



Java

Introduction à l'écosystème Android et Kotlin



Java



Présentation de l'écosystème Android

L'écosystème Android est un système d'exploitation open-source pour les smartphones, tablettes, téléviseurs et autres appareils mobiles.

Créé par Google, Android a connu une croissance rapide ces dernières années grâce à sa flexibilité, sa dynamique communauté de développeurs et sa grande popularité auprès des utilisateurs.





Versions d'Android et compatibilité

L'une des caractéristiques clés d'Android est sa compatibilité avec plusieurs versions du système d'exploitation en même temps.

Cependant, il est important de noter que différentes versions d'Android peuvent avoir des exigences matérielles différentes, ce qui peut affecter la façon dont les applications fonctionnent sur différents appareils.





Versions d'Android et compatibilité

Les développeurs doivent prendre en compte ces différences lors de la création de leur application.

Il est également important de comprendre les défis liés à la fragmentation d'Android, qui est une conséquence de la large variété d'appareils Android en circulation dans le monde.





Langage de programmation Kotlin





Syntaxe de base

La syntaxe de Kotlin est assez proche de celle de Java, ce qui facilite sa prise en main si vous avez déjà des connaissances en développement Android.





Déclaration d'une variable :

Dans Kotlin, les variables sont déclarées avec le mot-clé `var` ou `val`.

La différence entre ces deux mots-clés est que `var` permet de modifier la valeur d'une variable, alors que `val` permet de déclarer une constante.

```
var nomVariable: typeDonnée = valeurAssignée
```





Déclaration d'une fonction :

Les fonctions sont déclarées avec le mot-clé `fun`.

Elles peuvent prendre des paramètres et retourner une valeur.

```
fun nomFonction(paramètre1: typeDonnée1, paramètre2: typeDonnée2): typeDonnéeRetour {  
    // Code de la fonction  
    return valeurRetour  
}
```





Boucle for :

Les boucles for sont déclarées avec le mot-clé `for`.

```
for (i in 0..10) {  
    // Code de la boucle  
}
```





Boucle while :

Les boucles while sont déclarées avec le mot-clé `while`.

```
while (condition) {  
    // Code de la boucle  
}
```



Fonctionnalités avancées

En plus de sa syntaxe proche de celle de Java, Kotlin offre des fonctionnalités avancées qui facilitent le développement d'application Android.

L'une de ces fonctionnalités est la possibilité d'utiliser des expressions conditionnelles ternaires.

```
val valeur = if (condition) valeur1 else valeur2
```



Les null-safety

Kotlin permet d'éviter les erreurs liées aux références nulles en incluant cette notion de null-safety dans la syntaxe.

Par exemple :

```
var nomVariable: TypeDonnée? = null
```



Les lambdas

Les lambdas permettent de définir des fonctions anonymes, ce qui facilite l'écriture de code plus succinct et plus lisible.

Voici un exemple de lambda en Kotlin :

```
liste.forEach { element ->
    println(element)
}
```



Les extensions

Les extensions permettent d'ajouter des fonctionnalités à des classes existantes sans avoir besoin de les étendre.

Cette fonctionnalité facilite la réutilisation de code et permet d'améliorer la modularité de votre application.

Voici un exemple d'extension en Kotlin :

```
fun String.reverse(): String {  
    return this.reversed()  
}
```





Android Studio et Android SDK

Android Studio est l'EDI (environnement de développement intégré) officiel pour le développement Android.

Il fournit un éditeur de code et une suite complète d'outils de développement pour créer des applications, tester et déployer sur un emulateur ou un appareil Android.

Android Studio et Android SDK

L'Android SDK (Software Development Kit) est une collection d'API, de bibliothèques, de documentations et d'autres outils de développement qui simplifient le développement Android.

L'Android SDK permet également de créer des émulateurs de périphériques pour tester des applications dans différentes configurations.

Activités

Une activité est une classe qui représente une interface utilisateur avec laquelle l'utilisateur peut interagir.

Dans une application Android, une activité peut être considérée comme une "page" ou une "fenêtre" qui affiche un contenu particulier.

La classe héritera de la classe `Activity` ou `FragmentActivity` ou `AppCompatActivity`.



Cycle de vie d'une activité

Puisqu'une activité peut être lancée, interrompue et fermée plusieurs fois tout au long de l'exécution d'une application, il est important de comprendre son cycle de vie.





Création et gestion d'activités

La classe `Activity` est la classe de base pour toutes les activités Android.

Une activité est créée et gérée par un `ActivityManager`.

Son cycle de vie est : `onCreate()` , `onStart()` , `onResume()` , `onPause()` ,
`onStop()` , `onDestroy()` .



Fragments

Les fragments sont des éléments d'interface utilisateur réutilisables qui facilitent la modularité du code et la gestion des écrans d'une application Android.

Cycle de vie d'un fragment

Le Cycle de vie d'un fragment est le suivant :

1. Attachement : le fragment est attaché à l'activité.
2. Crédation : le fragment est créé.
3. Visualisation : le fragment est affiché à l'écran.
4. Activation : le fragment est prêt à interagir avec l'utilisateur.
5. Inactivation : le fragment est caché et plus interactif.
6. Détachement : le fragment est détaché de l'activité.



Communication entre fragments

Les fragments peuvent communiquer entre eux grâce à l'interface définie dans l'activité hôte :

```
interface FragmentCallbacks {  
    fun onItemSelected(item: String)  
}
```

Communication entre fragments

En outre, les fragments peuvent également communiquer avec l'activité hôte pour obtenir des données ou des informations via les méthodes de rappel :

```
class FragmentA : Fragment() {
    override fun onAttach(context: Context) {
        super.onAttach(context)
        if (context is FragmentCallbacks) {
            callbacks = context
        } else {
            throw RuntimeException("$context must implement FragmentCallbacks")
        }
    }
}
```



Services



Cycle de vie d'un service

Un service Android est un composant de l'application qui s'exécute en arrière-plan, sans interface utilisateur.

Les services sont principalement utilisés pour effectuer des opérations longues durées telles que des téléchargements de fichiers ou des traitements de données en arrière-plan.

Cycle de vie d'un service

Le cycle de vie d'un service est le suivant :

- 📌 Création : le service est créé.
- 📌 Démarrage : le service est démarré.
- 📌 Arrêt : le service est arrêté.
- 📌 Destruction : le service est détruit.



Intents et diffusion d'intentions (BroadcastReceiver)

L'un des moyens les plus courants pour les composants de communiquer entre eux est l'utilisation d'intents.

Les intents sont des objets qui déclenchent des actions dans l'application, que ce soit pour lancer une activité, un service ou envoyer des données à un autre composant.



Exemples d'intents

Cet Intent permet de lancer une activité :

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

Les intents

Les intents peuvent être explicites ou implicites.

Les intents explicites sont utilisés pour lancer un composant spécifique de l'application en utilisant son nom qualifié.

Les intents implicites sont utilisés lorsque l'application souhaite que le système d'exploitation recherche le composant qui peut traiter la demande.

Récepteurs de diffusion

Les récepteurs de diffusion sont des composants qui permettent aux applications de recevoir des messages de l'OS ou d'autres applications.

Les récepteurs de diffusion sont définis dans le fichier `AndroidManifest.xml` :

```
<receiver android:name=".MyReceiver">
    <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED" />
    </intent-filter>
</receiver>
```

ContentProvider

Le `ContentProvider` est un composant clé de l'architecture d'une application Android.

Il permet de fournir et de consommer des données entre différentes applications.

Concrètement, le `ContentProvider` permet à une application de fournir des données à d'autres applications, tout en s'assurant que ces données sont utilisées de manière sécurisée.

ContentProvider

Il permet également à une application de consommer des données fournies par d'autres applications, en mettant en place une infrastructure permettant de récupérer ces données de manière transparente.

La principale fonction d'un ContentProvider est de gérer l'accès aux données d'une application.

ContentProvider

Il peut s'agir de données stockées en interne, comme un fichier texte ou une base de données SQLite, ou de données provenant d'une source externe, comme un serveur web.

Le `ContentProvider` est un élément central de l'architecture Android, qui permet à différents composants d'une application de communiquer entre eux, de façon à créer une application robuste et modulaire.



La structuration de vos projets

Lors de la conception de vos projets informatiques, la structuration est essentielle pour assurer une bonne organisation et une collaboration efficace entre les membres de l'équipe.

Elle est la base de la bonne compréhension du code source et de la documentation.



Organisation des fichiers et dossiers

Organisation des fichiers et dossiers :

Il est important d'établir une structure claire et cohérente pour l'ensemble des fichiers et dossiers du projet.

Il est recommandé d'utiliser des noms de fichiers et de dossiers explicites et significatifs pour faciliter la recherche et la compréhension des données.



Utilisation de contrôle de version

L'utilisation d'un système de contrôle de version tel que Git permet de gérer l'historique des modifications apportées aux fichiers et de collaborer efficacement avec les autres membres de l'équipe.



Documentation

La documentation du projet est essentielle pour son évolution future.

Il est recommandé d'inclure une documentation détaillée du code source, des procédures d'installation et d'utilisation ainsi que des guides d'utilisation.



Test et validation

Avant la mise en production, il est important de tester et valider le projet pour s'assurer de sa fonctionnalité et de sa qualité.

Il est recommandé d'utiliser des outils de tests automatisés.

Maintenance

La maintenance est essentielle pour garantir le bon fonctionnement du projet sur le long terme.

Elle inclut les mises à jour, les corrections de bugs et les améliorations.

Il est recommandé d'adopter une méthode agile pour la gestion de la maintenance.

Structuration du code

La structuration du code est une étape importante pour faciliter la maintenance et l'évolutivité d'un projet.

Il existe différentes approches pour organiser son code en fonction de la taille du projet et de ses besoins.

Structure de projet

Voici quelques exemples de structures de projets :

Modèle MVC (Modèle-Vue-Contrôleur) : cette architecture sépare la logique de présentation (Vue), la logique métier (Modèle) et la logique de contrôle (Contrôleur).

Cette structure est souvent utilisée pour les applications web.



Arborescence en couches :

Cette structure sépare le code en couches en fonction de leur niveau d'abstraction.

Par exemple, une couche de données, une couche d'accès aux données, une couche métier et une couche de présentation.



Architecture hexagonale :

Cette architecture se concentre sur les cas d'utilisation de l'application en les plaçant au centre de l'architecture.

La logique métier est ensuite répartie autour de ces cas d'utilisation en différentes couches.



Gestion des dépendances et modules

La gestion des dépendances et des modules est une partie importante de la structure d'un projet.

Elle permet de simplifier le développement en utilisant des bibliothèques externes et en organisant son code en différents modules.



Utilisation de Gradle

Gradle est un système de build open-source utilisé pour automatiser le processus de construction, de test et de déploiement de logiciels.

Il permet de gérer les dépendances, les tâches et les modules.

Gradle permet :

Déclaration de dépendances : Gradle permet de déclarer les dépendances de l'application dans un fichier de configuration.

Définition de tâches : Gradle permet de définir des tâches à exécuter comme la compilation, le lancement des tests ou la génération de la documentation.

Gestion de modules : Gradle permet d'organiser son application en différents modules.



Conventions de nommage et organisation des fichiers

Nommage

Voici quelques bonnes pratiques à suivre :

- 📌 Les noms de classes devraient être écrits en PascalCase, c'est-à-dire que la première lettre de chaque mot est en majuscule.

Par exemple : MaClasse , MaClassePrincipale .



Les méthodes

Les noms de méthodes sont écrits en camelCase, c'est-à-dire que la première lettre du premier mot est en minuscule et la première lettre de chaque mot suivant est en majuscule.

Par exemple : `maMethod` , `maVariable` .



Les Constantes

Les noms de constantes sont écrits en MAJUSCULES, avec des mots séparés par des underscores.

Par exemple : MA_CONSTANTE .

La conception

Les étapes de la conception sont les suivantes :

- 📌 Analyse des besoins
- 📌 Conception globale
- 📌 Conception détaillée
- 📌 Développement
- 📌 Tests unitaires et d'intégration
- 📌 Déploiement

Structurer son code

Les bonnes pratiques :

- 📌 Écrire du code clair et facile à comprendre.
- 📌 Utiliser des noms de variables et de fonctions explicites.
- 📌 Éviter la répétition de code en utilisant des fonctions ou des classes
- 📌 Utiliser des commentaires pour rendre le code plus compréhensible.

Les types de versionning

Il existe plusieurs méthodes pour la gestion de versionning des logiciels :

- 📌 Version simple, avec un numéro de version incrémenté à chaque nouvelle version.
- 📌 Version avec des branches, pour gérer des versions alternatives en parallèle.
- 📌 Version avec des étiquettes, pour identifier des versions spécifiques.



Utiliser un système de versionning

Il est recommandé d'utiliser un système de versionning pour conserver les différentes versions d'un logiciel.

Ainsi, on utilisera un outil comme Git ou Subversion pour enregistrer les modifications effectuées sur le code source.

Les versions d'Android

Il existe plusieurs versions d'Android qui nécessitent une gestion de compatibilité :

- 📌 Android 4 : est une version obsolète d'Android.
- 📌 Android 5 et 6 : sont encore utilisées sur certains appareils.
- 📌 Android 7, 8 et 9 : sont des versions récentes d'Android.



Gérer la compatibilité entre les versions

Pour gérer la compatibilité entre les différentes versions d'Android, il est important d'utiliser des librairies compatibles avec chaque version d'Android.

De plus, il est recommandé d'utiliser des fonctionnalités de compatibilité pour assurer un fonctionnement uniforme entre les différentes versions.



Design des interfaces

Le design des interfaces est une étape cruciale dans le cycle de développement d'une application Android.

Il permet de concevoir et de visualiser les différentes interfaces graphiques de l'application.

Prototypage et maquettes

Pour concevoir une interface utilisateur, la première étape consiste à faire un prototypage ou des maquettes.

Il s'agit de créer une représentation graphique de l'interface de l'application.

Le but du prototypage est de définir et de valider la structure de l'interface, les interactions entre les différents éléments, les couleurs, les typographies ainsi que les icônes à utiliser.

Développement et tests

Pendant cette phase de développement, il est important de suivre les étapes suivantes pour garantir un fonctionnement optimal de l'application :

- 📌 Implémentation des fonctionnalités
- 📌 Tests unitaires



Implémentation des fonctionnalités

Cette étape consiste à programmer et intégrer toutes les fonctionnalités de l'application en respectant les spécifications de conception.

Il est important de procéder de manière méthodique pour éviter des erreurs de programmation qui pourraient compromettre le bon fonctionnement de l'application.

Tests unitaires

Ils consistent à utiliser des outils d'automatisation pour exécuter des tests sur chaque unité du code pour s'assurer qu'elle fonctionne correctement.

Les tests unitaires sont essentiels pour détecter les erreurs de programmation le plus tôt possible et pour garantir la qualité de l'application.

Vues et widgets

Une vue est un rectangle qui peut être dessiné sur l'écran et qui permet à l'utilisateur d'interagir avec l'application.

Les widgets sont des éléments de l'interface utilisateur, qui sont utilisés dans les vues pour aider l'utilisateur à interagir avec l'application.

Widgets

Voici quelques exemples de widgets couramment utilisés :

- 📌 TextView : utilisé pour afficher du texte
- 📌 EditText : utilisé pour saisir du texte
- 📌 Button : utilisé pour exécuter une action
- 📌 ImageView : utilisé pour afficher des images



Création de vues

Dans Android Studio, vous pouvez créer des vues et des widgets à partir du panneau Palette.

Vous pouvez facilement ajouter, supprimer et configurer les widgets pour créer l'interface de votre choix.

Pour créer une vue en XML, vous devez utiliser le balisage XML.



Fichiers XML

Vous pouvez créer un fichier XML pour chaque vue et utiliser le système de gestionnaire de ressources fourni par Android pour les récupérer dans votre code.

Voici un exemple de code XML pour créer une vue TextView :

```
<TextView  
    android:id="@+id/tv_hello_world"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="Hello World!" />
```

Layouts

Les layouts sont utilisés pour organiser les éléments de l'interface utilisateur sur un écran.

- 📌 LinearLayout : aligne les éléments dans un ordre linéaire
- 📌 RelativeLayout : positionne les éléments les uns par rapport aux autres
- 📌 ConstraintLayout : positionne les éléments les uns par rapport aux autres en utilisant des contraintes

LinearLayout

LinearLayout permet d'aligner les éléments dans un ordre linéaire, soit horizontalement ou verticalement.

Les éléments peuvent être centrés, alignés à gauche ou à droite, ou répartis de manière égale.



Exemple de LinearLayout horizontal :

```
<LinearLayout  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:orientation="horizontal">  
  
    <Button  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:text="Bouton 1" />  
  
</LinearLayout>
```

RelativeLayout

RelativeLayout permet de positionner les éléments les uns par rapport aux autres en utilisant des règles telles que "en haut de", "à gauche de", "en dessous de", etc.

Cela permet une plus grande flexibilité dans la conception de l'interface utilisateur.

```
<RelativeLayout  
    android:layout_width="match_parent"  
    android:layout_height="match_parent">  
  
    <TextView  
        android:id="@+id/titre"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:text="Titre" />  
  
    <Button  
        android:id="@+id/bouton1"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:text="Bouton 1"  
        android:layout_below="@id/titre" />  
  
</RelativeLayout>
```



FrameLayout

FrameLayout permet de superposer les éléments les uns sur les autres.

C'est utile lorsque l'on veut afficher des éléments tels que des boutons flottants ou des notifications de manière très visible.

```
<FrameLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <ImageView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:src="@drawable/image" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Notification"
        android:layout_gravity="top|end"
        android:padding="8dp"
        android:background="@color/colorAccent"
        android:textColor="@color/colorWhite" />

    <ImageButton
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/fab"
        android:layout_gravity="bottom|right"
        android:padding="16dp" />

</FrameLayout>
```



Ressources

Lors de la conception d'une interface utilisateur, il est important de prendre en compte les différentes ressources nécessaires pour créer une expérience utilisateur agréable et intuitive.

Elles sont stockées dans le dossier res. Et peuvent être utilisées dans le code de l'application comme des variables.



Les Images :

Images : Les images sont souvent utilisées pour les icônes, les boutons et les éléments de navigation pour aider les utilisateurs à comprendre visuellement le contenu du site ou de l'application.

Il est important de choisir des images de haute qualité et de les optimiser pour une performance optimale.



Les Chaînes de caractères :

Chaînes de caractères : Les chaînes de caractères sont utilisées pour les étiquettes, les titres et les descriptions.

Il est important de localiser ces chaînes pour une utilisation internationale et de les tester avec des utilisateurs pour s'assurer qu'elles sont claires et compréhensibles.



Les Couleurs :

Couleurs : Les couleurs jouent un rôle important dans l'expérience utilisateur en créant une ambiance et un style pour le site ou l'application.

Il est important de choisir des couleurs cohérentes avec la marque et de les utiliser avec parcimonie pour éviter une confusion visuelle.



Les Styles :

Styles : Les styles incluent les typographies, les couleurs et les dispositions pour les éléments visuels tels que les boutons, les liens et les sections.

Ils doivent être cohérents et suivre les conventions établies pour éviter la confusion chez les utilisateurs.

Bases de données SQLite

Base de données

SQLite est un système de gestion de bases de données relationnelles qui est très populaire en raison de sa simplicité et de son efficacité.

Il est souvent utilisé pour stocker des données sur des appareils mobiles car il ne nécessite pas de serveur.

Création d'une table:

Après avoir créé une base de données, vous pouvez créer une table pour stocker vos données avec la commande :

par exemple:

```
CREATE TABLE utilisateurs (
    id INTEGER PRIMARY KEY,
    nom TEXT NOT NULL,
    age INTEGER,
    email TEXT
);
```



Insérez des données dans la table

Pour insérer des données dans la table que vous avez créée, utilisez la commande :

```
INSERT INTO nom_table (colonne1, colonne2, ...) VALUES (valeur1, valeur2, ...);
```

par exemple:

```
INSERT INTO utilisateurs (nom, age, email) VALUES ('Pierre', 25, 'pierre@email.com');
```



Consultez les données:

Pour consulter les données, utilisez la commande :

```
SELECT colonne_1, colonne_2, ...  
FROM nom_table;
```

par exemple:

```
SELECT * FROM utilisateurs;
```



Mettre à jour les données

Pour mettre à jour les données existantes, utilisez la commande :

```
UPDATE nom_table SET colonne_1 = valeur1 WHERE condition;
```

par exemple:

```
UPDATE utilisateurs SET age = 26 WHERE nom = 'Pierre' ;
```



Supprimer des données

Pour supprimer des données de la table, utilisez la commande :

```
DELETE FROM nom_table WHERE condition;
```

par exemple:

```
DELETE FROM utilisateurs WHERE nom = 'Pierre';
```

Room Persistence Library

La **Room Persistence Library** est une bibliothèque qui facilite la gestion des bases de données dans le développement d'applications Android.

Elle est composée de trois éléments clés : les entités, les DAO et les bases de données.

Entités

Les **entités** représentent des objets qui peuvent être stockés dans la base de données.

Pour créer une entité, on peut utiliser l'annotation `@Entity` pour spécifier le nom de la table et les noms des colonnes.

Exemple :

```
@Entity(tableName = "utilisateurs")
data class Utilisateur(
    @PrimaryKey val id: Int,
    val nom: String,
    val age: Int
)
```

Dans cet exemple, nous avons créé une entité **Utilisateur** qui sera stockée dans une table nommée **utilisateurs**.

Cette table contiendra trois colonnes : **id**, **nom** et **age**.

DAO

Les **DAO** (Data Access Object) sont des interfaces qui permettent de définir les opérations à effectuer sur la base de données.

On peut annoter une interface avec `@Dao` pour le signaler à la bibliothèque Room.

Exemple :

```
@Dao
interface UtilisateurDao {
    @Query("SELECT * FROM utilisateurs")
    fun getAll(): List<Utilisateur>

    @Insert
    fun insert(utilisateur: Utilisateur)

    @Update
    fun update(utilisateur: Utilisateur)

    @Delete
    fun delete(utilisateur: Utilisateur)
}
```

Bases de données

Les **bases de données** sont des instances de la classe `RoomDatabase` qui contiennent les DAO et permettent de les utiliser pour interagir avec la base de données.

Exemple :

```
@Database(entities = [Utilisateur::class], version = 1)
abstract class MaBaseDeDonnees : RoomDatabase() {
    abstract fun utilisateurDao(): UtilisateurDao
}
```



REST API (JSON, XML)

Les REST API sont une méthode populaire pour communiquer entre les applications.

Elles utilisent des formats de données courants comme JSON et XML pour envoyer et recevoir des informations entre les clients et les serveurs.

JSON

JSON est un format de données léger qui est facilement compréhensible par les humains et les machines.

Il est couramment utilisé pour les échanges de données entre les clients et les serveurs Web.

```
{  
    "name": "John Doe",  
    "age": 30,  
    "email": "john.doe@email.com"  
}
```

XML

XML est un langage de balisage qui est également utilisé pour les échanges de données.

Il est plus verbeux que JSON, mais il a l'avantage d'être facilement lisible par les humains.

```
<person>
    <name>John Doe</name>
    <age>30</age>
    <email>john.doe@email.com</email>
</person>
```



Bibliothèques de réseau

Les bibliothèques de réseau permettent à une application de communiquer avec un serveur distant et de récupérer les données nécessaires pour l'exécution de l'application.



Retrofit

Retrofit est une bibliothèque de réseau pour Android qui permet d'effectuer des requêtes HTTP de manière simple et efficace.

Elle permet également de convertir les données JSON en objets Java.

OkHttp

OkHttp est une bibliothèque de réseau pour Android qui permet d'effectuer des requêtes HTTP de manière asynchrone.

Elle offre également de nombreuses fonctionnalités telles que la mise en cache des réponses, la compression des données et la redirection automatique.



Gestion des requêtes et réponses

La gestion des requêtes et réponses est cruciale dans toute application qui communique avec un serveur distant.

Il convient de comprendre les différentes étapes de la communication, depuis l'envoi de la requête jusqu'à la réception de la réponse, en passant par le traitement des erreurs éventuelles.

Envoi de la requête

L'envoi de la requête se fait à l'aide d'un objet Request qui contient toutes les informations nécessaires pour la transmission des données.



Réception de la réponse

La réception de la réponse se fait à l'aide d'un objet Response qui contient toutes les informations renvoyées par le serveur distant.

Traitement des erreurs

Il est important de gérer les erreurs qui peuvent survenir pendant la communication avec le serveur distant.

Les erreurs les plus courantes sont les erreurs de réseau, les erreurs de format de données et les erreurs de sécurité.

Il convient de définir une stratégie de gestion des erreurs et de la mettre en œuvre de manière cohérente dans toute l'application.



Gestion des autorisations et sécurité

La gestion des autorisations et la sécurité sont des concepts très importants pour l'utilisation des API.



Permissions

Les permissions permettent de définir les droits d'accès aux différentes parties d'un programme ou d'une API.

Il est important de définir avec soin les permissions pour assurer l'authentification correcte des utilisateurs et la protection des données.

OAuth

OAuth est un protocole d'autorisation pour les API.

Il permet à un utilisateur d'autoriser un site ou une application tierce à accéder à ses données sans avoir à entrer de nom d'utilisateur et de mot de passe.

OAuth est très utilisé pour les applications Web et mobiles, car il est facile à utiliser et à intégrer.

AsyncTask

L' `AsyncTask` permet d'exécuter des tâches longues en arrière-plan (par opposition à l'UI thread), tout en ayant la possibilité de mettre à jour l'interface utilisateur.

Cela est particulièrement utile pour améliorer la réactivité de l'application et la fluidité de l'interface.

Tests unitaires

Les tests sont une étape importante pour s'assurer de la qualité et de la fiabilité du code.

Il existe différents types de tests, notamment les tests unitaires et les tests d'intégration.

Les tests unitaires sont des tests qui vérifient le bon fonctionnement de chaque unité de code (par exemple une méthode ou une classe) de manière isolée.

Tests d'intégration

Les tests d'intégration, quant à eux, vérifient le bon fonctionnement de l'ensemble de l'application.

Pour la réalisation de ces tests, il est possible d'utiliser différents frameworks et outils.

Par exemple, JUnit est un framework très utilisé pour les tests unitaires en Java, tandis que Espresso est un framework pour les tests d'interface utilisateur sur Android.

Utilisation de JUnit et Espresso

JUnit est un framework de tests unitaires pour Java.

Il permet de définir des méthodes de tests pour chaque unité de code à tester.

Pour cela, il utilise des annotations spécifiques comme `@Test`, qui permet de définir une méthode de test.

Exemple

```
import org.junit.Test;
import static org.junit.Assert.*;

public class CalculatriceTest {

    @Test
    public void testSomme() {
        Calculatrice calc = new Calculatrice();
        assertEquals(4, calc.somme(2, 2));
    }
}
```

Explication

Ce test vérifie que la méthode somme de la classe Calculatrice retourne bien un résultat de 4 pour les paramètres 2 et 2.

Espresso, quant à lui, est un framework de tests pour les applications Android.

Il permet de tester les interactions et le comportement de l'interface utilisateur.

Exemple

```
@Test
public void testClicBouton() {
    // Récupération de la vue du bouton
    onView(withId(R.id.button)).perform(click());

    // Vérification que le texte affiché dans un TextView est bien celui attendu
    onView(withId(R.id.textView)).check(matches(withText("Bouton cliqué")));
}
```



Les Architecture de l'application



Architecture MVVC

L'architecture MVVC est une architecture de développement logiciel qui sépare les données de l'interface utilisateur.

Elle permet de séparer les différentes responsabilités de l'application et de faciliter la maintenance et l'extension du code.



Modèle

Le modèle représente les données de l'application.

Dans une application Android, le modèle est généralement représenté par des objets Entity.



Exemple

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



Vue

La vue représente l'interface utilisateur de l'application.

Dans une application Android, la vue est généralement représentée par des activités et des fragments.





Exemple

```
class MainActivity : AppCompatActivity() {  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
    }  
}
```

Vue-modèle

Le vue-modèle représente le code qui gère les interactions entre la vue et le modèle.

Dans une application Android, le vue-modèle est généralement représenté par des classes ViewModel.

En général, le vue-modèle est responsable de la logique d'affichage de l'application.

Exemple

```
class MainViewModel : ViewModel() {  
  
    private val _text = MutableLiveData<String>()  
    val text: LiveData<String> = _text  
  
    init {  
        _text.value = "Hello World"  
    }  
}
```