FORMEZ-VOUS AU NUMÉRIQUE AVEC L'OIF

Développez une première application CRUD locale avec Flutter et Sqflite

Formation en développement Mobile
Niveau Intermédiaire

Il s'agit d'un atelier sur le CRUD avec sqflite.

À la fin de l'atelier, les apprenants devraient être capables de créer une application Flutter simple qui permet d'ajouter, afficher, modifier et supprimer des données stockées localement dans une base de données SQLite via le package sqflite

Chapitre 1 - Introduction & Configuration

Objectif de ce module : Comprendre les concepts fondamentaux de CRUD, la nécessité d'une base de données locale pour la persistance des données, le rôle de sqflite, et configurer notre projet Flutter pour pouvoir utiliser ces outils.

---

1. Introduction

Bienvenue dans cet atelier ! Aujourd'hui, nous allons construire une application Flutter simple mais complète qui vous permettra de gérer des données directement sur votre appareil. Nous allons créer une application capable de :

- **Créer** de nouvelles données (par exemple, ajouter une nouvelle note ou un nouvel élément à une liste).
- **Lire** les données existantes (afficher toutes les notes ou éléments).
- **Modifier** les données (changer le contenu d'une note).
- **Supprimer** les données (retirer une note).

Ces quatre opérations forment l'acronyme **CRUD**, un concept fondamental dans le développement logiciel pour interagir avec des sources de données.

---

2. Qu'est-ce que le CRUD ?

CRUD est l'acronyme des quatre opérations de base utilisées pour interagir avec la persistance des données :

- **C** - **Create (Créer) :** Ajouter de nouvelles entrées ou enregistrements dans une base de données ou un système de stockage.
    - *Exemple :* Ajouter une nouvelle tâche à votre liste de tâches.
- **R** - **Read (Lire) :** Récupérer des données existantes à partir du système de stockage. Cela peut être la lecture d'un seul élément spécifique ou d'une collection d'éléments.
    - *Exemple :* Afficher toutes les tâches dans votre liste.
- **U** - **Update (Modifier) :** Changer ou mettre à jour les données existantes.
    - *Exemple :* Marquer une tâche comme terminée ou modifier son nom.

- **D** - **Delete (Supprimer) :** Retirer ou effacer des données du système de stockage.
    - *Exemple :* Supprimer une tâche de votre liste.

La maîtrise de ces quatre opérations est essentielle car elles sont au cœur de presque toutes les applications qui gèrent des informations.

3. Pourquoi une Base de Données Locale ? Pourquoi Sqflite ?

Lorsque vous développez une application, les données que l'utilisateur crée ou utilise doivent souvent être conservées même après la fermeture de l'application ou le redémarrage de l'appareil. C'est ce qu'on appelle la **persistance des données**.

Sans persistance, chaque fois que l'utilisateur ouvre l'application, il recommence à zéro, ce qui n'est pas l'expérience attendue pour la plupart des applications modernes.

Il existe plusieurs façons de stocker des données localement sur un appareil mobile :

1. **Stockage simple clé-valeur :** Pour des préférences utilisateur simples (comme shared_preferences dans Flutter). Utile pour des petits bouts d'information, pas pour des ensembles de données structurés.
2. **Fichiers :** Stocker des données dans des fichiers (texte, JSON, etc.). Utile pour des données non structurées ou des fichiers volumineux (images, vidéos).
3. **Bases de données locales :** Stocker des données structurées dans une base de données intégrée à l'application. C'est idéal pour des listes d'éléments, des catalogues, des notes, etc., où vous avez plusieurs enregistrements avec des champs définis.

Pour notre application CRUD, qui gérera plusieurs "éléments" avec différents attributs (comme un titre et un contenu pour une note), une base de données locale est la solution la plus adaptée.

Sur les plateformes mobiles (Android et iOS), **SQLite** est une base de données relationnelle embarquée très populaire. Elle est légère, rapide, ne nécessite pas de serveur distinct et stocke les données dans un simple fichier sur l'appareil.

Flutter, étant un framework multiplateforme, a besoin d'un moyen de communiquer avec la base de données SQLite native de chaque plateforme. C'est là qu'intervient le package **sqflite**.

sqflite est le plugin Flutter officiel et largement recommandé pour interagir avec les bases de données SQLite. Il fournit une API Dart conviviale pour effectuer toutes les opérations SQL (comme CREATE TABLE, INSERT, SELECT, UPDATE, DELETE) sans avoir à écrire de code natif (Kotlin/Java pour Android, Swift/Objective-C pour iOS).

En résumé, nous utilisons une base de données locale (SQLite via sqflite) pour :

- **Persistance :** Sauvegarder les données de manière durable.
- **Structure :** Organiser les données en tables avec des colonnes définies.
- **Performance :** Accéder et manipuler les données efficacement, même hors ligne.
- **Simplicité :** sqflite nous permet de faire tout cela facilement depuis notre code Dart/Flutter.

4. Configuration du Projet Flutter

Avant de pouvoir utiliser sqflite et commencer à construire notre application, nous devons configurer notre projet Flutter.

Étape 1 : Créer un nouveau projet Flutter

Si vous n'avez pas déjà un projet pour cet atelier, créez-en un.
Vous pouvez le faire via la ligne de commande ou votre IDE (VS Code, Android Studio).

- **Via la ligne de commande :**
  Ouvrez votre terminal ou invite de commandes et exécutez :

```
flutter create my_crud_app
```

Remplacez my_crud_app par le nom que vous souhaitez donner à votre projet.
Naviguez ensuite dans le répertoire du projet :

```
cd my_crud_app
```

- **Via un IDE (VS Code ou Android Studio) :**
  Utilisez les options de menu pour créer un nouveau projet Flutter (File > New > New Flutter Project). Choisissez l'option "Application".

Étape 2 : Ajouter les dépendances nécessaires

Nous devons indiquer à Flutter que notre projet a besoin des packages sqflite, path_provider, et path. Nous faisons cela en modifiant le fichier pubspec.yaml à la racine de votre projet.

Ouvrez le fichier pubspec.yaml et trouvez la section dependencies:. Ajoutez les lignes suivantes sous cupertino_icons: (ou à la suite des autres dépendances si vous en avez déjà) :

```
      dependencies:
  flutter:
    sdk: flutter

  cupertino_icons: ^1.0.2 # Exemple, votre version peut être
différente

  # Dépendances pour la base de données
  sqflite: ^2.0.0+3 # Ou la dernière version stable
  path_provider: ^2.0.11 # Ou la dernière version stable
  path: ^2.0.11 # Ou la dernière version stable


dev_dependencies:
  flutter_test:
    sdk: flutter

  flutter_lints: ^2.0.0 # Exemple, votre version peut être
différente

flutter:
  uses-material-design: true

  # Pour le Module 3, si vous utilisez une DB pré-existante,
ajoutez cette section
  # assets:
  #   - assets/db/
```

Explication des dépendances ajoutées :

- **sqflite :** C'est le package principal qui fournit les fonctions Dart pour interagir avec la base de données SQLite native.
- **path_provider :** Ce package est nécessaire pour trouver les chemins du système de fichiers standard de l'appareil (comme le répertoire des documents) où nous devrions stocker le fichier de notre base de données. On ne peut pas simplement choisir un chemin arbitraire.
- **path :** Ce package fournit des utilitaires pour manipuler les chemins de fichiers (comme joindre des segments de chemin) de manière indépendante du système d'exploitation. Cela garantit que votre code fonctionnera correctement sur Android (qui utilise /) et iOS (qui utilise potentiellement d'autres séparateurs ou conventions).

Important : L'indentation est cruciale en YAML. Assurez-vous que sqflite:, path_provider:, et path: sont alignés verticalement avec cupertino_icons:.

Étape 3 : Obtenir les dépendances

Une fois que vous avez modifié pubspec.yaml, vous devez demander à Flutter de télécharger ces packages et de les rendre disponibles pour votre projet.

- **Via la ligne de commande :**
  Exécutez la commande suivante dans le répertoire de votre projet :

```
flutter pub get
```

- **Via un IDE :**
  Votre IDE (VS Code ou Android Studio) détectera généralement les changements dans pubspec.yaml et vous proposera d'exécuter flutter pub get automatiquement (un bouton "Get Packages" ou similaire apparaîtra). Cliquez dessus.

Vous devriez voir des messages indiquant que Flutter télécharge et met en cache les packages. Une fois terminé, vous êtes prêt à utiliser sqflite et les autres packages dans votre code !

---

5. Petit ajustement dans lib/main.dart

Pour certaines opérations qui interagissent avec les services de la plateforme native (comme celles que sqflite utilise pour ouvrir la base de données sur le système de fichiers), vous devez vous assurer que les "Flutter Widgets Binding" sont initialisés *avant* d'appeler runApp(). C'est une bonne pratique, surtout lorsqu'on travaille avec des plugins qui touchent aux fonctionnalités natives.

Ouvrez le fichier lib/main.dart et ajoutez la ligne `WidgetsFlutterBinding.ensureInitialized();` au début de votre fonction main().

```
import 'package:flutter/material.dart';
// N'oubliez pas d'importer les autres packages si
nécessaire pour les modules suivants
// import 'package:sqflite/sqflite.dart';
// import 'package:path/path.dart';
// import 'package:path_provider/path_provider.dart';

void main() {
  // Assurez-vous que les services de la plateforme Flutter
sont initialisés
  // Ceci est souvent nécessaire pour les plugins qui
interagissent avec les fonctionnalités natives,
  // comme l'accès au système de fichiers pour la base de
données.
```

```
  WidgetsFlutterBinding.ensureInitialized();

  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Mon App CRUD',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: const MyHomePage(title: 'Application de Notes
CRUD'), // Nom de votre page d'accueil
    );
  }
}

// Vous allez créer la page MyHomePage dans les modules
suivants
class MyHomePage extends StatefulWidget {
  const MyHomePage({Key? key, required this.title}) :
super(key: key);

  final String title;

  @override
  State<MyHomePage> createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  // Le code de votre page ira ici
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      ),
      body: Center(
        child: Text('Bienvenue dans votre application CRUD
!'),
      ),
    );
  }
}
```

Explication de WidgetsFlutterBinding.ensureInitialized(); : Cette ligne garantit que l'instance WidgetsBinding est un singleton valide et prête à être utilisée. Elle est nécessaire avant d'utiliser certains plugins qui doivent interagir avec le code natif de la plateforme avant que l'application Flutter ne soit complètement démarrée par runApp(). sqflite, qui accède au système de fichiers et à la base de données native, en fait partie.

Conclusion

Dans ce module, nous avons jeté les bases de notre atelier. Nous avons compris ce qu'est le concept fondamental de CRUD et pourquoi les bases de données locales, en particulier SQLite via sqflite, sont une excellente solution pour la persistance des données dans les applications Flutter. Nous avons également configuré notre projet en ajoutant les dépendances nécessaires (sqflite, path_provider, path) et en ajustant la fonction main pour assurer une bonne initialisation.

Chapitre 2 - Construire l'Interface Utilisateur (Formulaire de Saisie)

Objectif de ce chapitre : Apprendre à créer une interface utilisateur simple dans Flutter pour permettre la saisie de données à l'aide de widgets de formulaire, comprendre leurs propriétés essentielles et mettre en place la validation basique.

1. Pourquoi un Formulaire ?

Dans toute application CRUD, la première étape pour "Créer" une donnée est de permettre à l'utilisateur de la saisir. C'est le rôle d'un formulaire. Un formulaire est une collection de champs où l'utilisateur entre des informations (texte, nombres, dates, choix, etc.) et un moyen de soumettre ces informations (un bouton).

Dans Flutter, nous allons utiliser un ensemble de widgets spécialement conçus pour les formulaires.

2. Structure de Base de la Page

Notre formulaire aura besoin d'être contenu dans une page. Une page typique dans une application Material Design utilise le widget Scaffold.

- **Scaffold :** Fournit une structure de base pour une page (AppBar, corps principal, FloatingActionButton, etc.).
- **AppBar :** La barre en haut de l'écran, souvent utilisée pour le titre de la page et les actions.
- **body :** La zone principale où se trouve le contenu de votre page.
- **Padding :** Un widget utile pour ajouter de l'espace autour de son enfant, évitant que le contenu ne touche les bords de l'écran.
- **ListView ou Column :** Pour organiser les widgets les uns au-dessus des autres. Pour un formulaire, ListView est souvent préférable car il gère automatiquement le défilement si le contenu dépasse la taille de l'écran (utile si le clavier virtuel apparaît et réduit l'espace disponible).

Commençons par créer un nouveau fichier pour notre page de formulaire (par exemple, lib/note_form_page.dart).

Code : lib/note_form_page.dart (Début)

```
import 'package:flutter/material.dart';
```

```
// Nous aurons besoin que cette page soit un StatefulWidget
car
// nous allons gérer l'état du formulaire (validation, valeurs
saisies)
class NoteFormPage extends StatefulWidget {
  const NoteFormPage({Key? key}) : super(key: key);

  @override
  _NoteFormPageState createState() => _NoteFormPageState();
}

class _NoteFormPageState extends State<NoteFormPage> {

  // Ici nous allons ajouter le code pour le formulaire

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Nouvelle Note'), // Ou Modifier
Note
      ),
      body: Padding( // Ajoute un peu d'espace autour du
contenu
        padding: const EdgeInsets.all(16.0),
        child: ListView( // Utilisation de ListView pour le
défilement si nécessaire
          children: <Widget>[
            // Ici iront les champs du formulaire et le bouton
          ],
        ),
      ),
    );
  }
}
```

3. Le Widget Form

Pour gérer un formulaire de manière efficace dans Flutter (en particulier pour la validation et la sauvegarde des champs), il est recommandé d'utiliser le widget Form.

- Le widget Form agit comme un conteneur pour vos champs de formulaire (TextFormField, Checkbox, DropdownButtonFormField, etc.).
- Il permet de valider tous les champs à la fois et de sauvegarder leurs valeurs.
- Pour interagir avec le Form (déclencher la validation ou la sauvegarde), nous avons besoin d'une GlobalKey<FormState>. Cette clé nous donne accès à l'état actuel du formulaire (FormState).

Code : lib/note_form_page.dart (Ajout de la clé et du widget Form)

```
      import 'package:flutter/material.dart';

class NoteFormPage extends StatefulWidget {
  const NoteFormPage({Key? key}) : super(key: key);

  @override
  _NoteFormPageState createState() => _NoteFormPageState();
}

class _NoteFormPageState extends State<NoteFormPage> {
  // 1. Déclarez une GlobalKey pour le Formulaire.
  // C'est nécessaire pour accéder et gérer l'état du
Formulaire.
  final _formKey = GlobalKey<FormState>();

  // Variables pour stocker les valeurs du formulaire après
sauvegarde
  String _title = '';
  String _content = '';

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Nouvelle Note'),
      ),
      body: Padding(
        padding: const EdgeInsets.all(16.0),
        child:
            // 2. Enveloppez vos champs de formulaire dans un
widget Form.
            Form(
          key: _formKey, // Assurez-vous d'assigner la clé au
Form
          child: ListView(
            // Utilisez Column si vous êtes certain que le
contenu ne dépassera jamais l'écran
            children: <Widget>[
              // Ici iront les champs du formulaire et le
bouton
            ],
          ),
        ),
      ),
    );
  }
}
```

Explication :

- final _formKey = GlobalKey<FormState>(); : Crée une clé unique qui peut être utilisée pour identifier le widget Form dans l'arbre de widgets et accéder à son FormState.
- Form(key: _formKey, ...) : Associe cette clé au widget Form.
- String _title = ''; String _content = ''; : Déclare des variables dans l'état de notre widget pour stocker les données saisies une fois que le formulaire est validé et sauvegardé.

4. Les Champs de Formulaire (TextFormField)

Le widget le plus couramment utilisé pour la saisie de texte est TextFormField. C'est une version enrichie du widget TextField qui s'intègre bien avec le widget Form.

Voici les propriétés essentielles de TextFormField pour cet atelier :

- **decoration (Type InputDecoration) :** Permet de personnaliser l'apparence du champ (bordures, icônes, labels, texte d'aide, etc.). C'est ici que vous définissez labelText et hintText.
    - labelText : Un libellé qui s'anime au-dessus du champ lorsqu'il est sélectionné.
    - hintText : Un texte d'exemple qui disparaît lorsque l'utilisateur commence à taper.
- **keyboardType (Type TextInputType) :** Configure le type de clavier affiché sur l'appareil (texte, numérique, email, multiligne, etc.).
- **validator (Type FormFieldValidator<String>?) :** Une fonction qui est appelée lorsque _formKey.currentState.validate() est exécuté.
    - Elle reçoit la valeur actuelle du champ (String? value).
    - Elle doit retourner null si la valeur est valide.
    - Elle doit retourner un message d'erreur (String) si la valeur est invalide. Ce message sera affiché sous le champ.
- **onSaved (Type FormFieldSetter<String>?) :** Une fonction qui est appelée lorsque _formKey.currentState.save() est exécuté.
    - Elle reçoit la valeur actuelle du champ (String? value).

    - C'est le bon endroit pour stocker la valeur du champ dans une variable (comme nos variables _title et _content).

- **controller (Type TextEditingController?) :** Une alternative ou un complément à onSaved. Un TextEditingController permet de :
    - Récupérer la valeur du champ en temps réel (controller.text).
    - Définir la valeur initiale du champ (controller.text = '...').
    - Effacer le champ (controller.clear()).
    - Pour cet atelier, nous allons principalement utiliser onSaved car il s'intègre bien avec le flux validation/sauvegarde déclenché par le bouton. Le controller sera utile plus tard pour effacer le formulaire après insertion.

Ajoutons deux TextFormField (pour le titre et le contenu de la note) à notre ListView à l'intérieur du Form.

Code : lib/note_form_page.dart (Ajout des TextFormField)

```
      import 'package:flutter/material.dart';

class NoteFormPage extends StatefulWidget {
  const NoteFormPage({Key? key}) : super(key: key);

  @override
  _NoteFormPageState createState() => _NoteFormPageState();
}

class _NoteFormPageState extends State<NoteFormPage> {
  final _formKey = GlobalKey<FormState>();

  String _title = '';
  String _content = '';

  // Controllers pour effacer les champs après insertion
(Optionnel mais pratique)
  final TextEditingController _titleController =
TextEditingController();
  final TextEditingController _contentController =
TextEditingController();


  // N'oubliez pas de disposer des controllers quand le widget
est supprimé
  @override
  void dispose() {
    _titleController.dispose();
    _contentController.dispose();
    super.dispose();
```

```
  }


  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Nouvelle Note'),
      ),
      body: Padding(
        padding: const EdgeInsets.all(16.0),
        child: Form(
          key: _formKey,
          child: ListView(
            children: <Widget>[
              // Champ pour le Titre
              TextFormField(
                controller: _titleController, // Associer le
controller
                decoration: const InputDecoration(
                  labelText: 'Titre',
                  hintText: 'Entrez le titre de la note',
                  border: OutlineInputBorder(), // Ajoute une
bordure stylisée
                ),
                // Validation : S'assurer que le champ n'est
pas vide
                validator: (value) {
                  if (value == null || value.isEmpty) {
                    return 'Veuillez entrer un titre';
                  }
                  return null; // Retourne null si la
validation réussit
                },
                // Sauvegarde la valeur lorsque
formKey.currentState.save() est appelé
                onSaved: (value) {
                  _title = value ?? ''; // Utilise une chaîne
vide si la valeur est null
                },
              ),
              const SizedBox(height: 16.0), // Ajoute un
espace vertical entre les champs

              // Champ pour le Contenu
              TextFormField(
                 controller: _contentController, // Associer
le controller
                decoration: const InputDecoration(
                  labelText: 'Contenu',
                  hintText: 'Entrez le contenu de la note',
```

```
                  border: OutlineInputBorder(),
                ),
                 keyboardType: TextInputType.multiline, //
Clavier pour plusieurs lignes
                 maxLines: null, // Permet au champ de grandir
avec le contenu (ou fixer un nombre)
                // Validation : S'assurer que le champ n'est
pas vide
                validator: (value) {
                  if (value == null || value.isEmpty) {
                    return 'Veuillez entrer un contenu';
                  }
                  return null;
                },
                // Sauvegarde la valeur
                onSaved: (value) {
                  _content = value ?? '';
                },
              ),

              // Le bouton sera ajouté ensuite
            ],
          ),
        ),
      ),
    );
  }
}
```

Explication des ajouts :

- Nous avons ajouté un TextFormField pour le titre et un pour le contenu.
- Chacun a une decoration pour l'apparence et les libellés. Nous avons ajouté border: OutlineInputBorder() pour un look plus moderne.
- Chacun a une fonction validator simple qui vérifie si le champ est vide.
- Chacun a une fonction onSaved qui stocke la valeur du champ dans la variable _title ou _content correspondante dans l'état du widget.
- const SizedBox(height: 16.0) : Un widget simple pour ajouter de l'espace vide entre les deux champs.
- keyboardType: TextInputType.multiline et maxLines: null : Permet au champ de contenu d'afficher et de permettre la saisie sur plusieurs lignes.
- TextEditingControllers et dispose(): Nous avons ajouté des controllers. Ils sont particulièrement utiles si vous voulez effacer le formulaire après l'enregistrement, ou pré-remplir le formulaire pour la modification. Il est *essentiel* de les dispose()

dans la méthode dispose() de votre StatefulWidget pour éviter les fuites de mémoire.

---

5. Le Bouton de Soumission

Enfin, nous avons besoin d'un bouton pour déclencher la validation et la sauvegarde du formulaire. Nous allons utiliser un ElevatedButton.

La logique principale sera dans la fonction onPressed du bouton :

1. Accéder à l'état du formulaire via _formKey.currentState.
2. Appeler validate() : Ceci exécute la fonction validator de *tous* les TextFormField à l'intérieur de ce Form. Si *tous* les validateurs retournent null, validate() retourne true. Si au moins un validateur retourne une String, validate() retourne false et les messages d'erreur s'affichent.
3. Si validate() retourne true (le formulaire est valide), appeler save() : Ceci exécute la fonction onSaved de *tous* les TextFormField à l'intérieur de ce Form. C'est à ce moment que nos variables _title et _content seront remplies avec les données saisies par l'utilisateur.
4. Une fois que les données sont sauvegardées, nous aurons les valeurs dans nos variables _title et _content, prêtes à être utilisées (dans le module suivant, pour l'insertion dans la base de données).

Ajoutons le bouton à la fin de la ListView.

Code : lib/note_form_page.dart (Ajout du ElevatedButton)

```
      import 'package:flutter/material.dart';

class NoteFormPage extends StatefulWidget {
  const NoteFormPage({Key? key}) : super(key: key);

  @override
  _NoteFormPageState createState() => _NoteFormPageState();
}

class _NoteFormPageState extends State<NoteFormPage> {
  final _formKey = GlobalKey<FormState>();

  String _title = '';
  String _content = '';

  final TextEditingController _titleController =
TextEditingController();
```

```
  final TextEditingController _contentController =
TextEditingController();


  @override
  void dispose() {
    _titleController.dispose();
    _contentController.dispose();
    super.dispose();
  }

  // Fonction qui sera appelée quand le bouton est pressé
  void _saveNote() {
    // Vérifie si l'état actuel du formulaire et l'état sont
non-nulls
    if (_formKey.currentState!.validate()) {
      // Si la validation passe, appelle onSaved pour chaque
champ.
      // Cela remplit nos variables _title et _content.
      _formKey.currentState!.save();

      // --- À CE POINT, LES DONNÉES SONT VALIDES ET
DISPONIBLES DANS _title et _content ---
      // Dans les modules suivants, nous allons insérer ces
données dans la base de données ici.

      // Pour l'instant, affichons-les dans la console pour
vérifier
      print('Titre: $_title');
      print('Contenu: $_content');

      // Optionnel : Afficher un message de succès à
l'utilisateur
      ScaffoldMessenger.of(context).showSnackBar(
        const SnackBar(content: Text('Données du formulaire
traitées !')),
      );

      // Optionnel : Effacer le formulaire après la sauvegarde
(si vous voulez ajouter une autre note)
      // Commenter si vous voulez voir les données rester dans
les champs après print/snackbar
      // _titleController.clear();
      // _contentController.clear();
      // _formKey.currentState!.reset(); // Réinitialise
l'état du formulaire (messages d'erreur etc.)
    }
  }


  @override
```

```
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: const Text('Nouvelle Note'),
    ),
    body: Padding(
      padding: const EdgeInsets.all(16.0),
      child: Form(
        key: _formKey,
        child: ListView(
          children: <Widget>[
            // Champ pour le Titre
            TextFormField(
              controller: _titleController,
              decoration: const InputDecoration(
                labelText: 'Titre',
                hintText: 'Entrez le titre de la note',
                border: OutlineInputBorder(),
              ),
              validator: (value) {
                if (value == null || value.isEmpty) {
                  return 'Veuillez entrer un titre';
                }
                return null;
              },
              onSaved: (value) {
                _title = value ?? '';
              },
            ),
            const SizedBox(height: 16.0),

            // Champ pour le Contenu
            TextFormField(
              controller: _contentController,
              decoration: const InputDecoration(
                labelText: 'Contenu',
                hintText: 'Entrez le contenu de la note',
                border: OutlineInputBorder(),
              ),
              keyboardType: TextInputType.multiline,
              maxLines: null,
              validator: (value) {
                if (value == null || value.isEmpty) {
                  return 'Veuillez entrer un contenu';
                }
                return null;
              },
              onSaved: (value) {
                _content = value ?? '';
              },
            ),
```

```
              const SizedBox(height: 24.0), // Plus d'espace
avant le bouton

              // Bouton de soumission
              ElevatedButton(
                onPressed: _saveNote, // Appel de notre
fonction de sauvegarde
                child: const Text('Enregistrer Note'),
              ),
            ],
          ),
        ),
      ),
    );
  }
}
```

Explication :

- Nous avons ajouté un ElevatedButton à la fin de la ListView.
- Sa propriété onPressed est définie pour appeler notre nouvelle fonction _saveNote().
- La fonction _saveNote() contient la logique pour valider (_formKey.currentState!.validate()) et sauvegarder (_formKey.currentState!.save()) le formulaire.
- Nous avons ajouté des print pour l'instant afin de vérifier que les données sont correctement récupérées.
- ScaffoldMessenger.of(context).showSnackBar(...) : Une façon simple d'afficher un petit message temporaire en bas de l'écran pour donner du feedback à l'utilisateur.
- Les lignes pour _titleController.clear();, _contentController.clear(); et _formKey.currentState!.reset(); sont commentées mais montrent comment on pourrait effacer le formulaire après une sauvegarde réussie si l'intention est de saisir plusieurs éléments à la suite.

1.6. Afficher la Page de Formulaire (Intégration)

Pour voir notre formulaire, nous devons naviguer vers cette page depuis notre page principale (MyHomePage dans lib/main.dart).

Ouvrez lib/main.dart et ajoutez un bouton ou une action (comme un FloatingActionButton) qui navigue vers NoteFormPage.

Code : lib/main.dart (Ajout d'un FloatingActionButton)

```
      import 'package:flutter/material.dart';
import 'package:my_crud_app/note_form_page.dart'; // Importez
la nouvelle page
// Assurez-vous que les chemins d'importation sont corrects
selon la structure de votre projet

void main() {
  WidgetsFlutterBinding.ensureInitialized();
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Mon App CRUD',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: const MyHomePage(title: 'Application de Notes
CRUD'),
      // Vous pouvez aussi définir les routes nommées ici si
vous préférez
      // routes: {
      //   '/newNote': (context) => const NoteFormPage(),
      // },
    );
  }
}

class MyHomePage extends StatefulWidget {
  const MyHomePage({Key? key, required this.title}) :
super(key: key);

  final String title;

  @override
  State<MyHomePage> createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  // Méthode pour naviguer vers la page du formulaire
  void _navigateToNoteForm() {
    Navigator.push(
      context,
      MaterialPageRoute(builder: (context) => const
NoteFormPage()),
```

```
    );
    // Si vous utilisez les routes nommées:
    // Navigator.pushNamed(context, '/newNote');
  }


  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      ),
      body: Center(
        // Pour l'instant, le corps est vide ou contient un
message
        child: Text('Liste des notes ira ici (bientôt !)'),
      ),
      // Ajoute un bouton flottant pour créer une nouvelle
note
      floatingActionButton: FloatingActionButton(
        onPressed: _navigateToNoteForm, // Appelle la fonction
de navigation
        tooltip: 'Ajouter une note',
        child: const Icon(Icons.add),
      ),
    );
  }
}
```

Explication :

- Nous avons importé note_form_page.dart.
- Nous avons ajouté un FloatingActionButton au Scaffold de MyHomePage.
- Lorsque ce bouton est pressé (onPressed), il appelle _navigateToNoteForm().
- _navigateToNoteForm() utilise Navigator.push pour afficher NoteFormPage. MaterialPageRoute est couramment utilisé pour les transitions d'écran standard.

Maintenant, si vous lancez votre application, vous devriez voir un bouton "+" en bas à droite. Cliquer dessus vous amènera à la page du formulaire. Essayez de remplir les champs, de cliquer sur "Enregistrer Note", et observez la console de débogage (souvent dans le terminal où vous avez lancé flutter run) pour voir les valeurs imprimées. Essayez aussi de cliquer sur "Enregistrer Note" sans remplir les champs pour voir la validation en action.

Chapitre 3 - Introduction à la Programmation Asynchrone (Async/Await)

Objectif de ce chapitre : Comprendre pourquoi la programmation asynchrone est nécessaire dans les applications UI, et apprendre à utiliser les concepts clés de Dart : Future, async, et await.

1. Le Problème des Opérations Bloquantes

Imaginez que votre application doit effectuer une tâche qui prend du temps. Cela pourrait être :

- Télécharger des données depuis Internet.
- Lire ou écrire un fichier sur le disque de l'appareil.
- Effectuer un calcul très complexe.
- **Accéder à une base de données (ce qui nous intéresse pour sqflite !).**

Dans une application traditionnelle qui exécute le code étape par étape de manière séquentielle ("synchrone"), si vous lancez une de ces tâches longues, l'application doit **attendre** que cette tâche soit terminée avant de passer à l'instruction suivante.

Sur un appareil mobile, toute l'interface utilisateur (dessin des widgets, réponse aux touchers, animations) s'exécute généralement sur un fil d'exécution unique, souvent appelé le **"thread principal"** ou **"UI thread"**.

Si une opération longue s'exécute sur ce UI thread de manière synchrone, le thread est **bloqué**. Il ne peut rien faire d'autre tant que l'opération n'est pas finie. Résultat ? L'interface utilisateur ne répond plus. Boutons qui ne réagissent pas, animations figées, écran qui ne se met pas à jour... l'application semble "gelée". C'est une mauvaise expérience utilisateur.

Solution : Exécuter ces opérations potentiellement longues de manière **asynchrone**.

2. La Programmation Asynchrone

La programmation asynchrone permet à votre programme de lancer une tâche qui prend du temps et, au lieu d'attendre passivement qu'elle se termine, de continuer à exécuter d'autres tâches (comme maintenir l'interface utilisateur réactive). Lorsque la tâche longue est terminée, l'application est notifiée et peut traiter le résultat.

Pensez à commander un café :

- **Synchrone :** Vous allez au comptoir, commandez, restez *planté* devant le comptoir sans bouger, attendant que le café soit préparé et vous soit remis. Pendant que vous attendez, vous ne pouvez rien faire d'autre. Le barista non plus s'il ne sert que vous.
- **Asynchrone :** Vous allez au comptoir, commandez, on vous donne un biper ou on vous appelle par votre nom. Pendant que le barista prépare votre café (tâche longue), vous pouvez aller vous asseoir, lire un journal, consulter votre téléphone (le UI thread reste réactif). Quand le café est prêt (tâche terminée), le biper sonne ou on vous appelle (notification), et vous allez chercher votre café (traiter le résultat).

Dart utilise le concept de Future et les mots-clés async et await pour gérer l'asynchronisme de manière élégante.

---

3. Les Future

Un Future<T> est un objet qui représente un résultat potentiel (T) ou une erreur qui sera disponible à un moment donné dans le futur. C'est une sorte de "promesse". Au moment où une fonction asynchrone retourne un Future, le résultat n'est pas encore prêt. Il le sera plus tard.

Un Future peut être dans l'un de ces états :

- **Uncompleted (Non complété) :** La tâche asynchrone n'est pas encore terminée.
- **Completed with a value (Complété avec une valeur) :** La tâche s'est terminée avec succès et a produit une valeur.
- **Completed with an error (Complété avec une erreur) :** La tâche a échoué et a renvoyé une erreur.

La plupart des fonctions dans les librairies Dart/Flutter qui effectuent des opérations I/O (comme les accès fichiers, réseau, base de données) sont asynchrones et retournent un Future.

Exemple simple de fonction retournant un Future :

```
      // Cette fonction simule une tâche qui prend du temps
Future<String> fetchUserData() {
  print('Début de la récupération des données
utilisateur...');
  // Future.delayed simule une attente de 3 secondes
  return Future.delayed(const Duration(seconds: 3), () {
    print('Données utilisateur récupérées.');
```

```
    return '{"name": "Alice", "age": 30}'; // Un exemple de
données
  });
}
```

Si vous appelez cette fonction de manière synchrone (ce qui n'est pas possible avec un Future, mais imaginons) ou si vous essayez d'afficher directement son retour sans gérer le Future :

```
      // N'ESSAYEZ PAS CECI POUR OBTENIR LA VALEUR !
// Ceci affiche juste l'instance du Future, pas le résultat.
print(fetchUserData()); // Affichera quelque chose comme :
Instance of 'Future<String>'
```

Pour obtenir la *valeur* contenue dans le Future lorsqu'il est complété, il faut utiliser async et await.

4. async et await

Les mots-clés async et await permettent d'écrire du code asynchrone qui ressemble beaucoup à du code synchrone, le rendant plus facile à lire et à gérer que les anciens systèmes basés sur des callbacks imbriqués.

- Le mot-clé **async** :
    - Placez async avant le corps d'une fonction (void functionName() async { ... } ou Future<ReturnType> functionName() async { ... }).
    - Cela indique que cette fonction contient potentiellement des opérations asynchrones (c'est-à-dire, qu'elle pourrait utiliser await).
    - Une fonction marquée async retourne *toujours* un Future. Si votre fonction async ne retourne pas explicitement un Future, Dart l'enveloppera automatiquement dans un Future<void>. Si elle retourne une valeur non-Future (return someValue;), Dart l'enveloppera dans un Future<TypeDeSomeValue>.
- Le mot-clé **await** :
    - Placez await *devant* une expression qui retourne un Future.
    - **await ne peut être utilisé qu'à l'intérieur d'une fonction marquée async.**

- Lorsque l'exécution atteint une ligne avec await, la fonction async est *mise en pause* à cet endroit précis.
- Pendant que la fonction async est en pause, le thread principal (UI thread) est libéré et peut faire autre chose.
- Une fois que le Future sur lequel await attend est complété (avec une valeur ou une erreur), l'exécution de la fonction async reprend immédiatement après l'await.
- L'expression await futureExpression évalue à la valeur contenue dans le Future une fois complété.

Exemple d'utilisation de async et await :

Reprenons notre fonction fetchUserData. Pour obtenir sa valeur, on l'appelle à l'intérieur d'une fonction async en utilisant await.

```
      Future<String> fetchUserData() {
  print('Début de la récupération des données
utilisateur...');
  return Future.delayed(const Duration(seconds: 3), () {
    print('Données utilisateur récupérées.');
    return '{"name": "Alice", "age": 30}';
  });
}

// Cette fonction est asynchrone et utilise await
void printUserData() async {
  print('Appel de printUserData...');

  // Mettre en pause l'exécution ici jusqu'à ce que
fetchUserData soit terminé
  // Pendant ce temps, l'UI peut continuer à se mettre à jour
  String userData = await fetchUserData();

  // Cette ligne ne sera exécutée qu'après que fetchUserData
soit complété
  print('Données reçues (dans printUserData) : $userData');

  print('Fin de printUserData.');
}

// Pour lancer l'exemple (par exemple dans main() ou un bouton
onPressed)
void main() {
  print('Début du programme.');
  printUserData(); // Appel de la fonction asynchrone
  print('Programme continue après avoir appelé printUserData
(qui s\'exécute en arrière-plan).');
```

```
  // Le programme principal peut faire d'autres choses pendant
que fetchUserData attend
}

/*
Sortie possible (l'ordre exact peut varier légèrement pour les
deux dernières lignes):
Début du programme.
Appel de printUserData...
Début de la récupération des données utilisateur...
Programme continue après avoir appelé printUserData (qui
s'exécute en arrière-plan).
(Pause de 3 secondes pendant que l'UI est réactive)
Données utilisateur récupérées.
Données reçues (dans printUserData) : {"name": "Alice", "age":
30}
Fin de printUserData.
*/
```

Analyse de l'exemple :

1. print('Début du programme.'); s'exécute.
2. printUserData(); est appelée.
3. print('Appel de printUserData...'); s'exécute.
4. fetchUserData(); est appelée.
5. print('Début de la récupération des données utilisateur...'); (dans fetchUserData) s'exécute.
6. Future.delayed(...) est lancé.
7. L'exécution atteint await fetchUserData();. La fonction printUserData est *mise en pause*. Le thread principal est *libéré*.
8. print('Programme continue après avoir appelé printUserData...'); (dans main) s'exécute immédiatement après que printUserData soit mise en pause, prouvant que le thread principal n'est pas bloqué.
9. Après 3 secondes, le Future de fetchUserData se complète.
10. La fonction printUserData reprend son exécution là où elle s'était arrêtée (await). La valeur du Future ("...") est attribuée à userData.
11. print('Données reçues...') s'exécute.
12. print('Fin de printUserData.'); s'exécute.

5. Gestion des Erreurs Asynchrones (try...catch)

```
Les Future peuvent se compléter avec une erreur. Pour gérer
ces erreurs de manière propre avec async/await, on utilise les
blocs try...catch comme avec le code synchrone.
Si une opération asynchrone après un await lève une exception,
cette exception peut être interceptée par un bloc catch juste
après le await.
      Future<String> fetchUserDataWithError() {
  print('Début de la récupération des données utilisateur
(avec erreur)...');
  return Future.delayed(const Duration(seconds: 2), () {
    print('Une erreur est survenue.');
    throw Exception('Échec de la récupération des données !');
// Simule une erreur
    // Si pas d'erreur, on retournerait une valeur : return
'Données valides';
  });
}

void processUserData() async {
  print('Appel de processUserData...');
  try {
    String userData = await fetchUserDataWithError();
    // Cette ligne ne sera PAS exécutée si
fetchUserDataWithError lève une erreur
    print('Données traitées avec succès : $userData');
  } catch (e) {
    // Ce bloc s'exécute si une erreur se produit pendant
l'attente (await)
    print('Erreur interceptée : $e');
  }
  print('Fin de processUserData.');
}

void main() {
  print('Début du programme principal (erreur).');
  processUserData();
  print('Programme continue après avoir appelé
processUserData.');
}

/*
Sortie possible:
Début du programme principal (erreur).
Appel de processUserData...
Début de la récupération des données utilisateur (avec
erreur)...
Programme continue après avoir appelé processUserData.
(Pause de 2 secondes)
Une erreur est survenue.
```

```
Erreur interceptée : Exception: Échec de la récupération des
données !
Fin de processUserData.
*/
```

6. Lien avec Sqflite

Maintenant que nous comprenons Future, async, et await, le lien avec sqflite devient clair.

Toutes les opérations que vous effectuerez avec la base de données (ouverture, création de tables, insertion, lecture, modification, suppression) sont des opérations d'I/O qui prennent du temps.

Par conséquent, les méthodes du package sqflite qui réalisent ces opérations retournent des **Future**.

Voici des exemples de signatures de méthodes sqflite (nous les verrons en détail plus tard) :

- Ouvrir la base de données : Future<Database> openDatabase(...)
- Exécuter une requête SQL : Future<List<Map<String, dynamic>>> rawQuery(String sql, [List? arguments])
- Insérer une ligne : Future<int> insert(String table, Map<String, Object?> values, {String? nullColumnHack, ConflictAlgorithm? conflictAlgorithm})
- etc.

Pour utiliser ces méthodes dans votre code, vous devrez donc :

1. Placer l'appel de la méthode sqflite derrière un mot-clé **await**.
2. Inclure cet appel dans une fonction que vous aurez marquée avec le mot-clé **async**.

Exemple (aperçu du prochain module) :

```
      // CECI EST UN APERÇU, NOUS CONSTRUIRONS LE CODE COMPLET
PLUS TARD
import 'package:sqflite/sqflite.dart'; // N'oubliez pas
l'import

// Fonction pour ouvrir la base de données (sera expliquée en
détail)
Future<Database> initDatabase() async {
  // ... logique pour trouver le chemin ...
  String path = await getDatabasesPath(); // getDatabasesPath
retourne un Future !
```

```
  String dbPath = join(path, 'notes.db'); // join est
synchrone

  // Ouvrir la base de données est une opération asynchrone
  Database db = await openDatabase(dbPath, version: 1,
onCreate: (db, version) async {
    // Créer la table est aussi asynchrone
    await db.execute('''
      CREATE TABLE notes(
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        title TEXT,
        content TEXT
      )
    ''');
  });
  return db;
}

// Fonction pour insérer une note (sera expliquée en détail)
Future<int> insertNote(Map<String, dynamic> noteData) async {
  // Obtenir l'instance de la DB - initDatabase retourne un
Future !
  Database db = await initDatabase(); // Si la DB est déjà
ouverte, initDatabase la retourne rapidement

  // Insérer la note est une opération asynchrone
  int id = await db.insert('notes', noteData); // 'notes' est
le nom de notre table
  return id;
}

// Exemple d'utilisation dans une fonction UI (comme onPressed
d'un bouton)
void handleSaveButtonPress(String title, String content) async
{ // LA FONCTION DOIT ÊTRE ASYNC
  Map<String, dynamic> newNote = {'title': title, 'content':
content};

  try {
    int newId = await insertNote(newNote); // Utilisation de
await
    print('Note insérée avec l\'ID : $newId');
    // Mettre à jour l'UI si nécessaire, naviguer, etc.
  } catch (e) {
    print('Erreur lors de l\'insertion de la note : $e');
    // Afficher un message d'erreur à l'utilisateur
  }
}
```

Comme vous pouvez le voir dans l'exemple ci-dessus, chaque interaction avec la base de données ou avec une fonction qui interagit avec la base de données nécessite l'utilisation de await à l'intérieur d'une fonction async.

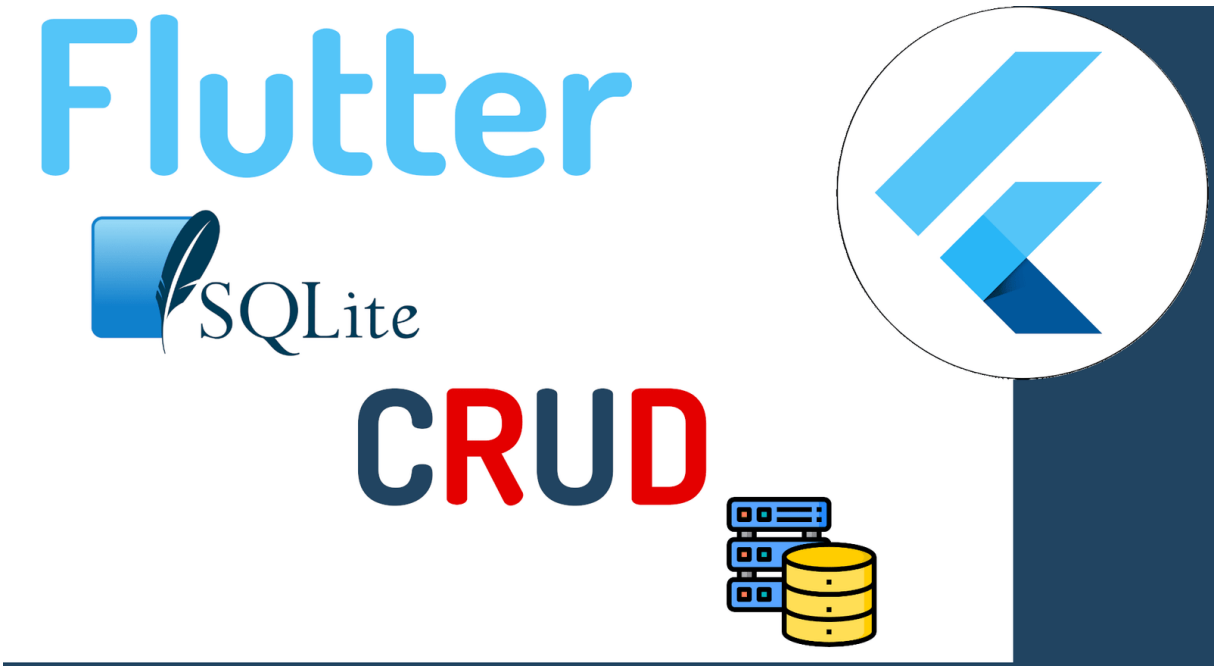
Conclusion

Nous avons exploré l'importance de la programmation asynchrone en Dart/Flutter pour maintenir une interface utilisateur fluide. Nous avons appris à utiliser les outils fondamentaux :

- Les **Future** représentent des valeurs qui seront disponibles plus tard.
- Les mots-clés **async** et **await** nous permettent d'écrire du code asynchrone de manière séquentielle et lisible, en mettant en pause l'exécution d'une fonction async en attendant qu'un Future se termine, sans bloquer le thread principal.
- Les blocs **try...catch** sont utilisés pour gérer les erreurs dans le code asynchrone.

Nous comprenons maintenant que les opérations de base de données avec sqflite sont asynchrones et nécessiteront l'utilisation intensive de async et await.

Dans le prochain chapitre, nous allons appliquer ces connaissances pour mettre en place notre base de données avec sqflite et la rendre prête à accueillir nos données.



Chapitre 4 - Déclaration et Initialisation de la Base de Données Sqflite

Objectif de ce chapitre : Apprendre à configurer et ouvrir une base de données SQLite avec sqflite, à définir sa structure (le schéma) lors de sa première création, et à gérer l'accès à la base de données dans notre application.

---

1. Où Mettre le Code de la Base de Données ?

Pour garder notre projet organisé et facile à maintenir, il est préférable de séparer la logique de gestion de la base de données du code de l'interface utilisateur (nos widgets).

Nous allons créer une classe dédiée qui sera responsable de l'ouverture de la base de données, de la création des tables, et plus tard, de l'exécution des opérations CRUD. C'est une pratique courante appelée le **"pattern Repository"** ou **"Database Helper"**.

Créons un nouveau fichier pour cela, par exemple lib/database_helper.dart.

---

2. Le Pattern Singleton pour la Base de Données

Une base de données est une ressource partagée. Il est important de s'assurer qu'il n'y a qu'une seule connexion à la base de données ouverte à la fois dans votre application. Ouvrir et fermer des connexions fréquemment peut être coûteux en performance et potentiellement causer des problèmes d'accès concurrents.
Le **pattern Singleton** est un modèle de conception qui garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès global à cette instance. C'est parfait pour notre DatabaseHelper.
Voici comment implémenter un Singleton basique en Dart :

```
      // Exemple de structure Singleton
class SingletonExample {
  // 1. Variable statique privée pour stocker l'instance unique
  static final SingletonExample _instance = SingletonExample._privateConstructor();

  // 2. Constructeur privé pour empêcher la création directe d'instances
  SingletonExample._privateConstructor();

  // 3. Factory constructor pour fournir le point d'accès global
```

```
  factory SingletonExample() {
    return _instance;
  }

  // Ajoutez ici les méthodes et propriétés de votre classe
  void doSomething() {
    print('Singleton fait quelque chose.');
  }
}

// Utilisation :
// SingletonExample singleton1 = SingletonExample(); // Obtient
l'instance unique
// SingletonExample singleton2 = SingletonExample(); // Obtient
la même instance unique
// print(identical(singleton1, singleton2)); // Affiche true
// singleton1.doSomething();
```

Nous allons appliquer cette structure à notre classe DatabaseHelper.

3. Trouver le Chemin de la Base de Données sur l'Appareil

Notre base de données sera stockée dans un fichier .db sur le système de fichiers de l'appareil. Nous ne pouvons pas choisir un chemin arbitraire ; nous devons utiliser un répertoire auquel notre application a accès.

Le package path_provider nous aide à trouver ces répertoires standard (comme le répertoire des documents de l'application). Le package sqflite fournit également une méthode pour obtenir le chemin par défaut recommandé pour les bases de données (getDatabasesPath). Nous utiliserons ce dernier, car il est conçu spécifiquement pour les bases de données.

Le package path nous aidera à joindre le chemin du répertoire et le nom de notre fichier de base de données (notes.db) de manière correcte, indépendamment de l'OS (Android utilise /, iOS gère différemment, etc.).

Ces opérations pour obtenir les chemins d'accès au système de fichiers sont **asynchrones**. Elles retournent donc des Future.

4. Ouvrir la Base de Données (openDatabase)

La fonction principale pour interagir avec la base de données est openDatabase, fournie par le package sqflite.

Signature simplifiée :

```
Future<Database> openDatabase(String path, {int? version,
OnDatabaseCreateFn? onCreate, OnDatabaseVersionChangeFn? onUpgrade, ...})
```

- **path :** Le chemin complet vers le fichier de la base de données sur l'appareil (celui qu'on a obtenu à l'étape 3.3).
- **version :** Un entier positif qui représente la version du schéma de votre base de données. C'est très important pour gérer les futures modifications de la structure de votre base de données (migrations). Pour le premier atelier, on commence à 1.
- **onCreate :** Une fonction de rappel (callback) qui est appelée *uniquement* la première fois que la base de données est ouverte, c'est-à-dire lorsque le fichier .db n'existe pas encore sur l'appareil et que sqflite doit le créer. C'est l'endroit parfait pour exécuter vos requêtes SQL de **création de tables (CREATE TABLE)**. Cette fonction callback est elle-même asynchrone (OnDatabaseCreateFn est de type Future<void> Function(Database db, int version)).
- **onUpgrade :** Une fonction de rappel appelée lorsque vous ouvrez la base de données avec un numéro de version *plus élevé* que la version actuelle de la base de données déjà présente sur l'appareil. C'est utilisé pour gérer les migrations (ajouter des colonnes, modifier des tables, etc.). Nous n'allons pas l'implémenter en détail dans ce premier atelier, mais il est bon de savoir qu'elle existe. Ce callback est également asynchrone.

La fonction openDatabase est **asynchrone** et retourne un Future<Database>. L'objet Database retourné est l'instance à travers laquelle vous effectuerez toutes vos opérations (INSERT, SELECT, UPDATE, DELETE).

5. Définir le Schéma de la Base de Données (CREATE TABLE)

À l'intérieur du callback onCreate de openDatabase, nous allons exécuter des requêtes SQL pour définir la structure de notre base de données. Nous aurons une seule table pour nos notes.

Une requête CREATE TABLE typique ressemble à ceci :

```
      CREATE TABLE table_name (
  column1_name DATATYPE CONSTRAINTS,
  column2_name DATATYPE CONSTRAINTS,
  ...
);
```

Pour notre table de notes, nous aurons besoin de :

- Un identifiant unique pour chaque note. INTEGER PRIMARY KEY AUTOINCREMENT est le type de colonne standard pour un ID unique qui s'incrémente automatiquement à chaque nouvelle insertion. C'est le plus simple à utiliser.
- Un champ pour le titre. Le type TEXT est approprié pour du texte.
- Un champ pour le contenu. Le type TEXT est également approprié.

Notre requête CREATE TABLE ressemblera donc à ceci :

```
      CREATE TABLE notes (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  title TEXT NOT NULL, -- NOT NULL indique que ce champ ne
peut pas être vide
  content TEXT NOT NULL
);
```

Nous exécuterons cette requête à l'intérieur du callback onCreate en utilisant la méthode db.execute().
db.execute() est également **asynchrone**.

6. Mettre Tout Ensemble : La Classe DatabaseHelper

Maintenant, combinons les concepts du Singleton, de la recherche de chemin, de openDatabase et de onCreate dans notre classe DatabaseHelper.

Code : lib/database_helper.dart

```
      import 'dart:async'; // Nécessaire pour Future

// Importation des packages
import 'package:sqflite/sqflite.dart';
import 'package:path/path.dart'; // Pour joindre les chemins


// Constante pour le nom de notre base de données
final String databaseName = "notes.db";
// Constante pour le nom de notre table
final String tableName = "notes";

// Constantes pour les noms des colonnes de notre table
'notes'
class NoteFields {
  static final String id = '_id'; // Souvent "_id" pour
l'identifiant
  static final String title = 'title';
  static final String content = 'content';
```

```
  // Vous pourriez ajouter d'autres champs ici si nécessaire
(date, etc.)

  static final List<String> values = [ // Utile pour les
requêtes SELECT *
    id, title, content
  ];
}


class DatabaseHelper {
  // 1. Variable statique pour l'instance de la base de
données
  // Elle sera privée pour qu'on ne puisse pas y accéder
directement
  static Database? _database;

  // 2. Variable statique pour l'instance unique de la classe
DatabaseHelper (Singleton)
  static final DatabaseHelper _instance =
DatabaseHelper._privateConstructor();

  // 3. Constructeur privé
  DatabaseHelper._privateConstructor();

  // 4. Factory constructor pour le point d'accès global
  factory DatabaseHelper() {
    return _instance;
  }

  // 5. Getter asynchrone pour obtenir l'instance de la base
de données
  // Si _database est null, on l'initialise
  // Sinon, on retourne l'instance existante
  Future<Database> get database async {
    if (_database != null) return _database!; // Retourne
l'instance si elle existe

    // Si l'instance n'existe pas, l'initialiser
    _database = await _initDatabase();
    return _database!;
  }

  // 6. Méthode privée asynchrone pour initialiser la base de
données
  Future<Database> _initDatabase() async {
    // Obtenir le chemin par défaut pour les bases de données
    String databasesPath = await getDatabasesPath();
    // Construire le chemin complet pour notre fichier de base
de données
    String path = join(databasesPath, databaseName);
```

```
    print("Chemin de la base de données : $path"); // Utile
pour déboguer

    // Ouvrir la base de données
    // La méthode openDatabase est asynchrone, donc on utilise
await
    return await openDatabase(
      path,
      version: 1, // Numéro de version de la base de données
      // Cette fonction onCreate est appelée uniquement la
première fois
      // que la base de données est créée
      onCreate: _onCreate,
      // onUpgrade: _onUpgrade, // Pour gérer les futures
versions, on le laisse de côté pour l'instant
    );
  }

  // 7. Méthode privée asynchrone pour créer les tables
(appelée par onCreate)
  Future _onCreate(Database db, int version) async {
    // Exécuter la requête SQL pour créer la table 'notes'
    // db.execute est asynchrone, donc on utilise await
    await db.execute('''
      CREATE TABLE $tableName (
        ${NoteFields.id} INTEGER PRIMARY KEY AUTOINCREMENT,
        ${NoteFields.title} TEXT NOT NULL,
        ${NoteFields.content} TEXT NOT NULL
      )
    ''');

    print("Table '$tableName' créée avec succès !");
  }

  // --- Plus tard, dans le chapitre suivant, nous ajouterons
les méthodes CRUD ici ---
  // Future<int> insert(Note note) async { ... }
  // Future<List<Note>> getNotes() async { ... }
  // Future<int> update(Note note) async { ... }
  // Future<int> delete(int id) async { ... }

  // Méthode pour fermer la base de données (optionnel,
souvent pas nécessaire dans les petites apps)
  Future close() async {
    final db = await _instance.database;
    db.close();
    _database = null; // Réinitialise l'instance après
fermeture
  }
}
```

Explications du code :

- Nous avons importé les packages nécessaires (sqflite, path, dart:async).
- Nous avons défini des constantes pour le nom de la base de données et de la table, ainsi qu'une classe NoteFields pour avoir les noms de colonnes sous forme de constantes (aide à éviter les fautes de frappe).
- static Database? _database; : Stocke l'instance de la base de données une fois qu'elle est ouverte. ? indique qu'elle peut être null au début.
- Le pattern Singleton (_instance, constructeur privé, factory constructor) est mis en place.
- Future<Database> get database async { ... } : C'est le point d'entrée public. Chaque fois que vous appelerez DatabaseHelper().database, il vérifiera si la base de données est déjà ouverte (_database != null). Si oui, il la retourne immédiatement. Sinon, il appelle _initDatabase() (en attendant son résultat avec await) pour l'ouvrir et stocke l'instance avant de la retourner. C'est la garantie du Singleton et de l'initialisation unique.
- Future<Database> _initDatabase() async { ... } : Cette méthode privée contient la logique d'ouverture : elle obtient le chemin, le joint avec le nom du fichier, puis appelle openDatabase. await est utilisé car getDatabasesPath et openDatabase retournent des Future.
- onCreate: _onCreate : Nous passons la référence de notre méthode _onCreate à openDatabase.
- Future _onCreate(Database db, int version) async { ... } : Cette méthode reçoit l'instance db fraîchement créée et le numéro de version. À l'intérieur, nous appelons db.execute() avec notre requête CREATE TABLE. await est utilisé ici aussi car db.execute retourne un Future.
- close() : Une méthode optionnelle pour fermer la base de données. Souvent, les applications mobiles laissent la base de données ouverte tant que l'application est en cours d'exécution, donc cette méthode n'est pas toujours utilisée, mais elle est là si besoin.

7. Initialiser la Base de Données au Démarrage de l'Application

Pour que la base de données soit prête à l'emploi lorsque nous en aurons besoin (par exemple, pour enregistrer une note depuis le formulaire), nous devons déclencher

l'initialisation de la base de données. Le plus simple est d'appeler le getter DatabaseHelper().database une première fois au démarrage de l'application.

Nous pouvons faire cela dans la fonction main() de notre fichier lib/main.dart, après WidgetsFlutterBinding.ensureInitialized();. Puisque DatabaseHelper().database est asynchrone (elle retourne un Future), la fonction main doit être marquée async.

Code : lib/main.dart (Mise à jour de main())

```
      import 'package:flutter/material.dart';
import 'package:my_crud_app/note_form_page.dart';
import 'package:my_crud_app/database_helper.dart'; // Importez
votre DatabaseHelper

// Transformez main en fonction asynchrone car nous allons
utiliser await à l'intérieur
Future<void> main() async {
  // Assurez-vous que les services de la plateforme Flutter
sont initialisés
  WidgetsFlutterBinding.ensureInitialized();

  // --- Initialisation de la base de données ---
  // Appeler le getter 'database' du Singleton force son
initialisation
  // (c'est-à-dire, la création de la base et des tables si
elles n'existent pas)
  // Nous utilisons await car le getter retourne un Future
  try {
    await DatabaseHelper().database;
    print("Base de données initialisée avec succès !");
  } catch (e) {
    print("Erreur lors de l'initialisation de la base de
données : $e");
    // Gérer l'erreur (afficher un message à l'utilisateur,
logguer, etc.)
  }
  // ------------------------------------------


  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Mon App CRUD',
      theme: ThemeData(
        primarySwatch: Colors.blue,
```

```
      ),
      home: const MyHomePage(title: 'Application de Notes
CRUD'),
    );
  }
}

class MyHomePage extends StatefulWidget {
  const MyHomePage({Key? key, required this.title}) :
super(key: key);

  final String title;

  @override
  State<MyHomePage> createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  void _navigateToNoteForm() {
    Navigator.push(
      context,
      MaterialPageRoute(builder: (context) => const
NoteFormPage()),
    );
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      ),
      body: const Center( // Utilisation de const car le
contenu est statique pour l'instant
        child: Text('Liste des notes ira ici (bientôt !)'),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _navigateToNoteForm,
        tooltip: 'Ajouter une note',
        child: const Icon(Icons.add),
      ),
    );
  }
}
```

Explication :

- Nous avons importé notre DatabaseHelper.
- Nous avons marqué la fonction main comme async.

- Nous appelons await DatabaseHelper().database; juste avant runApp().
- Lors du tout premier lancement de l'application, cet appel déclenchera _initDatabase, qui appellera openDatabase qui trouvera que le fichier notes.db n'existe pas. Cela déclenchera alors le callback _onCreate, qui exécutera la requête CREATE TABLE.
- Lors des lancements suivants, DatabaseHelper().database trouvera que le fichier existe, openDatabase s'ouvrira rapidement, et _onCreate ne sera *pas* appelé.
- Nous avons ajouté un bloc try...catch autour de l'appel asynchrone pour gérer d'éventuelles erreurs lors de l'initialisation.
- Les messages print dans _initDatabase, _onCreate, et main vous aideront à vérifier dans la console si la base de données est bien créée lors du premier lancement.

Lancez l'application sur un émulateur/appareil. La première fois, vous devriez voir les messages de création de la DB et de la table dans la console. Les fois suivantes, seul le message "Base de données initialisée avec succès !" (ou "Base de données ouverte avec succès !") devrait apparaître.

Conclusion

Nous avons fait un grand pas en avant ! Nous avons mis en place la structure pour gérer notre base de données locale SQLite en utilisant sqflite.

Nous avons appris à :

- Organiser le code lié à la base de données dans une classe dédiée (DatabaseHelper).
- Utiliser le pattern Singleton pour garantir une seule instance de la base de données.
- Trouver le chemin approprié sur l'appareil pour stocker le fichier .db.
- Utiliser openDatabase pour ouvrir la connexion.
- Définir le schéma de notre base de données (créer la table notes) en utilisant le callback onCreate avec une requête SQL CREATE TABLE.
- Utiliser **async** et **await** de manière intensive, car toutes ces opérations sont asynchrones.
- Initialiser notre DatabaseHelper au démarrage de l'application.

Notre base de données est maintenant prête et structurée. Dans le prochain module, nous allons implémenter les méthodes pour réaliser les opérations CRUD (Create, Read, Update, Delete) en interagissant avec cette base de données !

Chapitre 5 - Implémenter les Opérations CRUD

Objectif de ce chapitre : Apprendre à utiliser l'instance de base de données ouverte par sqflite pour effectuer les quatre opérations CRUD (Créer, Lire, Modifier, Supprimer) sur notre table notes.

1. Modèle de Données (Data Model)

Avant d'interagir avec la base de données, il est utile de définir comment nous allons représenter les données d'une seule "note" dans notre code Dart. Avoir une classe modèle simple rend le code plus lisible et gère les conversions entre nos objets Dart et le format utilisé par la base de données (qui est généralement une Map<String, dynamic>).

Créons un nouveau fichier lib/models/note.dart (vous pourriez créer un dossier models pour organiser cela).

Code : lib/models/note.dart

```
      // Utiliser les constantes de colonnes définies dans
database_helper.dart
import '../database_helper.dart';

class Note {
  // Propriétés de notre Note, correspondant aux colonnes de
la table
  final int? id; // L'ID est généré par la DB, donc il est
nullable lors de la création
  final String title;
  final String content;

  // Constructeur pour créer une instance de Note
  Note({
    this.id,
    required this.title,
    required this.content,
  });

  // Méthode pour convertir un objet Note en Map
  // Utile pour insérer ou mettre à jour dans la base de
données
  Map<String, dynamic> toMap() {
    // La clé de la map doit correspondre aux noms des
colonnes dans la DB
    return {
```

```
      NoteFields.title: title,
      NoteFields.content: content,
      // L'ID n'est inclus que s'il existe (pour les mises à
jour)
      if (id != null) NoteFields.id: id,
    };
  }

  // Méthode statique (ou factory constructor) pour créer un
objet Note à partir d'une Map
  // Utile pour lire les données depuis la base de données
  factory Note.fromMap(Map<String, dynamic> map) {
    return Note(
      id: map[NoteFields.id] as int?, // Conversion en int,
nullable si nécessaire
      title: map[NoteFields.title] as String, // Conversion en
String
      content: map[NoteFields.content] as String,
    );
  }

  // Optionnel : Override toString pour faciliter le débogage
  @override
  String toString() {
    return 'Note{id: $id, title: $title, content: $content}';
  }
}
```

Explication :

- La classe Note a des propriétés id, title, content qui correspondent à nos colonnes. id est int? car lors de la création d'une nouvelle note, son ID n'est pas encore connu (il sera généré par la DB).
- La méthode toMap() prend un objet Note et le convertit en Map<String, dynamic>. C'est le format attendu par les méthodes d'insertion et de mise à jour de sqflite. Notez que l'ID n'est ajouté à la map que s'il existe, ce qui est pertinent pour les mises à jour.
- La factory constructor Note.fromMap() prend une Map (qui est le format retourné par les requêtes de lecture de sqflite) et crée un objet Note. Nous utilisons as int? et as String pour caster explicitement les types.
- Nous utilisons les constantes NoteFields définies dans database_helper.dart pour les noms des clés dans la map, ce qui rend le code moins sujet aux erreurs de frappe.

2. Implémenter l'Opération "Create" (Insertion)

Nous allons ajouter une méthode à notre classe DatabaseHelper pour insérer une nouvelle note dans la table notes.

Code : lib/database_helper.dart (Ajout de la méthode insert)

```
      import 'dart:async';
import 'package:sqflite/sqflite.dart';
import 'package:path/path.dart';

// Importez votre modèle Note
import 'models/note.dart'; // Assurez-vous que le chemin est
correct

final String databaseName = "notes.db";
final String tableName = "notes";

class NoteFields {
  static final String id = '_id';
  static final String title = 'title';
  static final String content = 'content';

  static final List<String> values = [id, title, content];
}

class DatabaseHelper {
  static Database? _database;
  static final DatabaseHelper _instance =
DatabaseHelper._privateConstructor();
  DatabaseHelper._privateConstructor();
  factory DatabaseHelper() => _instance;

  Future<Database> get database async {
    if (_database != null) return _database!;
    _database = await _initDatabase();
    return _database!;
  }

  Future<Database> _initDatabase() async {
    String databasesPath = await getDatabasesPath();
    String path = join(databasesPath, databaseName);
    print("Chemin de la base de données : $path");
    return await openDatabase(
      path,
      version: 1,
      onCreate: _onCreate,
    );
  }

  Future _onCreate(Database db, int version) async {
```

```
    await db.execute('''
      CREATE TABLE $tableName (
        ${NoteFields.id} INTEGER PRIMARY KEY AUTOINCREMENT,
        ${NoteFields.title} TEXT NOT NULL,
        ${NoteFields.content} TEXT NOT NULL
      )
    ''');
    print("Table '$tableName' créée avec succès !");
  }

  // --- MÉTHODE INSERT (CREATE) ---
  // Insère une note dans la base de données
  // Retourne l'ID de la ligne insérée
  Future<int> create(Note note) async {
    // 1. Obtenir l'instance de la base de données
    // Grâce au getter 'database', cela initialise la DB si
elle n'est pas ouverte
    final db = await database; // C'est une opération
asynchrone !

    // 2. Insérer la note
    // db.insert est une opération asynchrone, on utilise
await
    // On passe le nom de la table et la map de données
(obtenue via note.toMap())
    // conflictAlgorithm gère les conflits (ici, ignore s'il y
a un conflit, mais peu probable pour une insertion simple)
    final id = await db.insert(
      tableName,
      note.toMap(),
      conflictAlgorithm: ConflictAlgorithm.ignore, // Gère les
conflits si ID spécifié
    );

    print("Note insérée avec l'ID : $id");
    return id; // Retourne l'ID de la nouvelle ligne insérée
  }

  // --- Les autres méthodes CRUD seront ajoutées ici ---
  // Future<List<Note>> readAllNotes() async { ... }
  // Future<Note?> readNote(int id) async { ... }
  // Future<int> update(Note note) async { ... }
  // Future<int> delete(int id) async { ... }


  Future close() async {
    final db = await _instance.database;
    db.close();
    _database = null;
  }
}
```

Explication de create(Note note) :

- Elle est async car elle utilise await.
- final db = await database; : Récupère l'instance de la base de données. Si c'est le premier appel après le démarrage, cela déclenchera l'initialisation complète (on a vu ca dans le chapitre 4). Sinon, cela retournera très rapidement l'instance déjà ouverte.
- await db.insert(...) : Appelle la méthode insert de l'objet Database.
    - Le premier argument est le nom de la table (tableName).
    - Le deuxième argument est la Map<String, dynamic> représentant la ligne à insérer. Nous l'obtenons en appelant note.toMap() sur l'objet Note passé en paramètre.
    - conflictAlgorithm est une option pour gérer les cas où l'insertion violerait une contrainte (comme un ID existant si vous spécifiez l'ID). ignore est souvent suffisant pour une insertion simple où la DB génère l'ID.
- db.insert() retourne un Future<int> qui, une fois complété, contient l'ID de la nouvelle ligne insérée. Nous attendons ce résultat et le stockons dans id.
- La méthode retourne cet id.

---

3. Implémenter l'Opération "Read" (Lecture/Requête)

Nous allons ajouter une méthode pour récupérer toutes les notes de la base de données.

Code : lib/database_helper.dart (Ajout de la méthode readAllNotes)

```
      // ... imports et code précédent ...

class DatabaseHelper {
  // ... Singleton et initialisation ...

  Future<int> create(Note note) async {
    // ... code de la méthode create ...
    final db = await database;
    final id = await db.insert(tableName, note.toMap(),
conflictAlgorithm: ConflictAlgorithm.ignore);
    print("Note insérée avec l'ID : $id");
    return id;
  }

  // --- MÉTHODE QUERY (READ - Lire tout) ---
  // Récupère toutes les notes de la table
```

```
  // Retourne une liste d'objets Note
  Future<List<Note>> readAllNotes() async {
    final db = await database; // Opération asynchrone

    // Effectuer la requête
    // db.query est une opération asynchrone, on utilise await
    // Elle retourne une List<Map<String, dynamic>>
    // On peut spécifier les colonnes, un ordre (orderBy) etc.
    // Si on ne spécifie rien, c'est SELECT * FROM tableName
    final resultMaps = await db.query(
      tableName,
      orderBy: '${NoteFields.id} DESC', // Exemple : Trier par
ID décroissant (dernières notes en premier)
      // columns: NoteFields.values, // Optionnel : Spécifier
les colonnes à récupérer
    );

    // Convertir la liste de Maps en liste d'objets Note
    // Utilisation de la méthode .map() et de la factory
Note.fromMap
    return resultMaps.map((map) =>
Note.fromMap(map)).toList();
  }

  // --- MÉTHODE QUERY (READ - Lire une seule) ---
  // Récupère une seule note par son ID
  // Retourne un objet Note ou null si non trouvé
  Future<Note?> readNote(int id) async {
    final db = await database; // Opération asynchrone

    // Effectuer la requête avec condition WHERE
    // Utilisation de '?' et 'whereArgs' pour éviter
l'injection SQL
    final maps = await db.query(
      tableName,
      columns: NoteFields.values, // Spécifier les colonnes
      where: '${NoteFields.id} = ?', // La condition WHERE
      whereArgs: [id], // Les valeurs correspondant aux '?'
      limit: 1, // On n'attend qu'un seul résultat
    );

    // Vérifier si un résultat a été trouvé et le convertir
    if (maps.isNotEmpty) {
      return Note.fromMap(maps.first); // maps.first est la
première (et seule) map trouvée
    } else {
      return null; // Retourne null si aucune note trouvée
avec cet ID
    }
  }
```

```
  // --- Les autres méthodes CRUD seront ajoutées ici ---
  // Future<int> update(Note note) async { ... }
  // Future<int> delete(int id) async { ... }


  Future close() async {
    final db = await _instance.database;
    db.close();
    _database = null;
  }
}
```

Explication de readAllNotes() et readNote() :

- Elles sont async.
- final db = await database; : Obtient l'instance de la base de données.
- await db.query(...) : Appelle la méthode query de l'objet Database.
    - Le premier argument est le nom de la table (tableName).
    - orderBy: '${NoteFields.id} DESC' : Un argument optionnel pour trier les résultats. Ici, par ID décroissant.
    - columns: NoteFields.values : Optionnel, spécifie quelles colonnes récupérer. Si omis, toutes les colonnes sont récupérées (équivalent à SELECT *).
    - where: '${NoteFields.id} = ?' et whereArgs: [id] : Dans readNote, ces arguments sont utilisés pour filtrer les résultats et ne récupérer que la ligne dont l'ID correspond à la valeur passée. C'est l'équivalent d'une clause WHERE id = X en SQL. L'utilisation de ? et whereArgs est la méthode sécurisée recommandée par sqflite pour passer des valeurs dans les requêtes afin d'éviter les injections SQL.
    - limit: 1 : Dans readNote, cela limite le résultat à une seule ligne (puisque l'ID est unique).
- db.query() retourne un Future<List<Map<String, dynamic>>>. Chaque Map dans la liste représente une ligne du résultat.
- resultMaps.map((map) => Note.fromMap(map)).toList(); : C'est une opération de transformation courante en Dart. Elle parcourt chaque map dans la liste resultMaps, applique la factory Note.fromMap() à chaque map pour créer un objet Note, et collecte tous ces objets Note dans une nouvelle liste (.toList()).

- readNote vérifie si la liste de maps retournée n'est pas vide (maps.isNotEmpty) avant de tenter d'accéder au premier élément (maps.first) et de le convertir. Si la liste est vide (aucun résultat), elle retourne null.

4. Implémenter l'Opération "Update" (Modification)

Nous allons ajouter une méthode pour modifier une note existante.

Code : lib/database_helper.dart (Ajout de la méthode update)

```
      // ... imports et code précédent ...

class DatabaseHelper {
  //  ...  Singleton,  initialisation,  create,  readAllNotes,
readNote ...

  // --- MÉTHODE UPDATE (UPDATE) ---
  // Met à jour une note existante dans la base de données
  // L'objet Note passé en paramètre DOIT avoir un ID non-null
  // Retourne le nombre de lignes affectées (devrait être 1 si
l'ID existe)
  Future<int> update(Note note) async {
    final db = await database; // Opération asynchrone

    // db.update est une opération asynchrone, on utilise await
    //  On  passe  le  nom  de  la  table,  la  map  de  données
(note.toMap()),
    // et la condition WHERE pour spécifier quelle(s) ligne(s)
mettre à jour
    final rowsAffected = await db.update(
      tableName,
      note.toMap(), // Utilise la map (qui contient l'ID si
present, mais WHERE est prioritaire)
      where: '${NoteFields.id} = ?', // Condition WHERE basée
sur l'ID
      whereArgs: [note.id], // Valeur de l'ID pour la condition
    );

    print("Note  avec  l'ID  ${note.id}  mise  à  jour.  Lignes
affectées : $rowsAffected");
    return  rowsAffected;  //  Retourne  le  nombre  de  lignes
modifiées
  }

  // --- Les autres méthodes CRUD seront ajoutées ici ---
  // Future<int> delete(int id) async { ... }

  Future close() async {
    final db = await _instance.database;
```

```
    db.close();
    _database = null;
  }
}
```

Explication de update(Note note) :

- Elle est async.
- final db = await database; : Obtient l'instance de la base de données.
- await db.update(...) : Appelle la méthode update de l'objet Database.
    - Le premier argument est le nom de la table (tableName).
    - Le deuxième argument est la Map<String, dynamic> contenant les nouvelles valeurs pour les colonnes. Nous utilisons note.toMap().
    - where: '${NoteFields.id} = ?' et whereArgs: [note.id] : Indiquent à sqflite quelles lignes mettre à jour. C'est crucial. Sans clause WHERE, la méthode update mettrait à jour *toutes* les lignes de la table ! Nous passons l'ID de la note à mettre à jour dans whereArgs. Notez que l'objet Note passé à cette méthode *doit* avoir un id non-null.
- db.update() retourne un Future<int> qui, une fois complété, contient le nombre de lignes qui ont été modifiées par l'opération.

5. Implémenter l'Opération "Delete" (Suppression)

Enfin, nous allons ajouter une méthode pour supprimer une note.

Code : lib/database_helper.dart (Ajout de la méthode delete)

```
      // ... imports et code précédent ...

class DatabaseHelper {
  // ... Singleton, initialisation, create, readAllNotes,
readNote, update ...

  // --- MÉTHODE DELETE (DELETE) ---
  // Supprime une note par son ID
  // Retourne le nombre de lignes supprimées (devrait être 1
si l'ID existe)
  Future<int> delete(int id) async {
    final db = await database; // Opération asynchrone

    // db.delete est une opération asynchrone, on utilise
await
```

```
    // On passe le nom de la table et la condition WHERE
    final rowsDeleted = await db.delete(
      tableName,
      where: '${NoteFields.id} = ?', // Condition WHERE basée
sur l'ID
      whereArgs: [id], // Valeur de l'ID pour la condition
    );

    print("Note avec l'ID $id supprimée. Lignes supprimées :
$rowsDeleted");
    return rowsDeleted; // Retourne le nombre de lignes
supprimées
  }

  // Méthode pour fermer la base de données
  // Souvent pas nécessaire dans les petites apps, mais bonne
pratique
  Future close() async {
    final db = await _instance.database;
    db.close();
    _database = null;
  }
}
```

Explication de delete(int id) :

- Elle est async.
- final db = await database; : Obtient l'instance de la base de données.
- await db.delete(...) : Appelle la méthode delete de l'objet Database.
    - Le premier argument est le nom de la table (tableName).
    - where: '${NoteFields.id} = ?' et whereArgs: [id] : Indiquent à sqflite quelles lignes supprimer. Sans clause WHERE, db.delete() supprimerait *toutes* les lignes de la table ! Nous passons l'ID de la note à supprimer dans whereArgs.
- db.delete() retourne un Future<int> qui, une fois complété, contient le nombre de lignes qui ont été supprimées.

---

6. Récapitulatif de DatabaseHelper (Code Complet de ce Module)

Voici à quoi devrait ressembler votre fichier lib/database_helper.dart après avoir ajouté toutes les méthodes CRUD :

```
      import 'dart:async';
```

```dart
import 'package:sqflite/sqflite.dart';
import 'package:path/path.dart';

import 'models/note.dart'; // Importez votre modèle Note

final String databaseName = "notes.db";
final String tableName = "notes";

// Constantes pour les noms des colonnes de notre table
'notes'
class NoteFields {
  static final String id = '_id'; // Souvent "_id" pour
l'identifiant
  static final String title = 'title';
  static final String content = 'content';
  // Vous pourriez ajouter d'autres champs ici si nécessaire
(date, etc.)

  static final List<String> values = [ // Utile pour les
requêtes SELECT *
    id, title, content
  ];
}


class DatabaseHelper {
  // 1. Variable statique pour l'instance de la base de
données
  static Database? _database;

  // 2. Variable statique pour l'instance unique de la classe
DatabaseHelper (Singleton)
  static final DatabaseHelper _instance =
DatabaseHelper._privateConstructor();

  // 3. Constructeur privé
  DatabaseHelper._privateConstructor();

  // 4. Factory constructor pour le point d'accès global
  factory DatabaseHelper() {
    return _instance;
  }

  // 5. Getter asynchrone pour obtenir l'instance de la base
de données
  Future<Database> get database async {
    if (_database != null) return _database!;
    _database = await _initDatabase();
    return _database!;
  }
```

```
  // 6. Méthode privée asynchrone pour initialiser la base de
données
  Future<Database> _initDatabase() async {
    String databasesPath = await getDatabasesPath();
    String path = join(databasesPath, databaseName);
    print("Chemin de la base de données : $path");
    return await openDatabase(
      path,
      version: 1,
      onCreate: _onCreate,
      // onUpgrade: _onUpgrade, // Pour gérer les futures
versions
    );
  }

  // 7. Méthode privée asynchrone pour créer les tables
(appelée par onCreate)
  Future _onCreate(Database db, int version) async {
    await db.execute('''
      CREATE TABLE $tableName (
        ${NoteFields.id} INTEGER PRIMARY KEY AUTOINCREMENT,
        ${NoteFields.title} TEXT NOT NULL,
        ${NoteFields.content} TEXT NOT NULL
      )
    ''');
    print("Table '$tableName' créée avec succès !");
  }

  // --- MÉTHODES CRUD ---

  // CREATE
  Future<int> create(Note note) async {
    final db = await database;
    final id = await db.insert(
      tableName,
      note.toMap(),
      conflictAlgorithm: ConflictAlgorithm.ignore,
    );
    print("Note insérée avec l'ID : $id");
    return id;
  }

  // READ (Tout)
  Future<List<Note>> readAllNotes() async {
    final db = await database;
    final resultMaps = await db.query(
      tableName,
      orderBy: '${NoteFields.id} DESC',
      columns: NoteFields.values, // Optionnel
    );
```

```
    return resultMaps.map((map) =>
Note.fromMap(map)).toList();
  }

   // READ (Un seul)
  Future<Note?> readNote(int id) async {
    final db = await database;
    final maps = await db.query(
      tableName,
      columns: NoteFields.values,
      where: '${NoteFields.id} = ?',
      whereArgs: [id],
      limit: 1,
    );
    if (maps.isNotEmpty) {
      return Note.fromMap(maps.first);
    } else {
      return null;
    }
  }


  // UPDATE
  Future<int> update(Note note) async {
    final db = await database;
    final rowsAffected = await db.update(
      tableName,
      note.toMap(),
      where: '${NoteFields.id} = ?',
      whereArgs: [note.id],
    );
    print("Note avec l'ID ${note.id} mise à jour. Lignes
affectées : $rowsAffected");
    return rowsAffected;
  }

  // DELETE
  Future<int> delete(int id) async {
    final db = await database;
    final rowsDeleted = await db.delete(
      tableName,
      where: '${NoteFields.id} = ?',
      whereArgs: [id],
    );
    print("Note avec l'ID $id supprimée. Lignes supprimées :
$rowsDeleted");
    return rowsDeleted;
  }

  // Fermer la base de données
  Future close() async {
```

```
    final db = await _instance.database;
    db.close();
    _database = null;
  }
}
```

Conclusion

Le code donné ci-dessus présente toutes les méthodes nécessaires dans le DatabaseHelper pour effectuer les opérations CRUD sur la base de données SQLite.

Nous avons appris à :

- Utiliser un modèle de données (Note) pour représenter les informations et faciliter la conversion entre objets Dart et Map.
- Implémenter la méthode create() en utilisant db.insert().
- Implémenter les méthodes readAllNotes() et readNote() en utilisant db.query() et en convertissant la List<Map> résultante en List<Note>.
- Implémenter la méthode update() en utilisant db.update() avec une clause WHERE pour spécifier quelle ligne modifier.
- Implémenter la méthode delete() en utilisant db.delete() avec une clause WHERE pour spécifier quelle ligne supprimer.
- Utiliser **async** et **await** pour toutes ces opérations car elles interagissent avec le système de fichiers/la base de données.
- Utiliser les arguments where et whereArgs de manière sécurisée.

Notre logique de données est maintenant prête. Dans le prochain chapitre, nous allons connecter cette logique avec l'interface utilisateur que nous avons commencée dans le deuxième chapitre, permettant ainsi à l'utilisateur d'interagir avec la base de données via le formulaire et d'afficher la liste des notes.

Chapitre6 - Intégrer l'Interface Utilisateur et la Base de Données

Objectif de ce chapitre : Connecter les widgets de notre interface (le formulaire de saisie et une future liste d'affichage) avec les méthodes CRUD de notre classe DatabaseHelper afin de permettre aux utilisateurs d'enregistrer, lire, modifier et supprimer des notes.

1. Connecter le Formulaire et l'Opération "Create"

Nous avons déjà un formulaire (NoteFormPage) capable de valider et de sauvegarder les données dans des variables locales (_title, _content) lorsque le bouton "Enregistrer Note" est pressé. Maintenant, au lieu de simplement afficher les données dans la console, nous allons utiliser notre méthode DatabaseHelper().create().

Rappelez-vous que DatabaseHelper().create() est une méthode **asynchrone** (Future<int> create(Note note) async). Cela signifie que la fonction qui l'appelle (_saveNote dans NoteFormPage) doit également être async et utiliser await.

Nous allons également ajouter une gestion d'erreur basique et naviguer de retour vers la page principale une fois que la note est enregistrée.

Code : lib/note_form_page.dart (Mise à jour de _saveNote)

```
      import 'package:flutter/material.dart';
// N'oubliez pas d'importer vos classes DatabaseHelper et Note
import 'database_helper.dart';
import 'models/note.dart';

class NoteFormPage extends StatefulWidget {
  // Le constructeur sera modifié plus tard pour la mise à
jour
  const NoteFormPage({Key? key}) : super(key: key);

  @override
  _NoteFormPageState createState() => _NoteFormPageState();
}

class _NoteFormPageState extends State<NoteFormPage> {
  final _formKey = GlobalKey<FormState>();

  // Variables pour stocker les valeurs du formulaire après
sauvegarde
  String _title = '';
```

```
  String _content = '';

  final TextEditingController _titleController =
TextEditingController();
  final TextEditingController _contentController =
TextEditingController();

  @override
  void dispose() {
    _titleController.dispose();
    _contentController.dispose();
    super.dispose();
  }

  // Rendre la fonction _saveNote asynchrone
  Future<void> _saveNote() async {
    // Vérifie si le formulaire est valide
    if (_formKey.currentState!.validate()) {
      // Sauvegarde les données du formulaire dans nos
variables locales
      _formKey.currentState!.save();

      // --- Ici, on utilise DatabaseHelper pour créer la note
---
      // Créer un objet Note à partir des données du
formulaire
      // L'ID est null pour une nouvelle note, il sera généré
par la DB
      final newNote = Note(
        title: _title,
        content: _content,
      );

      // Appeler la méthode create de notre DatabaseHelper
      // Utilisation de try-catch pour gérer les erreurs
potentielles de la DB
      try {
        // Utilisation de await car DatabaseHelper().create()
retourne un Future
        final id = await DatabaseHelper().create(newNote);
        print('Note insérée avec l\'ID : $id');

        // Afficher un message de succès
        ScaffoldMessenger.of(context).showSnackBar(
          const SnackBar(content: Text('Note enregistrée avec
succès !')),
        );

        // Revenir à la page précédente (la liste des notes)
```

```
        // await permet d'attendre que la navigation soit
terminée avant de potentiellement rafraîchir la liste sur la
page précédente
        Navigator.pop(context, true); // On peut passer 'true'
pour indiquer que l'opération a réussi (utile pour rafraîchir
la liste)

      } catch (e) {
        // Gérer l'erreur d'insertion
        print('Erreur lors de l\'enregistrement de la note :
$e');
        ScaffoldMessenger.of(context).showSnackBar(
          SnackBar(content: Text('Erreur: Impossible
d\'enregistrer la note.')),
        );
      }
      // ----------------------------------------------------
---

      // Optionnel : Effacer le formulaire après la sauvegarde
(si on ne revient pas en arrière)
      // _titleController.clear();
      // _contentController.clear();
      // _formKey.currentState!.reset();
    }
  }


  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Nouvelle Note'), // Sera rendu
dynamique plus tard
      ),
      body: Padding(
        padding: const EdgeInsets.all(16.0),
        child: Form(
          key: _formKey,
          child: ListView(
            children: <Widget>[
              // Champ pour le Titre
              TextFormField(
                controller: _titleController,
                decoration: const InputDecoration(
                  labelText: 'Titre',
                  hintText: 'Entrez le titre de la note',
                  border: OutlineInputBorder(),
                ),
                validator: (value) {
                  if (value == null || value.isEmpty) {
```

```
                    return 'Veuillez entrer un titre';
                  }
                  return null;
                },
                onSaved: (value) {
                  _title = value ?? '';
                },
              ),
              const SizedBox(height: 16.0),

              // Champ pour le Contenu
              TextFormField(
                controller: _contentController,
                decoration: const InputDecoration(
                  labelText: 'Contenu',
                  hintText: 'Entrez le contenu de la note',
                  border: OutlineInputBorder(),
                ),
                keyboardType: TextInputType.multiline,
                maxLines: null,
                validator: (value) {
                  if (value == null || value.isEmpty) {
                    return 'Veuillez entrer un contenu';
                  }
                  return null;
                },
                onSaved: (value) {
                  _content = value ?? '';
                },
              ),
              const SizedBox(height: 24.0),

              // Bouton de soumission
              ElevatedButton(
                onPressed: _saveNote, // Appel de notre
fonction asynchrone
                child: const Text('Enregistrer Note'), // Sera
rendu dynamique plus tard
              ),
            ],
          ),
        ),
      ),
    );
  }
}
```

Explication :

- La méthode _saveNote est maintenant marquée async.

- À l'intérieur, après la sauvegarde du formulaire (_formKey.currentState!.save()), nous créons un objet Note avec les données _title et _content.
- Nous appelons await DatabaseHelper().create(newNote);. L'utilisation de await fait que l'exécution de _saveNote se met en pause jusqu'à ce que l'opération d'insertion dans la base de données soit terminée.
- Un bloc try...catch enveloppe l'appel à la base de données pour intercepter les erreurs potentielles.
- ScaffoldMessenger est utilisé pour afficher un message (SnackBar) à l'utilisateur indiquant le succès ou l'échec.
- Navigator.pop(context, true); est appelé pour revenir à l'écran précédent (notre page principale). Le deuxième argument true est un résultat qui peut être récupéré sur la page appelante.

Testez en lançant l'application, en cliquant sur le bouton "+", en remplissant le formulaire, et en cliquant sur "Enregistrer Note". Si tout se passe bien, vous devriez voir les messages "Note insérée..." dans la console et un SnackBar à l'écran, puis vous serez redirigé vers la page principale. La note est maintenant dans la base de données, mais nous ne la voyons pas encore.

5.2. Afficher les Données (Opération "Read")

Nous allons maintenant modifier notre page principale (MyHomePage) pour afficher la liste des notes stockées dans la base de données. Comme la lecture des données est une opération **asynchrone** (DatabaseHelper().readAllNotes() retourne un Future<List<Note>>), nous allons utiliser un FutureBuilder.

- FutureBuilder est un widget très utile qui permet de construire l'interface utilisateur en fonction de l'état d'un Future. Il peut afficher un indicateur de chargement pendant que le Future est en cours, un message d'erreur s'il échoue, ou les données une fois qu'il est complété.

Code : lib/main.dart (Mise à jour du corps de MyHomePage)

```
      import 'package:flutter/material.dart';
import 'package:my_crud_app/note_form_page.dart';
import 'package:my_crud_app/database_helper.dart';
import 'package:my_crud_app/models/note.dart'; // Importez le
modèle Note

Future<void> main() async {
  WidgetsFlutterBinding.ensureInitialized();
  try {
```

```
    await DatabaseHelper().database;
    print("Base de données initialisée avec succès !");
  } catch (e) {
    print("Erreur lors de l'initialisation de la base de
données : $e");
    // Ici, dans une vraie application, vous voudriez afficher
une erreur à l'utilisateur ou quitter.
  }
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Mon App CRUD',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: const MyHomePage(title: 'Application de Notes
CRUD'),
    );
  }
}

class MyHomePage extends StatefulWidget {
  const MyHomePage({Key? key, required this.title}) :
super(key: key);

  final String title;

  @override
  State<MyHomePage> createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {

  // Déclarer une variable pour stocker le Future qui récupère
les notes.
  // Nous l'initialiserons dans initState et la re-assignerons
pour rafraîchir la liste.
  late Future<List<Note>> _notesFuture;

  // Initialiser le Future la première fois que le widget est
créé
  @override
  void initState() {
    super.initState();
```

```
    _refreshNotesList(); // Appeler la méthode pour charger
les notes
  }

  // Méthode pour charger/rafraîchir la liste des notes
  void _refreshNotesList() {
    // setState déclenchera une reconstruction du widget
    // Ré-assigner _notesFuture à un nouveau Future force le
FutureBuilder à attendre à nouveau
    setState(() {
      _notesFuture = DatabaseHelper().readAllNotes();
    });
  }

  // Méthode pour naviguer vers la page du formulaire (pour
création ou modification)
  void _navigateToNoteForm({Note? note}) async {
    // Utilisation de await Navigator.push pour attendre le
retour de la page du formulaire
    final result = await Navigator.push(
      context,
      MaterialPageRoute(
        builder: (context) => NoteFormPage(
          // Plus tard, nous passerons l'objet note ici pour
la modification
          // note: note,
        ),
      ),
    );

    // Vérifier si la page du formulaire a renvoyé un résultat
indiquant un changement
    // (dans notre cas, true si une note a été
enregistrée/modifiée)
    if (result == true) {
      _refreshNotesList(); // Rafraîchir la liste si une note
a été enregistrée/modifiée
    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      ),
      // --- Corps de la page : Utilisation de FutureBuilder
pour afficher la liste ---
      body: FutureBuilder<List<Note>>( // Spécifier le type
attendu par le Future
        future: _notesFuture, // Le Future à attendre
```

```
        builder: (context, snapshot) {
          // snapshot contient l'état actuel du Future et les
données/erreur
          if (snapshot.connectionState ==
ConnectionState.waiting) {
            // Si le Future est encore en cours d'exécution,
afficher un indicateur de chargement
            return const Center(child:
CircularProgressIndicator());
          } else if (snapshot.hasError) {
            // Si une erreur est survenue
            print('Erreur FutureBuilder: ${snapshot.error}');
// Pour déboguer
            return Center(child: Text('Erreur:
${snapshot.error}'));
          } else if (!snapshot.hasData ||
snapshot.data!.isEmpty) {
            // Si le Future est terminé mais il n'y a pas de
données
            return const Center(child: Text('Aucune note
trouvée. Créez-en une !'));
          } else {
            // Si le Future est terminé et a des données,
afficher la liste
            final notes = snapshot.data!; // Récupérer la
liste des notes

            return ListView.builder(
              itemCount: notes.length,
              itemBuilder: (context, index) {
                final note = notes[index];
                // Widget simple pour afficher chaque note
dans la liste
                return ListTile(
                  title: Text(note.title),
                  subtitle: Text(
                    note.content,
                    maxLines: 2, // Afficher seulement les 2
premières lignes
                    overflow: TextOverflow.ellipsis, //
Ajouter '...' si le texte est tronqué
                  ),
                  // --- Ajouterons les actions Update et
Delete ici plus tard ---
                   onTap: () {
                       // Action de modification (sera
implémentée dans les sections suivantes)
                       // _navigateToNoteForm(note: note);
                   },
                );
              },
```

```
            );
          }
        },
      ),
      // ------------------------------------------------------
------------------

      floatingActionButton: FloatingActionButton(
        onPressed: _navigateToNoteForm, // Appelle la fonction
de navigation (pour créer une note)
        tooltip: 'Ajouter une note',
        child: const Icon(Icons.add),
      ),
    );
  }
}
```

Explication :

- Nous avons ajouté import 'package:my_crud_app/models/note.dart';.
- Dans _MyHomePageState, nous avons déclaré late Future<List<Note>> _notesFuture;. Le mot-clé late signifie que la variable sera initialisée plus tard (dans initState).
- Dans initState, nous appelons _refreshNotesList() pour lancer la première lecture des notes.
- _refreshNotesList() : Cette méthode simple utilise setState et ré-assigne _notesFuture au résultat de DatabaseHelper().readAllNotes(). Le setState indique à Flutter que l'état du widget a changé et qu'il doit se reconstruire. En ré-assignant _notesFuture, le FutureBuilder verra un nouveau Future à attendre.
- Le body du Scaffold est maintenant un FutureBuilder<List<Note>>.
- Sa propriété future est définie sur _notesFuture.
- Son builder est une fonction qui reçoit le context et un snapshot. Le snapshot contient l'état du Future.
- Nous vérifions snapshot.connectionState pour afficher différents widgets :
    - ConnectionState.waiting : CircularProgressIndicator (roue de chargement).
    - snapshot.hasError : Un message d'erreur si quelque chose s'est mal passé.
    - !snapshot.hasData || snapshot.data!.isEmpty : Un message si la liste est vide.

    - Sinon (quand ConnectionState.done et hasData est vrai) : On utilise ListView.builder pour construire une liste d'éléments (ListTile) à partir des données snapshot.data!.

- Dans le ListTile, nous affichons le titre et le contenu de chaque note. maxLines et overflow: TextOverflow.ellipsis sont ajoutés pour gérer le contenu long.
- La méthode _navigateToNoteForm dans MyHomePageState est maintenant async et utilise await sur le Navigator.push. Cela nous permet de récupérer le résultat qui est potentiellement retourné par la page cible (le true que nous avons ajouté dans Navigator.pop sur NoteFormPage). Si le résultat est true, cela signifie qu'une note a été sauvegardée, et nous appelons _refreshNotesList() pour mettre à jour l'affichage.

Lancez l'application. Si vous avez déjà ajouté des notes dans le module précédent, elles devraient maintenant s'afficher dans la liste. Si la base de données est vide, vous verrez le message "Aucune note trouvée...". Ajouter une nouvelle note via le bouton "+" devrait maintenant enregistrer la note *et* rafraîchir automatiquement la liste à votre retour.

---

3. et 4. Implémenter l'Opération "Update" (Modification)

La modification implique deux étapes principales :

1. Permettre à l'utilisateur de sélectionner une note à modifier (depuis la liste).
2. Pré-remplir le formulaire avec les données de cette note et modifier la logique du bouton "Enregistrer" pour appeler DatabaseHelper().update() au lieu de create().

Étape 1 : Rendre les éléments de la liste cliquables et naviguer

Nous allons modifier le ListTile dans MyHomePage pour qu'il utilise sa propriété onTap afin de naviguer vers la NoteFormPage, en lui passant l'objet Note à modifier.

Code : lib/main.dart (Mise à jour de _navigateToNoteForm et ListTile.onTap)

```
      // ... imports et code précédent ...

class _MyHomePageState extends State<MyHomePage> {
  late Future<List<Note>> _notesFuture;

  @override
  void initState() {
    super.initState();
    _refreshNotesList();
  }

  void _refreshNotesList() {
```

```
    setState(() {
      _notesFuture = DatabaseHelper().readAllNotes();
    });
  }

  // Méthode pour naviguer vers la page du formulaire (pour
création ou modification)
  // Le paramètre optionnel 'note' permet de passer l'objet
Note à modifier (null pour création)
  void _navigateToNoteForm({Note? note}) async {
    final result = await Navigator.push(
      context,
      MaterialPageRoute(
        builder: (context) => NoteFormPage(
          note: note, // Passage de l'objet note au
constructeur de NoteFormPage
        ),
      ),
    );

    if (result == true) {
      _refreshNotesList();
    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      ),
      body: FutureBuilder<List<Note>>(
        future: _notesFuture,
        builder: (context, snapshot) {
          if (snapshot.connectionState ==
ConnectionState.waiting) {
            return const Center(child:
CircularProgressIndicator());
          } else if (snapshot.hasError) {
             print('Erreur FutureBuilder: ${snapshot.error}');
            return Center(child: Text('Erreur:
${snapshot.error}'));
          } else if (!snapshot.hasData ||
snapshot.data!.isEmpty) {
            return const Center(child: Text('Aucune note
trouvée. Créez-en une !'));
          } else {
            final notes = snapshot.data!;

            return ListView.builder(
              itemCount: notes.length,
```

```
              itemBuilder: (context, index) {
                final note = notes[index];
                return ListTile(
                  title: Text(note.title),
                  subtitle: Text(
                    note.content,
                    maxLines: 2,
                    overflow: TextOverflow.ellipsis,
                  ),
                  // --- Action de modification : Appeler
_navigateToNoteForm avec la note ---
                   onTap: () {
                       _navigateToNoteForm(note: note); //
Passe la note à modifier
                   },
                  // --- L'icône de suppression sera ajoutée
plus tard ---
                  // trailing: IconButton(...),
                );
              },
            );
          }
        },
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _navigateToNoteForm, // Toujours utilisé
pour la création (note est null par défaut)
        tooltip: 'Ajouter une note',
        child: const Icon(Icons.add),
      ),
    );
  }
}
```

Étape 2 : Modifier NoteFormPage pour gérer la modification

Nous allons modifier NoteFormPage pour accepter une Note optionnelle dans son constructeur. Si une note est passée, cela signifie qu'on est en mode modification.

- Le initState vérifiera si une note a été passée et pré-remplira les champs du formulaire avec ses données en utilisant les TextEditingController.
- La méthode _saveNote vérifiera si un ID existe pour la note. S'il existe, elle appellera update(). Sinon, elle appellera create().
- Le titre de l'AppBar et le texte du bouton seront mis à jour en fonction du mode (Création ou Modification).

Code : lib/note_form_page.dart (Mise à jour pour gérer modification)

```
      import 'package:flutter/material.dart';
import 'database_helper.dart';
import 'models/note.dart';

class NoteFormPage extends StatefulWidget {
  // Ajout d'un paramètre optionnel 'note' pour la
modification
  final Note? note;

  const NoteFormPage({
    Key? key,
    this.note, // Le paramètre note est optionnel
  }) : super(key: key);

  @override
  _NoteFormPageState createState() => _NoteFormPageState();
}

class _NoteFormPageState extends State<NoteFormPage> {
  final _formKey = GlobalKey<FormState>();

  // Les variables _title et _content stockeront les valeurs
finales après save()
  // Elles peuvent être initialisées avec les valeurs de la
note si mode modification
  late String _title;
  late String _content;

  // Controllers pour les champs de texte (utiles pour pré-
remplir et effacer)
  final TextEditingController _titleController =
TextEditingController();
  final TextEditingController _contentController =
TextEditingController();

  // Variable pour stocker l'ID de la note si on est en mode
modification
  int? _noteId;

  // Initialisation de l'état du widget
  @override
  void initState() {
    super.initState();

    // Si une note a été passée au constructeur (mode
modification)
    if (widget.note != null) {
      _noteId = widget.note!.id; // Stocker l'ID
      _title = widget.note!.title; // Initialiser les
variables locales
      _content = widget.note!.content;
```

```
      // Pré-remplir les champs de texte avec les valeurs de
la note
      _titleController.text = _title;
      _contentController.text = _content;
    } else {
      // Si aucune note n'a été passée (mode création)
      _title = ''; // Initialiser les variables locales à vide
      _content = '';
      // Les controllers sont déjà vides par défaut, rien à
faire ici.
    }
  }


  @override
  void dispose() {
    // N'oubliez pas de disposer des controllers
    _titleController.dispose();
    _contentController.dispose();
    super.dispose();
  }

  // Fonction pour enregistrer/modifier la note
  Future<void> _saveNote() async {
    if (_formKey.currentState!.validate()) {
      _formKey.currentState!.save();

      // --- Logique pour CREATE ou UPDATE ---
      final noteToSave = Note(
        id: _noteId, // Inclure l'ID si mode modification
(sera null en création)
        title: _title,
        content: _content,
      );

      try {
        if (_noteId == null) {
          // Si _noteId est null, c'est une nouvelle note ->
CRÉER
          final id = await
DatabaseHelper().create(noteToSave);
          print('Note créée avec l\'ID : $id');
          ScaffoldMessenger.of(context).showSnackBar(
            const SnackBar(content: Text('Note enregistrée
avec succès !')),
          );
        } else {
          // Si _noteId n'est pas null, c'est une note
existante -> MODIFIER
          final rowsAffected = await
DatabaseHelper().update(noteToSave);
```

```
          print('Note avec ID $_noteId mise à jour. Lignes
affectées: $rowsAffected');
           ScaffoldMessenger.of(context).showSnackBar(
            const SnackBar(content: Text('Note mise à jour
avec succès !')),
          );
        }

        // Revenir à la page précédente, en indiquant que
l'opération a réussi
        Navigator.pop(context, true);

      } catch (e) {
        // Gérer l'erreur d'insertion/mise à jour
        print('Erreur lors de l\'enregistrement/mise à jour :
$e');
        ScaffoldMessenger.of(context).showSnackBar(
          SnackBar(content: Text('Erreur: Impossible
d\'enregistrer la note.')),
        );
      }
      // -----------------------------------
    }
  }

  @override
  Widget build(BuildContext context) {
    // Déterminer le titre de l'AppBar en fonction du mode
(création ou modification)
    final appBarTitle = _noteId == null ? 'Nouvelle Note' :
'Modifier Note';
    final buttonText = _noteId == null ? 'Enregistrer Note' :
'Mettre à jour Note';


    return Scaffold(
      appBar: AppBar(
        title: Text(appBarTitle), // Titre dynamique
      ),
      body: Padding(
        padding: const EdgeInsets.all(16.0),
        child: Form(
          key: _formKey,
          child: ListView(
            children: <Widget>[
              // Champ pour le Titre (lié au controller pour
pré-remplissage)
              TextFormField(
                controller: _titleController,
                decoration: const InputDecoration(
                  labelText: 'Titre',
```

```
                hintText: 'Entrez le titre de la note',
                border: OutlineInputBorder(),
              ),
              validator: (value) {
                if (value == null || value.isEmpty) {
                  return 'Veuillez entrer un titre';
                }
                return null;
              },
              // onSaved est toujours nécessaire pour
capturer la valeur finale après validation
              onSaved: (value) {
                _title = value ?? '';
              },
            ),
            const SizedBox(height: 16.0),

            // Champ pour le Contenu (lié au controller pour
pré-remplissage)
            TextFormField(
              controller: _contentController,
              decoration: const InputDecoration(
                labelText: 'Contenu',
                hintText: 'Entrez le contenu de la note',
                border: OutlineInputBorder(),
              ),
              keyboardType: TextInputType.multiline,
              maxLines: null,
              validator: (value) {
                if (value == null || value.isEmpty) {
                  return 'Veuillez entrer un contenu';
                }
                return null;
              },
               // onSaved est toujours nécessaire pour
capturer la valeur finale après validation
              onSaved: (value) {
                _content = value ?? '';
              },
            ),
            const SizedBox(height: 24.0),

            // Bouton de soumission
            ElevatedButton(
              onPressed: _saveNote,
              child: Text(buttonText), // Texte du bouton
dynamique
            ),
          ],
        ),
      ),
```

```
        ),
      );
    }
}
```

Explication :

- Le constructeur de NoteFormPage accepte maintenant un Note? note.
- Dans _NoteFormPageState, initState vérifie widget.note. S'il n'est pas null, il initialise _noteId, _title, _content et pré-remplit les TextEditingControllers.
- La méthode _saveNote crée un objet noteToSave. Crucialement, elle inclut l'_noteId s'il est défini.
- Un simple if (_noteId == null) permet de choisir entre appeler DatabaseHelper().create() ou DatabaseHelper().update(). Notez que update a besoin de l'ID pour savoir quelle ligne modifier.
- Les textes de l'AppBar et du bouton sont rendus dynamiques en fonction de si _noteId est null ou non.
- On conserve onSaved sur les TextFormField car il est toujours nécessaire pour la sauvegarde *après validation*, même si les controllers sont utilisés pour le pré-remplissage. Les controllers gèrent l'état du champ *pendant* la saisie, onSaved capture la valeur *finale* validée.

Lancez l'application. Vous devriez pouvoir ajouter de nouvelles notes (elles apparaissent après le rafraîchissement). En tapant sur une note existante dans la liste, vous devriez naviguer vers la page du formulaire avec les champs pré-remplis, et le titre/bouton devraient indiquer "Modifier Note". Modifier les champs et cliquer sur "Mettre à jour Note" devrait sauvegarder les changements et rafraîchir la liste.

5.5. Implémenter l'Opération "Delete" (Suppression)

La suppression est plus simple. Nous allons ajouter une icône à chaque ListTile dans MyHomePage. En tapant sur l'icône, nous appellerons DatabaseHelper().delete() et rafraîchirons la liste.

Code : lib/main.dart (Ajout de IconButton pour supprimer)

```
      import 'package:flutter/material.dart';
import 'package:my_crud_app/note_form_page.dart';
import 'package:my_crud_app/database_helper.dart';
import 'package:my_crud_app/models/note.dart';
```

```
Future<void> main() async {
  WidgetsFlutterBinding.ensureInitialized();
  try {
    await DatabaseHelper().database;
    print("Base de données initialisée avec succès !");
  } catch (e) {
    print("Erreur lors de l'initialisation de la base de
données : $e");
    // Gérer l'erreur...
  }
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Mon App CRUD',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: const MyHomePage(title: 'Application de Notes
CRUD'),
    );
  }
}

class MyHomePage extends StatefulWidget {
  const MyHomePage({Key? key, required this.title}) :
super(key: key);

  final String title;

  @override
  State<MyHomePage> createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {

  late Future<List<Note>> _notesFuture;

  @override
  void initState() {
    super.initState();
    _refreshNotesList();
  }

  void _refreshNotesList() {
    setState(() {
```

```
      _notesFuture = DatabaseHelper().readAllNotes();
    });
  }

  void _navigateToNoteForm({Note? note}) async {
    final result = await Navigator.push(
      context,
      MaterialPageRoute(
        builder: (context) => NoteFormPage(
          note: note,
        ),
      ),
    );

    if (result == true) {
      _refreshNotesList();
    }
  }

  // --- Méthode pour gérer la suppression ---
  void _deleteNote(int id) async {
    // Afficher une boîte de dialogue de confirmation (bonne
pratique !)
    bool confirm = await showDialog(
      context: context,
      builder: (BuildContext context) {
        return AlertDialog(
          title: const Text('Confirmer la suppression'),
          content: const Text('Êtes-vous sûr de vouloir
supprimer cette note ?'),
          actions: <Widget>[
            TextButton(
              onPressed: () =>
Navigator.of(context).pop(false), // Non, retourner false
              child: const Text('Annuler'),
            ),
            TextButton(
              onPressed: () =>
Navigator.of(context).pop(true), // Oui, retourner true
              child: const Text('Supprimer'),
            ),
          ],
        );
      },
    ) ?? false; // Utilise false si showDialog retourne null
(par ex, dialogue fermé sans choisir)


    if (confirm) {
      // Si l'utilisateur a confirmé la suppression
      try {
```

```
        final rowsDeleted = await DatabaseHelper().delete(id);
// Appeler la méthode delete
        print('Note avec ID $id supprimée. Lignes supprimées:
$rowsDeleted');

        // Afficher un message de succès
        ScaffoldMessenger.of(context).showSnackBar(
          const SnackBar(content: Text('Note supprimée !')),
        );

        // Rafraîchir la liste après la suppression
        _refreshNotesList();

      } catch (e) {
        // Gérer l'erreur de suppression
        print('Erreur lors de la suppression : $e');
        ScaffoldMessenger.of(context).showSnackBar(
          SnackBar(content: Text('Erreur: Impossible de
supprimer la note.')),
        );
      }
    }
  }
  // ----------------------------------------


  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      ),
      body: FutureBuilder<List<Note>>(
        future: _notesFuture,
        builder: (context, snapshot) {
          if (snapshot.connectionState ==
ConnectionState.waiting) {
            return const Center(child:
CircularProgressIndicator());
          } else if (snapshot.hasError) {
             print('Erreur FutureBuilder: ${snapshot.error}');
            return Center(child: Text('Erreur:
${snapshot.error}'));
          } else if (!snapshot.hasData ||
snapshot.data!.isEmpty) {
            return const Center(child: Text('Aucune note
trouvée. Créez-en une !'));
          } else {
            final notes = snapshot.data!;

            return ListView.builder(
```

```
              itemCount: notes.length,
              itemBuilder: (context, index) {
                final note = notes[index];
                return ListTile(
                  title: Text(note.title),
                  subtitle: Text(
                    note.content,
                    maxLines: 2,
                    overflow: TextOverflow.ellipsis,
                  ),
                  onTap: () {
                       _navigateToNoteForm(note: note); //
Naviguer pour modifier
                   },
                  // --- Action de suppression : Ajouter un
IconButton ---
                  trailing: IconButton(
                    icon: const Icon(Icons.delete),
                    color: Colors.red, // Optionnel : couleur
rouge pour supprimer
                    onPressed: () {
                      // Appeler notre méthode de suppression,
en lui passant l'ID de la note
                       _deleteNote(note.id!); // Utiliser
note.id! car les notes lues ont un ID
                    },
                  ),
                  // ----------------------------------------
-----------
                );
              },
            );
          }
        },
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _navigateToNoteForm, // Toujours utilisé
pour la création
        tooltip: 'Ajouter une note',
        child: const Icon(Icons.add),
      ),
    );
  }
}
```

Explication :

- Nous avons ajouté une nouvelle méthode _deleteNote(int id) dans _MyHomePageState. Elle est async car elle appelle une opération de base de données asynchrone.

- Une boîte de dialogue de confirmation (showDialog) est utilisée (bonne pratique UI) avant de procéder à la suppression effective. showDialog retourne un Future qui se complète avec le résultat passé à Navigator.of(context).pop(). Nous utilisons await showDialog(...) pour attendre le choix de l'utilisateur.
- Si l'utilisateur confirme (confirm == true), nous appelons await DatabaseHelper().delete(id); à l'intérieur d'un try...catch.
- Après la suppression réussie (ou l'échec), nous affichons un SnackBar pour informer l'utilisateur.
- Crucialement, après une suppression réussie, nous appelons _refreshNotesList() pour forcer le FutureBuilder à recharger la liste depuis la base de données et mettre à jour l'affichage.
- Dans le ListView.builder, nous avons ajouté un trailing: IconButton(...) à chaque ListTile.
- L'onPressed de cet IconButton appelle _deleteNote, en lui passant l'ID de la note courante (note.id!). Nous utilisons ! car nous savons que les notes lues depuis la base de données ont un ID non-null.

Lancez l'application. Vous devriez maintenant voir une icône de suppression à droite de chaque note. Cliquez dessus, confirmez la suppression, et la note devrait disparaître de la liste et de la base de données.

6. Réflexion sur le Rafraîchissement de l'UI

Notez le pattern de rafraîchissement que nous avons utilisé :

1. L'UI charge initialement les données en appelant une fonction de lecture (DatabaseHelper().readAllNotes()) et en stockant le Future dans une variable d'état (_notesFuture).
2. FutureBuilder attend ce Future et affiche la liste.
3. Lorsqu'une opération CRUD (Créer, Modifier, Supprimer) a lieu :
    - La logique de la base de données est appelée (await dbHelper.create/update/delete(...)).
    - Une fois l'opération terminée, on appelle setState(() { _notesFuture = DatabaseHelper().readAllNotes(); });. Cela déclenche une reconstruction du widget MyHomePage.

    - Lors de la reconstruction, le FutureBuilder voit que sa propriété future a changé (c'est maintenant un *nouveau* Future obtenu par le deuxième appel à readAllNotes()).
    - FutureBuilder attend alors ce nouveau Future, affiche l'indicateur de chargement brièvement, puis affiche les nouvelles données.

Cette méthode de recharger les données complètes après chaque modification n'est pas la plus efficace pour des applications très complexes ou avec d'énormes quantités de données (on pourrait mettre à jour la liste en mémoire directement), mais pour la plupart des applications CRUD simples, c'est une approche courante, facile à comprendre et suffisante en termes de performance. C'est un bon point de départ.

Pour des applications plus grandes, l'utilisation d'une gestion de l'état plus avancée (comme Provider, Bloc, Riverpod, etc.) qui notifie les widgets uniquement des changements spécifiques peut améliorer les performances et la structure du code.

Conclusion

Vous avez construit une application CRUD locale fonctionnelle avec Flutter et Sqflite !

Vous avez appris à :

- Connecter les actions de l'interface utilisateur (clic sur un bouton, clic sur un élément de liste) aux méthodes de base de données dans votre DatabaseHelper.
- Utiliser **async** et **await** de manière cohérente pour toutes les interactions avec la base de données.
- Utiliser FutureBuilder pour afficher élégamment des données provenant d'un Future (notre opération de lecture).
- Passer des données entre écrans lors de la navigation (passer un objet Note pour la modification).
- Mettre à jour la logique du formulaire (NoteFormPage) pour gérer à la fois la création et la modification d'une note en fonction des données passées.
- Implémenter l'opération de suppression et inclure une boîte de dialogue de confirmation.
- Mettre en place un mécanisme simple pour rafraîchir l'affichage de la liste après chaque opération CRUD (Create, Update, Delete) en ré-exécutant la requête de lecture et en utilisant setState pour déclencher la reconstruction du FutureBuilder.

Vous avez maintenant une application complète capable de gérer des notes persistantes localement. C'est une compétence fondamentale pour de nombreux types d'applications mobiles !