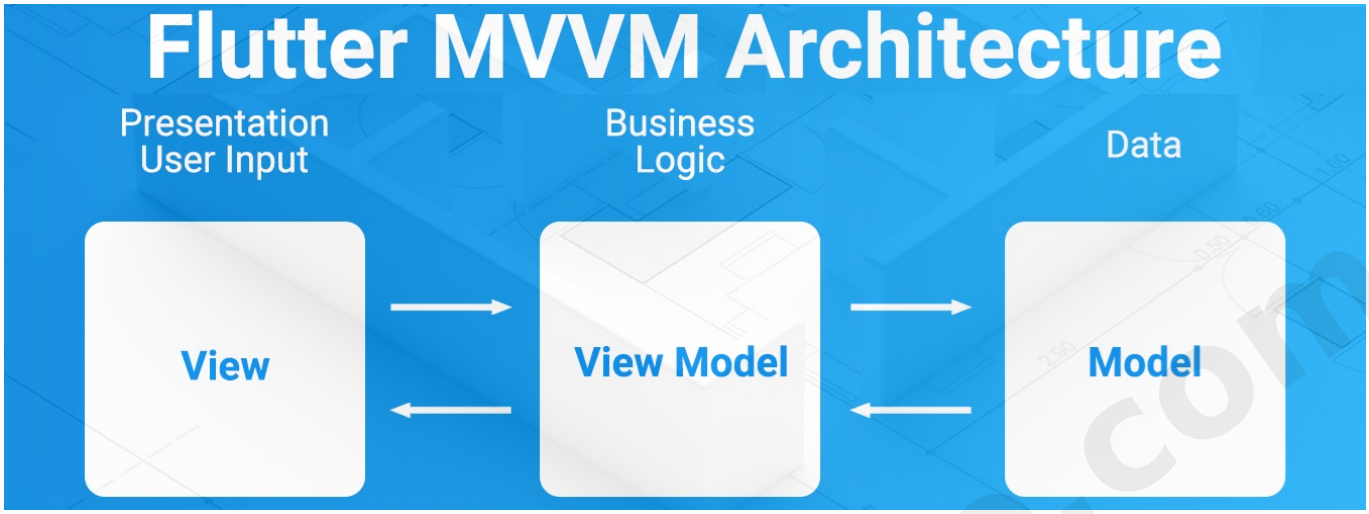
Comprendre l'Architecture MVVM dans Flutter

Comprendre l'Architecture MVVM dans Flutter



1. **Objectif**
   - Comprendre les Fondamentaux de l’Architecture MVVM
   - Appliquer MVVM dans un Projet Flutter
   - Créer une Application Modulaire et Testable
   - Acquérir des Pratiques pour des Applications de Grande Échelle

2. **Présentation**
   - Le développement d'applications mobiles modernes nécessite une architecture solide pour assurer la maintenabilité, la testabilité et la scalabilité du code.
   - MVVM (Model-View-ViewModel) est un modèle architectural utilisé dans le développement de logiciels, notamment dans Flutter, pour séparer les préoccupations d'une application en trois composants distincts : `Model`, `View` et `ViewModel`. Cette séparation des préoccupations permet de créer une base de code bien organisée et maintenable.
   - Le modèle `MVVM` est une variante du modèle de conception classique Modèle-Vue-Contrôleur (MVC) et est principalement utilisé pour créer des interfaces utilisateur.
   - Le modèle **Modèle-Vue-VueModel (MVVM)** sépare l'interface utilisateur d'une application en trois composants distincts : le modèle, la vue (View) et le ViewModel. Chaque élément joue un rôle spécifique dans l'architecture globale de l'application.

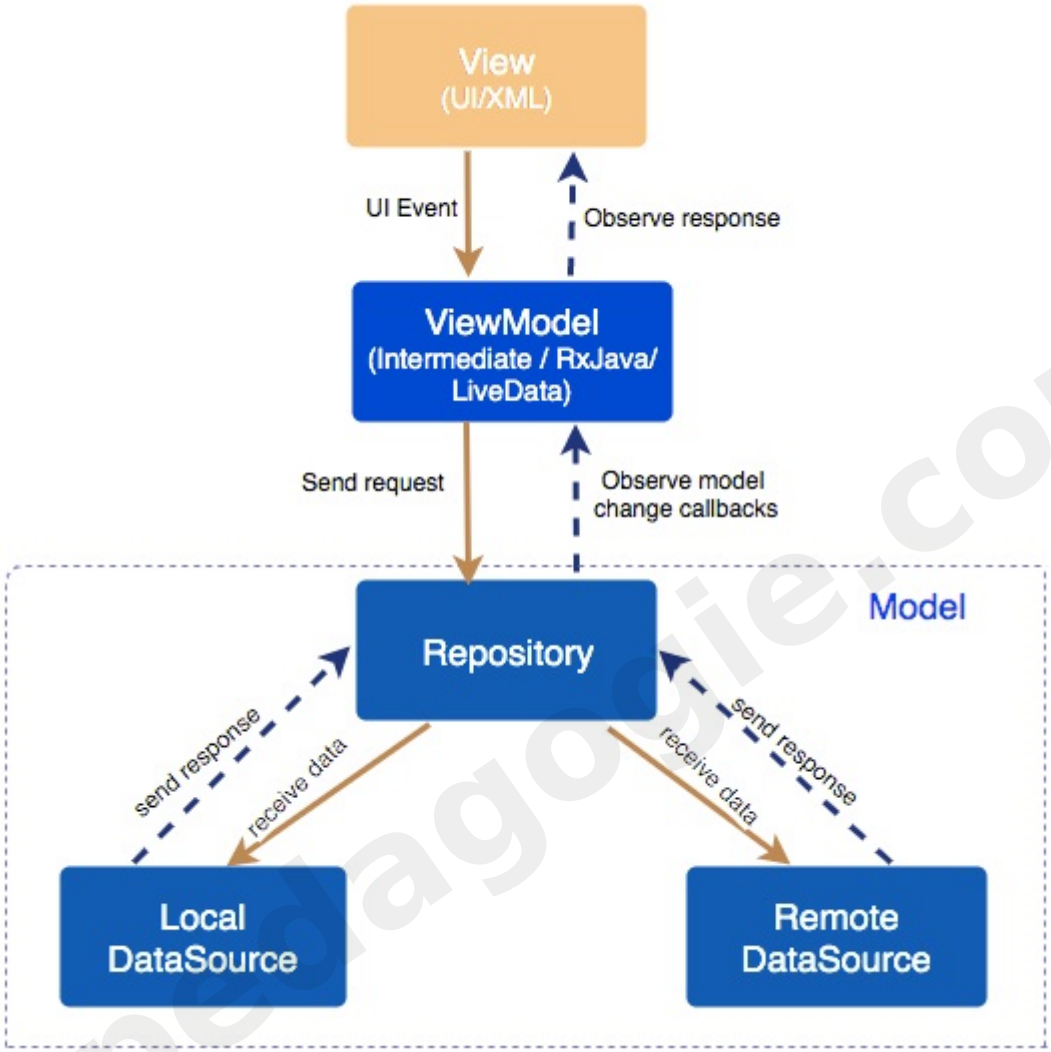
apcpedagogie.com

3. Qu'est-ce que Modèle-Vue-VueModèle ?

- L'architecture `Model-View-ViewModel (MVVM)` est une combinaison unique de modèles d'architecture logicielle qui prend en charge la séparation de l'interface utilisateur (qui est View) du développement de la logique métier ou de la logique backend (Model).
- Le modèle de vue à l'intérieur de **MVVM** est le pont responsable de la conversion des données d'une manière qui se comporte en fonction des changements qui se produisent dans l'interface utilisateur.
- De plus, pour connaître les responsabilités des trois composants, il est également important de comprendre comment les composants interagissent entre eux. Au plus haut niveau, la vue « connaît » le modèle de vue et le modèle de vue « connaît » le modèle , mais le modèle n'a pas connaissance du modèle de vue et le modèle de vue n'a pas connaissance de la vue .

apcpedagogie.com

4. `Model`**(Modèle)**



- Le modèle représente les données et la logique métier de l'application.
- Il est responsable du stockage, de la récupération et de la manipulation des données dont l'application a besoin pour fonctionner.
- Le modèle doit être complètement indépendant de la vue et du `ViewModel`, et ne doit contenir que la logique liée aux données elles-mêmes.
- Le modèle dans l'architecture MVVM est le cœur de l'application. Il est responsable de la gestion de toute la logique métier et de la gestion des données. Le modèle comprend toutes les classes de modèle, toutes les classes de référentiel et toutes les classes utilitaires.
- Les classes utilitaires fournissent des fonctions d'assistance utilisées dans l'ensemble de l'application. Il peut s'agir de fonctions de formatage de date, de manipulation de

apcpedagogie.com

chaînes ou de vérifications de connectivité réseau.

> *Le modèle représente les données et la logique métier de l'application. Il encapsule les structures de données, les interactions de base de données et toute autre logique d'application principale. Dans Flutter, le modèle peut inclure des classes ou des modèles de données qui définissent le schéma de données, ainsi que des services permettant de récupérer ou de manipuler des données.*

5. `View` **(Vue)**
   - La vue dans l'architecture MVVM est responsable de l'**affichage de l'interface utilisateur** et de **la capture de tous les événements utilisateur**. Elle comprend toutes les classes liées à la vue, comme la classe HomeScreen. La vue interagit avec le ViewModel pour gérer les événements utilisateur et mettre à jour l'interface utilisateur.
   - La vue est généralement implémentée sous forme d'arborescence de widgets dans Flutter. Elle utilise le contexte BuildContext pour interagir avec le ViewModel et mettre à jour l'interface utilisateur en fonction des données qu'elle reçoit. La vue est également chargée de capturer les événements utilisateur, comme les clics sur les boutons ou la saisie de texte, et de les transmettre au ViewModel.

6. `ViewModel`
   - Le `ViewModel` est le pont entre le modèle et la vue. Il comprend toutes les classes liées au ViewModel qui gèrent les demandes d'événements utilisateur et mettent à jour la vue. Le ViewModel utilise le contexte BuildContext pour interagir avec le modèle et la vue.
   - Le `ViewModel` est responsable de la gestion de tous les événements utilisateur et de la logique métier au sein de l'application. Il interagit avec le modèle pour traiter les demandes d'événements utilisateur et mettre à jour la vue. Le ViewModel avertit également la vue lorsque les données changent, ce qui permet à l'interface utilisateur de se mettre à jour automatiquement.

     > *Dans l'`architecture MVVM`, le ViewModel sert effectivement de pont entre le modèle (Model) et la vue (View). Il extrait, transforme, et prépare les données issues du modèle pour qu'elles soient directement utilisables par la vue. En Flutter, cela est rendu possible grâce aux solutions de gestion d'état telles que Provider, BLoC, Riverpod, GetX, etc., chacune offrant ses avantages pour structurer l'interaction entre les différentes couches de l'application.*

   - le `ViewModel` agit comme un intermédiaire entre la vue et le modèle , de telle manière qu'il fournit des données à l'interface utilisateur. Le ViewModel peut également exposer

apcpedagogie.com

des méthodes permettant de maintenir l'état de la vue, de mettre à jour le modèle en fonction des actions sur une vue et de déclencher des événements sur la vue. Dans Flutter, nous avons un écouteur appelé qui permet au ViewModel d'informer ou de mettre à jour la vue chaque fois que les données sont mises à jour.`ChangeNotifier`

1. **Rôle du ViewModel dans le MVVM**

    - **Préparer les données**: Le ViewModel reçoit les données du modèle sous leur forme brute (par exemple, des objets JSON d'une API) et les transforme au besoin. Par exemple, il peut convertir une date en un format plus lisible ou calculer des valeurs dérivées à partir des données reçues.
    - **Notifier la vue**: Dans une architecture réactive, comme avec Provider ou Riverpod, le ViewModel informe la vue de tout changement d'état (ajout, suppression, ou modification de données). Cela garantit que la vue reste synchronisée avec les données du modèle.
    - **Gérer les interactions utilisateur**: Le ViewModel réagit aux actions utilisateur (clics, saisies, etc.) et met à jour les données en conséquence. Par exemple, si l'utilisateur effectue une recherche, le ViewModel récupère les résultats et met à jour la vue.

2. **Qu'est-ce que Flutter ChangeNotifier ?**

    - `ChangeNotifier` est une classe qui fournit des notifications de changement à ses auditeurs.
    - `ChangeNotifier` est une classe importante dans Flutter utilisée pour la gestion d'état. Elle fait partie de la bibliothèque de base de Flutter et est souvent utilisée avec le `package Provider` pour une gestion d'état réactive et efficace.

7. Solutions de Gestion d'État Courantes pour MVVM dans Flutter

- Voici un aperçu de quelques solutions populaires en Flutter :
    - **Provider** : Solution native à Flutter et recommandée par l'équipe Flutter, Provider est léger, simple à implémenter, et bien intégré avec les widgets Flutter pour la gestion d'état dans MVVM.
    - **BLoC (Business Logic Component)** : Fondé sur le modèle de flux (streams) et de réactivité, BLoC utilise les Streams pour gérer l'état. Idéal pour des applications qui nécessitent une séparation stricte des couches et une gestion asynchrone.
    - **Riverpod** : Une amélioration de Provider, avec une syntaxe plus claire et des fonctionnalités avancées (comme la détection de code mort). Riverpod est idéal

apcpedagogie.com

pour une gestion d'état complexe, tout en restant simple et intuitif.

- **GetX** : Solution rapide et complète pour la gestion de l’état, la navigation et les dépendances. GetX est simple à utiliser, mais est moins prescriptif quant à la structuration en couches.

- En fonction de la complexité et des besoins de l’application, chaque solution peut être utilisée pour organiser et structurer le ViewModel, apportant ainsi modularité et réactivité à votre projet Flutter.

8. Mettre en Place MVVM dans Flutter

- **Organiser le code source selon MVVM**
  - Une structure de dossier typique pour MVVM dans un projet Flutter pourrait ressembler à ceci:

```
lib/
  ├── models/
  │   └── ..._model.dart
  ├── views/
  │   └── ..._view.dart
  ├── viewmodels/
  │   └── ..._viewmodel.dart
  ├── services/
  │   └── ..._service.dart
  └── main.dart
```

    - **models/**: Contient les classes définissant les données.
    - **views/**: Contient les widgets Flutter responsables de l'affichage à l'écran.
    - **viewmodels/**: Contient la logique liée à la manière dont les informations sont présentées à l'utilisateur.
    - **services/** : (Optionnel) Utilisé pour les logiques métier externes, comme les appels API.
    - Cette organisation claire facilite la navigation dans votre code et permet à chaque composant d'avoir un objectif précis.
- **Conventions de Nommage**
  - **Modèles**: Les fichiers qui contiennent des classes de modèle doivent être nommés avec le **suffixe _model**.
  - Par exemple, pour un modèle représentant un utilisateur, le fichier serait nommé : **user_model.dart**
  - **Vues**: Les fichiers qui contiennent des widgets responsables de l'affichage doivent être nommés avec le suffixe _view. Par exemple : user_view.dart

apcpedagogie.com

- **ViewModels**: Les fichiers qui contiennent la logique de présentation doivent être nommés avec le suffixe _view_model. Par exemple : **user_view_model.dart**
- **Services** : Les fichiers qui contiennent des services ou des logiques métier externes peuvent être nommés simplement selon leur fonction, sans suffixe spécifique, ou avec un suffixe comme **_service**. Par exemple :**user_service.dart**

- **Exemple (Gestion des données avec un fichier JSON en Flutter)**
  - Vous allez créer une application Flutter permettant de gérer une liste d'éléments (items). Ces éléments seront stockés localement dans un fichier JSON, lu et écrit par l'application.
  - **Objectifs**
    - Comprendre la séparation des responsabilités avec les dossiers models, services, viewmodels et views.
    - Implémenter la gestion des fichiers JSON pour stocker et lire les données.
    - Utiliser le package provider pour gérer l'état de l'application.
  - **Dans cette application, vous devrez :**
    - Afficher une liste d'éléments à l'écran.
    - Permettre à l'utilisateur d'ajouter un nouvel élément.
    - Permettre à l'utilisateur de supprimer un élément existant.
    - Stocker les données dans un fichier JSON local pour persistance entre les sessions.

```
lib/
  ├── models/
  │   └── item_model.dart
  ├── views/
  │   └── item_view.dart
  ├── viewmodels/
  │   └── item_viewmodel.dart
  ├── services/
  │   └── item_service.dart
  └── main.dart
```

1. **models/item_model.dart**

```
/// Représente un élément avec un identifiant unique et un nom.
class Item {
  /// Identifiant unique de l'élément.
  final int id;
```

apcpedagogie.com

```
  /// Nom de l'élément.
  final String name;

  /// Constructeur de la classe `Item` qui initialise les propriétés
[id] et [name].
  Item({required this.id, required this.name});

  /// Crée une copie de l'élément actuel en remplaçant éventuellement
les propriétés spécifiées.
  ///
  /// - [id] : Si fourni, remplace l'identifiant actuel.
  /// - [name] : Si fourni, remplace le nom actuel.
  ///
  /// Retourne un nouvel objet `Item` avec les propriétés mises à jour.
  Item copyWith({int? id, String? name}) {
    return Item(
      id: id ?? this.id, // Utilise l'id actuel si aucun nouveau n'est
fourni.
      name: name ?? this.name, // Utilise le nom actuel si aucun nouveau
n'est fourni.
    );
  }

  /// Convertit l'objet `Item` en une carte (Map) clé-valeur.
  ///
  /// Exemple de structure de la carte :
  /// ```dart
  /// {'id': 1, 'name': 'Exemple'}
  /// ```
  /// Retourne une carte contenant l'identifiant et le nom de l'élément.
  Map toMap() {
    return {
      'id': id, // Ajoute l'identifiant à la carte.
      'name': name, // Ajoute le nom à la carte.
    };
  }

  /// Crée un objet `Item` à partir d'une carte (Map).
  ///
  /// La carte doit contenir les clés suivantes :
  /// - `id` : Identifiant de l'élément.
  /// - `name` : Nom de l'élément.
```

apcpedagogie.com

```
  ///
  /// Exemple d'utilisation :
  /// ```dart
  /// final item = Item.fromMap({'id': 1, 'name': 'Exemple'});
  /// ```
  /// Retourne un nouvel objet `Item` initialisé avec les valeurs de la
carte.
  static Item fromMap(Map map) {
    return Item(
      id: map['id'], // Récupère l'identifiant à partir de la carte.
      name: map['name'], // Récupère le nom à partir de la carte.
    );
  }
}
```

2. **services/item_service.dart**

- Ce fichier gère la communication avec la base de données.

> - ***Rôle du Service***
>   - *Le service est responsable de :*
>   - *Gérer les données persistantes (lecture, écriture, suppression).*
>   - *Interagir avec des sources de données externes, comme une base de données ou un fichier JSON dans notre cas.*
>   - *Fournir des données "pures" au reste de l'application, mais sans gestion des états locaux.*
>   - *Dans ce cas, le service est conçu uniquement pour lire, écrire, ajouter et supprimer des données dans le fichier* ***items.json****. Il agit comme une couche d'accès aux données (DAL, Data Access Layer).*

Gestion des fichiers (Accès et Création)

- **Code concerné :**

```
Future<File> _getFile() async {
  final directory = await getApplicationDocumentsDirectory();
  return File('${directory.path}/$fileName');
}
```

- **Pourquoi c'est obligatoire ?**

apcpedagogie.com

- Cette méthode permet d'obtenir le fichier JSON (items.json) où seront stockés les données des items.
- Elle garantit que les données persistantes sont enregistrées dans un emplacement valide (dossier des documents de l'application).
- Sans cette méthode, aucune opération de lecture ou d'écriture ne peut être réalisée.

Lecture des données

- **Code concerné :**

```
Future<List<Item>> getItems() async {
  try {
    final file = await _getFile();
    if (!await file.exists()) {
      return [];
    }
    final contents = await file.readAsString();
    final List<dynamic> jsonData = json.decode(contents);
    return jsonData.map((data) =>
Item.fromMap(data)).toList();
  } catch (e) {
    print("Erreur lors de la lecture du fichier JSON: $e");
    return [];
  }
}
```

- **Pourquoi c'est obligatoire ?**
  - Cette méthode lit les données du fichier JSON et les convertit en objets Item.
  - Elle renvoie une liste d'éléments qui peut être utilisée par d'autres parties de l'application.
  - Elle gère également les erreurs, comme si le fichier n'existe pas encore.

Ajout d'un élément

- **Code concerné :**

```
Future<void> addItem(Item item) async {
  final items = await getItems();
  items.add(item);
  await _writeToFile(items);
}
```

apcpedagogie.com

- **Pourquoi c'est obligatoire ?**
  - Cette méthode ajoute un nouvel élément à la liste existante.
  - Elle utilise la méthode getItems pour récupérer les données actuelles, puis la méthode _writeToFile pour les enregistrer avec l'élément ajouté.
  - C'est essentiel pour garantir que les nouveaux items sont persistants dans le fichier JSON.

Suppression d'un élément

- **Code concerné :**

```
Future<void> deleteItem(int id) async {
  final items = await getItems();
  final updatedItems = items.where((item) => item.id !=
id).toList();
  await _writeToFile(updatedItems);
}
```

- **Pourquoi c'est obligatoire ?**
  - Cette méthode supprime un élément existant en filtrant la liste pour exclure l'élément avec l'ID donné.
  - Elle garantit que les modifications sont reflétées dans le fichier JSON grâce à _writeToFile.

Écriture des données dans le fichier

- **Code concerné :**

```
Future<void> _writeToFile(List items) async {
  final file = await _getFile();
  final jsonData = json.encode(items.map((item) =>
item.toMap()).toList());
  await file.writeAsString(jsonData);
}
```

- **Pourquoi c'est obligatoire ?**
  - Cette méthode prend une liste d'objets Item, les convertit en JSON et les enregistre dans le fichier.
  - C'est crucial pour la persistance des données. Sans elle, les modifications effectuées sur les données (ajout ou suppression) ne seraient pas sauvegardées.

apcpedagogie.com

```
import 'dart:async';
import 'dart:convert';
import 'dart:io';
import 'package:path_provider/path_provider.dart';
import '../models/item_model.dart';

class ItemService {
  String fileName = "items.json";

  // Obtenir le chemin du fichier JSON
  Future<File> _getFile() async {
    final directory = await getApplicationDocumentsDirectory();
    return File('${directory.path}/$fileName');
  }

  // Lire les données du fichier JSON
  Future<List<Item>> getItems() async {
    try {
      final file = await _getFile();

      // Si le fichier n'existe pas, retourner une liste vide
      if (!await file.exists()) {
        return [];
      }

      final contents = await file.readAsString();
      final List<dynamic> jsonData = json.decode(contents);

      return jsonData.map((data) => Item.fromMap(data)).toList();
    } catch (e) {
      print("Erreur lors de la lecture du fichier JSON: $e");
      return [];
    }
  }

  // Ajouter un élément au fichier JSON
  Future<void> addItem(Item item) async {
    final items = await getItems();
    items.add(item);

    await _writeToFile(items);
  }
```

apcpedagogie.com

```
  // Supprimer un élément du fichier JSON
  Future<void> deleteItem(int id) async {
    final items = await getItems();
    final updatedItems = items.where((item) => item.id != id).toList();

    await _writeToFile(updatedItems);
  }

  // Écrire la liste des items dans le fichier JSON
  Future<void> _writeToFile(List items) async {
    final file = await _getFile();
    final jsonData = json.encode(items.map((item) =>
item.toMap()).toList());

    await file.writeAsString(jsonData);
  }
}
```

3. **viewmodels/item_viewmodel.dart**

> - ***Rôle du ViewModel***
>   - *Le ViewModel (ou une classe similaire) est responsable de :*
>   - *Gérer l'état local de l'application ou d'une vue.*
>   - *Fournir des méthodes comme notifyListeners() pour informer l'interface utilisateur (UI) des changements d'état.*
>   - *Transformer les données brutes fournies par le service en un format directement utilisable par l'UI.*
>   - *La méthode updateItem inclut une gestion locale des données, car elle modifie la liste locale _items et utilise notifyListeners() pour notifier les widgets dépendants des changements. Cela appartient naturellement au ViewModel, pas au service.*

Gestion des données (état local):

- Vous devez disposer d'une liste ou d'un ensemble de données représentant l'état actuel des éléments dans l'application.
- **Code concerné :**

apcpedagogie.com

```
List _items = [];
List get items => _items;
```

- **Pourquoi c'est obligatoire ?**
  - Appelle le service pour récupérer les données.
  - `_items` stocke la liste des éléments localement dans le ViewModel.
  - `items` expose la liste à l'interface utilisateur (UI).

Service de données :

- Le ViewModel doit communiquer avec un service responsable de la persistance des données (lecture, écriture, suppression).
- **Code concerné :**

```
final ItemService _itemService = ItemService();
```

- **Pourquoi c'est obligatoire ?**
  - `ItemService` fournit les méthodes pour interagir avec les données persistantes (fichier JSON dans ce cas).

Méthode pour récupérer les données (`fetchItems`)

- Une méthode obligatoire dans tout ViewModel est de charger les données depuis le service au démarrage de l'application ou lors d'un rafraîchissement :
- **Code concerné :**

```
Future fetchItems() async {
  _items = await _itemService.getItems();
  notifyListeners();
}
```

- **Pourquoi c'est obligatoire ?**
  - Appelle le service pour récupérer les données.
  - Met à jour l'état local (`_items`).
  - Notifie l'interface utilisateur que les données ont changé (`notifyListeners()`).

Méthode pour ajouter un élément (`addItem`)

- Le ViewModel doit permettre d'ajouter de nouveaux éléments via l'UI :
- **Code concerné :**

apcpedagogie.com

```
Future addItem(String name) async {
  final newItem = Item(id:
DateTime.now().millisecondsSinceEpoch, name: name);
  await _itemService.addItem(newItem);
  await fetchItems();
}
```

- **Pourquoi c'est obligatoire ?**
  - Crée un nouvel élément (génère un ID unique basé sur le timestamp).
  - Utilise le service pour persister cet élément.
  - Recharge les données pour synchroniser l'UI.

Méthode pour supprimer un élément (`deleteItem`)

- Le ViewModel doit gérer la suppression des éléments :
- **Code concerné :**

```
Future deleteItem(int id) async {
  await _itemService.deleteItem(id);
  await fetchItems();
}
```

- **Pourquoi c'est obligatoire ?**
  - Supprime l'élément via le service.
  - Recharge les données pour refléter la suppression.

Méthode pour modifier un élément (`updateItem`)

- Le ViewModel doit gérer les mises à jour d'éléments existants :
- **Code concerné :**

```
void updateItem(int id, String newName) {
  final index = _items.indexWhere((item) => item.id == id);
  if (index != -1) {
    _items[index] = _items[index].copyWith(name: newName);
    notifyListeners();
  }
}
```

- **Pourquoi c'est obligatoire ?**
  - Trouve l'élément dans l'état local par son ID.
  - Crée une copie de l'élément avec le nouveau nom (en utilisant copyWith).

apcpedagogie.com

- Notifie l'UI pour appliquer les changements.

```
import 'package:flutter/material.dart';
import '../models/item_model.dart';
import '../services/item_service.dart';

class ItemViewModel extends ChangeNotifier {
  final ItemService _itemService = ItemService();
  List _items = [];

  List get items => _items;

  Future fetchItems() async {
    _items = await _itemService.getItems();
    notifyListeners();
  }

  Future addItem(String name) async {
    final newItem = Item(id: DateTime.now().millisecondsSinceEpoch,
name: name);
    await _itemService.addItem(newItem);
    await fetchItems();
  }

  Future deleteItem(int id) async {
    await _itemService.deleteItem(id);
    await fetchItems();
  }
  void updateItem(int id, String newName) {
  final index = _items.indexWhere((item) => item.id == id);
  if (index != -1) {
    _items[index] = _items[index].copyWith(name: newName); // Assurez-
vous que la méthode `copyWith` existe dans votre modèle.
    notifyListeners();
  }
}

}
```

4. **views/item_view.dart**

- La vue qui affiche les éléments et permet l'ajout et la suppression.

```
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import '../viewmodels/item_viewmodel.dart';

class ItemView extends StatelessWidget {
  final TextEditingController _controller = TextEditingController();

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Liste des éléments'),
        centerTitle: true,
      ),
      body: Column(
        children: [
          Expanded(
            child: Consumer(
              builder: (context, viewModel, child) {
                if (viewModel.items.isEmpty) {
                  return Center(
                    child: Text(
                      'Aucun élément trouvé.',
                      style: TextStyle(fontSize: 18, color:
Colors.grey),
                    ),
                  );
                }
                return ListView.builder(
                  itemCount: viewModel.items.length,
                  itemBuilder: (context, index) {
                    final item = viewModel.items[index];
                    return Card(
                      elevation: 4,
                      margin: const EdgeInsets.symmetric(
                          vertical: 6.0, horizontal: 10.0),
                      child: ListTile(
                        title: Text(
                          item.name,
                          style: TextStyle(
                            fontSize: 16,
                            fontWeight: FontWeight.bold,
```

apcpedagogie.com

```
                          ),
                        ),
                        trailing: Row(
                          mainAxisSize: MainAxisSize.min,
                          children: [
                            IconButton(
                              icon: Icon(Icons.edit, color:
Colors.blue),
                              onPressed: () => _editItem(context, item),
                              tooltip: 'Modifier cet élément',
                            ),
                            IconButton(
                              icon: Icon(Icons.delete, color:
Colors.red),
                              onPressed: () =>
viewModel.deleteItem(item.id),
                              tooltip: 'Supprimer cet élément',
                            ),
                          ],
                        ),
                      ),
                    );
                  },
                );
              },
            ),
          ),
          Padding(
            padding: const EdgeInsets.all(10.0),
            child: Row(
              children: [
                Expanded(
                  child: TextField(
                    controller: _controller,
                    decoration: InputDecoration(
                      border: OutlineInputBorder(),
                      labelText: 'Ajouter un élément',
                      hintText: 'Nom de l'élément',
                    ),
                  ),
                ),
                SizedBox(width: 8),
                ElevatedButton(
```

apcpedagogie.com

```
                  onPressed: () {
                    if (_controller.text.isNotEmpty) {
                      context.read().addItem(_controller.text);
                      _controller.clear();
                    }
                  },
                  child: Text('Ajouter'),
                  style: ElevatedButton.styleFrom(
                    padding: EdgeInsets.symmetric(horizontal: 20,
vertical: 15),
                    shape: RoundedRectangleBorder(
                      borderRadius: BorderRadius.circular(10),
                    ),
                  ),
                ),
              ],
            ),
          ),
        ],
      ),
    );
  }

  // Fonction pour modifier un élément
  void _editItem(BuildContext context, dynamic item) {
    final TextEditingController editController =
        TextEditingController(text: item.name);

    showDialog(
      context: context,
      builder: (context) {
        return AlertDialog(
          title: Text('Modifier l'élément'),
          content: TextField(
            controller: editController,
            decoration: InputDecoration(
              border: OutlineInputBorder(),
              labelText: 'Nom de l'élément',
              hintText: 'Entrer le nouveau nom',
            ),
          ),
          actions: [
            TextButton(
```

apcpedagogie.com

```
              onPressed: () => Navigator.pop(context),
              child: Text('Annuler'),
            ),
            ElevatedButton(
              onPressed: () {
                if (editController.text.isNotEmpty) {
                  // Appeler la méthode de mise à jour
                  context
                      .read()
                      .updateItem(item.id, editController.text);
                  Navigator.pop(context);
                }
              },
              child: Text('Sauvegarder'),
            ),
          ],
        );
      },
    );
  }
}
```

5. **main.dart**

   - Le point d'entrée principal de l'application.

```
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import 'views/item_view.dart';
import 'viewmodels/item_viewmodel.dart';

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return ChangeNotifierProvider(
      create: (_) => ItemViewModel()..fetchItems(),
      child: MaterialApp(
        debugShowCheckedModeBanner: false,
```

apcpedagogie.com

```
        title: 'Item Manager',
        theme: ThemeData(primarySwatch: Colors.blue),
        home: ItemView(),
      ),
    );
  }
}
```

9. Utiliser le modèle MVVM avec Sqflite dans une application Flutter,

- Pour utiliser le modèle **MVVM** avec `Sqflite` dans une application Flutter, vous devez structurer votre code en plusieurs classes standard qui représentent les différentes couches de l'architecture.
- Voici un aperçu des classes typiques que vous devriez déclarer :

1. **Modèle (Model)**

   - Cette classe représente la structure de vos données et peut inclure des méthodes pour convertir entre des objets et des formats de données (comme des maps pour la base de données).
   - **Elle sert uniquement à définir les attributs et les méthodes de transformation des données, par exemple, pour convertir en et depuis un Map.**

```
class Item {
  final int id;
  final String name;

  Item({required this.id, required this.name});

  // Convertir un objet Item en Map pour Sqflite
  Map toMap() {
    return {
      'id': id,
      'name': name,
    };
  }

  // Créer un objet Item à partir d'un Map
  factory Item.fromMap(Map map) {
    return Item(
```

apcpedagogie.com

```
      id: map['id'],
      name: map['name'],
    );
  }
}
```

2. **Service de Base de Données**

   - Cette classe gère toutes les opérations liées à la base de données, telles que l'insertion, la mise à jour et la récupération des données.

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

class DatabaseService {
  static final DatabaseService _instance = DatabaseService._internal();
  Database? _database;

  factory DatabaseService() {
    return _instance;
  }

  DatabaseService._internal();

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

  Future _initDatabase() async {
    String path = join(await getDatabasesPath(), 'items.db');
    return await openDatabase(path, version: 1, onCreate: (db, version)
{
      return db.execute(
        'CREATE TABLE items(id INTEGER PRIMARY KEY, name TEXT)',
      );
    });
  }

  Future insertItem(Item item) async {
    final db = await database;
```

apcpedagogie.com

```
    await db.insert('items', item.toMap());
  }

  Future> getItems() async {
    final db = await database;
    final List> maps = await db.query('items');
    return List.generate(maps.length, (i) {
      return Item.fromMap(maps[i]);
    });
  }
}
```

3. **ViewModel**

   - Cette classe interagit avec le modèle et le service de base de données. Elle contient la logique métier et notifie les vues des changements d'état.

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

class ItemViewModel with ChangeNotifier {
  final DatabaseService _databaseService = DatabaseService();
  List _items = [];

  List get items => _items;

  Future fetchItems() async {
    _items = await _databaseService.getItems();
    notifyListeners();
  }

  Future addItem(Item item) async {
    await _databaseService.insertItem(item);
    await fetchItems(); // Récupérer les éléments après ajout
  }
}
```

4. **Vue (View)**

   - Les widgets Flutter qui affichent les données et interagissent avec l'utilisateur. Ils écoutent les changements dans le ViewModel.

apcpedagogie.com

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

class ItemListView extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final itemViewModel = Provider.of(context);

    return Scaffold(
      appBar: AppBar(title: Text('Items')),
      body: FutureBuilder(
        future: itemViewModel.fetchItems(),
        builder: (context, snapshot) {
          if (snapshot.connectionState == ConnectionState.waiting) {
            return Center(child: CircularProgressIndicator());
          }
          return ListView.builder(
            itemCount: itemViewModel.items.length,
            itemBuilder: (context, index) {
              return ListTile(
                title: Text(itemViewModel.items[index].name),
              );
            },
          );
        },
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () {
          // Ajouter un nouvel élément
          itemViewModel.addItem(Item(id: 0, name: 'New Item'));
        },
        child: Icon(Icons.add),
      ),
    );
  }
}
```

5. **Main App**

   - Enfin, configurez votre application pour utiliser Provider et initialiser le ViewModel.

apcpedagogie.com

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

void main() {
  runApp(
    ChangeNotifierProvider(
      create: (context) => ItemViewModel(),
      child: MyApp(),
    ),
  );
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'MVVM avec Sqflite',
      home: ItemListView(),
    );
  }
}
```

6. **Conclusion**

7. Cette structure de classes vous permet de suivre le modèle MVVM tout en utilisant Sqflite pour la persistance des données. Assurez-vous d'adapter et d'étendre ces classes selon les besoins spécifiques de votre application.

Cours Flutter

Source : https://apcpedagogie.com/comprendre-larchitecture-mvvm-dans-flutter · téléchargé le 05/11/2026 · © APC Pédagogie · apcpedagogie.com