Travail pratique sur Flutter — Découverte de l'architecture et du fichier main.dart.

1. Qu'est-ce que Flutter ? Expliquez brièvement son rôle dans le développement d'applications mobiles.

   Flutter est un **framework open-source créé par Google** qui sert à développer des **applications multiplateformes** à partir **d'un seul code source**.

   Avec flutter, on peut créer :

- des applications **Android**
- des applications **iOS**
- des applications **Desktop** (Windows, Linux, macOS)
- des applications **Web**

Le **rôle de Flutter** dans le développement des applications mobiles est d'**accélérer, simplifier et unifier** la création d'applications **Android et iOS** à partir d'un **seul code source**.

Flutter sert à :

- créer les écrans (pages)
- boutons, formulaires, menus
- animations et transitions

Tout est réalisé avec des **widgets**.

Flutter gère :

- les actions des boutons
- la navigation entre écrans
- la validation des formulaires
- les calculs (moyennes, factures, etc.)

Flutter permet de :

- ✓ se connecter à une **base de données** (via API)
- ✓ communiquer avec **PHP + MySQL**
- ✓ envoyer / recevoir des données (JSON)

Flutter peut utiliser :

- caméra
- GPS
- notifications
- fichiers
- empreinte digitale (via plugins)

2. Lorsqu'on crée un nouveau projet Flutter, plusieurs dossiers sont générés automatiquement. Citez au moins trois dossiers créés par Flutter et donnez le rôle de chacun.

Les dossiers nécessaires l'or de la création du projet flateur :

- **Dossier android/ :**

Il Contient tout ce qui concerne **Android**

- configuration Android (Gradle)
- permissions
- version SDK
- icône de l'application Android

on ne peut rien faire à ce fichier

- **dossier ios/**

il Contient tout ce qui concerne **iOS**

- configuration Xcode
- permissions
- certificats Apple

on ne peut y toucher que si on compile pour iphone

- **le dossier lib/** (le plus important)

C'est ici qu'on peut saisir les code de l'application

- fichiers `.dart`
- écrans (pages)
- logique métier

Par défaut le fichier :

main.dart qui le **point d'entrée de l'application**

- **le dossier test/**

Sert aux **tests automatisés**

- tests unitaires
- tests de widgets

(Optionnel pour débutant)

- **le dossier web/**

Fichiers pour la version **Web**

- index.html
- configuration navigateur

- **le dossier windows/ | linux/ | macos/**

un dossier Dossiers pour les applications **desktop**

- le dossier **build/**

Dossier **généré automatiquement a n'est pas modifier manuellement**

- fichiers compilés
- APK, AAB, etc.

Certains fichiers nécessaires

pubspec.yaml

Fichier très important

- nom de l'application
- dépendances (packages)
- images, polices

pubspec.lock

Versions exactes des packages installés
(généré automatiquement)

analysis_options.yaml

Règles de qualité du code (lint)

README.md

Description du projet

3. Quel est le dossier dans lequel le développeur Flutter écrit principalement son code ? Justifiez votre réponse.

C'est dans le dossier : **lib/**

C'est ici qu'on écrit les codes de l'application.

- fichiers `.dart`
- écrans (pages)
- logique métier

4. À quoi sert le fichier pubspec.yaml dans un projet Flutter ?

Le fichier **`pubspec.yaml`** est le **cœur de configuration** d'un projet Flutter.
Sans lui, l'application **ne peut pas fonctionner correctement**.

Voici son rôle expliqué simplement 👇

Il sert à **décrire et configurer l'application Flutter**.

C'est lui qui dit à Flutter :

- comment s'appelle l'application
- quelles bibliothèques (packages) utiliser
- quelles images, polices et ressources charger
- quelle version de Flutter/Dart est requise

Il Définir les informations du projet

Ex :

```
name: gestion_etudiants
description: Application de gestion scolaire
version: 1.0.0+1
```

- `name` : nom interne du projet
- `version` : version de l'application (utile pour Play Store)

il Gère les dépendances (packages)

C'est son rôle **le plus important**.

Exemple :

```
dependencies:
  flutter:
    sdk: flutter
  http: ^1.2.0
  provider: ^6.0.5
```

Flutter télécharge automatiquement ces packages avec :

```
flutter pub get
```

Sans `pubspec.yaml`, **pas de packages**.

Il Déclare les images et autres ressources

Flutter **n'utilise pas automatiquement** les images.
Il faut les déclarer ici.

```
flutter:
  assets:
    - assets/images/
```

Ensuite dans le code :

```
Image.asset('assets/images/logo.png');
```

il Déclarer les polices (fonts)

```
flutter:
  fonts:
    - family: Roboto
      fonts:
        - asset: assets/fonts/Roboto-Regular.ttf
```

il Fixer la version de Dart / Flutter

```
environment:
  sdk: ">=3.0.0 <4.0.0"
```

il faut éviter les erreurs de compatibilité.

- **Indentation obligatoire** (espaces, pas de tabulation)
- chaque modification nécessite souvent :

```
flutter pub get
```

- une erreur ici peut bloquer tout le projet

5. Quel est le rôle du fichier main.dart dans une application Flutter ?

Le fichier **`main.dart`** joue un rôle **fondamental** dans une application Flutter :
c'est **le point de départ de toute l'application**.

Sans `main.dart`, **l'application ne démarre pas**.

`main.dart` sert à :

- **démarrer l'application**
- **charger le premier écran**
- **configurer l'application globale** (thème, routes, etc.)

Le point d'entrée de l'application

Comme en Dart classique, Flutter commence toujours par la fonction `main()` :

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

`main()` est la **première fonction exécutée**
`runApp()` lance l'application Flutter

il Charge le widget principal

`MyApp` est le **widget racine** de l'application :

```
class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Mon application',
      home: HomePage(),
    );
  }
}
```

Ce widget :

- définit le **style général**
- indique **la première page à afficher**

Il Définit le premier écran

Dans `main.dart`, on choisit :

- soit `home: HomePage()`
- soit des **routes** (navigation)

Exemple :

```
MaterialApp(
  home: LoginPage(),
);
```

Ici, l'application démarre sur la page de connexion.

Configuration globale

`main.dart` permet aussi de définir :

- le **thème** (couleurs, polices)
- la langue
- le mode clair/sombre
- la navigation entre pages

```
MaterialApp(
  theme: ThemeData(primarySwatch: Colors.blue),
);
```

6. Expliquez pourquoi la fonction main() est obligatoire dans un programme Flutter.

La fonction **`main()` est obligatoire** dans un programme Flutter parce qu'elle est le **point d'entrée officiel** du langage **Dart**, sur lequel Flutter est construit.

Voici l'explication claire et logique 👇

1. Point de départ du programme

En Dart (comme en C, Java, Python), **le programme doit savoir par où commencer**.

Dart commence **toujours** l'exécution par :

```
void main() { }
```

Sans cette fonction :

- Dart ne sait pas quoi exécuter
- Flutter ne peut pas démarrer
- l'application échoue immédiatement

2. Lancer l'application Flutter

Dans Flutter, `main()` appelle :

```
runApp(MyApp());
```

`runApp()` :

- initialise le moteur Flutter
- charge l'interface graphique
- affiche le premier écran

Sans `main()` :

- `runApp()` n'est jamais appelé
- l'interface ne s'affiche pas

3. Initialisation avant l'interface

`main()` permet aussi :

- d'initialiser des services (base de données, Firebase, API)
- de configurer l'environnement
- de préparer les données avant l'affichage

Exemple :

```
void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  await Firebase.initializeApp();
```

```
  runApp(MyApp());
}
```

Tout ça doit se faire **avant** l'interface.

4. Respect des règles du langage Dart

Flutter **n'invente pas** son propre point d'entrée.
Il respecte la règle du langage Dart :

Un programme Dart valide **doit** avoir une fonction `main()`.

Sans main il y a :

- erreur de compilation
- application impossible à lancer
- Flutter affiche un message du type :

  No main method found

7. Que fait l'instruction runApp() ? Pourquoi est-elle importante ?

Que fait l'instruction `runApp()` en Flutter ?

`runApp()` est l'instruction qui **démarre réellement l'application Flutter** et **affiche l'interface graphique à l'écran**.

Flutter fait plusieurs choses **automatiquement** :

1. **Initialise le moteur Flutter**
   - prépare le rendu graphique
   - connecte Flutter au système (Android / iOS)
2. **Charge le widget racine**
   - ici `MyApp`
   - c'est le **premier widget** de toute l'application
3. **Construit l'interface utilisateur**
   - lance la méthode `build()`
   - affiche les widgets à l'écran
4. **Démarre le cycle de vie de l'application**
   - gestion des événements (clics, navigation, etc.)

`runApp()` est si importante par ce que

- Sans `runApp()`, l'app ne s'affiche pas

Même si le code est correct :

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

Rien ne s'affiche, car Flutter n'a pas reçu l'ordre de lancer l'UI.

- Elle définit le point de départ visuel

`runApp()` indique à Flutter :

« Voici le widget racine de mon application »

Tout le reste de l'interface **dépend de ce widget**.

- Elle relie Dart à l'interface graphique

- Dart = logique
- Flutter = interface

`runApp()` fait le **pont entre le code Dart et l'écran du téléphone**.

- Une seule fois dans l'application

- `runApp()` est appelée **une seule fois**
- tous les changements d'écran se font ensuite via les widgets

Exemple simple

```
void main() {
  runApp(
    const MaterialApp(
      home: Scaffold(
        body: Center(
          child: Text('Bonjour Flutter'),
        ),
      ),
    ),
  );
}
```

Dans cet exemple

- `runApp()` lance Flutter
- `MaterialApp` configure l'application
- `Text` s'affiche à l'écran

8. Qu'est-ce qu'un widget en Flutter ? Donnez une définition simple avec vos propres mots.

Un **widget en Flutter**, c'est **un élément de base qui sert à réaliser l'interface et les évènements d'une application**.

Tout dans Flutter est un widget.

Un widget est un objet qui décrit à quoi ressemble une partie de l'application et comment elle se comporte.

Ça peut être :

- un texte
- un bouton
- une image
- un écran entier
- un menu
- même l'application complète

Exemples de widgets courants

- `Text` → afficher du texte
- `Button` / `ElevatedButton` → bouton
- `Image` → image
- `Row` / `Column` → organiser les éléments
- `Scaffold` → structure d'un écran
- `MaterialApp` → application complète

Comment Flutter utilise les widgets

Flutter **assemble des widgets dans d'autres widgets**, sous forme arborescence

```
MaterialApp
 └── Scaffold
     └── Center
         └── Text
```

Chaque widget :

- reçoit des informations
- décrit ce qu'il doit afficher
- peut contenir d'autres widgets

les widgets sont importants

- ils rendent l'interface **modulaire**
- ils facilitent la réutilisation du code
- ils rendent l'application claire et organisée
- ils permettent des interfaces rapides et fluides

Flutter est une application construite uniquement avec des widgets.

Sans widgets :

- pas d'écran
- pas de bouton
- pas d'application

9. Dans le code par défaut de Flutter, quelle est la classe principale de l'application ? Quel est son rôle ?

Dans le **code par défaut d'un projet Flutter**, la **classe principale de l'application** s'appelle généralement **`MyApp`.**

Voici la classe principale ?

```
class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      home: MyHomePage(),
    );
  }
}
```

`MyApp` est la **classe principale** (le widget racine).

Quel est son rôle ?

La classe **`MyApp`** sert à :

- Être le widget racine de l'application

- elle est passée à `runApp(MyApp())`
- tous les autres widgets sont **enfants de `MyApp`**

- Configurer l'application globalement

Dans `MyApp`, on définit :

- le **thème** (couleurs, polices)
- le **titre** de l'application
- la **navigation** (routes)
- la **page de démarrage**

```
MaterialApp(
  theme: ThemeData(primarySwatch: Colors.blue),
  home: MyHomePage(),
);
```

- Définir le premier écran

`MyApp` indique :

- quelle page s'affiche au lancement (`home`)
- ou quelles routes utiliser

10. Expliquez brièvement la différence entre :

- StatelessWidget et StatefulWidget

Voici la différence entre **`StatelessWidget`** et **`StatefulWidget`**

🔷 StatelessWidget

Un **widget sans état**.

Définition : Un `StatelessWidget` est un widget **dont le contenu ne change pas** après son affichage.

Il est Caractérisé par

- pas de données modifiables
- interface **fixe**
- ne se met pas à jour tout seul

Exemple

```
class Titre extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return const Text('Bonjour');
  }
}
```

Ici Le texte reste toujours le même.

Utilisation

- textes
- icônes
- pages statiques
- logos

🔷 StatefulWidget

Un **widget avec état**.

Définition : Un `StatefulWidget` est un widget **dont le contenu peut changer** pendant l'exécution de l'application.

Caractérisé par :

- possède un **State**
- se met à jour avec `setState()`

- interface **dynamique**

Exemple

```
class Compteur extends StatefulWidget {
  @override
  State<Compteur> createState() => _CompteurState();
}

class _CompteurState extends State<Compteur> {
  int count = 0;

  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      onPressed: () {
        setState(() {
          count++;
        });
      },
      child: Text('Compteur : $count'),
    );
  }
}
Ici Le texte change à chaque clic.

À retenir facilement
```

- **StatelessWidget** → *ne change jamais*
- **StatefulWidget** → *peut changer pendant l'exécution*

  Notion des constante et variable.

11. Quelle classe représente la première page affichée dans l'application Flutter par défaut ?

Dans un **projet Flutter créé par défaut**, la **première page affichée** est représentée par la classe **`MyHomePage`**.

Explication

Dans le code par défaut, on a généralement :

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

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
```

```
        primarySwatch: Colors.blue,
      ),
      home: const MyHomePage(title: 'Flutter Demo Home Page'), // ←
première page
    );
  }
}
```

- `MaterialApp` configure l'application.
- L'attribut **`home`** indique **le widget à afficher au démarrage**.
- Ici : `home: const MyHomePage(...)` → **MyHomePage** est donc **la première page affichée**.

- Le Rôle de `MyHomePage`

- Afficher l'**écran principal** de l'application.
- Contient généralement un **Scaffold** avec :
  - AppBar (barre en haut)
  - Body (contenu principal)
  - FloatingActionButton (bouton flottant, optionnel)
- Peut être modifiée pour créer **n'importe quelle page de démarrage**.

12. Pourquoi Flutter utilise-t-il une classe séparée pour gérer l'état (State) de la page ?

Flutter utilise une **classe séparée pour gérer l'état (`State`)** d'une page pour **séparer clairement la logique dynamique de l'interface statique**, ce qui rend le code plus organisé, réutilisable et performant. Voici l'explication détaillée

Séparation entre UI et état

- **Widget (`StatefulWidget`)** : décrit la **structure et l'apparence** de l'interface (UI)
- **State (`_MyWidgetState`)** : contient **les données qui peuvent changer** pendant l'exécution

Exemple :

```
class Compteur extends StatefulWidget {
  @override
  State<Compteur> createState() => _CompteurState();
}

class _CompteurState extends State<Compteur> {
  int count = 0; // état dynamique

  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      onPressed: () {
        setState(() {
          count++;
        });
      },
      child: Text('Compteur : $count'),
```

```
    );
  }
}
```

- `Compteur` → **déclare le widget**
- `_CompteurState` → **gère la valeur `count` et les mises à jour**

Avantages de cette séparation

a) Organisation du code

- Le widget reste **propre et déclaratif**
- L'état dynamique est **dans une autre classe**

b) Réutilisabilité

- Tu peux réutiliser le même `StatefulWidget` avec différents états
- L'UI et la logique sont découplées

c) Optimisation des performances

- Quand l'état change, Flutter ne reconstruit **que le widget avec son State**, pas toute l'application

d) Cycle de vie clair

- La classe `State` a ses méthodes :
  - `initState()` → initialisation
  - `dispose()` → nettoyage
  - `setState()` → mise à jour de l'UI

NB :

Flutter sépare l'état de la page pour que l'UI reste statique et claire, tandis que la logique dynamique est dans State, permettant des mises à jour efficaces et bien structurées.

13. À quoi sert la méthode build() dans une application Flutter ?

La méthode **`build()`** en Flutter est **la fonction qui crée et décrit l'interface utilisateur** d'un widget.

C'est **le cœur du fonctionnement de l'UI** dans Flutter.

Rôle principal de `build()`

- Elle **retourne un arbre de widgets** qui définit ce que l'utilisateur voit à l'écran.
- Flutter utilise cet arbre pour **afficher l'interface graphique**.
- Chaque fois que l'état change (`setState()`), **`build()` est rappelée** pour reconstruire l'UI avec les nouvelles données.

Où place `build()` ?

- Dans un **StatelessWidget** :

```
class Titre extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return const Text('Bonjour Flutter');
  }
}
```

- Dans un **StatefulWidget** (dans la classe State) :

```
class _CompteurState extends State<Compteur> {
  int count = 0;

  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      onPressed: () {
        setState(() {
          count++;
        });
      },
      child: Text('Compteur : $count'),
    );
  }
}
```

`Points clés à retenir`

1. **Retourne toujours un widget** (ou un arbre de widgets)
2. **Reconstruction automatique** quand l'état change (`setState`)
3. **Ne pas stocker l'état ici** — juste afficher ce qui doit être vu

14. Expliquez, étape par étape, ce qui se passe depuis le lancement de l'application jusqu'à l'affichage de l'interface à l'écran.

Voici une explication **étape par étape** de ce qui se passe dans Flutter **depuis le lancement de l'application jusqu'à l'affichage de l'interface à l'écran** ;

1) Lancement de l'application

- Le système d'exploitation (Android ou iOS) **exécute le programme Flutter**.
- Dart commence par chercher la **fonction `main()`** : c'est le **point d'entrée obligatoire**.

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

2) Appel de runApp()

- `runApp()` est appelé avec le **widget racine** (`MyApp`).
- Flutter initialise :
  - le **moteur graphique**
  - la **liaison entre Dart et l'écran**
  - le **cycle de vie des widgets**
- `runApp()` indique à Flutter :

le widget racine à afficher.

3) Création du widget racine

- Flutter construit le **widget MyApp**.
- `MyApp` retourne généralement un **MaterialApp** ou **CupertinoApp**.
- Cette étape définit :
  - le **thème global** (couleurs, polices)
  - le **titre de l'application**
  - la **première page à afficher** (`home`)

```
MaterialApp(
  home: const MyHomePage(),
)
```

4) Construction du premier écran

- Flutter appelle la **méthode `build()`** du widget racine.
- `build()` crée **l'arbre de widgets** (UI) :
  - Scaffold (structure de l'écran)
  - AppBar (barre en haut)
  - Body (contenu principal)
  - Boutons, textes, images
- Chaque widget dans cet arbre peut contenir **d'autres widgets imbriqués**.

5) Initialisation des widgets avec état

- Si le widget est un **StatefulWidget**, Flutter :
  1. Crée une instance de sa **classe State**.
  2. Appelle **`initState()`** pour initialiser les variables ou services.
  3. Appelle **`build()`** pour construire l'UI en utilisant l'état initial.

6) Affichage à l'écran

- Flutter passe l'arbre de widgets au **moteur de rendu**.
- Chaque widget est transformé en **éléments graphiques réels** :
  - texte, image, bouton cliquable
- L'interface apparaît à l'écran **prête à interagir avec l'utilisateur**.

7) Gestion des interactions

- Lorsque l'utilisateur interagit (clic, scroll, saisie) :
  - Flutter détecte l'événement
  - Si nécessaire, on appelle **`setState()`** pour mettre à jour l'état
  - Flutter **reconstruit uniquement les widgets affectés** via `build()`

Structure explicative :

```
 [OS lance l'app]
        ↓
[Dart exécute main()]
        ↓
[runApp(MyApp())]
        ↓
[Création widget racine MyApp]
        ↓
[build() de MyApp → MaterialApp → home: MyHomePage]
        ↓
[Création arbre de widgets du premier écran]
        ↓
[Moteur Flutter rend les widgets à l'écran]
        ↓
[Interface visible et interactive]
```

`Points essentiels à retenir`

1. **main()** → point de départ
2. **runApp()** → lance Flutter et widget racine
3. **build()** → construit l'UI
4. **State** → gère les données dynamiques
5. **setState()** → met à jour l'UI quand l'état change

15. Pourquoi dit-on que Flutter est basé sur une architecture de widgets ?

On dit que **Flutter est basé sur une architecture de widgets** parce que **tout dans Flutter – l'interface, la mise en page, même l'application entière – est construit avec des widgets**. C'est le **principe fondamental de Flutter**. Voici l'explication détaillée :

Tout est un widget

- En Flutter, **chaque élément à l'écran est un widget**, que ce soit :
  - un **texte** (Text)
  - un **bouton** (ElevatedButton)
  - une **image** (Image)
  - une **barre d'outils** (AppBar)

- un **écran complet** (Scaffold)
- l'**application entière** (MaterialApp)

Même les conteneurs, colonnes ou lignes sont des widgets !

Widgets imbriqués (arborescence)

- Les widgets sont **imbriqués les uns dans les autres**, formant un **arbre de widgets**.
- Cet arbre décrit **la structure, l'apparence et le comportement** de l'interface.

Exemple simple :

```
MaterialApp(
  home: Scaffold(
    appBar: AppBar(title: Text('Mon App')),
    body: Center(
      child: ElevatedButton(
        onPressed: () {},
        child: Text('Clique moi'),
      ),
    ),
  ),
);
```

- `MaterialApp` → widget racine
- `Scaffold` → structure de l'écran
- `AppBar` et `Center` → widgets enfants
- `ElevatedButton` et `Text` → widgets petits, contenus dans d'autres widgets

Avantages de cette architecture

1. **Modularité** : chaque widget est une **brique réutilisable**
2. **Hiérarchie claire** : l'UI est structurée en arbre, facile à comprendre et maintenir
3. **UI dynamique** : les widgets peuvent être **stateless** ou **stateful**, ce qui permet de gérer des interfaces fixes ou changeantes
4. **Performance** : Flutter reconstruit uniquement les widgets nécessaires lors des mises à jour

A retenir

Flutter est basé sur une architecture de widgets parce que **tout, du plus petit élément à l'application entière, est un widget**, formant un **arbre hiérarchique** qui décrit l'interface et son comportement.

16. Si vous deviez personnaliser la première application Flutter, quelles informations simples pourriez-vous ajouter à l'écran (sans entrer dans les détails du code) ?

Pour personnaliser la **première application Flutter** sans entrer dans le code, je peux simplement **changer ou ajouter des informations visibles à l'écran**.

Un message comme :

- « Bonjour Flutter ! »
- « Bienvenue dans mon application »
- « Gestion des étudiants »

Image ou logo

- Ajouter un **logo ou une image** au centre de l'écran
- Par exemple : le logo duprojet ou une illustration

Boutons simples

- Un **bouton cliquable** pour :

  dire « Appuyez ici »
  naviguer vers une autre page (exemple : « Voir la liste des étudiants »)

Informations personnelles sur mon projet :

- **nom** ou le **titre du projet**
- Exemple :

- Projet Flutter de Jean Jacques
- Version 1.0

Couleurs et thèmes simples

- Changer la **couleur de fond** de l'écran
- Modifier la **couleur du texte**
- Exemple : fond bleu ciel avec texte blanc

17. Selon vous, pourquoi est-il important de bien comprendre l'architecture Flutter avant de créer des applications plus complexes ?

Comprendre l'**architecture Flutter** avant de créer des applications complexes est **crucial** pour plusieurs raisons :

1) Pour structurer correctement son application

- Flutter repose sur **l'arbre de widgets** et la **séparation entre UI et état**.
- Si tu comprends bien cette architecture, tu sauras comment organiser tes fichiers et dossiers, créer des widgets réutilisables, et structurer les pages de manière logique.
- Exemple : savoir où mettre les **StatefulWidget**, les **StatelessWidget**, et les **services** (API, base de données).

2) Pour gérer efficacement l'état

- Dans Flutter, certaines pages ont des données qui changent (formulaires, compteur, liste d'étudiants).

- Si tu maîtrises l'architecture, tu sais **quand utiliser StatefulWidget**, **comment utiliser setState()**, et **où placer la logique métier**.
- Sinon, tu risques d'avoir un **code spaghetti** difficile à maintenir.

3) Pour faciliter la réutilisation et la maintenance

- Comprendre la hiérarchie des widgets et la séparation des responsabilités permet de :

- réutiliser des widgets dans plusieurs pages
- éviter de dupliquer du code
- mettre à jour facilement une partie de l'application sans tout casser

4) Pour optimiser les performances

- Flutter reconstruit l'UI **en fonction de l'état**.
- Si tu ne comprends pas l'architecture, tu risques :
    - de reconstruire **tout l'arbre de widgets** inutilement
    - de provoquer des **ralentissements** sur l'application

5) Pour gérer les applications complexes

- Une application avec plusieurs écrans, formulaires, bases de données, notifications ou navigation nécessite une **architecture claire**.
- Sans compréhension, tu risques :
    - de te perdre dans les widgets
    - de mélanger UI et logique
    - d'avoir des bugs difficiles à corrige

En conclusion ;

Comprendre l'architecture Flutter me permet de créer des applications **claires, performantes, modulaires et faciles à maintenir**.

C'est comme construire une maison : si tu ne connais pas le plan (fondations, murs, étages), tu risques de tout faire mal et de devoir tout recommencer.