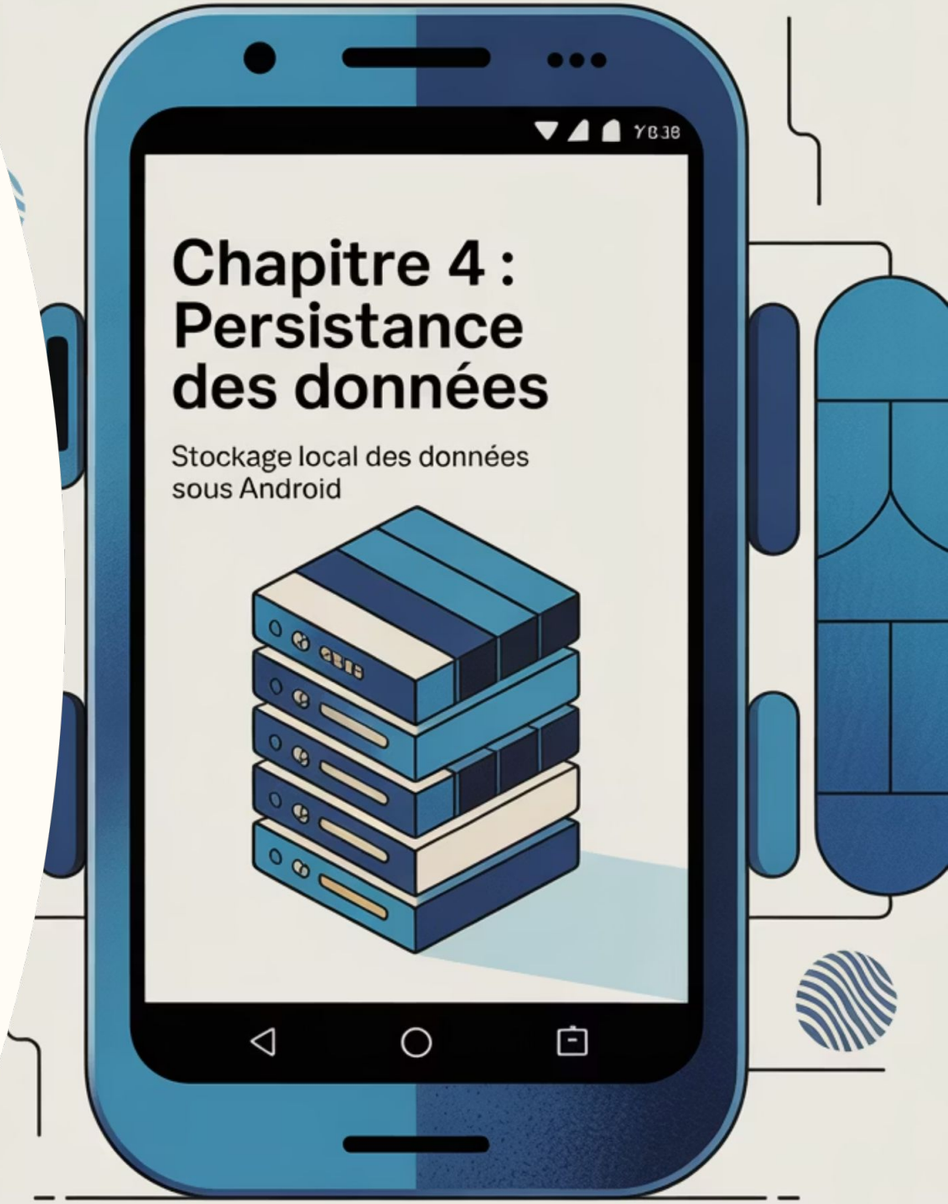
Chapitre 4 : Persistance des données

Stockage local des données sous Android



Persistance des Données

Ce chapitre explore les différentes méthodes de stockage et de persistance des données au sein des applications Android, essentielles pour gérer l'information de manière durable et efficace.

Nous aborderons les points suivants :

01

Accès aux Fichiers

Comment lire, écrire et stocker des données dans le système de fichiers de l'appareil, gérant ainsi les informations de manière locale.

02

Système de Fichiers & Préférences

Manipulation du système de fichiers interne et externe, et gestion des préférences utilisateur pour une expérience personnalisée.

03

Bases de Données SQLite

Introduction et utilisation des bases de données SQLite, une solution robuste pour le stockage structuré de grandes quantités de données.

04

Mode Offline

Stratégies et meilleures pratiques pour développer des applications mobiles fonctionnant sans connexion internet, garantissant l'accès aux données à tout moment.

Persistance des Données

Comprendre comment une application Android peut conserver ses données au-delà d’une seule exécution, garantissant la disponibilité et l'intégrité des informations même après la fermeture de l'application ou le redémarrage de l'appareil. Ce chapitre explorera les différentes options de **stockage locale** offertes par Android, de la simple sauvegarde de préférences aux bases de données complexes.

4.1 Accès aux fichiers : Lecture, Écriture, Stockage

Android offre la possibilité de stocker des données directement dans des fichiers locaux sur l'appareil. Pour cela, deux types d'espaces de stockage distincts sont disponibles, chacun avec ses particularités.

Stockage Interne

Cet espace est propre à l'application, ce qui signifie que les données y sont isolées et non accessibles par d'autres applications. Il est fortement recommandé pour le stockage des informations confidentielles ou des données essentielles au fonctionnement exclusif de votre application.

Stockage Externe

Ce stockage est partagé avec d'autres applications et est donc adapté aux fichiers volumineux ou aux contenus que l'utilisateur souhaite partager (photos, documents). L'accès à cet espace nécessite la gestion appropriée des permissions utilisateur.

Les opérations fondamentales que nous explorerons incluent la **lecture** (récupération de données), l'**écriture** (enregistrement d'informations) et le **stockage** (conservation durable des données même après la fermeture de l'application).

4.1 Bonnes pratiques pour la gestion des fichiers

La manipulation des fichiers nécessite une attention particulière pour garantir la stabilité et la sécurité de votre application. Adoptez ces bonnes pratiques :

1 Vérification d'existence

Toujours vérifier la présence d'un fichier avant toute tentative de lecture pour éviter les exceptions et les erreurs inattendues.

2 Fermeture des flux

Il est crucial de fermer systématiquement tous les flux (`InputStream`, `OutputStream`) après usage afin de libérer les ressources système.

3 Protection des données sensibles

Pour les informations confidentielles, utilisez le stockage interne et évitez absolument le stockage externe, potentiellement accessible par d'autres applications.

4 Gestion des erreurs

Mettez en place des blocs `try-catch` robustes pour gérer les cas d'erreurs (fichier manquant, espace insuffisant, permissions refusées).

5 Maintenance des fichiers

Prévoyez des mécanismes pour la suppression ou la mise à jour régulière des fichiers obsolètes afin d'optimiser les performances et l'espace de stockage.

4.2 Manipulation du système de fichiers et préférences utilisateurs

La gestion des données passe par deux approches fondamentales : le système de fichiers pour les documents et les informations structurées, et les préférences utilisateurs pour les paramètres et les petites données clés-valeur.

Système de fichiers Android

Android fournit un environnement sécurisé pour le stockage des fichiers, essentiel pour la persistance des données. Comprendre son fonctionnement est crucial.

Structure Hiérarchique

Le système est organisé comme un arbre (racine, dossiers, fichiers), facilitant une gestion logique et intuitive des données.

Sandboxing

Chaque application dispose de son propre espace de stockage interne isolé, garantissant la confidentialité et l'intégrité de ses données.

Accès API

Des API Android dédiées permettent aux applications d'accéder et de manipuler des dossiers spécifiques pour des opérations de lecture et d'écriture sécurisées.

Préférences Utilisateurs

Les SharedPreferences sont la méthode standard et la plus simple pour stocker des paires clé-valeur discrètes, souvent utilisées pour les configurations de l'application ou les états légers.

Utilisation Simple

Idéal pour des paramètres comme le thème de l'application, l'état de connexion de l'utilisateur ou de petites données de préférence qui n'évoluent pas fréquemment.

Données Légères

Ce mécanisme n'est pas conçu pour stocker de grandes quantités de données ou des structures complexes. Il est optimisé pour des informations clés-valeur rapides d'accès et de petite taille.

Persistance Facile

Les données stockées via SharedPreferences sont sauvegardées même après la fermeture de l'application ou le redémarrage de l'appareil, garantissant ainsi la persistance des réglages utilisateur.

SharedPreferences

Les SharedPreferences sont un mécanisme simple pour stocker des données légères sous forme de paires clé-valeur, utilisées principalement pour les préférences utilisateur et les réglages d'application.

A. Obtenir l'Instance

```
SharedPreferences prefs = getSharedPreferences("MonAppPrefs",
Context.MODE_PRIVATE);
```

Cette ligne récupère une instance de `SharedPreferences` pour stocker des données clé-valeur. "MonAppPrefs" est le nom du fichier XML où les préférences sont stockées, et `MODE_PRIVATE` assure que seul votre application peut y accéder.

B. Lire des Données

Utilisez les méthodes `get` de `SharedPreferences` pour lire les données stockées. Chaque méthode nécessite une clé et une valeur par défaut, utilisée si la clé n'existe pas ou n'est pas trouvée, garantissant ainsi la stabilité de l'application.

```
String nom = prefs.getString("nomUtilisateur", "Invité");
int age = prefs.getInt("age", 0);
boolean notifs = prefs.getBoolean("notificationsActives", false);
```

C. Supprimer des Données

```
editor.remove("nomUtilisateur");

editor.apply();
```

Utiliser `remove()` pour supprimer une clé spécifique.

D. Écrire des Données

Pour ajouter ou modifier des paires clé-valeur, obtenez un `SharedPreferences.Editor`, effectuez vos modifications, puis appliquez-les.

```
SharedPreferences.Editor editor = prefs.edit();
editor.putString("nomUtilisateur", "John Doe");
editor.putInt("age", 30);
editor.putBoolean("notificationsActives", true);
editor.apply();
```

apply() vs commit()

apply()

- Asynchrone
- Plus rapide
- Pas de valeur de retour

commit()

- Synchrone
- Bloque le thread
- Retourne un booléen

Recommandation : Utiliser apply() dans la plupart des cas.

4.3 Bonnes pratiques des préférences utilisateurs

Pour une gestion optimale et sans accroc des `SharedPreferences`, suivez ces recommandations essentielles :

Données Simples

Ne stockez pas de données complexes (objets, listes) dans les préférences. Elles sont conçues pour des paires clé-valeur simples et légères.

Formats Structurés

Bien qu'Android utilise XML en interne, privilégiez des valeurs de type primitif. Pour des structures plus complexes, orientez-vous vers d'autres solutions de persistance.

Clés Explicites

Utilisez des noms de clés clairs et descriptifs. Cela facilite la lecture, la maintenance et l'évite les conflits entre développeurs.

Qestion du Stockage

Supprimez les préférences devenues inutiles (ex: après une désinstallation, une déconnexion) pour éviter l'encombrement et garantir la pertinence des données.

4.4 Accès aux bases de données SQLite

SQLite est une solution de base de données relationnelle légère et efficace, idéale pour la persistance des données directement sur l'appareil Android. Elle permet de gérer des informations structurées de manière autonome, sans nécessiter de serveur externe.

Base de données Embarquée

SQLite est entièrement intégrée à l'application. Elle ne nécessite aucune installation ou configuration serveur, fonctionnant comme un fichier local sur l'appareil.

Stockage Structuré

Elle permet d'organiser et de stocker des données dans des tables, avec des colonnes et des lignes, facilitant la gestion de relations complexes entre les informations.

Autonome et Locale

Son principal avantage est l'absence de serveur, ce qui réduit la complexité de déploiement et permet un fonctionnement hors ligne complet de l'application.

Compatible SQL

SQLite supporte la majorité des commandes SQL standards, ce qui rend son apprentissage et son utilisation intuitifs pour ceux qui connaissent déjà ce langage.

Opérations Clés avec SQLite

La puissance de SQLite réside dans sa capacité à gérer localement les données structurées. Voici les opérations fondamentales que vous effectuerez pour interagir avec votre base de données embarquée :

Initialisation de la Base

Création d'une base de données SQLite locale sur l'appareil, servant de conteneur pour toutes les informations de votre application.

Définition des Tables

Mise en place des structures de données (tables) avec leurs colonnes et types, pour organiser vos informations de manière logique et efficace.

Manipulation des Données

Exécution des requêtes SQL pour insérer, lire, mettre à jour et supprimer des enregistrements dans vos tables.

Gestion du Cycle de Vie

Utilisation de la classe `SQLiteOpenHelper` pour faciliter la création et les mises à jour (migrations) de votre schéma de base de données.

Syntaxe des Opérations Clés SQL dans SQLite

Interagir avec une base de données SQLite dans Android Studio ne se limite pas à des requêtes SQL brutes. Pour une gestion robuste et pratique, notamment la création et la mise à jour du schéma, il est essentiel d'utiliser la classe `SQLiteOpenHelper`. Cette approche encapsule les détails complexes et fournit une API simplifiée pour manipuler votre base de données locale.

Création et Gestion de la Base de Données avec `SQLiteOpenHelper`

`SQLiteOpenHelper` est une classe utilitaire fournie par Android pour faciliter la création, l'ouverture et la gestion des versions de votre base de données. Elle simplifie grandement la tâche de gérer le cycle de vie de la base de données.

1. Étendre `SQLiteOpenHelper`

Créez une classe personnalisée qui hérite de `SQLiteOpenHelper`. C'est dans cette classe que vous définirez la structure de votre base de données et les opérations de base.

```
public class DatabaseHelper extends SQLiteOpenHelper {
    public static final String DATABASE_NAME = "MaBaseDeDonnees.db";
    public static final int DATABASE_VERSION = 1; // Incrémentez pour les mises à jour

    public DatabaseHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);@
```

2. Implémenter `onCreate(SQLiteDatabase db)`

Cette méthode est appelée une seule fois, lors de la première création de la base de données. C'est ici que vous exécutez vos commandes `CREATE TABLE`.

```
    @Override
    public void onCreate(SQLiteDatabase db) {
        String CREATE_TABLE = "CREATE TABLE produits (" +
                              "id INTEGER PRIMARY KEY AUTOINCREMENT," +
                              "nom TEXT," +
                              "prix REAL," +
                              "quantite INTEGER)";
        db.execSQL(CREATE_TABLE);
    }
```

3. Implémenter `onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)`

Appelée lorsque la version de la base de données (définie dans `DATABASE_VERSION`) change. Utilisez-la pour gérer les migrations (ajouter des colonnes, modifier des tables, etc.) en préservant les données existantes, si possible.

```
    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        // Pour l'exemple, nous supprimons et recréons. En production, gérez les
migrations.
        db.execSQL("DROP TABLE IF EXISTS produits");
        onCreate(db);
    }
```

4.Opérations CRUD Pratiques

Ajoutez des méthodes à votre `DatabaseHelper` pour encapsuler les opérations d'insertion, de lecture, de mise à jour et de suppression (CRUD).

1. Insertion de Données (`insertData`)

```
    public boolean insertData(String nom, double prix, int quantite) {
        SQLiteDatabase db = this.getWritableDatabase();
        ContentValues contentValues = new ContentValues();
        contentValues.put("nom", nom);
        contentValues.put("prix", prix);
        contentValues.put("quantite", quantite);
        long result = db.insert("produits", null, contentValues);
        return result != -1; // Retourne true si l'insertion a réussi
    }
```

2. Lecture de Données (`getData`)

```
    public Cursor getData() {
        SQLiteDatabase db = this.getWritableDatabase();
        Cursor res = db.rawQuery("SELECT * FROM produits", null);
        return res;
    }
```

3. Mise à Jour de Données (`updateData`)

```
    public boolean updateData(String id, String nom, double prix, int quantite) {
        SQLiteDatabase db = this.getWritableDatabase();
        ContentValues contentValues = new ContentValues();
        contentValues.put("id", id);
        contentValues.put("nom", nom);
        contentValues.put("prix", prix);
        contentValues.put("quantite", quantite);
        db.update("produits", contentValues, "id = ?", new String[] { id });
        return true;
    }
```

4. Suppression de Données (`deleteData`)

```
    public Integer deleteData(String id) {
        SQLiteDatabase db = this.getWritableDatabase();
        return db.delete("produits", "id = ?", new String[] { id });
    }
```

Utilisation dans une `Activity` Android

Pour interagir avec votre base de données, instanciez `DatabaseHelper` dans votre `Activity` ou `Fragment` et appelez ses méthodes.

```
public class MainActivity extends AppCompatActivity {
    DatabaseHelper myDb;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        myDb = new DatabaseHelper(this);

        // Exemple d'insertion
        boolean isInserted = myDb.insertData("Livre", 19.99, 50);
        if (isInserted) Toast.makeText(this, "Données insérées", Toast.LENGTH_LONG).show();
        else Toast.makeText(this, "Échec de l'insertion", Toast.LENGTH_LONG).show();

        // Exemple de lecture
        Cursor res = myDb.getData();
        if (res.getCount() == 0) {
            // Afficher message "Rien trouvé"
            return;
```

```
    }
        StringBuffer buffer = new StringBuffer();
        while (res.moveToNext()) {
            buffer.append("ID :"+ res.getString(0)+"\n");
            buffer.append("Nom :"+ res.getString(1)+"\n");
            buffer.append("Prix :"+ res.getString(2)+"\n");
            buffer.append("Quantité :"+ res.getString(3)+"\n\n");
        }
        // Afficher toutes les données (par ex. dans un AlertDialog)
        // showMessage("Données", buffer.toString());
```

4.5 Utilisation d’applications mobiles en mode Offline

Le développement d'applications mobiles robustes implique souvent de prévoir des scénarios où la connectivité Internet est limitée ou inexistante. Le mode offline est une fonctionnalité essentielle pour garantir une expérience utilisateur ininterrompue et fiable.

Accès Continu

Permet aux utilisateurs d'interagir avec l'application et ses données, même en l'absence de connexion Internet.

Fiabilité en Mobilité

Essentiel pour garantir une expérience utilisateur fluide dans les zones à faible connectivité, les transports ou sans réseau.

Stockage Localisé

Repose sur des mécanismes de persistance des données directement sur l'appareil : fichiers, préférences partagées ou bases de données SQLite.

Principes du Mode Offline

Le mode offline repose sur des mécanismes bien définis pour assurer une continuité de service et une intégrité des données, même en l'absence de réseau.

Stockage Local

Les données sont stockées et mises à jour directement sur l'appareil, garantissant un accès et des modifications immédiats, même sans connexion.

Synchronisation Intelligente

Dès que la connectivité est rétablie, l'application initie la synchronisation pour transférer les modifications locales et récupérer les mises à jour du serveur distant.

Qestion des Conflits

Des mécanismes sont intégrés pour gérer les éventuels conflits entre les données locales et distantes, assurant ainsi la cohérence et l'intégrité de l'information. La synchronisation peut être différée si nécessaire.

Avantages du mode Offline

L'intégration d'un mode hors ligne offre de nombreux bénéfices, tant pour l'application que pour l'expérience utilisateur finale.

Accessibilité continue

Les utilisateurs peuvent accéder aux fonctionnalités et aux données de l'application à tout moment, sans être contraints par la disponibilité d'une connexion internet.

Meilleure réactivité

Les opérations locales sur les données stockées directement sur l'appareil sont généralement plus rapides que les requêtes réseau, améliorant ainsi la fluidité de l'application.

Réduction de la dépendance au réseau

L'application reste fonctionnelle même dans des zones à faible couverture réseau ou sans aucune connectivité, évitant les frustrations liées aux interruptions de service.

Expérience utilisateur améliorée

La combinaison d'une accessibilité constante et d'une réactivité accrue se traduit par une satisfaction globale plus élevée pour l'utilisateur.

Vers des approches plus modernes : La bibliothèque Room

Bien que SQLite soit la pierre angulaire de la persistance locale sur Android, la complexité de sa manipulation directe peut être source d'erreurs et de code répétitif. Pour simplifier et moderniser le développement de bases de données, Google a introduit des bibliothèques d'abstraction.

Parmi elles, la **bibliothèque Room**, qui fait partie d'Android Jetpack, offre une couche d'abstraction puissante au-dessus de SQLite. Elle permet de travailler avec des objets Java (ou Kotlin) plutôt qu'avec des requêtes SQL brutes, réduisant ainsi le code répétitif et offrant une vérification au moment de la compilation pour les requêtes SQL.

Room simplifie l'interaction avec la base de données, améliore la lisibilité du code et renforce la robustesse des applications grâce à sa gestion des migrations et sa sécurité au moment de la compilation, évitant de nombreux bugs liés à SQLite.