer du fichier « **AndroidManifest.xml** » ou la mettre à « **false** ». android:debuggable="false"

- **Numéro de la version**

  android:versionName

  - Donne une valeur sous la forme d'une chaîne de caractères à la version de votre application (par exemple « **1.0 alpha** » ou « **2.8.1b** »).
  - Cet attribut est visible à l'utilisateur

  android:versionCode

  - Cet attribut n'est pas visible à l'utilisateur.
  - Il ne peut contenir que des nombres entiers.
  - Si votre ancien numéro était à « **1** », en le mettant à une valeur supérieure à « **1** », le Store va conclure qu'il s'agit d'une version plus récente.

- **Nom du package**

  C'est le nom utilisé pour identifier votre application. Vous ne pouvez pas le changer entre deux versions. Ce nom est unique et permanent. Par ailleurs, il faut qu'il se démarque.
  (com.companyName.appName.flavor)

- **Icône**

  C'est un détail pour vous, oui, mais … c'est le premier contact.

https://material.io/design/iconography/product-icons.html#design-principles

https://developer.android.com/distribute/best-practices/develop/use-material-design?_hl=FR
https://www.usabilis.com/material-design-lui-selon-google/

http://kunzisoft.blogspot.com/2017/02/creer-un-icone-materila-design-android.html

- **Licence d'utilisation (facultatif)**
  À vous de décider le type de licence que vous voulez associer avec votre application.
- **La version ciblée**
  Dans le manifeste, vous devez décider de la version minimale de l'API. Un choix permet soit de restreindre le nombre d'utilisateurs ou de l'élargir.

- **Tester, tester et encore tester**
  Ne pas oublier de faire des tests exhaustifs en tant que développeur afin de valider la robustesse de votre application.
  https://developer.android.com/training/testing/fundamentals

  Faire l'exercice pratique (« codelab ») :

  https://codelabs.developers.google.com/codelabs/android-testing/#0

Signature de l'application

- Les applications dans « **Google Play** » sont représentées par un fichier au format « **apk** ».
- On commence par exporter le projet de l'interface de développement sous la forme d'un fichier « **apk** ».
- Cette procédure fait en sorte que le projet est exporté en mode « **release** » et non pas « **debug** ».
- Durant ce processus, nous sommes ramenés à signer l'application.
- Android exige d'une application qu'elle soit préalablement signée avant d'être installée ou mise à jour sur un appareil.

- **Signer une application permet de la sécuriser :**
  - On garantit ainsi son intégrité.
  - On définit l'auteur de l'application.
  - La mise à jour de l'application ne peut avoir lieu que si elle possède une signature provenant du même certificat.

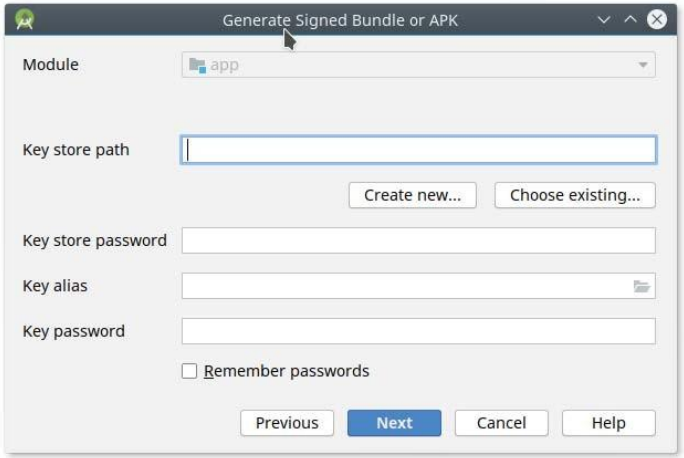
- **Utilisez sa propre clé :**
  - Vous évitez ainsi une signature générique « simple » à trouver.
  - Utilisez la même clé pour signer toutes ses applications. Ces applications vont pouvoir fonctionner dans un même processus (dans « une » seule application) si elles le désirent. Elles peuvent aussi échanger et partager des données de manière sécuritaire.
  - Utilisez un mot de passe abracadabrant.
  - Évitez de perdre votre clé! Sinon impossible de mettre à jour votre application.
  - Évitez aussi de vous la faire voler!

- **Signez l'application en utilisant Android Studio :**

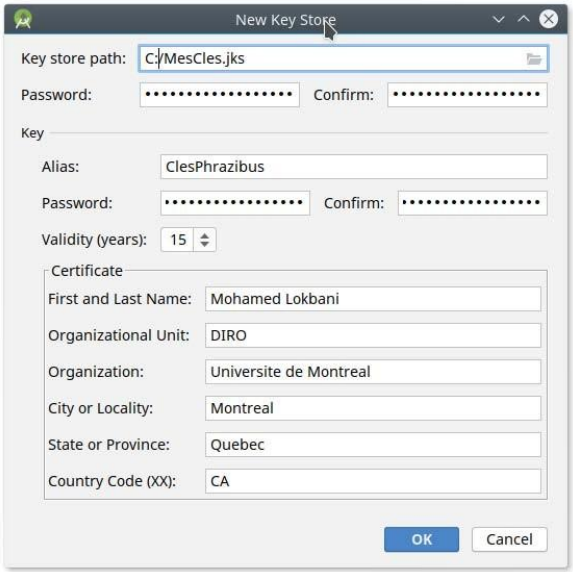
  https://developer.android.com/studio/publish/app-signing

  o Dans le menu, cliquez sur « build » puis « Generate signed Bundle/APK… » , cochez « APK », « Next »

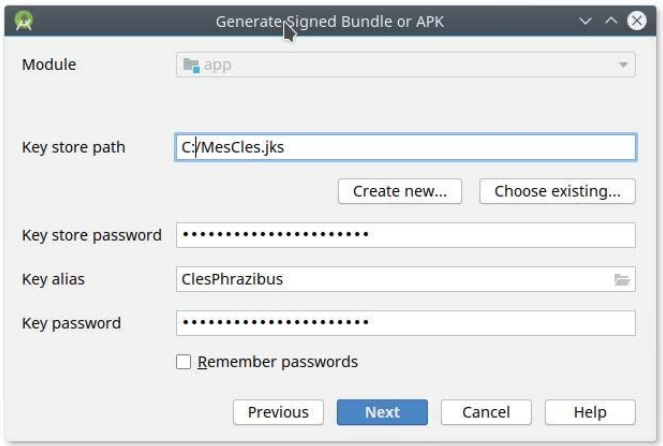
- Nous devons préciser une clé. Pour cela, nous allons commencer par générer un dépôt (« **Key store** ») pour les clés. Cliquez sur « **Create new** » :



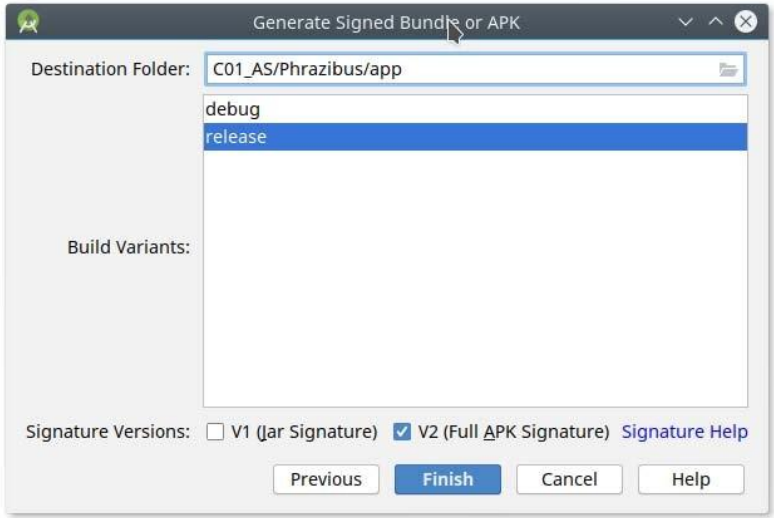
o Complétez les différents champs et cliquez sur « **OK** »



o Vous allez obtenir cette fenêtre complétée, cliquez sur « **Next** » :



- Précisez le répertoire où le fichier « **apk** » sera stocké, la variante « **release** » et la version « **V2** » puis cliquez sur « **Finish** » :



- Si l'opération a réussi, vous allez obtenir ce message :

  Generate Signed APK
  APK(s) generated successfully for 1 module:
  Module 'app': locate or analyze the APK.

- Notez le chemin où le paquetage « APK » sera sauvegardé.

- Il est possible aussi de configurer « Android Studio » pour qu'il signe automatiquement votre application. Voir pour cela les indications décrites sur le lien mentionné au début de cette section.

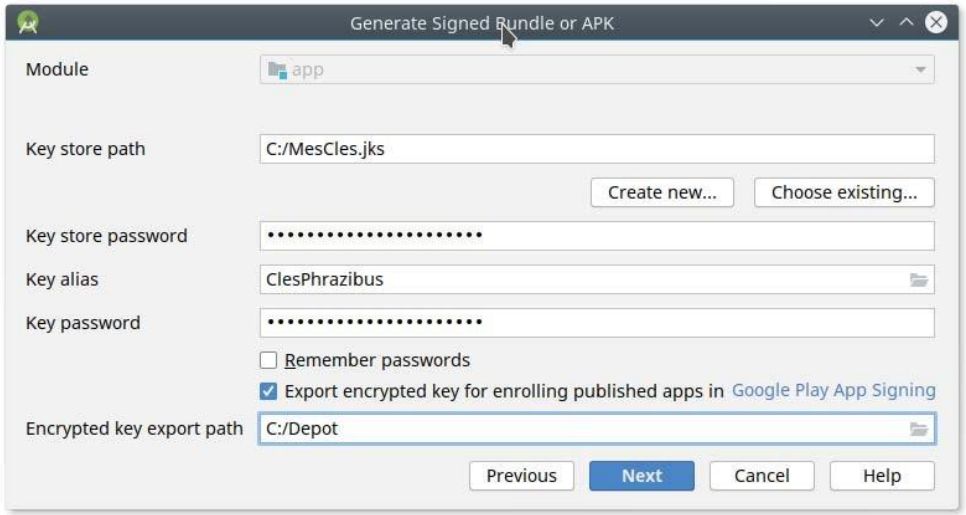
- **Signez un « bundle »**

Avant la version « 3.2 » d'Android Studio, le développeur devait générer une version du fichier « APK » pour les différentes variantes de son application (taille de l'écran, type de l'appareil, etc.). Il pouvait aussi générer un seul fichier « APK » qui permettait de couvrir toutes ces variantes. Or la taille de ce fichier était très large. Ajouter à cela que la création de plusieurs variantes nécessite d'inclure des heures de travail supplémentaires pour réaliser cette tâche. Une étude menée par Google a montré la taille de l'application, a une influence directe sur la décision de l'utilisateur de l'installer ou pas. Pour pallier à tous ces problèmes, Google a proposé une livraison dynamique. Le développeur doit inclure les différentes images pour les différents appareils dans un fichier compressé « bundle ». Au moment de l'installation, Google va générer de manière dynamique à partir de ce « bundle », le fichier « APK » qui correspond aux caractéristiques de l'appareil.

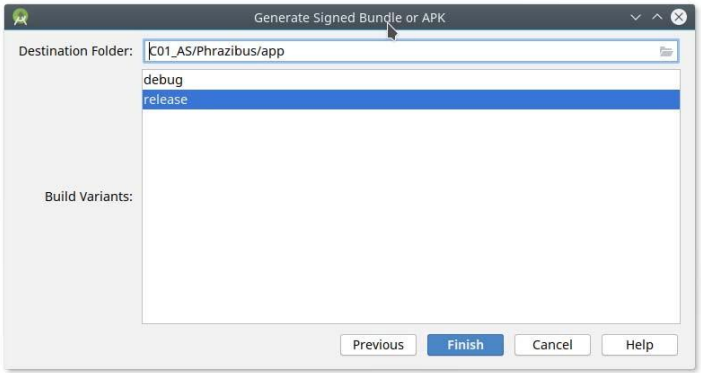
C'est l'approche recommandée par Google.

o Dans le menu, cliquez sur « **build** » puis « **Generate signed Bundle/APK…** » , cochez « **Android App Bundle** », « **Next** » :

o Complétez les différents champs, puis cliquez sur « **Next** »



- Choisissez la destination et la variante, puis cliquez sur « **Finish** ». Attention l'application doit inclure aussi un numéro de version comme mentionné au début de chapitre.



- Si l'opération est un succès, vous allez obtenir cette série de messages :

Generate Signed Bundle

App bundle(s) generated successfully:
Module 'app': locate or analyze the app bundle.
Locate exported key file.

- Signez l'application « **manuellement** »
    - Générez une clé :

keytool -genkey -v -keystore my-release-key.keystore -alias alias_name -keyalg RSA -keysize 2048 -validity 18250

« **my-release-key.keystore** » : votre fichier de clés.
« **validity** » : durée de validité de la clé. 50 ans ~ 18250 jours.

```
signingConfigs {
    releaseconfig {
        keyAlias 'release_key'
        keyPassword 'release_pwd'
        //storeFile file('C:\\Users\\lenovo\\AndroidStudioProjects\\app\\Release.jks')
        storeFile file('/Users/lenovo/StudioProjects/app/Release.jks')

        storePassword 'smssurtymar'
    }
}
```

```
buildTypes {

    release {
        minifyEnabled false
        proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        signingConfig signingConfigs.releaseconfig

        firebaseCrashlytics {
            mappingFileUploadEnabled false
        }
        manifestPlaceholders = [crashlyticsCollectionEnabled:"true"]
    }
```

- Compilez votre projet en mode « **release** ».
- On obtient ainsi une application en mode « **release** » signée.

Distribuer l'application

- Assurez-vous d'abord d'avoir suivi les recommandations pour déployer l'application.

https://support.google.com/googleplay/android-developer/answer/7159011
Le déploiement peut se faire de plusieurs manières :

http://developer.android.com/distribute/tools/open-distribution.html

- Manuelle : via le web, espace local ou sur le réseau.

- Boutique en ligne : une boutique qui peut héberger des applications gratuites ou payantes, dédiées entre autres à des appareils Android. C'est l'option idéale si l'on veut ratisser large.

Déploiement manuel

- Vous pouvez donner accès à l'application via le web par exemple. Vous n'avez qu'à fournir le lien « URL » vers la page web en question.

  Telecharger App

  - Vous devez activer dans votre appareil l'option « **installation à partir d'une source inconnue** ».

  - Le serveur qui héberge la page doit ajouter ce « MIME » :

    « **application/vnd.android.package-archive** »

Exercice : **Déploiement manuel**

1- il faut créer un projet Android (JAVA ou Kotlin) : avec 2 Activities (MainActivity1 et MainActivity2 qui sont reliées par un Button), ensuite créer un APK

2- Créer une Page HTML (Image, titre et détails) et insérer dedans un lien pour l'APK généré.

3- hébérger la Page HTML pour donner accès à l'application via le web

- Vous pouvez aussi la déposer dans un espace partagé. Dans ce cas, l'utilisateur lambda doit connaître les étapes à suivre pour installer l'application.

    - **adb -s nom_de_appareil install monpaquetage.apk**

- Vous pouviez copier l'application localement sur votre appareil à travers le port USB. Par la suite, vous allez utiliser l'explorateur de fichiers pour vous rendre à l'endroit où le fichier a été sauvegardé. Finalement, il suffit de cliquer sur le fichier pour effectuer l'installation de l'application.

    - Vous devez activer dans votre appareil l'option « **installation à partir d'une source inconnue** ».

Déploiement à travers « Google Play »

- « **Google Play** » store est une place de marché qui a été créée et exploitée par Google qui permet aux Développeurs enregistrés dans certains pays de distribuer des Produits directement aux utilisateurs d'Appareils.

- Créée le 6 mars 2012 de la fusion de « Android Market », « Google Music » et « Google eBookstore ».

- En date du 17 Septembre 2021, la boutique contient 2,705,806 des applications gratuites 96.3 % et 102,850 applications payantes 3.7%.

  http://www.appbrain.com/stats/free-and-paid-android-applications

- La boutique est utilisable par l'intermédiaire d'un compte Google, c'est-à-dire « **Gmail** ».

- Ce compte peut-être associé à une personne physique ou morale.

- La personne gère les accès à la boutique pour tous les appareils dont elle est propriétaire, sans restriction de nombre.

- Ainsi elle peut déployer une application payante ou gratuite acquise sur la boutique pour l'ensemble des appareils dont elle est propriétaire, en ne payant qu'une seule fois l'application (payante).

- Pour rendre disponible une application via la boutique, il faut avoir un compte développeur.

- Un compte développeur est un compte de publication attribué aux développeurs qui permet la distribution de Produits via le Play Store.

- Ce compte requiert des frais d'inscription uniques de 25USD.

- Pour une application payante : 30% des revenus sont pour Google, 70% pour le développeur.

- Le compte développeur nécessite aussi un compte « Gmail ».

https://developer.android.com/distribute/googleplay/start.html

- Avec votre compte « Gmail », commencez par vous connecter à cette adresse :

  https://play.google.com/apps/publish/signup/

- Acceptez les conditions générales associées à la distribution sur « Google Play » pour les développeurs, puis cliquez « Continue to payment ».

- Réglez les frais d'inscription de 25USD.

- L'étape finale consiste à fournir des informations relatives à votre compte (nom du développeur, adresse courriel et numéro de téléphone).

- Google vous redirige par la suite sur la console développeur de « Google Play » :

https://developer.android.com/distribute/googleplay/developer-console.html

- Cette interface va vous permettre d'importer des applications, de définir les prix, d'ajouter des utilisateurs de comptes et gérer les autorisations et finalement, de consulter des rapports, des statistiques et des informations relatives à vos applications.

- L'interface pour ajouter un nouvel utilisateur du compte :

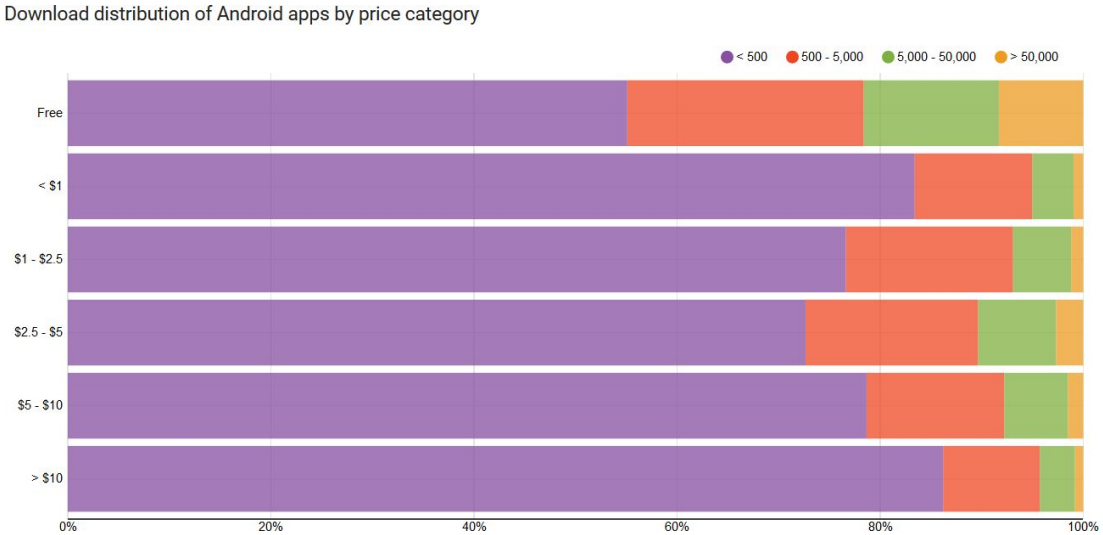
- La publication d'une application nécessite l'envoi du fichier « aab » et de fournir des captures d'écran et des métadonnées relatives à votre application.

- Si une application est proposée gratuitement, elle ne peut pas changer d'état.

- Si une application payante est rendue gratuite, elle ne peut plus changer d'état.

- Une application peut-être :
  - **Payante** : elle est facturée avant qu'elle ne soit téléchargée.
  - **Gratuite** : mais vraiment gratuite sans les « extras » !
  - **Mixte** : offrir une version gratuite basic et une version améliorée payante.
  - **Financée par la Pub** : elle est gratuite, mais vous allez lui inclure de la pub.
  - **Produits intégrés** : elle est gratuite, mais vous allez lui inclure un contenu qui peut-être acheté au fur et à mesure (une arme pour un jeu, un véhicule pour une course de voitures, etc.).
  - **Abonnements** : le contenu nécessite le paiement d'un abonnement périodique. Cette technique est utilisée pour contrer le fait qu'un compte peut servir plusieurs appareils dont il est responsable. L'abonnement proposé est généralement lié à un compte et à un appareil.

- Quel prix fixé?



- 90% des applications dont le prix est supérieur à 10$ ont été téléchargées moins de 500 fois! (http://www.appbrain.com/stats/free-and-paid-android-applications)

- Visibilité

https://support.google.com/googleplay/android-developer/answer/4448378?hl=fr&ref_topic=3450986

« La fonctionnalité de recherche de Google Play tient compte de l'expérience utilisateur générale que votre application procure en se basant sur le comportement et les commentaires des utilisateurs. Les applications sont classées en fonction de plusieurs facteurs tels que les notes, les avis ou le nombre de téléchargements. Bien que le poids et la valeur de chaque facteur soient confidentiels en raison de leur appartenance à l'algorithme de recherche de Google, vous pouvez réaliser les opérations ci-dessous afin d'améliorer la visibilité de votre application :

- Créez une expérience utilisateur durable et enrichissante pour vos utilisateurs.
- Améliorez votre application en y apportant des mises à jour régulières.
- Encouragez vos utilisateurs à laisser un avis et à donner une note à votre application.
- Fournissez un service client de qualité en répondant aux utilisateurs et en résolvant leurs problèmes. »

Déploiement sur d'autres boutiques alternatives

- Noyer dans la masse : 2,5 millions d'applications sur « Google Play »!
- Le marché chinois : il est possible de télécharger de la boutique « Google Play » que les applications gratuites.
- Avoir un autre ratio de partage des gains que (70/30) de « Google Play ».
- Publier ailleurs l'application « rejetée » par « Google Play ».
- Difficile d'ignorer des boutiques alternatives devenues trop grandes.

- L'alternative doit donc permettre de publier aussi des applications payantes, de fournir des statistiques adéquates, de proposer une interface dans une langue couramment utilisée (ou plusieurs langues).

http://en.wikipedia.org/wiki/List_of_mobile_software_distribution_platforms

- 2 exemples :
  - Amazon App Store : il surfe sur la vague du magasin en ligne « Amazon ». Il offre des applications pour Kindle et Android.
  - SlideMe : réseau alternatif #1 à « Google Play ». Il est installé par défaut sur de nombreux terminaux.

Toutes les applications
Boîte de réception 40
Conformité aux règles
Utilisateurs et autorisations
Gestion des commandes
Télécharger des rapports
Détails du compte
Page du développeur
Comptes de développeur associés
Journal d'activité
Configuration

Toutes les applications

Consulter les applications et les jeux auxquels vous avez accès dans votre compte de développeur

Créer une application

Applications épinglées

Épinglez des applications ici pour y accéder rapidement et consulter les statistiques clés

9 applications

Filtrer par

Tout

Faire une recherche par nom d'application...

Application	Audience ayant installé l'application	État de l'application	État de la mise à jour	Dernière mise à jour
Ghazala Horaires & tarifs - غزالة com.stanly.ghazala	10	Application suspendue par Google		27 déc. 2018
My Beautiful Selfie - Filter Cam... com.stanly.selfie	0	Supprimé		27 sept. 2018
My puzzle game com.stanly.puzzle	0	Supprimé par Google		29 déc. 2018

Créer une application

Informations sur l'application

Nom de l'application

Le nom de votre application apparaîtra tel quel sur Google Play. Soyez concis et n'incluez pas de prix, de classement, d'emoji ni de symboles répétitifs. 0/50

Langue par défaut

Anglais (États-Unis) – en-US

Application ou jeu

Vous pouvez modifier la catégorie plus tard dans les paramètres du Play Store

- Application
- Jeu

Application gratuite ou payante

Vous pouvez modifier cette application plus tard sur la page de l'application payante

- Gratuit
- Payant

Tableau de bord

Commencer à configurer votre application

Lors de la configuration, les tâches à accomplir pour rendre votre application opérationnelle sont indiquées dans le tableau de bord. Vous trouverez ainsi des recommandations pour gérer, tester et promouvoir votre application. Dès que vous avez terminé une tâche, revenez ici pour découvrir ce que vous pouvez faire d'autre.

Masquer

Démarrer les tests maintenant

Publier votre application de manière anticipée en vue de réaliser des tests internes sans vérification

Partagez votre application avec un maximum de 100 testeurs internes pour identifier les problèmes et bénéficier de précieux retours

Masquer les tâches

- Sélectionner les testeurs >

CRÉER ET DÉPLOYER UNE RELEASE

- Créer une release >
- Vérifier et déployer la release

Tableau de bord

Fournir des informations sur votre application et configurer sa fiche Play Store

Décrivez-nous le contenu de votre application et gérez sa présentation sur Google Play

Masquer les tâches

DÉCRIRE LE CONTENU DE VOTRE APPLICATION

- Accès aux applications ›
- Annonces ›
- Classification du contenu ›
- Cible ›
- Applications d'actualités ›
- Applis sur le suivi des contacts et le statut COVID-19 ›

GÉRER L'ORGANISATION ET LA PRÉSENTATION DE VOTRE APPLICATION

- Sélectionner la catégorie de votre application et indiquer vos coordonnées ›
- Configurer une fiche Play Store ›

Publier votre application

Tester votre application auprès d'un plus large groupe de testeurs que vous contrôlez

Les tests fermés vous permettent de tester votre application auprès de plus larges groupes de personnes. Vous pouvez contrôler l'accès à l'aide des adresses e-mail ou de Google Groupes.

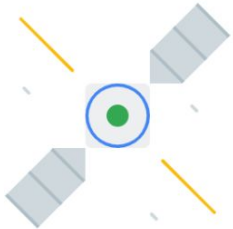
Effectuez d'abord les tâches de configuration initiales Masquer les tâches

CONFIGURER LE CANAL DE TEST FERMÉ

- Sélectionner les pays et les régions
- Sélectionner les testeurs

CRÉER ET DÉPLOYER UNE RELEASE

- Créer une release
- Vérifier et déployer la release



Publier votre application sur Google Play

Publiez votre application afin qu'elle soit disponible pour tous les utilisateurs sur Google Play en la diffusant dans le canal de production

Effectuez d'abord les tâches de configuration initiales Masquer les tâches

Sélectionner les pays et les régions

CRÉER ET DÉPLOYER UNE RELEASE

Créer une release

Vérifier et déployer la release

D.2 – Maîtriser Git

Qu’est-ce que Git ?

- **Git** est un **système de contrôle de version distribué** utilisé principalement pour gérer le code source des projets.
- Il permet de suivre les modifications des fichiers au fil du temps et de revenir à des versions antérieures.
- **Git** a été créé par **Linus Torvalds** en 2005 pour gérer le code source de **Linux**.
- Contrairement à d'autres systèmes comme **SVN** ou **CVS**, **Git** est **distribué**, ce qui signifie que chaque utilisateur possède une copie complète de l’historique du projet.

Pourquoi utiliser Git ?

- **Historique des changements** : Git garde une trace détaillée des modifications apportées à chaque fichier, permettant de revenir à n’importe quel état antérieur du projet.

- **Collaboration efficace** : Plusieurs développeurs peuvent travailler sur le même projet sans risque de conflits grâce aux branches et aux fusions.

- **Gestion des versions** : Git vous permet de gérer facilement les différentes versions d'un projet, ce qui facilite les tests et les corrections de bugs.

Git vs autres outils

Outil	Type	Avantages
Git	Distribué	Travail hors ligne, gestion complète de l’historique localement
SVN	Centralisé	Reposant sur un serveur central, plus simple à comprendre mais moins flexible

Installer Git

Télécharger Git depuis le site officiel https://git-scm.com et suivez les instructions selon votre système d'exploitation (Windows, macOS, Linux).

Sous Linux :

```
$ sudo apt install git
```

Vérifier l'installation :

```
$ git --version
```

Configurer Git

Après l’installation, il est important de configurer votre **nom** et **email** pour associer chaque commit à un utilisateur unique :

Nom :

```
$ git config --global user.name "Votre Nom"
```

Email :

```
$ git config --global user.email "vous@example.com"
```

Créer un dépôt local

Pour commencer un projet Git, initialisez un dépôt local avec la commande suivante :

```
$ git init
```

Cela crée un répertoire caché **.git** dans votre dossier, qui contiendra l'historique du projet.

Cloner un dépôt

Pour travailler sur un projet existant, clonez un dépôt distant avec cette commande :

```
$ git clone https://github.com/user/repo.git
```

Cela crée une copie locale du projet.

Vérifier l’état du dépôt

Pour voir les fichiers modifiés ou non suivis par Git, utilisez la commande suivante :

```
$ git status
```

Cela vous aide à savoir quels fichiers ont été modifiés depuis le dernier commit.

Ajouter un fichier au suivi

Vous pouvez ajouter un fichier spécifique au suivi de Git avec cette commande :

$ git add fichier.txt

Ajouter tous les fichiers

Pour ajouter tous les fichiers modifiés au suivi, utilisez la commande suivante :

```
$ git add .
```

Cette commande ajoute tous les fichiers modifiés et non suivis.

Faire un commit

Pour enregistrer les modifications dans l'historique de Git, utilisez la commande **commit** avec un message décrivant les changements :

```
$ git commit -m "Message du commit"
```

Voir l’historique des commits

Pour voir la liste des commits effectués dans le projet, utilisez :

$ git log

Cela affichera les commits avec leurs identifiants, dates et messages associés.

Voir les différences

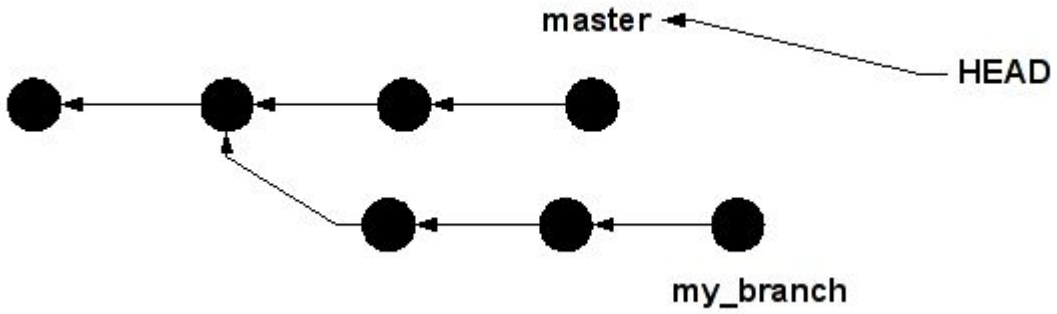
Pour afficher les différences entre l'état actuel des fichiers et la dernière version validée, utilisez :

$ git diff

Créer une branche

Pour ajouter une nouvelle fonctionnalité, il est recommandé de créer une branche dédiée :
$ git branch nouvelle-branche

Cette branche permet de travailler indépendamment sans affecter la branche principale (par défaut, main ou master).



Lister les branches

Pour voir toutes les branches existantes dans votre dépôt, utilisez :

$ git branch

Changer de branche

Pour passer d'une branche à une autre, utilisez :

$ git checkout nom-branche

Cela vous permet de basculer entre les différentes branches du projet.

Créer et changer de branche en une commande

Pour créer une nouvelle branche et y basculer immédiatement :

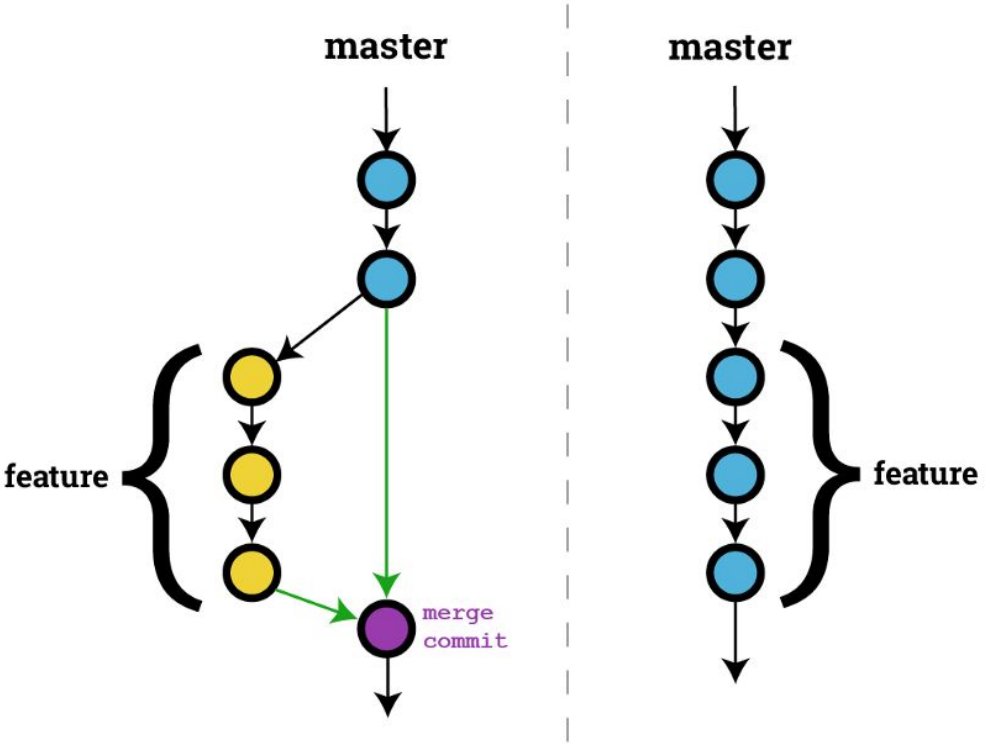
```
$ git checkout -b nouvelle-branche
```

Fusionner des branches

Une fois que vous avez terminé de travailler sur une branche, vous pouvez la fusionner avec la branche principale :

$ git merge nom-branche

Cela permet d'intégrer les changements de la branche dans la branche actuelle.



Supprimer une branche

Pour supprimer une branche locale une fois qu'elle n'est plus nécessaire :

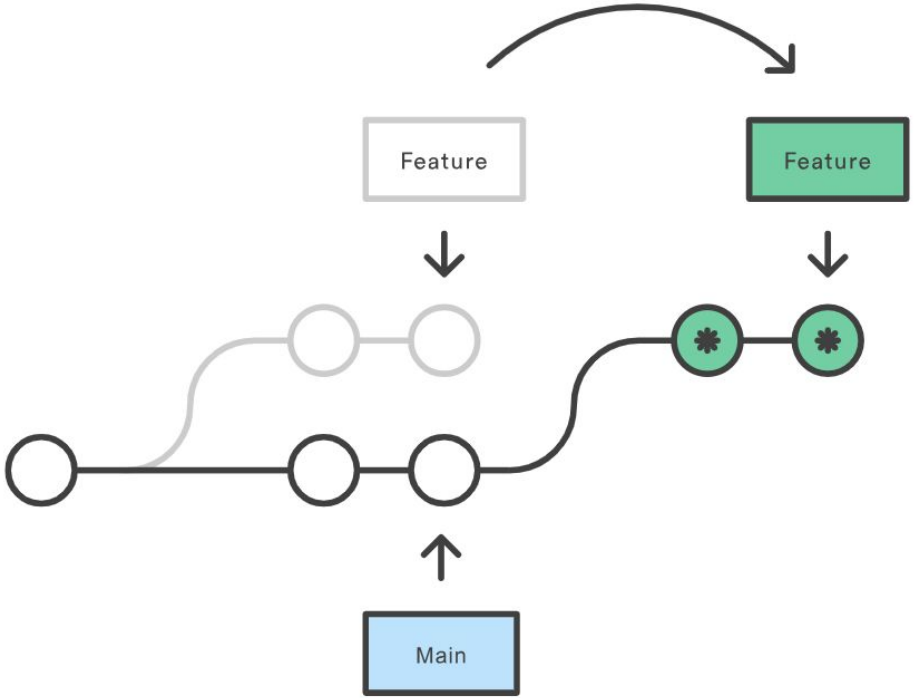
$ git branch -d nom-branche

Fusionner des branches

$ git rebase

vous permet de changer facilement une série de commits, en modifiant l'historique de votre dépôt.

Vous pouvez effectuer la réorganisation, la modification des commits de manière groupée.



Ajouter un dépôt distant

Pour lier votre projet local à un dépôt distant, utilisez la commande suivante :

$ git remote add origin https://github.com/user/repo.git

Voir les dépôts distants

Pour vérifier les dépôts distants associés à votre projet :

```
$ git remote -v
```

Envoyer les changements à un dépôt distant

Une fois que vous avez fait vos commits, vous pouvez envoyer vos modifications au dépôt distant :

$ git push origin main

Récupérer les changements depuis un dépôt distant

Pour récupérer les changements effectués par d'autres contributeurs, utilisez :

$ git pull

Cette commande fusionne automatiquement les changements du dépôt distant avec votre copie locale.

Télécharger sans fusionner

Si vous souhaitez seulement télécharger les dernières modifications sans les fusionner avec votre travail local, utilisez :

```
$ git fetch
```

Ignorer des fichiers

Créez un fichier **.gitignore** à la racine de votre projet pour indiquer les fichiers que vous ne souhaitez pas suivre (par exemple, les fichiers temporaires, logs, etc.) :

```
*.log
node_modules/
.env
```

Annuler un ajout

Si vous avez ajouté un fichier par erreur, vous pouvez le retirer de l’index (sans supprimer le fichier) avec :

```
$ git reset fichier.txt
```

Annuler un commit

Pour revenir en arrière et annuler le dernier commit tout en conservant les modifications dans votre répertoire de travail :

$ git reset --soft HEAD~1

Voir la configuration Git

Pour afficher la configuration de votre dépôt Git (nom, email, etc.) :

```
$ git config --list
```

Résumé

Git est un outil puissant pour la gestion de versions et la collaboration entre développeurs.

Il permet de suivre les modifications, de travailler sur différentes branches, et d'intégrer les modifications de manière fluide.

Git facilite grandement la gestion de projets complexes grâce à ses fonctionnalités avancées comme les branches, les commits et les fusions.

La liste des commandes GIT :

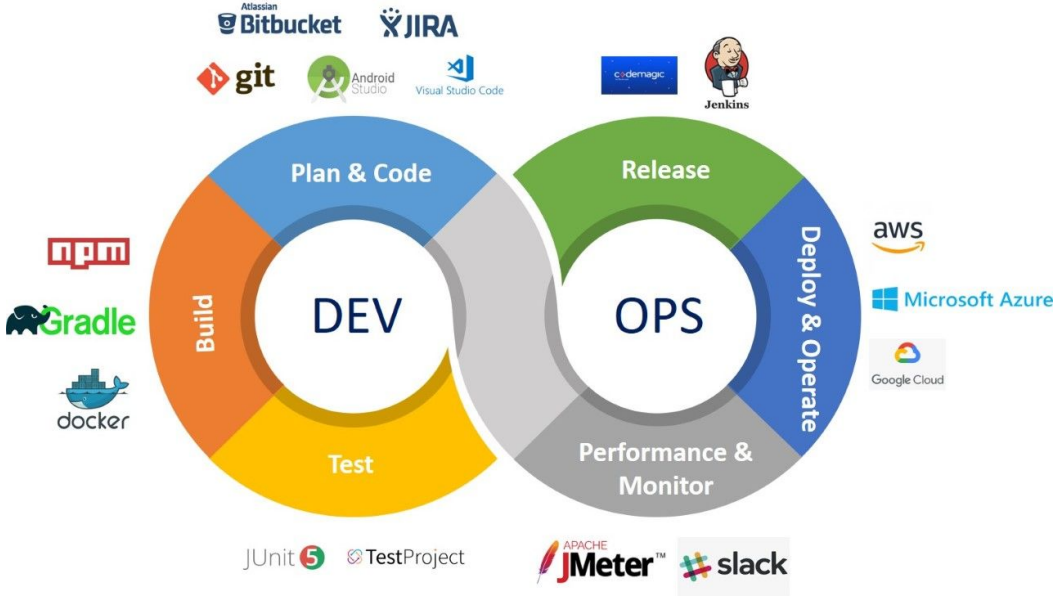
https://www.ionos.fr/digitalguide/fileadmin/DigitalGuide/Downloads/IONOS_Digital-Guide_Git-Cheat-Sheet_FR.pdf

TP 9

D.3– Utiliser des plate-formes d’intégration et déploiements continus
CI/CD

CI/CD (Intégration Continue / Déploiement Continu)

Désigne un ensemble de pratiques et de méthodologies visant à améliorer le processus de développement logiciel en automatisant l'intégration, les tests et les déploiements du code.



CI

L'intégration continue (Continuous integration, CI), est un ensemble de pratiques utilisées en génie logiciel consistant à vérifier à chaque modification de code source que le résultat des modifications ne produit **pas de régression** dans l'application développée.

Outils populaires de CI :

- **Jenkins** : Outil open-source très flexible et extensible.
- **GitHub Actions** : Directement intégré à GitHub, idéal pour les projets hébergés sur cette plateforme.
- **GitLab CI/CD** : Intégré à GitLab, offre une gestion complète des pipelines.
- **CircleCI** : Rapide et flexible, avec une bonne intégration Docker.
- **Travis CI** : Simple et bien intégré à GitHub, mais payant pour les dépôts privés.
- **Bitbucket Pipelines** : Intégré à Bitbucket, simple à configurer.

CD

La livraison continue (Continuous delivery, CD) est une approche d'ingénierie logicielle dans laquelle les équipes produisent des logiciels dans des cycles courts, ce qui permet de le mettre à disposition à n'importe quel moment.

Le but est de **construire**, **tester** et **diffuser** un logiciel plus rapidement.

Outils populaires de CD :

- **Argo CD** : Spécialement conçu pour Kubernetes, basé sur GitOps.
- **Jenkins X** : Extension de Jenkins pour Kubernetes, orientée microservices.
- **GitHub Actions** : Peut être utilisé pour le déploiement avec des workflows adaptés.
- **GitLab CD** : Offre un déploiement continu intégré avec GitLab CI.
- **CircleCI** : Possède des fonctionnalités pour automatiser les déploiements.

CI/CD & DevOps

Aspect	DevOps	CI/CD
Définition	Culture et méthodologie pour unifier développement et opérations	Pratiques pour automatiser l'intégration et le déploiement du code
Objectif	Collaboration, rapidité, stabilité, et amélioration continue	Automatisation des tests, intégration, et déploiement
Portée	Large, inclut gestion des configurations, surveillance, sécurité, et IaC	Plus spécifique, se concentre sur la gestion des versions et la livraison rapide
Exemples d'outils	Docker, Kubernetes, Ansible, Terraform	Jenkins, GitHub Actions, GitLab CI, Argo CD
Focus	Culture, automatisation, et collaboration	Qualité du code et rapidité des livraisons

Utilisation de Gitlab CI

GitLab CI est une fonctionnalité de GitLab permettant d'automatiser les tâches de CI/CD via des pipelines. Ces pipelines sont définis par un fichier .gitlab-ci.yml et exécutent diverses étapes comme la compilation, les tests, le déploiement, etc.

Utilisation de Gitlab CI

Prérequis

- Un compte sur **GitLab**.
- Un projet hébergé sur **GitLab**.
- **GitLab Runner** installé (GitLab Runner est un agent qui exécute les jobs définis dans votre pipeline).

Utilisation de Gitlab CI

La **gestion de pipeline** dans GitLab CI consiste à définir, organiser et contrôler les différentes étapes de votre pipeline CI/CD.

Un **pipeline** est un ensemble de jobs qui s'exécutent de manière séquentielle ou parallèle, chacun ayant un objectif spécifique (par exemple, construire le code, exécuter des tests, déployer l’application).

Utilisation de Gitlab CI

Qu'est-ce qu'un Pipeline ?

Un **pipeline** est une série de **jobs** organisés par **stages** dans un fichier `.gitlab-ci.yml`.

Chaque job exécute des commandes et peut dépendre des résultats des jobs précédents.

L’objectif principal est d’automatiser les tâches courantes de développement, comme la compilation, l'exécution des tests, ou le déploiement.

Utilisation de Gitlab CI

Structure d'un fichier .gitlab-ci.yml

Le fichier **.gitlab-ci.yml** définit les étapes du pipeline. Il contient plusieurs sections clés :

Exemple simple d'un fichier .gitlab-ci.yml :

stages : Définition des étapes du pipeline (ici, build, test, deploy).

job : Chaque étape peut contenir un ou plusieurs jobs qui définissent les actions à effectuer. Un job est associé à une étape du pipeline.

```
stages:
  - build
  - test
  - deploy

# Job pour construire l'application
build_job:
  stage: build
  script:
    - echo "Building the project..."

# Job pour tester l'application
test_job:
  stage: test
  script:
    - echo "Running tests..."

# Job pour déployer l'application
deploy_job:
  stage: deploy
  script:
    - echo "Deploying the project..."
```

Utilisation de Gitlab CI

Ajouter un GitLab Runner

Pour exécuter des jobs, vous devez configurer un **GitLab Runner**. Un runner peut être installé sur une machine locale ou dans le cloud (ex : AWS, Google Cloud).

Étapes pour installer un GitLab Runner :

1. Téléchargez le **GitLab Runner**.
2. Enregistrez le runner en suivant les instructions dans l'interface GitLab (ex : gitlab-runner register).
3. Liez-le à votre projet GitLab avec un token d'authentification.

Utilisation de Gitlab CI

Déploiement continu avec GitLab CI

GitLab CI peut être utilisé pour déployer automatiquement des applications après un succès dans les étapes précédentes.

Les **artéfacts** sont des fichiers produits par un job qui peuvent être utilisés dans d'autres jobs ou téléchargés après l'exécution du pipeline. Par exemple, vous pouvez stocker les résultats de tests ou les fichiers compilés pour une utilisation ultérieure.

Utilisation de Gitlab CI

Bonnes pratiques avec GitLab CI

- **Tests** : Exécuter toujours des tests automatisés avant de déployer. Cela garantit la stabilité.
- **Branches** : Utiliser des pipelines séparés pour différentes branches (par exemple, `develop` pour la préproduction et `master` pour la production).
- **Sécurité** : Utiliser des variables sécurisées pour gérer les informations sensibles comme les clés API et les mots de passe.

Utilisation de Gitlab CI

GitLab CI est un outil puissant pour automatiser les processus de développement et de déploiement.

En utilisant un fichier **.gitlab-ci.yml**, vous pouvez définir des workflows complexes pour vos projets.

Il existe encore de nombreuses fonctionnalités avancées comme la gestion des environnements, la parallélisation des jobs, et l'utilisation de Docker dans vos pipelines.

Manipulation de Fastlane

Fastlane

Fastlane est un outil open-source qui permet de faire du Continuous Delivery sous IOS et Android. Il permet d'automatiser un
certain nombre de tâches fastidieuses comme gérer les screenshots, les certificats, déployer votre app…
https://fastlane.tools/

Fastlane

Principales fonctionnalités :

1. **Automatisation des builds** - Crée automatiquement les builds pour différentes configurations.
2. **Gestion des certificats** - Automatise l'approvisionnement des certificats et des profils de provisioning.
3. **Distribution** - Publie les applications sur les app stores ou via des services comme TestFlight, Firebase App Distribution ou Play Store.
4. **Screenshots** - Capture automatiquement des captures d'écran pour différentes langues et résolutions.
5. **Gestion des changelogs** - Gère automatiquement les notes de version.

Fastlane

Installation :

```
# Installation via RubyGems
sudo gem install fastlane

# Vérification de l'installation
fastlane --version
```

Fastlane

Configuration :

Pour un projet Android, commencez par créer une configuration de base avec :

fastlane init

Puis, ajoutez un fichier **Fastfile** comme ceci :

Fastfile est un fichier **ruby** qui définit toutes vos **lanes**, on ajoute ce fichier dans le projet ensuite on le pousse, une fois l'outil CI/CD détecte sa présence, il l'exécute.

```
default_platform(:android)

platform :android do
  desc "Génère un APK pour production"
  lane :build do
    gradle(task: "assembleRelease")
  end

  desc "Publie l'APK sur le Play Store"
  lane :deploy do
    build
    supply(track: "production")
  end
end
```

TP 10

C’est la fin du module

Bon courage !