Kotlin

Persistance de données

Accès à une base de données

Concepts de bases de données

Les bases de données sont des systèmes permettant de stocker, d'organiser et de manipuler de grandes quantités de données structurées de manière cohérente.

Pour cela, elles suivent souvent un modèle relationnel.

Modèle relationnel

Le modèle relationnel regroupe les données en tables où chaque ligne représente un enregistrement et chaque colonne une donnée de cet enregistrement.

Les tables sont reliées entre elles via des clés primaires et étrangères.

Entités et relations

Dans le modèle relationnel, les entités représentent les éléments que l'on souhaite stocker et les relations définissent les liens entre ces entités.

Clés primaires et étrangères

La clé primaire d'une table est l'attribut qui permet d'identifier de manière unique un enregistrement dans cette table.

La clé étrangère d'une table permet de relier cette table à une autre table en référençant une clé primaire de cette dernière.

Exemple d’accès à une base de données

Voici un exemple basique d’accès à une base de données en Kotlin :

```
class MyDatabaseHelper(context: Context) : SQLiteOpenHelper(context, DATABASE_NAME, null,
 DATABASE_VERSION) {

    companion object {
        private const val DATABASE_NAME = "my_database.db"
        private const val DATABASE_VERSION = 1

        // Définition de la table des utilisateurs
        private const val TABLE_USERS = "users"
        private const val COLUMN_ID = "id"
        private const val COLUMN_NAME = "name"
        private const val COLUMN_EMAIL = "email"
    }

    override fun onCreate(db: SQLiteDatabase) {
        // Création de la table des utilisateurs
        val CREATE_USERS_TABLE = ("CREATE TABLE $TABLE_USERS ("
                + "$COLUMN_ID INTEGER PRIMARY KEY,"
                + "$COLUMN_NAME TEXT,"
                + "$COLUMN_EMAIL TEXT)")
        db.execSQL(CREATE_USERS_TABLE)
    }

    override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {
        // Suppression de la table des utilisateurs si elle existe déjà
        db.execSQL("DROP TABLE IF EXISTS $TABLE_USERS")

        // Recréation de la table des utilisateurs
        onCreate(db)
    }
override fun onCreate(db: SQLiteDatabase) {
        // Création de la table des utilisateurs
        val CREATE_USERS_TABLE = ("CREATE TABLE $TABLE_USERS ("
                + "$COLUMN_ID INTEGER PRIMARY KEY,"
                + "$COLUMN_NAME TEXT,"
                + "$COLUMN_EMAIL TEXT)")
        db.execSQL(CREATE_USERS_TABLE)
    }
}
```

Création et ouverture d’une base de données

La première étape pour utiliser une base de données est de créer et ouvrir une connexion avec celle-ci.

```
class MyDatabaseHelper(context: Context) : SQLiteOpenHelper(context, DATABASE_NAME, null,
 DATABASE_VERSION) {

    companion object {
        private const val DATABASE_NAME = "my_database.db"
        private const val DATABASE_VERSION = 1

        // Définition de la table
    }
}
```

Création de la structure de la base de données

Une fois la connexion établie, il est temps de créer la structure de la base de données.

Cela implique de définir les tables, les colonnes et les types de données qui y seront stockés.

```
override fun onCreate(db: SQLiteDatabase) {
        // Création de la table des utilisateurs
        val CREATE_USERS_TABLE = ("CREATE TABLE $TABLE_USERS ("
                + "$COLUMN_ID INTEGER PRIMARY KEY,"
                + "$COLUMN_NAME TEXT,"
                + "$COLUMN_EMAIL TEXT)")
        db.execSQL(CREATE_USERS_TABLE)
    }
```

Connexion à la base de données

Une fois que la structure de la base de données est définie,nous pourrons y avoir acces depuis l’objet `writableDatabase`

```
// Suppression de la table des utilisateurs si elle existe déjà
db.execSQL("DROP TABLE IF EXISTS $TABLE_USERS")
// Recréation de la table des utilisateurs
onCreate(db)
```

Requêtes de sélection (SELECT)

Pour obtenir des enregistrements à partir de la base de données, on utilise la méthode `rawQuery`

Cette méthode permet d'avoir acces a un `cursor` qui sera un iterateur sur la structure de donne

---

Exemple :

```
fun getAllUsers(): List<User> {
        val selectQuery = "SELECT * FROM $TABLE_USERS"

        val db = this.readableDatabase
        val cursor = db.rawQuery(selectQuery, null)

        if (cursor.moveToFirst()) {
            do {
                 // Recuperation de la data
            } while (cursor.moveToNext())
        }

        cursor.close()
        db.close()

        return userList
    }
```

Requêtes d'insertion (INSERT)

Pour ajouter des enregistrements dans la base de données, on utilise la méthode `insert` de la classe `SQLiteDatabase`.

Cette méthode prend en paramètre le nom de la table et un objet ContentValues qui contient les paires clé-valeur à insérer.

Exemple :

```
fun insertUser(user: User) {
    val db = writableDatabase
    val values = ContentValues()
    values.put("name", user.name)
    values.put("email", user.email)
    db.insert("users", null, values)
```

```
    db.close()
}
```

Requêtes de mise à jour (UPDATE)

Pour modifier des enregistrements dans la base de données, on utilise la méthode `update` de la classe `SQLiteDatabase`.

Cette méthode prend en paramètre le nom de la table, l'objet ContentValues qui contient les nouvelles valeurs, ainsi que les critères de filtrage (WHERE).

Exemple :

```
// Fonction pour mettre à jour un utilisateur dans la base de données
fun updateUser(user: User) {
    val db = this.writableDatabase

    val values = ContentValues()
    values.put(COLUMN_NAME, user.name)
    values.put(COLUMN_EMAIL, user.email)

    db.update(TABLE_USERS, values, "$COLUMN_ID = ?", arrayOf(user.id.toString()))

    db.close()
}
```

Requêtes de suppression (DELETE)

Pour supprimer des enregistrements dans la base de données, on utilise la méthode `delete` de la classe `SQLiteDatabase`.

Cette méthode prend en paramètre le nom de la table ainsi que les critères de filtrage (WHERE).

Exemple :

```
fun deleteUser(id: Int): Int {
    val db = this.writableDatabase
    val selection = "$COLUMN_ID = ?"
    val selectionArgs = arrayOf(id.toString())
```

```
    val result = db.delete(TABLE_USERS, selection, selectionArgs)
    db.close()
    return result
}
```

Jointures et sous-requêtes

Il est également possible de faire des jointures entre plusieurs tables avec la méthode `query` en utilisant les clauses INNER JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTER JOIN.

De même, il est possible d'utiliser des sous-requêtes en utilisant la méthode `rawQuery`.

Les différents types de stockages

Il existe plusieurs façons de stocker des données dans une application Kotlin, voici les principales :

1. **Stockage interne**

Le stockage interne permet de stocker des fichiers directement sur le périphérique.

Ces fichiers sont accessibles uniquement par l'application qui les a créés et ils sont automatiquement supprimés lorsque l'application est désinstallée.

Avantages et inconvénients

Les avantages du stockage interne sont :

- Facile d'accès
- Fichiers toujours disponibles, même en mode hors ligne
- Pas besoin de demander la permission de l'utilisateur pour accéder au stockage interne de l'application

Les inconvénients sont :

- Espace limité
- Difficulté à gérer les mises à jour de l'application en rapport avec les données stockées en interne

- Risque de perte des données si l'application est désinstallée

Méthodes d'accès et d'utilisation

Pour accéder au stockage interne dans une application Kotlin, il faut utiliser la classe `Context` et sa méthode `openFileOutput()` pour écrire des données dans un fichier et la méthode `openFileInput()` pour lire des données à partir d'un fichier.

Voici un exemple basique d'utilisation :

```
// Ecrire dans un fichier
val file = "example.txt"
val fileContent = "Bonjour le monde !"
openFileOutput(file, Context.MODE_PRIVATE).use {
    it.write(fileContent.toByteArray())
}

// Lire depuis un fichier
val inputStream: FileInputStream = openFileInput(file)
val content = inputStream.bufferedReader().use(BufferedReader::readText)
```

Stockage externe

Avantages

Le stockage externe permet une grande capacité de stockage pour les données, ce qui est très utile pour les applications qui manipulent des fichiers volumineux.

De plus, il offre une flexibilité de stockage, car il peut être facilement transporté d'un appareil à un autre.

Enfin, il facilite la séparation des données entre les différents appareils, ce qui peut être utile dans certains cas.

Inconvénients

Le stockage externe peut être sujet à des problèmes de sécurité, car il est facilement perdu ou volé.

De plus, l’accès et l’utilisation de ces données peuvent être plus lents que le stockage interne.

Enfin, il est important de veiller à la compatibilité des différents appareils avec le stockage externe.

Méthodes d’accès et d’utilisation

Pour accéder au stockage externe, il faut utiliser des API spécifiques.

Kotlin propose plusieurs bibliothèques pour ce faire, dont Room et SQLite.

Ces bibliothèques facilitent la manipulation de données stockées dans des fichiers externes, en offrant des fonctions optimisées pour l’accès et la manipulation de fichiers.

Accès en lecture et/ou écriture au système de fichiers

Pour accéder aux fichiers, Kotlin fournit la classe File.

Elle permet d’accéder aux fichiers de manière synchronisée, garantissant ainsi l’intégrité des données.

Modes d’accès aux fichiers

Il existe trois modes d’accès aux fichiers :

- **Lecture seule** : permet de lire le contenu d’un fichier.
- **Écriture seule** : permet d’écrire dans un fichier existant ou de créer un nouveau fichier.
- **Lecture et écriture** : permet de lire et d’écrire dans un fichier.

Modes d’ouverture (création, ajout, etc.)

La classe File permet également de spécifier le mode d’ouverture des fichiers :

- **Création** (utilisé avec l’écriture seule) : permet de créer un nouveau fichier.

Si le fichier existe déjà, son contenu est remplacé.

- **Ajout** (utilisé avec l'écriture seule) : permet d'écrire à la fin d'un fichier existant, sans supprimer son contenu existant.
- **Écrasement** (utilisé avec la lecture et l'écriture) : permet de remplacer le contenu d'un fichier existant.

Si le fichier n'existe pas, il est créé.

Utiliser des buffers pour améliorer les performances

Bufferisation en lecture et en écriture

La classe **BufferedReader** permet de lire le contenu d'un fichier en utilisant un buffer.

Le buffer ajoute une mémoire tampon en mémoire, ce qui permet de réduire le nombre d'accès au disque dur.

Pour utiliser cette classe, vous devez créer un objet **FileReader** et le passer en paramètre au constructeur de **BufferedReader**.

Voici un exemple :

```
val fr = FileReader(file)
val br = BufferedReader(fr)
```

La classe **BufferedWriter**, quant à elle, permet d'écrire dans un fichier en utilisant également un buffer.

Pour créer un **BufferedWriter**, vous devez créer un objet **FileWriter** et le passer en paramètre au constructeur de la classe **BufferedWriter**.

Voici un exemple :

```
val fw = FileWriter(file)
val bw = BufferedWriter(fw)
```

Flushing et fermeture des buffers

Une fois que vous avez fini de lire ou d'écrire dans un fichier, il est important de fermer le flux et de vider le buffer.

Pour ce faire, vous pouvez utiliser les méthodes **close()** et **flush()**.

La méthode **flush()** permet de vider le buffer et d'écrire le contenu en mémoire sur le disque dur.

La méthode **close()** permet quant à elle de fermer le flux (et donc de libérer les ressources), tout en vidant le buffer si cela n'a pas déjà été fait.

La gestion des préférences

Les SharedPreferences sont un mécanisme simple pour stocker et récupérer des données de préférence dans une application Android.

Ils permettent de stocker des données simples telles que des chaînes, des entiers, des booléens, etc.

Présentation des SharedPreferences

Les SharedPreferences sont stockées dans un fichier XML sur le périphérique de stockage interne de l'application.

Les données sont stockées sous forme de paires clé-valeur.

Pour accéder aux SharedPreferences, vous devez utiliser l'objet SharedPreferences obtenu à partir de l'objet Context.

Avantages et inconvénients

Les avantages des SharedPreferences sont la simplicité et la rapidité d'accès aux préférences de l'application.

Cependant, les SharedPreferences ne sont pas adaptées au stockage de grandes quantités de données ou de données complexes.

Ils sont également moins sécurisés en raison de leur stockage sur le stockage interne du périphérique.

Cas d’utilisation courants

Les SharedPreferences sont utiles pour stocker des préférences utilisateur simples telles que les paramètres de langue, les paramètres d’affichage ou les paramètres de connexion à un service.

Exemple d’utilisation:

```
// Récupération des SharedPreferences
val sharedPreferences = context.getSharedPreferences(
    "my_preferences", // Nom du fichier
    Context.MODE_PRIVATE // Mode de fonctionnement
)

// Récupération d'une valeur
val defaultValue = "Hello world"
val hello = sharedPreferences.getString("hello", defaultValue)

// Stockage d'une valeur
val editor = sharedPreferences.edit()
editor.putString("hello", "Bonjour monde")
editor.apply()
```

Créer, lire et modifier les préférences

La persistance de données est un élément essentiel dans le développement d’applications sur Kotlin.

Les préférences sont l’une des méthodes les plus simples pour stocker des données.

Les préférences sont utilisées pour stocker des données sous forme de paires clé-valeur.

Les données stockées dans les préférences sont privées à l’application et sont accessibles uniquement à cette dernière.

Création d’un fichier de préférences

Pour créer un fichier de préférences dans Kotlin, nous allons utiliser la classe `SharedPreferences`.

Cette classe est utilisée pour stocker et récupérer des données à partir d'un fichier de préférences.

```
val sharedPreferences = getSharedPreferences("Nom_de_fichier", Context.MODE_PRIVATE)
```

Lecture des valeurs des préférences

Pour lire une valeur de préférence dans Kotlin, nous allons utiliser la classe `SharedPreferences` que nous avons créée précédemment.

Par exemple, pour récupérer une valeur de préférence stockée sous la clé "Nom_utilisateur":

```
val nomUtilisateur = sharedPreferences.getString("Nom_utilisateur", "")
```

Modification des valeurs des préférences

Pour modifier une préférence dans Kotlin, nous utiliserons la méthode `edit()` de la classe `SharedPreferences`.

Par exemple, pour sauvegarder le nom de l'utilisateur dans le fichier de préférences:

```
val editor = sharedPreferences.edit()
editor.putString("Nom_utilisateur", "John Doe")
editor.apply()
```

Gérer les préférences avec les préférences partagées

Les préférences partagées permettent de stocker des données de manière persistante dans l'application.

Cela peut être utile pour stocker des préférences utilisateur telles que le thème de l'application, ou encore des informations qui ne changent pas fréquemment.

Communications avec des systèmes externes

Types de communications dans les applications Android

Services web

Les services web sont des applications qui permettent à des systèmes hétérogènes de communiquer entre eux par le biais du réseau.

Les services web utilisent généralement des protocoles standardisés tels que HTTP, SOAP ou REST.

Pour communiquer avec des services web en Kotlin, les développeurs peuvent utiliser des bibliothèques tierces telles que Retrofit ou Volley.

Communications HTTP

Les communications HTTP sont un moyen de transférer des données entre un client et un serveur.

Kotlin propose des classes intégrées telles que UrlConnection qui permettent aux développeurs de créer des connexions HTTP pour envoyer et recevoir des données en utilisant différents formats tels que JSON, XML ou les simples chaînes.

Communications par sockets

Les communications par sockets permettent à des applications de communiquer en temps réel en envoyant des données entre un client et un serveur via une seule connexion réseau.

Pour ce faire, les développeurs peuvent utiliser des bibliothèques de sockets telles que SocketIO ou OkSocket pour Kotlin.

Accès aux ressources REST

L’accès aux ressources REST est un moyen de transférer des données entre un client et un serveur en utilisant le protocole REST.

Pour ce faire, les développeurs peuvent utiliser des bibliothèques tierces telles que Retrofit, Volley ou Fuel pour Kotlin.

Utilisation de sockets réseau

Les sockets réseau sont un moyen permettant la communication à travers un réseau informatique.

Ils permettent à un programme informatique d’envoyer et de recevoir des données sur un réseau, tel qu’Internet.

Qu’est-ce qu’un socket réseau ?

Un socket réseau peut être défini comme un point de terminaison d’une connexion bidirectionnelle entre deux programmes pratiquant la communication réseau.

Les sockets réseau prennent en charge différents types de protocoles de communication, tels que TCP, UDP, SCTP, etc.

Types de sockets

Il existe deux types principaux de sockets : les sockets clients et les sockets serveurs.

Les sockets clients sont utilisés pour initier une connexion avec un serveur.

Ils envoient une demande de connexion à un serveur spécifique et attendent une réponse.

Une fois la connexion établie, les sockets client et serveur peuvent échanger des données.

Les sockets serveurs, quant à eux, sont utilisés pour accepter de nouvelles connexions de socket client.

Ils sont généralement configurés pour écouter une certaine adresse IP et un port spécifique et attendent de nouvelles connexions.

Avantages et inconvénients

Les sockets réseau sont un moyen flexible et polyvalent de communication en réseau.

Ils offrent une capacité de communication peer-to-peer, sans passer par un serveur central.

Cependant, les sockets réseau peuvent être plus compliqués à mettre en place que d'autres méthodes de communication en réseau.

Création d'un socket réseau en Kotlin

La création d'un socket réseau en Kotlin est très simple.

Tout d'abord, il faut importer les librairies nécessaires, qui sont déjà fournies par le langage Kotlin.

Il s'agit principalement de la librairie `java.net.Socket`.

Une fois que les librairies sont importées, il faut créer un socket, qui permettra d'établir la connexion avec la machine distante.

Importation des librairies

```
import java.net.Socket
```

Création d'un socket

```
val serverName = "localhost" // Nom de la machine distante
val port = 8080 // Numéro de port de la connexion

val socket = Socket(serverName, port)
```

Configuration des options du socket

Une fois le socket créé, il est possible de modifier ses options en fonction de ses besoins.

Par exemple, il est possible d’ajuster le timeout de la connexion, ou encore de modifier le buffer de lecture/écriture.

Modification du timeout

```
val TIMEOUT = 5000 // 5 secondes
socket.soTimeout = TIMEOUT
```

Modification du buffer de lecture/écriture

```
val BUFFER_SIZE = 1024 // 1 Mo
socket.receiveBufferSize = BUFFER_SIZE
socket.sendBufferSize = BUFFER_SIZE
```

Envoi et réception de données avec un socket réseau

Envoi de données

Pour envoyer des données sur un socket réseau, il est nécessaire de créer un objet de type `DataOutputStream` à partir de la sortie du socket.

On peut ensuite utiliser la méthode `writeBytes` pour écrire les données à envoyer:

```
val socket = Socket("localhost", 1234)
val outputStream = socket.getOutputStream()
val writer = DataOutputStream(outputStream)
writer.writeBytes("Bonjour!".toByteArray())
```

Réception de données

Pour recevoir des données sur un socket réseau, il est nécessaire de créer un objet de type `DataInputStream` à partir de l'entrée du socket.

On peut ensuite utiliser la méthode `readBytes` pour lire les données reçues:

```
val socket = Socket("localhost", 1234)
val inputStream = socket.getInputStream()
val reader = DataInputStream(inputStream)
val buffer = ByteArray(1024)
val length = reader.readBytes(buffer)
val message = String(buffer, 0, length)
println(message)
```

Fermeture du socket

Une fois que la communication est terminée, il est important de bien fermer le socket pour libérer les ressources associées.

On peut utiliser la méthode `close` pour fermer le socket:

```
val socket = Socket("localhost", 1234)
...
socket.close()
```

Exemple

```
import java.net.ServerSocket

fun main() {
    val serverSocket = ServerSocket(1234)
    val socket = serverSocket.accept()
    val inputStream = socket.getInputStream()
    val reader = DataInputStream(inputStream)
    val outputStream = socket.getOutputStream()
    val writer = DataOutputStream(outputStream)

    val buffer = ByteArray(1024)
    val length = reader.readBytes(buffer)
    val message = String(buffer, 0, length)
    writer.writeBytes("Bienvenue, $message!".toByteArray())

    writer.close()
```

```
    reader.close()
    output.close()
    input.close()
    socket.close()
}
```

Consommation de services web

Introduction aux services web

Les services web sont des systèmes de communication qui permettent à des applications de se parler mutuellement à travers le réseau.

Ils sont basés sur des protocoles standards tels que HTTP/SOAP et REST.

Définition

Un service web peut être défini comme une méthode de communication entre deux programmes informatiques sur le web.

Il vous permet de demander des données à une source distante et de recevoir une réponse sous la forme d'un fichier JSON ou XML.

Types de services web

Il existe deux types de services web :

1. REST (Representational State Transfer) : c'est un style d'architecture de logiciel qui utilise des requêtes HTTP pour accéder et manipuler des ressources.
2. SOAP (Simple Object Access Protocol) : c'est un protocole de communication basé sur XML, utilisé pour échanger des données structurées entre deux systèmes distants.

Comparaison entre REST et SOAP

Principes de REST

Le principe de REST est de fournir des services web qui exposent des ressources sur le web.

Ces ressources sont identifiées par des URLs et peuvent être manipulées avec des actions HTTP standard (GET, POST, PUT, DELETE, etc.).

Les services REST sont souvent utilisés pour construire des applications basées sur une architecture orientée ressources.

Principes de SOAP

Le principe de SOAP est de fournir des services web qui utilisent un format de message XML pour échanger des données.

Les services SOAP utilisent des opérations et des messages pour décrire leurs fonctions, qui sont ensuite interprétées par le client et le serveur.

OkHttp

OkHttp est une bibliothèque HTTP qui permet la consommation de services web.

Elle peut être utilisée pour effectuer des requêtes HTTP synchrones ou asynchrones.

Exemple

Voici un exemple de requête simple :

```
val client = OkHttpClient()

fun run(url: String) {
  val request = Request.Builder()
      .url(url)
      .build()

  client.newCall(request).execute().use { response ->
    println(response.body!!.string())
  }
}
```

Retrofit

Retrofit est une bibliothèque de type “RESTful web service” qui permet la consommation de services web.

Elle simplifie la tâche en associant chaque requête HTTP à une méthode dans une interface.

Voici un exemple:

```
interface ApiService {
  @GET("users/{id}/repos")
  fun listRepos(@Path("id") userId: Int): Call<List<Repo>>
}

val retrofit = Retrofit.Builder()
    .baseUrl("https://api.github.com/")
    .build()

fun run() {
  val service = retrofit.create(ApiService::class.java)
  val repos = service.listRepos(12).execute().body()
}
```

Communication HTTP

La communication HTTP (HyperText Transfer Protocol) est un protocole de communication utilisé pour échanger des informations sur le réseau.

Les méthodes HTTP sont les actions que l’on peut effectuer sur une ressource d’un serveur web.

Les méthodes principales sont GET, POST, PUT et DELETE.

Les méthodes HTTP

GET

La méthode GET est utilisée pour demander une ressource spécifique sur un serveur web.

Elle est généralement utilisée pour récupérer des données à partir d'un serveur.

POST

La méthode POST est utilisée pour envoyer des données à un serveur.

Elle est généralement utilisée pour envoyer des formulaires en ligne.

PUT

La méthode PUT est utilisée pour mettre à jour une ressource sur un serveur web.

Elle est généralement utilisée pour mettre à jour des informations sur un utilisateur enregistré sur un serveur.

DELETE

La méthode DELETE est utilisée pour supprimer une ressource sur un serveur web.

Elle est généralement utilisée pour supprimer des informations sur un utilisateur enregistré sur un serveur.

Communications par sockets

Les sockets sont des interfaces de communication qui permettent l'échange de données entre des systèmes différents via un réseau.

En Kotlin, il est possible de créer un serveur socket pour recevoir des connexions entrantes et envoyer des données.

Création d'un serveur socket en Kotlin

Importation des librairies

La première étape pour créer un serveur socket en Kotlin est d'importer les librairies nécessaires.

Pour les sockets, nous allons utiliser la librairie java.net.

```
import java.net.ServerSocket
import java.net.Socket
```

Création et configuration du serveur

Ensuite, nous pouvons créer un objet de type ServerSocket et le configurer selon nos besoins.

Pour spécifier le port sur lequel le serveur va écouter, nous pouvons passer en paramètre du constructeur un entier représentant ce port.

```
val server = ServerSocket(8080)
```

Écoute des connexions entrantes

La dernière étape de la création du serveur consiste à écouter les connexions entrantes.

Pour cela, nous allons utiliser une boucle infinie qui va attendre une nouvelle connexion à chaque itération.

Lorsqu'une connexion est établie, un objet Socket est créé pour représenter cette connexion.

```
while (true) {
    val client = server.accept()
    // traitement de la connexion avec le client
}
```

Gestion des clients et de la communication avec les clients

La gestion des clients peut être réalisée en utilisant la classe `ServerSocket`.

Cette classe permet de lancer un serveur socket qui attendra des connexions de clients.

Une fois qu'un client est connecté, on peut gérer la communication avec le client en utilisant la classe `Socket`.

Gestion des connexions

Pour gérer les connexions, on peut utiliser la classe `Socket`.

On peut récupérer les flux d'entrée et de sortie de la connexion en utilisant ses méthodes `getInputStream()` et `getOutputStream()`.

Une fois qu'on a récupéré ces flux, on peut échanger des données avec le client.

Échange de données avec les clients

L'échange de données peut être réalisé en utilisant les flux d'entrée et de sortie de la connexion.

On peut envoyer des données au client en écrivant dans le flux de sortie, et recevoir des données du client en lisant dans le flux d'entrée.

Exemple :

```
val serverSocket = ServerSocket(8080)
val client = serverSocket.accept()

val input = BufferedReader(InputStreamReader(client.getInputStream()))
val output = PrintWriter(client.getOutputStream())

output.println("Bienvenue sur le serveur!")
output.flush()

val message = input.readLine()
println("Le client a envoyé le message suivant: $message")
```

```
client.close()
serverSocket.close()
```

Fermeture des connexions

Il est important de fermer les connexions avec les clients une fois qu'on a terminé la communication.

On peut fermer une connexion en utilisant la méthode `close()` de la classe `Socket`.

Exemple de fermeture de connexion :

```
client.close()
```

Exemple d'application de chat en temps réel avec des sockets

Dans cet exemple, nous allons créer une application de chat en temps réel qui permettra à plusieurs utilisateurs de communiquer entre eux.

Architecture de l'application

L'application de chat en temps réel se compose d'un serveur et de plusieurs clients.

Le serveur est responsable de la gestion des messages entre les clients.

Mise en place du serveur

La première étape consiste à mettre en place le serveur de chat.

Nous allons créer une classe `Server` qui sera responsable de la communication avec les clients.

Voici le code pour créer cette classe:

```
import java.net.ServerSocket

class Server(private val port: Int) {
```

```
    fun start() {
        val serverSocket = ServerSocket(port)
        println("Serveur démarré sur le port $port")

        while (true) {
            val client = serverSocket.accept()

            // Do something with the client
        }
    }
}
```

Mise en place du client

Une fois que le serveur est en place, nous pouvons maintenant mettre en place les clients.

Nous allons créer une classe `Client` qui sera responsable de la communication avec le serveur.

Voici le code pour créer cette classe:

```
import java.net.Socket

class Client(private val serverAddress: String, private val serverPort: Int) {

    fun connect() {
        val socket = Socket(serverAddress, serverPort)

        // Do something with the socket
    }
}
```

Nous avons une fonction `connect` qui s’occupe de se connecter au serveur.

Gestion des messages

Maintenant que nous avons mis en place le serveur et les clients, nous devons nous assurer que les messages sont correctement gérés entre les différents utilisateurs.

Voici le code pour gérer l’envoi de messages par un client:

```
import java.io.OutputStream

class Client(private val serverAddress: String, private val serverPort: Int) {

    fun sendMessage(message: String) {
        val socket = Socket(serverAddress, serverPort)
        val outputStream: OutputStream = socket.outputStream

        outputStream.write(message.toByteArray())
        outputStream.flush()

        socket.close()
    }
}
```

Nous avons une nouvelle fonction dans la classe `Client`, `sendMessage`, qui s'occupe d'envoyer un message au serveur.

Voici le code pour gérer la réception de messages côté serveur:

```
import java.net.Socket

class Server(private val port: Int) {

    fun start() {
        val serverSocket = ServerSocket(port)
        println("Serveur démarré sur le port $port")

        while (true) {
            val client = serverSocket.accept()

            val input = client.getInputStream()
            val buffer = ByteArray(1024)

            while (input.read(buffer) != -1) {
                val message = String(buffer).trim()
                println("Nouveau message reçu: $message")
            }

            client.close()
        }
    }
}
```

Nous avons modifié la boucle `while` de la classe `Server` pour gérer la réception des messages.

Nous utilisons la méthode `getInputStream` pour récupérer le flux entrant, puis nous utilisons une boucle `while` pour gérer tous les messages entrants.

Exemple d'utilisation

Voici un exemple basique pour illustrer l'utilisation de notre application de chat:

```
fun main() {
    // Démarrer le serveur
    val server = Server(8080)
    server.start()

    // Se connecter au serveur et envoyer un message
    val client = Client("localhost", 8080)
    client.connect()
    client.sendMessage("Bonjour à tous!")
}
```

Dans cet exemple, nous avons créé une instance de la classe `Server` et avons démarré le serveur.

Ensuite, nous avons créé une instance de la classe `Client`, nous avons connecté le client au serveur et nous avons envoyé un message depuis le client.

Accès à des ressources REST et exploitation de données JSON

Introduction aux API REST

Les API REST (Representational State Transfer) sont des interfaces de programmation qui permettent à des systèmes externes de communiquer entre eux.

Elles sont basées sur HTTP et reposent sur l'utilisation de méthodes de requête telles que GET, POST, PUT et DELETE.

Définition

Une API REST est une interface de programmation qui permet à des systèmes externes de communiquer entre eux en utilisant le protocole HTTP.

Les données sont généralement transférées au format JSON (JavaScript Object Notation).

Principes

Les API REST reposent sur les principes suivants :

- Utilisation des méthodes de requête HTTP : GET pour récupérer des données, POST pour créer des ressources, PUT pour mettre à jour des ressources et DELETE pour supprimer des ressources.
- Utilisation de l'URI (Uniform Resource Identifier) pour identifier les ressources.
- Utilisation du format JSON pour transférer les données.

Avantages et inconvénients

Les API REST présentent plusieurs avantages :

- Elles sont simples à mettre en œuvre.
- Elles sont compatibles avec de nombreux langages de programmation.
- Elles permettent d'assurer la séparation des couches (front-end et back-end).

Cependant, elles présentent également des inconvénients :

- Elles peuvent être vulnérables aux attaques de type XSS ou CSRF.
- Elles peuvent être difficiles à maintenir si les ressources évoluent fréquemment.

Utilisation de l'API Gson pour la sérialisation et la désérialisation JSON en Kotlin

---

Introduction à Gson

Gson est une bibliothèque Java open source qui permet de transformer les objets Java en format JSON et vice versa.

En utilisant Gson, vous pouvez sérialiser des objets Java en fichiers JSON pour les stocker ou les envoyer sur le réseau, et vous pouvez également utiliser JSON pour désérialiser des fichiers JSON en objets Java.

Sérialisation JSON

La sérialisation JSON consiste à prendre un objet Java et à le transformer en une chaîne JSON.

Pour sérialiser un objet Java en JSON à l'aide de Gson, vous devez créer un objet Gson et appeler la méthode toJson() avec l'objet Java à sérialiser en tant que paramètre.

Voici un exemple :

```
data class Person(val name: String, val age: Int)

val person = Person("John Doe", 30)
val gson = Gson()
val personJson = gson.toJson(person)

println(personJson)
```

Ce code prend un objet Person et le transforme en une chaîne JSON.

La chaîne JSON résultante sera :

```
{
   "name": "John Doe",
   "age": 30
}
```

Il est important de noter que Gson utilise les noms des propriétés de l'objet Java pour générer les noms des clés JSON.

Si vous voulez personnaliser la conversion, vous pouvez annoter les propriétés avec des noms de clés JSON personnalisés.

Désérialisation JSON

La désérialisation JSON consiste à prendre une chaîne JSON et à la transformer en un objet Java.

Pour désérialiser un fichier JSON en objet Java à l'aide de Gson, vous devez créer un objet Gson et appeler la méthode fromJson() avec la chaîne JSON à désérialiser en tant que paramètre.

Voici un exemple :

```
val gson = Gson()
val personJson = "{ \"name\": \"John Doe\", \"age\": 30 }"
val person = gson.fromJson(personJson, Person::class.java)

println(person.name) // affiche "John Doe"
println(person.age) // affiche 30
```

Ce code prend une chaîne JSON et la transforme en un objet Person.

Il utilise la méthode fromJson() de Gson et spécifie le type de l'objet (Person::class.java) pour que Gson sache comment désérialiser le fichier JSON.

Personnalisation de la conversion

Enfin, vous pouvez personnaliser la conversion en ajoutant des adaptateurs personnalisés à Gson.

Ces adaptateurs peuvent traiter des types de données spécifiques ou modifier la façon dont les données sont sérialisées ou désérialisées.

Voici un exemple d'adaptateur personnalisé qui traite les dates au format ISO8601 :

```
val gson = GsonBuilder()
   .registerTypeAdapter(Date::class.java, Iso8601DateDeserializer())
   .registerTypeAdapter(Date::class.java, Iso8601DateSerializer())
   .create()
```

Ici, nous avons créé un objet GsonBuilder et avons ajouté deux adaptateurs personnalisés pour les dates.

L’adaptateur Iso8601DateDeserializer est appelé pour désérialiser les dates, tandis que l’adaptateur Iso8601DateSerializer est appelé pour sérialiser les dates.

Exemple de base

Pour illustrer les concepts abordés dans ce chapitre, voici un exemple de base qui utilise l’API Gson pour sérialiser et désérialiser un objet Kotlin en format JSON :

```
data class Person(val name: String, val age: Int)

fun main() {
   val person = Person("John Doe", 30)

   // sérialisation JSON avec Gson
   val gson = Gson()
   val personJson = gson.toJson(person)
   println(personJson)

   // désérialisation JSON avec Gson
   val personFromJson = gson.fromJson(personJson, Person::class.java)
   println(personFromJson.name)
   println(personFromJson.age)
}
```

Ce code crée un objet Person, le transforme en format JSON à l’aide de l’API Gson, puis le transforme à nouveau en un objet Person à l’aide de là API Gson.

Les résultats de cette sérialisation et désérialisation sont affichés dans la console.

Voilà pour ce chapitre sur l’utilisation de l’API Gson pour la sérialisation et la désérialisation JSON en Kotlin.

Exemple d’application consommant une API REST et exploitant des données JSON

Imaginons que nous souhaitons récupérer les derniers articles d’un blog via l’API REST disponible à l’adresse `https://blog.example.com/api/last-articles`.

Cette API retourne un tableau d’objets JSON contenant pour chaque article les champs `title`, `body` et `date`.

Présentation de l’exemple

Notre application Kotlin va donc effectuer une requête HTTP GET pour récupérer les derniers articles via l’URL de l’API REST.

Nous allons ensuite extraire les données JSON de la réponse pour les afficher dans une vue de l’application.

Mise en place

Tout d’abord, nous avons besoin de la bibliothèque `org.json.JSONObject` pour manipuler les données JSON.

Cette bibliothèque est incluse dans la bibliothèque standard de Kotlin.

Nous avons également besoin d’une bibliothèque pour effectuer des requêtes HTTP.

Dans cet exemple, nous utilisons la bibliothèque Volley qui peut être incluse à l’aide de Gradle :

```
dependencies {
    implementation 'com.android.volley:volley:1.1.0'
}
```

Enfin, nous allons créer une vue dans notre application pour afficher les derniers articles.

Dans cet exemple, nous utilisons une ListView.

Récupération des données

Nous allons maintenant récupérer les données JSON de l’API via une requête HTTP GET.

Nous utilisons la bibliothèque Volley pour effectuer cette requête :

```
val queue = Volley.newRequestQueue(context)
val url = "https://blog.example.com/api/last-articles"

val request = JsonObjectRequest(Request.Method.GET, url, null,
    Response.Listener { response ->
        // Traitement de la réponse JSON
```

```
    }, Response.ErrorListener { error ->
        // Traitement de l'erreur
    })

queue.add(request)
```

Dans cet exemple, nous avons créé un objet `JsonObjectRequest` représentant notre requête HTTP GET.

Nous avons défini l'URL de l'API ainsi qu'un callback pour traiter la réponse JSON.

En cas d'erreur, nous avons également défini un callback pour traiter le cas d'erreur.

Dans le callback de traitement de la réponse JSON, nous allons extraire les données JSON et les afficher dans la ListView :

```
val articlesList = mutableListOf<Map<String, String>>()

for (i in 0 until response.length()) {
    val article = response.getJSONObject(i)
    val title = article.getString("title")
    val body = article.getString("body")
    val date = article.getString("date")

    val articleMap = mapOf(
        "title" to title,
        "body" to body,
        "date" to date
    )

    articlesList.add(articleMap)
}

val articleAdapter = SimpleAdapter(
    context,
    articlesList,
    R.layout.article_layout,
    arrayOf("title", "date"),
    intArrayOf(R.id.article_title, R.id.article_date)
)

listView.adapter = articleAdapter
```

Dans cet exemple, nous avons créé une liste d'articles `articlesList` pour stocker les données JSON.

Pour chaque objet JSON dans la réponse, nous avons extrait les champs `title`, `body` et `date` pour les stocker dans une Map correspondant à un article `articleMap`.

Nous avons ensuite créé un `SimpleAdapter` pour afficher les articles dans la ListView grâce au Layout `article_layout`.

La méthode `setAdapter` de la ListView permet d'afficher les articles dans la vue.

Multimédia

Introduction au multimédia sur Android

L'un des avantages d'Android est qu'il dispose de fonctionnalités multimédia avancées.

Le développement d'applications multimédia sur Android est donc un sujet d'actualité.

Dans ce chapitre, nous allons introduire les fonctionnalités multimédia d'Android.

Présentation des fonctionnalités multimédia

Les fonctionnalités multimédia d'Android permettent d'effectuer les opérations suivantes :

- La lecture et l'écriture de fichiers audio et vidéo ;
- La capture de photos et de vidéos depuis la caméra ;
- La lecture et l'écriture de données depuis des médias externes ;
- L'extraction d'informations des fichiers multimédias comme la taille de la vidéo, la durée de l'audio, etc. ;
- La gestion de la qualité et du format des médias.

Formats supportés et compatibilité des codecs

Android prend en charge une gamme étendue de formats de fichier multimédia tels que les formats WAV, MP3, MP4, AAC, MKV, et bien d'autres encore.

Les codecs de ces formats peuvent être natifs ou tiers.

La prise en charge des codecs dépend de la version d'Android.

Exemple

Supposons que vous ayez besoin de lire une vidéo dans votre application.

Vous pouvez utiliser la bibliothèque ExoPlayer pour réaliser cette tâche.

Il suffit de créer un objet ExoPlayer et de lui fournir l'adresse de la vidéo à lire.

Voici un exemple de code :

```
val player = ExoPlayer.Builder(context).build()
val uri = Uri.parse("http://example.com/video.mp4")
val mediaItem = MediaItem.fromUri(uri)
player.setMediaItem(mediaItem)
player.prepare()
player.play()
```

Lecture audio

La lecture audio est une fonctionnalité courante dans les applications mobiles, qu'il s'agisse de musique, de podcasts ou d'autres formes de contenus audio.

Dans Kotlin, la classe MediaPlayer est utilisée pour gérer la lecture audio.

Cette classe fournit des méthodes pour contrôler la lecture et gérer les événements.

Utilisation de MediaPlayer pour la lecture audio

Pour commencer à utiliser MediaPlayer, vous devrez créer une instance de la classe et spécifier le fichier audio que vous souhaitez lire.

Vous pouvez le faire en utilisant l'une des méthodes suivantes :

```
val mediaPlayer = MediaPlayer.create(context, R.raw.my_audio_file)
```

ou

```
val mediaPlayer = MediaPlayer()
mediaPlayer.setDataSource(context, myAudioUri)
mediaPlayer.prepare()
```

Gestion des événements et du cycle de vie de MediaPlayer

Pour gérer les événements de MediaPlayer, vous pouvez utiliser la méthode setOnCompletionListener, qui est appelée lorsque la lecture est terminée :

```
mediaPlayer.setOnCompletionListener {
    // Do something when playback is complete
}
```

Vous pouvez également mettre en pause ou arrêter la lecture en utilisant les méthodes pause et stop :

```
mediaPlayer.pause()
mediaPlayer.stop()
```

Pour mieux gérer le cycle de vie de MediaPlayer, vous pouvez l'initialiser dans la méthode onCreate de votre activité et le libérer dans la méthode onDestroy :

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

    mediaPlayer = MediaPlayer.create(this, R.raw.my_audio_file)
}

override fun onDestroy() {
    super.onDestroy()

    mediaPlayer.release()
}
```

Contrôle de la lecture (play, pause, stop, seek)

Pour contrôler la lecture audio, vous pouvez utiliser les méthodes suivantes :

```
mediaPlayer.start() // Démarre la lecture
mediaPlayer.pause() // Met la lecture en pause
mediaPlayer.stop() // Arrête la lecture
mediaPlayer.seekTo(positionInMillis) // Positionne la lecture à un moment donné (en millis
econdes)
```

Lecture vidéo

La lecture de vidéo est très importante dans la programmation multiplateforme.

Dans Kotlin, nous utilisons la classe VideoView pour lire les vidéos.

La classe VideoView est une sous-classe de SurfaceView qui peut être utilisée pour lire les vidéos et elle est compatible avec plusieurs formats de fichiers vidéo tels que MP4 et 3GP.

Utilisation de VideoView et MediaController pour la lecture vidéo

La première étape pour la lecture vidéo consiste à inclure la balise VideoView dans votre fichier de design.

Il doit être configuré avec un URL ou un chemin vers le fichier vidéo que vous souhaitez lire.

Après l’ajout du VideoView, le MediaController doit également être ajouté.

Il s’agit d’un élément de l’interface utilisateur qui facilite le contrôle de la lecture vidéo à l’utilisateur.

Il permet de contrôler la lecture (play, pause, stop, seek) de la vidéo.

Gestion des événements et du cycle de vie de VideoView

La classe VideoView est liée aux événements, tels que les événements de lecture, qui se produisent lors de la lecture d’une vidéo.

Nous pouvons écouter les événements en utilisant les écouteurs associés à la classe VideoView.

La gestion du cycle de vie de VideoView est très importante pour éviter les problèmes de performances dans nos applications.

Par exemple, nous pouvons arrêter la lecture vidéo en cours lors de la pause ou de la destruction de l'activité qui contient le VideoView.

Contrôle de la lecture (play, pause, stop, seek)

Le MediaController permet aux utilisateurs de contrôler la lecture de la vidéo en utilisant des boutons pour la lecture, la pause, l'arrêt et le retour.

En tant que développeur, nous avons besoin de comprendre comment utiliser ces boutons pour contrôler efficacement la lecture de la vidéo.

Utilisation de ExoPlayer (bibliothèque externe)

Présentation de ExoPlayer et avantages par rapport à MediaPlayer

ExoPlayer est une bibliothèque externe pour la lecture de contenu multimédia sous Android.

Contrairement à MediaPlayer, ExoPlayer est modulaire, ce qui signifie que vous pouvez choisir les fonctionnalités spécifiques dont vous avez besoin et utiliser uniquement celles-ci.

Les avantages d'ExoPlayer par rapport à MediaPlayer sont nombreux, notamment:

- La prise en charge de formats de fichiers multimédias plus larges
- La gestion plus efficace de la mémoire tampon et du décodage
- La possibilité de personnaliser l'interface utilisateur de la lecture

Intégration et utilisation d'ExoPlayer pour la lecture audio et vidéo

Pour intégrer ExoPlayer dans votre application Android, vous devez ajouter les dépendances dans votre fichier build.gradle, puis initialiser ExoPlayer dans votre code Kotlin.

Voici un exemple de code pour la lecture d'un fichier audio avec ExoPlayer:

```
val player = SimpleExoPlayer.Builder(context).build()
val mediaItem = MediaItem.fromUri("https://example.com/audio.mp3")
player.setMediaItem(mediaItem)
player.prepare()
player.play()
```

Personnalisation et contrôle de la lecture avec ExoPlayer

ExoPlayer offre de nombreuses options de personnalisation pour améliorer l'expérience de lecture multimédia dans votre application.

Par exemple, vous pouvez ajouter des boutons de lecture, de pause et d'avance/retour rapide personnalisés à votre interface utilisateur.

En outre, ExoPlayer propose des API pour contrôler la lecture, telles que:

- La possibilité de mettre en pause, de reprendre ou d'arrêter la lecture.
- La possibilité d'accélérer ou de ralentir la lecture.
- La possibilité de sauter en avant ou en arrière dans le contenu.

Voici un exemple de code pour contrôler la lecture avec ExoPlayer:

```
// Pause
player.pause()

// Reprise
player.play()

// Sauter en avant de 10 secondes
player.seekTo(player.currentPosition + 10000)
```

Exemple

Voici un exemple de lecture d’une vidéo avec ExoPlayer:

```
val player = SimpleExoPlayer.Builder(context).build()
val mediaItem = MediaItem.fromUri("https://example.com/video.mp4")
player.setMediaItem(mediaItem)
player.prepare()
player.play()
```

En conclusion, ExoPlayer est une bibliothèque externe efficace et personnalisable pour la lecture de contenu multimédia sous Android.

Avec les fonctionnalités qu’il offre, il est facile de contrôler et personnaliser l’expérience de lecture multimédia dans votre application.

Affichage de graphiques

L’affichage de graphiques est l’un des domaines clés de la programmation multimédia.

En Kotlin, cela peut être accompli à l’aide de la combinaison de Canvas et Paint.

Utilisation de Canvas et Paint pour dessiner des formes et du texte

Canvas est un objet qui fournit une zone de dessin pour les graphiques.

Paint est un objet qui fournit des fonctions pour le dessin de formes et de texte sur le Canvas.

Voici un exemple de code simple qui dessine un cercle rouge sur un Canvas :

```
// Création d'un objet Paint rouge
val paint = Paint().apply {
    color = Color.RED
}

// Dessin d'un cercle sur le Canvas
canvas.drawCircle(100f, 100f, 50f, paint)
```

Gestion des événements tactiles pour interagir avec les graphiques

Pour interagir avec les graphiques, nous pouvons utiliser la gestion des événements tactiles.

En Kotlin, cela peut être accompli en mettant en œuvre les méthodes onTouchEvent () et onDraw () de la classe View.

```
override fun onTouchEvent(event: MotionEvent): Boolean {
    when (event.action) {
        MotionEvent.ACTION_DOWN -> {
            // Lorsque l'utilisateur appuie sur l'écran
        }
        MotionEvent.ACTION_MOVE -> {
            // Lorsque l'utilisateur déplace son doigt sur l'écran
        }
        MotionEvent.ACTION_UP -> {
            // Lorsque l'utilisateur soulève son doigt de l'écran
        }
    }
    return true
}

override fun onDraw(canvas: Canvas) {
    // Dessinez des graphiques dans le Canvas
}
```

Création de vues personnalisées pour l’affichage de graphiques

En Kotlin, nous pouvons créer des vues personnalisées pour afficher des graphiques à l’aide de la classe View.

```
class CustomGraphView(context: Context?) : View(context) {
    override fun onDraw(canvas: Canvas) {
        // Code de dessin mis ici
    }
}
```

Enregistrement audio et vidéo

L’enregistrement audio et vidéo est une fonctionnalité clé pour de nombreuses applications multimédias.

Kotlin offre plusieurs outils pour faciliter l’enregistrement audio et vidéo.

Utilisation de MediaRecorder pour l’enregistrement audio

La classe `MediaRecorder` est l’outil principal pour l’enregistrement audio en Kotlin.

Pour l’utiliser, il faut d’abord gérer les permissions nécessaires et le cycle de vie de la `MediaRecorder` dans l’application.

Ensuite, il suffit d’initialiser une instance de `MediaRecorder` avec les options souhaitées et de lancer l’enregistrement.

Voici un exemple de code pour l’enregistrement audio en Kotlin :

```
val recorder = MediaRecorder()
recorder.setAudioSource(MediaRecorder.AudioSource.MIC)
recorder.setOutputFormat(MediaRecorder.OutputFormat.MPEG_4)
recorder.setOutputFile(outputFile.getAbsolutePath())
recorder.setAudioEncoder(MediaRecorder.AudioEncoder.AAC)
recorder.prepare()
recorder.start()
// Arrêter l'enregistrement avec recorder.stop()
```

Enregistrement vidéo avec Camera2 API

Pour l’enregistrement vidéo, Kotlin utilise la `Camera2 API`, qui permet de contrôler la caméra pour capturer des images et des vidéos en haute qualité.

La `Camera2 API` offre également une grande flexibilité pour gérer les paramètres de la caméra et les formats d’enregistrement.

Voici un exemple de code pour l’enregistrement vidéo en Kotlin avec la `Camera2 API` :

```
val captureBuilder = cameraDevice.createCaptureRequest(CameraDevice.TEMPLATE_RECORD)
captureBuilder.addTarget(surface)
captureBuilder.set(CaptureRequest.CONTROL_MODE, CameraMetadata.CONTROL_MODE_AUTO)
cameraDevice.createCaptureSession(listOf(surface), object : CameraCaptureSession.StateCall
back() {
```

```
    override fun onConfigured(session: CameraCaptureSession) {
        val builder = session.device.createCaptureRequest(CameraDevice.TEMPLATE_RECORD)
        builder.addTarget(surface)
        val surfaces = listOf(surface, recorder.surface)
        session.setRepeatingRequest(builder.build(), null, null)
        recorder.start()
    }
    override fun onConfigureFailed(session: CameraCaptureSession) {
        // Gerer les erreurs de configuration
    }
}, null)
```