

Introduction à Flutter : Développement d'application multiplateformes

Introduction à Flutter

Qu'est-ce que Flutter ?

Flutter est un framework Open Source développé par Google pour la création d'applications mobiles, web et desktop. Sa version Alpha (v0.0.6) a vu le jour en le 12 mai 2017. Il permet de développer des applications multiplateformes à partir d'une seule base de code. Flutter utilise le langage de programmation Dart aussi créé par Google, et fournit une multitude de widget et d'outils pour créer des interfaces performantes.

Avantages de Flutter

Flutter offre de nombreux avantages pour les développeurs d'applications, en voici quelques-uns :

- **Multiplateforme** : Permet de créer des applications pour iOS, Android, web et même desktop à partir d'une seule base de code. En d'autres mots, avec Flutter, tu peux écrire ton code source (Dart) qu'une seule fois et celui-ci peut être déployé sur plusieurs plateformes. L'idée ici est que la logique et l'interface utilisateur sont les mêmes pour toutes les plateformes. Avec plusieurs bases de code, le code source est différent pour chaque plateforme. Par exemple, pour Android, tu écrirais ton code en Java ou Kotlin et, pour iOS, tu écrirais ton code en Swift ou Objective-C. En résumé, l'avantage avec Flutter c'est la réduction considérable du temps de développement et de maintenance plus une cohérence de l'application sur toutes les plateformes.
- **Performance native** : Flutter compile directement en code machine, ce qui permet une performance proche de celle des applications natives.
- **Hot Reload** : Cette fonctionnalité permet aux développeurs de voir instantanément les modifications qu'ils apportent à l'interface utilisateur sans avoir à redémarrer l'application.
- **Facilité d'apprentissage** : Le langage Dart est relativement facile à apprendre, et la documentation Flutter est très complète, ce qui en fait un choix attrayant pour les nouveaux développeurs

Désavantages de Flutter


Bien que Flutter présente de nombreux avantages, il comporte également certains désavantages. Voici quelques-uns des principaux inconvénients de Flutter :

- **Taille de l'application** : Les applications Flutter peuvent être plus volumineuses par rapport aux applications natives. Le framework Flutter inclut le moteur de rendu, les widgets, et d'autres dépendances dans chaque application, ce qui peut entraîner une taille de fichier plus importante.
- **Le langage Dart** : Flutter utilise le langage Dart, développé par Google, bien qu'il soit relativement facile à apprendre, il n'est pas autant populaire que d'autres langages comme

JavaScript ou Java. Pour les développeurs venant d'autres framework, cela peut être un obstacle.

Installation de Flutter

Bien qu'il soit possible de développer des applications sur Flutter sur l'environnement de développement Android Studio, notre introduction se fera avec l'éditeur de texte VS Code. VS Code peut être utilisé par la grosse majorité des ordinateurs contrairement à Android Studio qui demande des performances élevées. De plus, l'interface est simple et minimaliste pour une personne débutante avec ce framework. Voici l'installation nécessaire pour pouvoir commencer ton premier projet. Si ce n'est pas déjà fait installer [Visual Studio Code](#).

1. Se rendre sur le site officiel de Flutter (<https://docs.flutter.dev/get-started/install>) et télécharger la dernière version stable pour ton système d'exploitation (Windows, macOS, Linux, ChromeOS)
2. Choisir le type d'application (Android, Web, Desktop). Dans notre cas, nous allons choisir Android.
3. Installer les outils de développement. Il va falloir installer [Git pour Windows](#) 2.27 ou une version ultérieure et [Android Studio](#) 2023.3.1 (Jellyfish) ou une version ultérieure. Suivre les instructions sur leur site respectif en choisissant les options de téléchargement de base.
4. Ouvrez Android Studio. Vous allez devoir installer les outils ci-dessus. Pour ce faire, aller dans Tools → SDK Manager et installer les éléments suivants :
 - ✓ Android SDK Platform, VanillaIceCream API (35) – ABI (x86_64) – Target (Android 15.0 Google APIs)
 - ✓ Android SDK Command-line Tools
 - ✓ Android SDK Build-Tools
 - ✓ Android SDK Platform-Tools
 - ✓ Android Emulator
5. Configurer un émulateur virtuel. Lorsque vous serez rendu, dans la fenêtre de vérification, cliquez sur montrer des paramètres avancés, puis dans performance émulée et changer les graphiques à « Hardware ». Au besoin, se référer à ce tutoriel : <https://developer.android.com/studio/run/managing-avds?hl=fr>
6. Retourner sur la page d'installation de flutter et télécharger le dossier zip qui inclut le SDK Flutter et extraire le dossier ici C:\.
7. Ajouter la variable d'environnement à votre PATH. Voici à quoi le chemin devrait ressembler : C:\flutter\bin.
8. Ouvrir VS Code et installer l'extension Flutter.

9. Ouvrir une invite de commande et exécuter la ligne suivante :
`flutter doctor --android-licenses` et accepter les licences Android Studio
10. Finalement, pour s'assurer que Flutter est bien installé, vous pouvez exécuter la ligne suivante : `flutter doctor`.

Pour plus d'information sur l'installation Flutter : <https://docs.flutter.dev/get-started/install/windows/mobile>

Introduction au langage Dart

Avant de plonger dans la création d'un projet Flutter, il est essentiel de comprendre Dart. Comprendre Dart est crucial pour tirer le meilleur parti de Flutter, car c'est ce langage qui nous permettra de concevoir des applications fluides et performantes. Passons maintenant à ses bases avant de démarrer un projet. La syntaxe de Dart est similaire à celle de Java, mais plus légère et optimisée pour le développement d'UI.

Structure de base d'un programme Dart

Un programme Dart commence par une fonction `main()` qui est le point d'entrée de l'application.

```
void main() {  
  print('Bonjour le monde');  
}
```

Variables, types et opérateurs

Les variables peuvent être déclarées avec des mots-clés comme `var`, `final`, `const` ou `dynamic`. Ces mots-clés permettent de répondre à différents besoins, que ce soit pour des valeurs fixes à la compilation (`const`), des valeurs immuables calculées dynamiquement (`final`), ou des variables dont le type peut changer à l'exécution (`dynamic`). Les opérateurs permettent de manipuler les variables et les valeurs. Ils incluent des opérateurs

```
int age = 25;  
var name = "Alice"; // Type inféré  
final currentTime = DateTime.now(); // Dépend de l'exécution  
const pi = 3.14; // Connu à la compilation  
dynamic anything = 42; // Peut changer de type  
anything = "Hello!";
```

```
// Arithmétiques  
int sum = 5 + 3; // Addition  
int product = 4 * 2; // Multiplication  
  
// Relationnels  
bool isGreater = 5 > 3; // Renvoie true  
  
// Logiques  
bool result = true && false; // Renvoie false  
  
// Null-aware  
String? name;  
print(name ?? "Default Name"); // Utilise "Default Name" si name est null
```

arithmétiques, relationnels, logiques, et des opérateurs modernes comme les opérateurs `null-aware` pour gérer les valeurs nulles efficacement.

Niveaux d'accès

En Dart, il n'y a pas de mot-clé spécifique comme `public`, `private` ou `protected` (comme en Java). Seul le préfixe « `_` » distingue les membres privés des autres.

```
int publicValue = 42; // Accessible partout  
int _privateValue = 10; // Accessible seulement dans ce fichier
```

Fonctions & Méthodes

Les fonctions en Dart fonctionnent de la même manière que Java.

```
int multiply(int a, int b) => a * b;
void printMessage(String message) => print(message); // Fonction courte
```

Classe & Objets

Voici un exemple d'une classe Personne en Dart

```
class Personne {
  String _nom;
  int _age;

  // Constructeur
  Personne(this._nom, this._age);

  // Getter
  String get nom => _nom;

  // Setter
  set nom(String valeur) {
    if (valeur.isNotEmpty) _nom = valeur;
  }

  // Redéfinition de toString()
  @override
  String toString() {
    return 'Personne: $_nom, Âge: $_age';
  }
}

void main() {
  var personne = Personne('Alice', 30);
  print(personne); // Appel de toString() automatiquement
  personne.nom = 'Bob'; // Utilisation du setter
  print(personne); // Affiche 'Personne: Bob, Âge: 30'
}
```

Premiers Pas avec Flutter

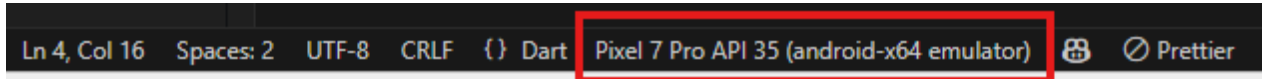
Créer un nouveau projet Flutter

1. Allez dans Affichage > Palette de commandes...
2. Vous pouvez également appuyer sur **Ctrl + Shift + P**.
3. Tapez **flutter**.
4. Sélectionnez **Flutter: New Project**.
5. Appuyez sur **Entrée**.
6. Sélectionnez **Application**.
7. Appuyez sur **Entrée**.
8. Choisissez un emplacement pour le projet.
9. Entrez le nom souhaité pour votre projet.

Exécution de l'application sur un émulateur/simulateur

Préalable : Avoir créer un émulateur mobile avec Android Studio

1. Dans la barre de statut, en bas à droite de votre écran, choisir l'émulateur que vous souhaitez

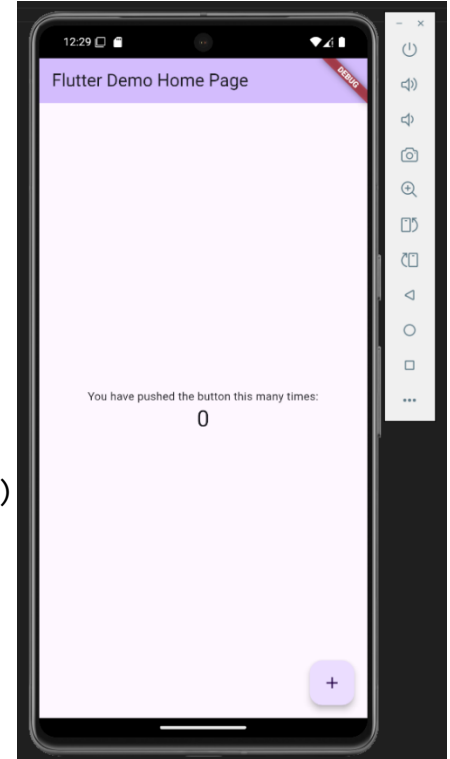


2. Faites un clic droit sur le fichier main.dart et sélectionner l'option « Run Without Debugging »
3. Si tout fonctionne l'affichage normal devrait ressembler à ceci :

Structure d'un projet Flutter

Lorsque vous créez un nouveau projet Flutter, la structure suivante est générée automatiquement :

```
exemple_flutter_projet/
├── android/      # Code et configuration spécifiques à Android
├── ios/          # Code et configuration spécifiques à iOS
├── lib/          # Contient votre code Dart principal
│   └── main.dart # Point d'entrée de l'application Flutter
├── test/        # Fichiers pour les tests unitaires
├── web/         # Code et configuration pour le web (si activé)
├── pubspec.yaml # Dépendances et configuration du projet
├── README.md    # Documentation du projet
└── .gitignore   # Fichiers à exclure du contrôle de version
```



Description des principaux fichiers et dossiers :

- **lib/main.dart** : Le fichier principal où vous définissez votre widget racine.
- **pubspec.yaml** : Utilisé pour définir les dépendances, les assets (images, polices) et les configurations globales du projet.
- **android/ et ios/** : Contiennent les fichiers spécifiques aux plateformes respectives pour personnaliser votre application (comme le nom ou les permissions).
- **test/** : Contient les tests unitaires pour vérifier le fonctionnement de votre code

Material Design

Material Design est un langage de conception visuelle développé par Google en 2014, conçu pour créer des interfaces utilisateurs cohérentes, intuitives et esthétiques à travers différentes plateformes et appareils. Il repose sur des principes clairs qui combinent une hiérarchie visuelle, des animations fluides, et une utilisation judicieuse des couleurs et des espacements.

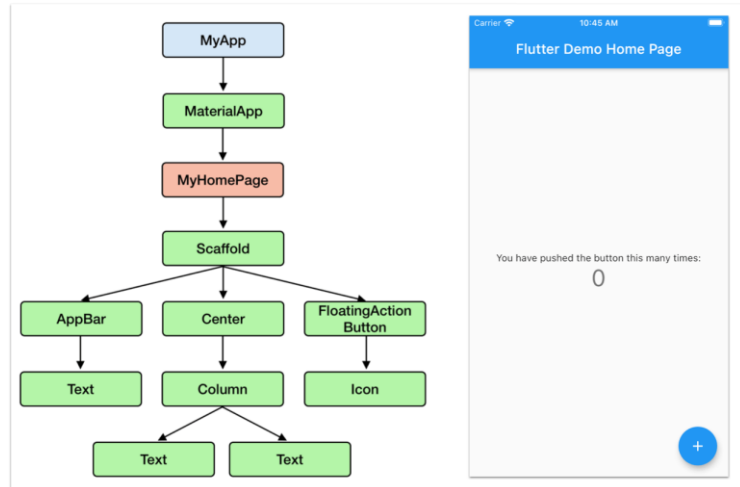
Flutter intègre de manière native Material Design dans ses widget et ses composants d'interface. En utilisant Flutter, les développeurs peuvent rapidement créer des applications respectant les principes de Material Design, avec des éléments comme les barres d'applications, les boutons flottants et les menus latéraux. Grâce à MaterialApp et Scaffold, Flutter facilite l'implémentation de ces concepts visuels tout en offrant des multiples différents outils de personnalisation.

Widgets de base

Dans flutter, un widget est un composant d'interface utilisateur. Chaque élément visible ou structure logique est un widget.

Introduction aux Widgets

C'est l'organisation des widgets sous forme de parent et d'enfants. Chaque widget peut contenir d'autres widgets, créant ainsi une hiérarchie. On peut visualiser ce concept dans cette image. Dans les pages suivantes, on va vous expliquer les différents widgets et leur fonctionnement.



Il existe trois types principaux :

- **Widgets structurels** : comme `Column`, `Row` ou `Container`, qui définissent la disposition.
- **Widgets interactifs** : comme `ElevatedButton` ou `TextField`, qui permettent les interactions.
- **Widgets décoratifs** : comme `Padding`, `Align`, ou `Center`, qui modifient l'apparence ou la position.

MaterialApp

Le **MaterialApp** est un widget racine qui configure les paramètres globaux d'une application Flutter basée sur Material Design. Il sert de point d'entrée pour structurer et configurer l'application entière. Voici une structure de base :

Scaffold

Le **Scaffold** est un widget qui fournit une structure de base pour un écran individuel. C'est comme un "squelette" pour chaque page de l'application. Il va se trouver dans le widget MaterialApp. Voici une structure de base :

```
MaterialApp({  
  Key? key,  
  String title = '', // Titre de l'application  
  ThemeData? theme, // Thème visuel global  
  Widget? home, // Premier écran à afficher  
  Map<String, WidgetBuilder>? routes, // Navigation avec des noms de routes  
  Locale? locale, // Langue par défaut  
  bool debugShowCheckedModeBanner = true, // Cache le bandeau 'debug'  
})
```

```
Scaffold({  
  Key? key,  
  PreferredSizeWidget? appBar, // Barre d'application en haut  
  Widget? body, // Contenu principal  
  Widget? floatingActionButton, // Bouton flottant  
  Widget? drawer, // Menu latéral  
  Widget? bottomNavigationBar, // Barre de navigation en bas  
  Widget? bottomSheet, // Contenu qui glisse depuis le bas  
  Color? backgroundColor, // Couleur d'arrière-plan  
})
```

```
void main() {
  runApp(MaterialApp(
    home: Scaffold(
      appBar: AppBar(
        title: const Text('Accueil'),
        backgroundColor: Colors.red[700],
      ), // AppBar
      body: const Center(
        child: Text('Bienvenue sur mon application'),
      ), // Center
    ), // Scaffold
  )); // MaterialApp
}
```



Dans cet exemple, le widget **MaterialApp** introduit le widget **Scaffold**, qui fournit une structure de base pour l'interface de l'application, incluant des éléments comme la barre d'application (AppBar) et le corps principal (body).

Le mot-clé **const** permet de déclarer des objets immuables, optimisant ainsi la performance en évitant leur recréation pendant l'exécution de l'application.

StatelessWidget VS StatefulWidget

StatelessWidget

Le **StatelessWidget** est un concept fondamental dans le développement Flutter, représentant un widget dont l'état ne change pas au fil du temps. Ce type de widget est important car il simplifie le développement, réduit la complexité et améliore la performance, puisque l'interface n'a pas besoin d'être redessinée lorsqu'elle reste inchangée. De plus, il est compatible avec la fonctionnalité Hot Reload de Flutter, permettant aux développeurs de visualiser instantanément les modifications apportées au code sans avoir à redémarrer l'application. Reprenons l'exemple ci-dessus, en ajoutant un StatelessWidget.

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

  @override
  Widget build(BuildContext context) {
    return const Text('Hello World',
      style: TextStyle(fontSize: 30,
        color: Colors.teal)); // TextStyle // Text
  }
}
```

body: const Home()

Main



Hot Reload (CTRL + S)

StatefulWidget

Un **StatefulWidget** est un widget qui possède un état mutable, c'est-à-dire que ses données peuvent changer pendant la durée de vie du widget, et ces changements doivent être reflétés dans l'interface utilisateur. Il est utilisé lorsque vous avez besoin d'une interaction ou d'un contenu dynamique.

Un **StatefulWidget** est divisé en deux classes :

1. La classe principale, qui étend StatelessWidget (par exemple, CounterWidget).

2. Une classe associée qui gère son état, et qui étend State (par exemple, `_CounterWidgetState`).

Chaque fois que l'état change, Flutter reconstruit l'interface utilisateur en appelant la méthode `build()` de la classe associée au widget.

Qu'est-ce que `setState()` ?

La méthode `setState()` est utilisée dans un **StatefulWidget** pour mettre à jour son état. Elle signale à Flutter qu'une modification de l'état s'est produite et que l'interface utilisateur doit être reconstruite.

```
void _incrementCounter() {  
  setState(() {  
    _counter++;  
  });  
}
```

Qu'est-ce que `CounterWidget` ?

CounterWidget est un exemple de **StatefulWidget** qui implémente un compteur interactif. Ce widget contient un bouton flottant permettant d'incrémenter une valeur affichée sur l'écran.

Structure de `CounterWidget` :

1. **Classe principale (`CounterWidget`) :**
 - Définit le widget comme un **StatefulWidget**.
 - Est liée à une classe d'état (`_CounterWidgetState`) qui gère la logique.
2. **Classe d'état (`_CounterWidgetState`) :**
 - Contient la variable `_counter` pour stocker la valeur actuelle du compteur.
 - Implémente la méthode `setState()` pour mettre à jour la valeur du compteur et reconstruire l'interface.

```
class CounterWidget extends StatefulWidget {  
  const CounterWidget({super.key});  
  
  @override  
  State<CounterWidget> createState() => _CounterWidgetState();  
}  
  
class _CounterWidgetState extends State<CounterWidget> {  
  int _counter = 0;  
  
  void _incrementCounter() {  
    setState(() {  
      _counter++;  
    });  
  }  
  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: AppBar(  
        title: const Text('Exemple StatefulWidget'),  
      ), // AppBar  
      body: Center(  
        child: Column(  
          mainAxisAlignment: MainAxisAlignment.center,  
          children: [  
            const Text(  
              'Vous avez appuyé sur le bouton ce nombre de fois :',  
            ), // Text  
            Text(  
              '$_counter',  
              style: const TextStyle(fontSize: 24, fontWeight: FontWeight.bold),  
            ), // Text  
          ],  
        ), // Column  
      ), // Center  
      floatingActionButton: FloatingActionButton(  
        onPressed: _incrementCounter,  
        child: const Icon(Icons.add),  
      ), // FloatingActionButton  
    ); // Scaffold  
  }  
}
```

Pourquoi utiliser un `StatefulWidget` ?

Un **StatefulWidget** est idéal dans des scénarios où l'interface utilisateur doit changer en réponse aux actions de l'utilisateur, comme :

- Cliquer sur un bouton.
- Modifier un champ de texte.
- Réagir à des données venant d'un backend ou d'une API.

Cela le distingue des **StatelessWidgets**, qui ne peuvent pas gérer d'état mutable.

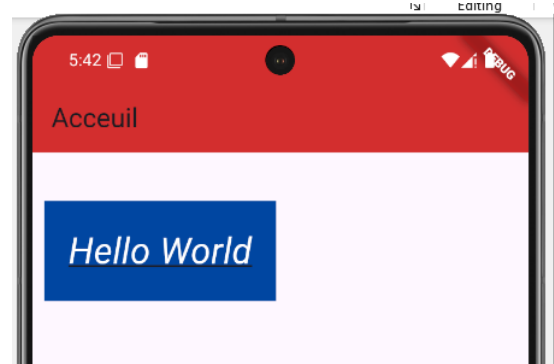
Widget de mise en page

1. Container

Le **Container** est un widget polyvalent dans Flutter, utilisé pour styliser et positionner d'autres widgets. Il permet de définir des propriétés telles que la taille, les marges, les bordures, les couleurs de fond et bien plus encore.

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

  @override
  Widget build(BuildContext context) {
    return Container (
      color: Colors.blue[900],
      // width: 300,
      // height: 300,
      padding: const EdgeInsets.all(20),
      margin: const EdgeInsets.fromLTRB(10, 40, 0, 0),
      child: const Text('Hello World', style: TextStyle(
        color: Colors.white,
        decoration: TextDecoration.underline,
        fontStyle: FontStyle.italic,
        fontSize: 30,)), // TextStyle
    ); // Container
  }
}
```

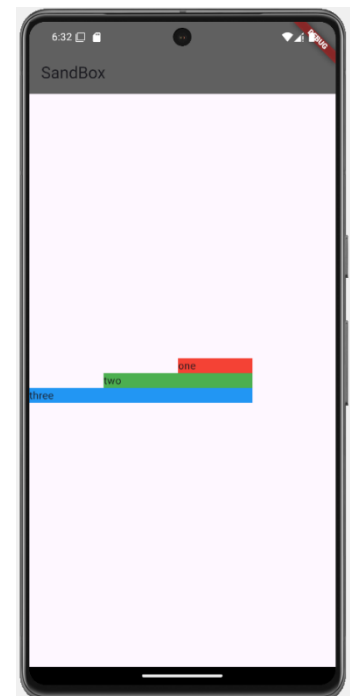


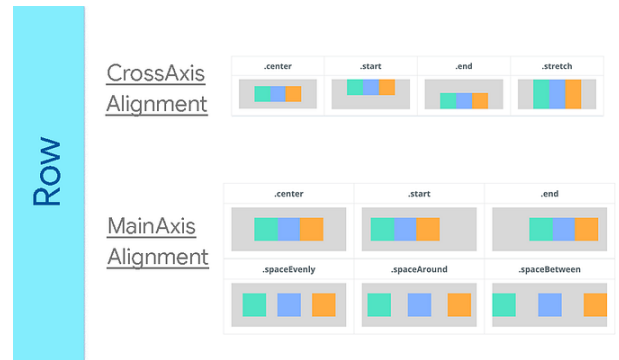
2. Column

Le **Column** est un widget dans Flutter qui permet d'organiser les éléments enfants verticalement. Le **mainAxisAlignment** et le **crossAxisAlignment** sont utilisés pour contrôler l'alignement des éléments dans un Column. Le mainAxisAlignment définit l'alignement sur l'axe principal (vertical pour une colonne), tandis que le crossAxisAlignment ajuste l'alignement sur l'axe transversal (horizontal).

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

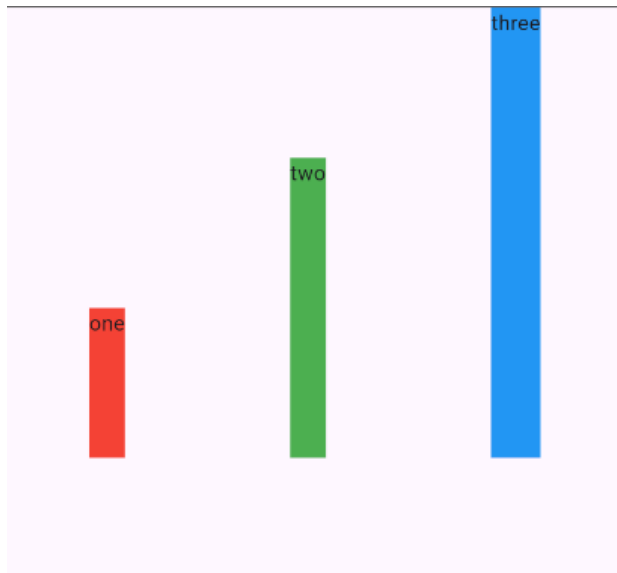
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Sandbox'),
        backgroundColor: Colors.grey[700],
      ), // AppBar
      body: Column(
        crossAxisAlignment: CrossAxisAlignment.end,
        mainAxisAlignment: MainAxisAlignment.center,
        children: [
          Container(
            width: 100,
            color: Colors.red,
            child: const Text('one'),
          ), // Container
          Container(
            width: 200,
            color: Colors.green,
            child: const Text('two'),
          ), // Container
          Container(
            width: 300,
            color: Colors.blue,
            child: const Text('three'),
          ), // Container
        ],
      ), // Column
    ); // Scaffold
  }
}
```





3. Row

Le **Row** est un widget dans Flutter qui permet d'organiser les éléments enfants horizontalement. Comme avec le **Column**, il offre des options d'alignement et d'espacement pour personnaliser la disposition des enfants sur l'axe horizontal.

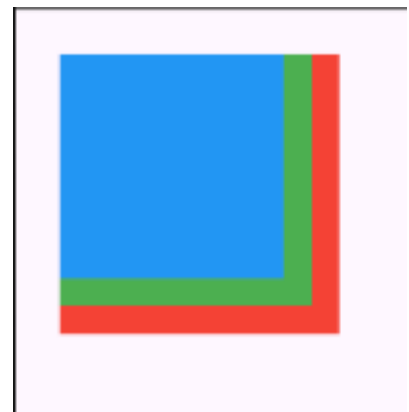


```
class Sandbox extends StatelessWidget {
  const Sandbox({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Sandbox'),
        backgroundColor: Colors.grey[700],
      ), // AppBar
      body: Row(
        mainAxisAlignment: MainAxisAlignment.spaceAround,
        crossAxisAlignment: CrossAxisAlignment.end,
        children: [
          Container(
            height: 100,
            color: Colors.red,
            child: const Text('one'),
          ), // Container
          Container(
            height: 200,
            color: Colors.green,
            child: const Text('two'),
          ), // Container
          Container(
            height: 300,
            color: Colors.blue,
            child: const Text('three'),
          ), // Container
        ],
      ), // Row
    ); // Scaffold
  }
}
```

4. Stack

Le **Stack** est un widget puissant de Flutter qui permet de superposer plusieurs widgets les uns sur les autres, offrant ainsi une grande flexibilité pour la création d'interfaces utilisateur complexes. Contrairement aux widgets comme **Column** ou **Row**, qui organisent les enfants en lignes ou colonnes, le **Stack** positionne ses enfants selon leur ordre dans la liste et leur configuration.



5. SizedBox

Le **SizedBox** est un widget simple mais puissant dans Flutter, principalement utilisé pour gérer les tailles et les espacements dans l'interface utilisateur. Il permet de définir des dimensions fixes (largeur et hauteur) pour ses enfants, ou

de créer un espacement vide entre les widgets. Il est souvent utilisé pour assurer une mise en page fluide et cohérente dans l'application.

```
body: Center(  
  child: SizedBox(  
    width: 100, // Largeur fixe de 100 pixels  
    height: 50, // Hauteur fixe de 50 pixels  
    child: Container(  
      color: Colors.blue, // Couleur du Container  
    ), // Container  
  ), // SizedBox
```

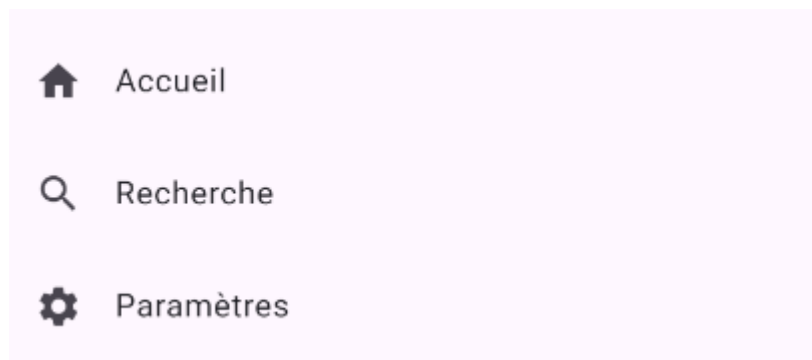
6. ListView

Le widget **ListView** en Flutter est un widget scrollable qui permet d'afficher une liste d'éléments dans une interface. Il est souvent utilisé pour présenter des données sous forme de listes verticales ou horizontales. Ce widget est très flexible et performant, en particulier lorsqu'il s'agit de longues listes, car il charge dynamiquement les éléments visibles à l'écran.

Un des widgets couramment utilisés avec **ListView** est **ListTile**. **ListTile** est un widget pratique qui permet de créer des éléments de liste avec des propriétés comme une icône, un titre, une souscription ou une action cliquable. Il est utilisé pour simplifier la création d'éléments de liste standardisés, offrant un bon design sans trop de complexité.

```
body: ListView(  
  children: const <Widget>[  
    ListTile(  
      leading: Icon(Icons.home),  
      title: Text('Accueil'),  
    ), // ListTile  
    ListTile(  
      leading: Icon(Icons.search),  
      title: Text('Recherche'),  
    ), // ListTile  
    ListTile(  
      leading: Icon(Icons.settings),  
      title: Text('Paramètres'),  
    ), // ListTile  
  ], // <Widget>[]  
) // ListView
```

Dans un **ListView**, tu peux insérer une grande variété de widgets : **ListTile**, **Divider**, **Container**, **Image**, **Card**, **Text**, et bien d'autres encore. Cela te permet de créer des listes très variées et interactives en fonction de tes besoins.



Pour plus d'informations sur les différentes fonctionnalités du **ListView** visiter la documentation flutter : <https://api.flutter.dev/flutter/widgets/ListView-class.html>

Widgets interactifs

1. TextField

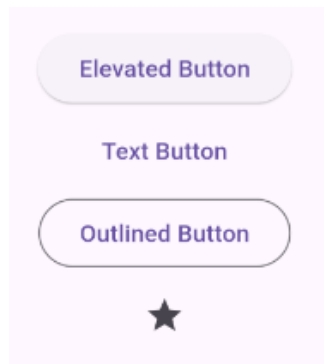
Un **TextField** dans Flutter est un widget utilisé pour permettre à l'utilisateur de saisir du texte. Il est couramment utilisé dans les formulaires et les interfaces où l'utilisateur doit entrer des informations, comme des noms, des mots de passe, des adresses e-mail, etc.

Il peut être personnalisé avec des décorations (comme un label, une bordure, ou un style de texte), et il offre des fonctionnalités telles que la gestion du texte saisi et la possibilité de récupérer ou de modifier le contenu à l'aide d'un **TextEditingController**.

```
class MyApp extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      home: Scaffold(  
        appBar: AppBar(title: const Text('Exemple Simple de TextField')),  
        body: const Center(  
          child: TextField(  
            decoration: InputDecoration(  
              labelText: 'Entrez du texte ici', // Label pour le champ  
            ), // InputDecoration  
          ), // Center  
        ), // Scaffold  
      ); // MaterialApp  
    )  
  }  
}
```

2. Button

- **ElevatedButton** : Un bouton élevé avec un fond coloré et des coins arrondis. Il est souvent utilisé pour les actions principales dans l'interface
- **TextButton** : Un bouton avec un texte simple sans fond ni bordure, utilisé pour des actions moins importantes ou secondaires.
- **OutlinedButton** : Un bouton avec une bordure visible, mais sans fond coloré. Il est souvent utilisé pour des actions secondaires, mais qui doivent être mises en évidence.
- **IconButton** : Un bouton qui affiche uniquement une icône (ici, l'icône d'une étoile). Il peut être utilisé dans des contextes où l'icône représente une action spécifique.



```
children: [  
  // Bouton Elevated (élevé)  
  ElevatedButton(  
    onPressed: () {},  
    child: const Text('Elevated Button'),  
  ), // ElevatedButton  
  // Bouton Text (texte uniquement)  
  TextButton(  
    onPressed: () {},  
    child: const Text('Text Button'),  
  ), // TextButton  
  // Bouton Outlined (bordure)  
  OutlinedButton(  
    onPressed: () {},  
    child: const Text('Outlined Button'),  
  ), // OutlinedButton  
  // Bouton Icône (avec icône)  
  IconButton(  
    onPressed: () {},  
    icon: const Icon(Icons.star),  
    tooltip: 'Icône bouton',  
  ), // IconButton  
]
```

3. GestureDetector

Le **GestureDetector** est un widget puissant dans Flutter qui permet de détecter et de répondre à différents gestes effectués par l'utilisateur, tels que les tapotements, les glissements, les pressions longues, les double-tap et d'autres interactions tactiles. Contrairement aux widgets comme les boutons qui réagissent à des événements spécifiques comme `onPressed`, le **GestureDetector** peut être utilisé avec n'importe quel widget enfant pour capturer des gestes plus variés.

Il est particulièrement utile pour rendre les interfaces utilisateur plus interactives et dynamiques. Par exemple, vous pouvez l'utiliser pour détecter un tapotement sur une image, un glissement sur une liste, ou même un geste de zoom avec deux doigts. Ce widget ne se limite pas seulement aux boutons, il

peut être appliqué à une large gamme de widgets comme les

images, les cartes, les containers, etc.

En utilisant

GestureDetector, vous pouvez configurer plusieurs types de gestes tels que **onTap**, **onDoubleTap**, **onLongPress**, et bien d'autres, pour améliorer l'interaction avec l'utilisateur dans vos applications Flutter. Cela permet de créer des interfaces plus réactives et intuitives.

Pour démontrer son fonctionnement, on peut créer un **StatefulWidget** avec un **Container** dans lequel sa couleur de fond va changer à chaque fois qu'il est cliqué. Il va falloir utiliser une fonction avec `setState`.

```
// Fonction pour changer la couleur
void _changeColor() {
  setState(() {
    // Change la couleur du carré à chaque tap
    _color = _color == Colors.blue ? Colors.red : Colors.blue;
  });
}
```

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: const Text('Exemple GestureDetector')),
    body: Center(
      child: GestureDetector(
        onTap: _changeColor, // Détecte le tap et change la couleur
        child: Container(
          width: 200,
          height: 200,
          color: _color, // Applique la couleur du carré
          child: const Center(
            child: Text(
              'Clique moi !',
              style: TextStyle(color: Colors.white, fontSize: 20),
            ), // Text
          ), // Center
        ), // Container
      ), // GestureDetector
    ), // Center
  ); // Scaffold
}
```

Widgets Visuels

1. Image

Le widget **Image** est utilisé pour afficher des images dans une application Flutter. Il peut charger des images à partir de différentes sources, telles que des fichiers locaux, des ressources réseau, des fichiers sur le système de fichiers, ou des données en mémoire (bytes). Ce widget est flexible, performant et essentiel pour créer des interfaces graphiques visuellement attrayantes.

Afficher une image depuis les Assets

Il faut d'abord créer le dossier Assets à la racine de votre projet, puis créer un dossier

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

images à l'intérieur de celui-ci. Il va

aussi falloir aussi déclarer le dossier dans le fichier pubspec.yaml. Après avoir modifié pubspec.yaml, exécutez la commande suivante pour rafraîchir les assets : **flutter pub get**

```
const Text('1. Image depuis les assets'),  
Image.asset(  
  'assets/images/example.png',  
  width: 200,  
  height: 150,  
  fit: BoxFit.cover,  
), // Image.asset
```

```
const Text('2. Image depuis une URL'),  
Image.network(  
  'https://pngimg.com/d/mario_PNG125.png',  
  width: 200,  
  height: 150,  
  fit: BoxFit.scaleDown,  
  loadingBuilder: (context, child, progress) {  
    if (progress == null) return child;  
    return const Center(  
      child: CircularProgressIndicator(),  
    ); // Center  
  },  
), // Image.network
```

Afficher une image depuis une URL

Pour charger une image depuis une URL, on utilise **image.network**. Le **loadingBuilder** affiche un loader pendant que l'image est téléchargée. Le chargement d'une image avec **Image.network** est généralement plus long que de charger une image depuis les assets ou un fichier local.

Navigation

La navigation dans Flutter permet de passer d'une page à une autre au sein de l'application. Flutter fournit une méthode simple et flexible pour gérer cela grâce à **Navigator**, qui permet de manipuler la pile de pages (ou "stack") en ajoutant ou en retirant des pages.

Navigator.push et Navigator.pop

- **Navigator.push(context, route)** : Cette méthode permet de naviguer vers une nouvelle page (ou route). Elle ajoute la page au sommet de la pile de navigation.
- **Navigator.pop(context)** : Cette méthode permet de revenir à la page précédente en retirant la page actuelle du sommet de la pile.

Pourquoi Navigator.push et Navigator.pop ?

Bien qu'il existe d'autres méthodes de navigation dans Flutter, comme les routes nommées ou des packages tiers (par exemple **GoRouter**), utiliser **Navigator.push** et **Navigator.pop** est la méthode la plus simple et la plus directe pour gérer la navigation. Elle est flexible, facile à comprendre et fonctionne bien pour des applications de taille moyenne.

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

Run | Debug | Profile

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

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: HomePage(),
    ); // MaterialApp
  }
}
```

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

```
class HomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Page d\'accueil')),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            Navigator.push(
              context,
              MaterialPageRoute(builder: (context) => SecondPage()),
            );
          },
          child: const Text('Aller à la page suivante'),
        ), // ElevatedButton
      ), // Center
    ); // Scaffold
  }
}
```

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

```
class SecondPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Deuxième page')),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            Navigator.pop(context);
          },
          child: const Text('Retour à la page précédente'),
        ), // ElevatedButton
      ), // Center
    ); // Scaffold
  }
}
```

```
ElevatedButton(
  onPressed: () async {
    // Navigue vers la deuxième page et attend un objet Item en retour
    final result = await Navigator.push(
      context,
      MaterialPageRoute(builder: (context) => SecondPage()),
    );

    // Si un objet a été retourné, on l'ajoute à la liste
    if (result != null) {
      addItem(result);
    }
  },
  child: const Text('Ajouter un item'),
), // ElevatedButton
```

```
// Retourne la chaîne de caractère à la première page
Navigator.pop(context, "Bonjour");
```

Séparation des fichiers

Pour mieux organiser ton application, il est recommandé de séparer chaque page dans son propre fichier.

1. **main.dart** : Le fichier principal qui lance l'application et configure la navigation de base.
2. **home_page.dart** : Le fichier contenant la première page de l'application.
3. **second_page.dart** : Le fichier contenant la deuxième page, vers laquelle tu navigueras.

Pour passer des données du page à l'autre, il va falloir ajouter quelques lignes de code pour que ça puisse fonctionner adéquatement.

Pour en apprendre plus sur la navigation et sur comment passer des objets d'une page à l'autre, veuillez visiter la documentation suivante : [Passage de Données](#)

Publication sur le Google Play Store

A. Configurez votre application

1. Ouvrez `android/app/src/main/AndroidManifest.xml`
2. Assurez vous que l'application a les permissions à l'internet
3. Vérifiez que le nom et l'icône de l'application sont corrects dans la balise `<application>`.
4. Dans le fichier `android/app/build.gradle`, il faut définir l'ID unique de l'application, la version et le SDK qui est minimal. Vous devriez voir les champs `applicationID`, `minSdkVersion`, `targetSdkVersion`, `VersionCode` et `VersionName`

- `ApplicationID`

L'`applicationId` est l'identifiant unique de votre application sur le Google Play Store. Cet identifiant est justement utilisé pour que il puisse y avoir une différence entre votre application et celle des autres.

- `MinSdkVersion`

Le champ `minSdkVersion` indique la version minimale d'Android requise pour exécuter l'application, définie par un niveau d'API.

- `TargetSdkVersion`

La `targetSdkVersion` spécifie la version d'Android avec laquelle l'application est conçue pour fonctionner de manière optimale. `VersionCode`

- `VersionCode`

Le `versionCode` est un entier incrémental qui identifie de manière unique chaque version de l'application. Il est utilisé par le Play Store pour gérer les mises à jour.

- `VersionName`

Le `versionName` est une chaîne de caractères qui représente la version visible de l'application, comme "1.0.0". Contrairement au `versionCode`, il est affiché aux utilisateurs dans le Play Store et dans les paramètres de l'application.

Générez une clé de signature

Une clé de signature est comme une empreinte numérique unique qui identifie le développeur de l'application. Chaque mise à jour future de l'application doit être signée avec la même clé, sinon les utilisateurs ne pourront pas l'installer. Cela empêche également d'autres personnes de publier des versions modifiées de votre application sous votre identité.

Étapes :

1. **Créer une clé de signature** : C'est un fichier sécurisé qui contient vos informations de signature. Ce fichier est stocké localement sur votre ordinateur.
Exemple : `"keytool -genkey -v -keystore ~/chemin/vers/mon-keystore.jks -keyalg RSA -keysize 2048 -validity 10000 -alias mon-alias"`
Placez le fichier .jks dans le répertoire android/.
Configurez un fichier key.properties : Créez un fichier key.properties dans le dossier android/ avec les champs storePassword, keyPassword, keyAlias, storeFile. Les deux premiers champs représentent le mot de passe, le troisième quant à lui s'agit de l'alias, et le dernier champ s'agit du chemin vers le keystore.
2. **Configurer le projet Android** : Une fois la clé générée, vous indiquez à votre application d'utiliser cette clé pour signer le fichier final avant publication, soit en modifiant build.gradle. On indique à Gradle où trouver la clé et les mots de passe associés.
3. **Signer automatiquement votre application** : Lors de la génération du fichier final pour le Play Store, Android utilisera la clé pour ajouter une signature numérique.

Publication sur le Google Play Store

A. Accédez à la Google Play Console

1. Connectez-vous à Google Play Console.
2. Créez un nouveau projet :
 - a. Fournissez un **nom d'application**, une **langue par défaut** et d'autres détails de base.

B. Préparez la fiche Play Store

1. Ajoutez des **captures d'écran** de votre application (obtenez-les en utilisant l'émulateur ou un appareil réel).
2. Ajoutez une **icône haute résolution** (512x512 pixels).
3. Rédigez une **description courte** et une **description longue**.

C. Téléversez l'application

1. Accédez à **Production > Créer une nouvelle version**.
2. Téléversez le fichier .aab généré précédemment.
3. Passez les vérifications de compatibilité.

D. Déclarez les détails de l'application

1. **Évaluation du contenu** : Remplissez un questionnaire pour classer votre application.
2. **Public cible** : Déclarez l'audience visée (ex. enfants, adultes).
3. **Politique de confidentialité** : Ajoutez un lien vers votre politique de confidentialité.

6. Soumission et examen

1. Une fois toutes les sections complétées, soumettez l'application pour examen.

2. Google examinera votre application (1 à 7 jours en général).
3. Une fois approuvée, votre application sera disponible sur le Play Store.

Bibliographie

"Documentation Flutter." *Flutter Dev Docs*, Flutter, <https://docs.flutter.dev/>. Consulté le 15 décembre 2024.

Bhanderi, Parth. "Les avantages et inconvénients de Flutter." *Medium*, 25 oct. 2020, <https://medium.com/@parthbhanderi01/flutters-benefits-and-drawbacks-b268c1fe0f7f>. Consulté le 15 décembre 2024.

"Tutoriel Flutter." *GeeksforGeeks*, https://www.geeksforgeeks.org/flutter-tutorial/?ref=header_outind. Consulté le 15 décembre 2024.

"Flutter (logiciel)." *Wikipédia*, [https://en.wikipedia.org/wiki/Flutter_\(software\)](https://en.wikipedia.org/wiki/Flutter_(software)). Consulté le 15 décembre 2024.

Dummies. *Flutter for Dummies*. John Wiley & Sons, Inc., 2020, p. 261-262.

Annexe

Source humaine

Ayman Adas Developpeur Flutter Chez HulSoft Technologies

Introduction

Je travaille chez HulSoft Technologies depuis 2017. Je travaille avec Flutter, un outil qui permet de créer des applications pour iOS et Android avec le même code. Contrairement à des langages comme Swift, Kotlin, Objective-C ou Java, Flutter facilite le travail des programmeurs en utilisant un seul code pour plusieurs plateformes. Par rapport à React Native, Flutter offre des avantages comme le hot reload, une plus grande vitesse de développement et des designs plus attrayants.

Avantages principaux de Flutter

- **Hot reload** : Permet de voir les changements instantanément.
- **Développement rapide** : Idéal pour créer des applications rapidement.
- **Bonne performance** : Les applications sont fluides et rapides.

Erreurs courantes avec Flutter

- **En général** :
 - Mettre tout le code dans un seul fichier, ce qui rend le projet difficile à comprendre.
 - Ne pas suivre les règles de base du codage ou les bonnes pratiques.
 - Ne pas utiliser des noms clairs, ce qui complique la maintenance.
- **Avec Flutter** :
 - Ne pas utiliser **const** pour les widgets qui ne changent pas, ce qui peut ralentir l'application.
 - Ne pas rendre l'application **adaptable** à différents écrans.
 - Utiliser **setState** pour tout, ce qui recharge toute la page au lieu de juste une partie.
 - Ne pas charger les données au fur et à mesure, ce qui peut rendre l'application lente avec beaucoup de contenu.
- **Autres erreurs** :
 - Publier un fichier de test au lieu d'une version finale pour les utilisateurs.
 - Oublier de modifier le fichier **AndroidManifest** quand on ajoute des fonctions comme la caméra.

Gestion d'état

- J'ai utilisé des outils comme **Bloc** et **Provider** pour gérer l'état des applications.
- J'aime bien **Bloc**, mais je peux m'adapter à d'autres outils selon le besoin.

Limites de Flutter

- **Flutter Web** n'est pas encore parfait et a des restrictions.
- Les dossiers Android et iOS, comme Gradle et Runner, pourraient être plus simples à utiliser.

Conclusion

Flutter est un excellent outil pour créer des applications sur plusieurs plateformes. Cependant, il est important de bien comprendre les bases et d'éviter les erreurs courantes pour tirer le meilleur parti de ce framework.

Trois moteurs de recherche spécialisés

1. Google Scholar

The screenshot shows the Google Scholar interface with the search term 'flutter'. The search results are displayed in a list format. On the left, there are filters for 'Any time' (Since 2024, Since 2023, Since 2020, Custom range...), 'Sort by relevance' (Sort by date), 'Any type' (Review articles), and checkboxes for 'include patents' and 'include citations'. A 'Create alert' button is at the bottom left. The search results list includes:

- Aeroelastic design of a drone for research on active flutter control** (PDF) hal.science. N Fabbiane, V Bouillaut, A Lepage - IFASD 2024, 2024 - hal.science. ... onset of the flutter instability. This leads to very peculiar specifications for the nominal flutter behaviour of the aircraft and, therefore, to adapted solutions in the design process. The flutter ...
- Flutter in action** (book) Flutter in action. E Windmill - 2020 - books.google.com. ... to use Flutter, but you'll understand why using Flutter in the ... found challenging when moving to Flutter. Between these pages ... a simple Flutter example app to explain how Flutter works, ...
- Influence of a hysteretic damper on the flutter instability** (PDF) hal.science. A Malher, O Doaré, C Touzé - Journal of Fluids and Structures, 2017 - Elsevier. ... This paper focuses on the control of the airfoil classical flutter ... flutter velocity. The goal of the control is then to increase the ... the reader is referred to the web version of this article.) ...
- Towards unsteady approach for future flutter calculations** (PDF) hal.science. L Mouton, A Leroyer, G Deng, P Queutey - Journal of Sailing ..., 2018 - hal.science. ... d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés. ... Two main approaches of flutter evaluation have been investigated in this article. The first ...
- Integrated structural and control design (Co-Design) for active flutter suppression**. E Faisse - 2022 - theses.fr.

On the right, there is a user profile for Tony Mourrah (tony.mourrah04@gmail.com) with buttons for 'My Account', 'Add account', and 'Sign out'.

2. Google books

Google Livres

Rechercher dans Google Livres

Recherche avancée

Revenir à la version classique de Google Livres

Flutter For Dummies
De Barry Burd

Introduction 1

Part 1: Getting Ready 7

CHAPTER 1: What Is Flutter? 9

CHAPTER 2: Setting Up Your Computer for Mobile App Development 29

Part 2: Flutter: A Burd's-Eye View 67

CHAPTER 3: "Hello" from Flutter 69

CHAPTER 4: Hello Again 105

CHAPTER 5: Making Things Happen 131

CHAPTER 6: Laying Things Out 163

Part 3: Details, Details 205

CHAPTER 7: Interacting with the User 207

CHAPTER 8: Navigation, Lists, and Other Goodies 247

CHAPTER 9: Moving Right Along 297

Part 4: The Part of Tens 321

CHAPTER 10: Ten Ways to Avoid Mistakes 323

CHAPTER 11: Ten Ways to Enhance Your App Development Career 327

CHAPTER 12: Ten Chapters about Flutter App Development 331

Part 5: Appendices 335

APPENDIX: Doris's Dating App 337

Index 347

flutter

1 sur 98

Tout afficher

Page v

3. Google Groups

Groupes

Créer un groupe

Mes groupes

Groupes récents

Groupes favoris

Conversations marquées d'une étoile

Tous les groupes et... flutter

Groupes




requête correspondant à « flutter »

1-89 of about 234

Groupes	Nbre de membres	Activité
Lifeboat Technologies is the best flutter training institute in Hyderabad providin...		
flutter-users@googlegroups.com Share flutter skills with other members. If you have problem with flutter you can ...	-	-
flutter-google-play-testers@googlegroups.com flutter android testers Apps	-	-
flutter-a-coruna@googlegroups.com Si te gusta flutter, ya seas programador o curioso, este es tu sitio.	-	-
flutter-devs@googlegroups.com A community to request help with your flutter code.	-	< 1 message par jour Dernier message : 13 déc.
flutter-developers@googlegroups.com This group is made for connecting all flutter developers in one place those are s...	-	-
flutter-canarias@googlegroups.com Group for Flutter devs in the Canary Islands	-	Aucun au cours des 30 derniers jours Dernier message : 2021-11-11
flutter-web-hot-restart-issue@googlegroups.com while using flutter web through Android Studio/VS Code, on hot restart , all the p...	-	-
		Aucun au cours des 30 derniers

Confidentialité • Conditions

Le suivi de mots-clés (5) avec Google Alertes



Alerts

Monitor the web for interesting new content

My alerts (5)

Flutter Canada

Flutter trends

Flutter vs React Native

Flutter Quebec

Flutter performance

How often

As-it-happens

Sources

Automatic

Language

French

Region

Canada

How many

All results

Deliver to

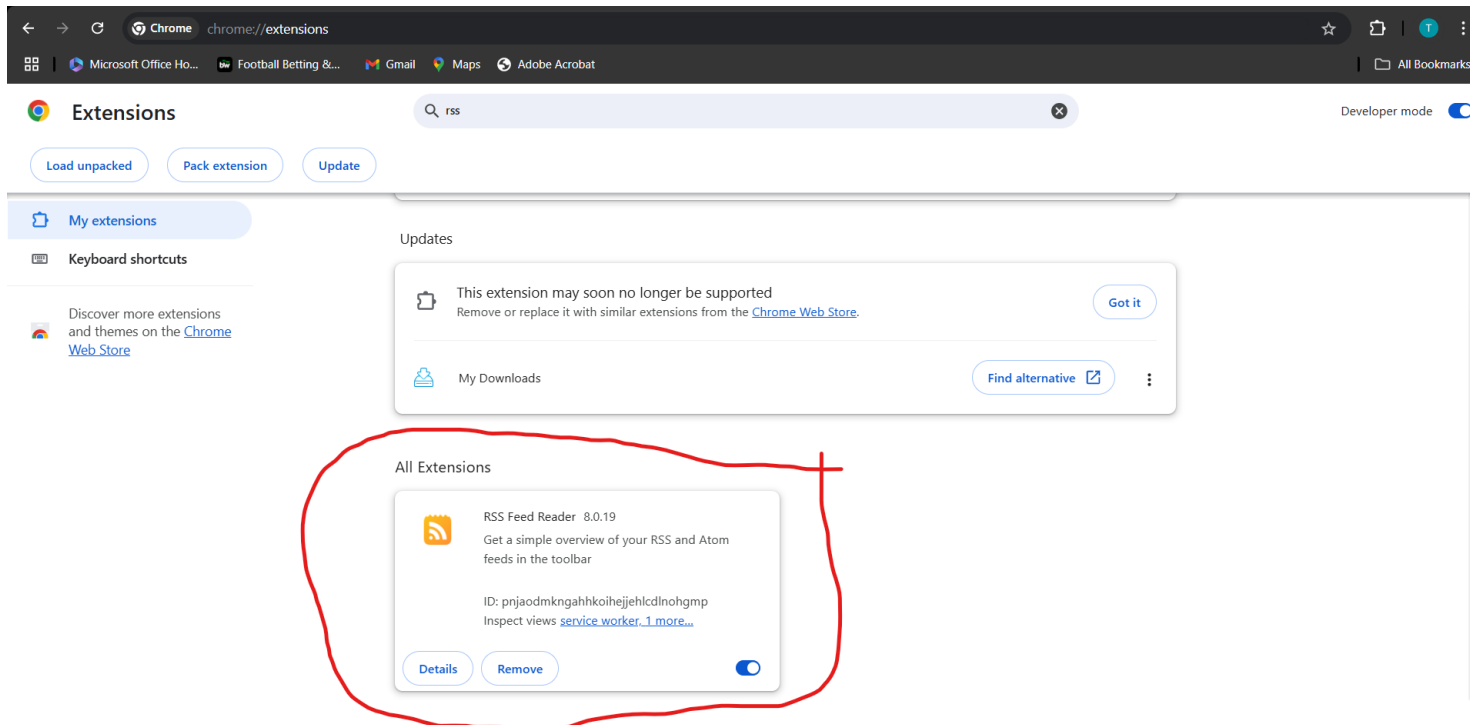
tony.mourrah04@gmail.com

Update alert

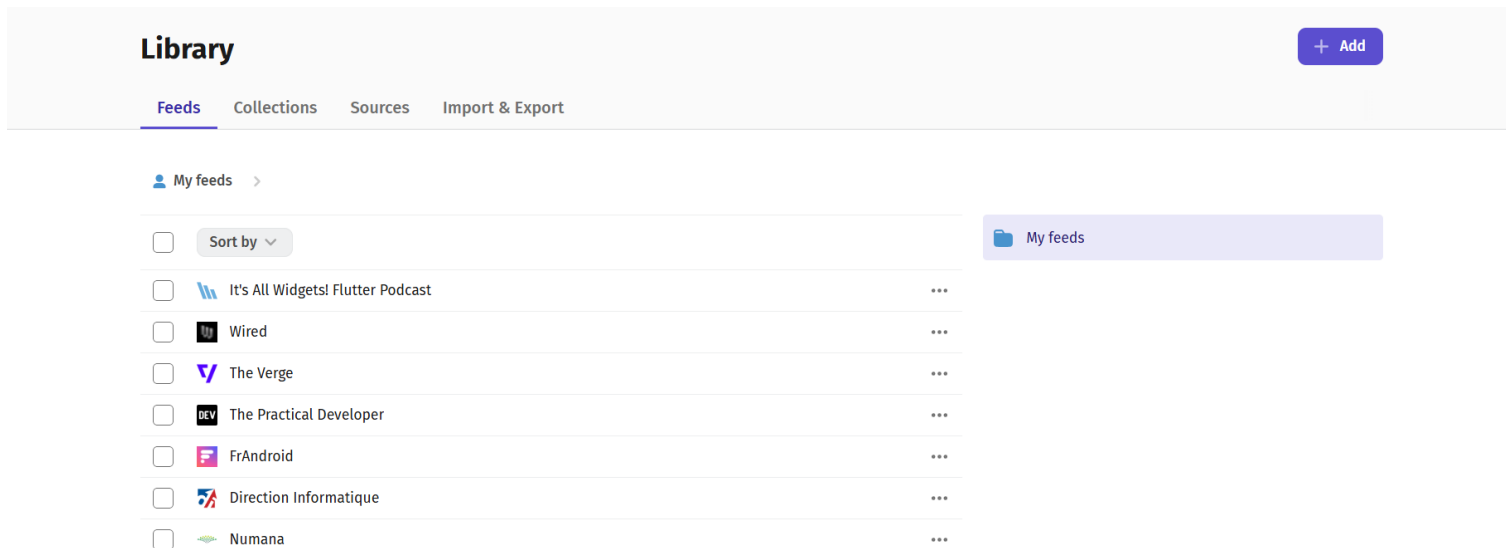
Hide options ▲

L'abonnement à six flux RSS provenant de sites Web différents

Utilisation extension RSS Feed Reader sur Chrome



L'abonnement aux flux RSS (6 sites webs)



La mise en oeuvre d'un agent de surveillance qui monitore les changements de cinq pages Web

Utilisation Infominder

Subscription Expires on 12/14/2025 >> Upgrade

Welcome, Tony Mourrah [Logout](#)

Search Pages [Go](#)

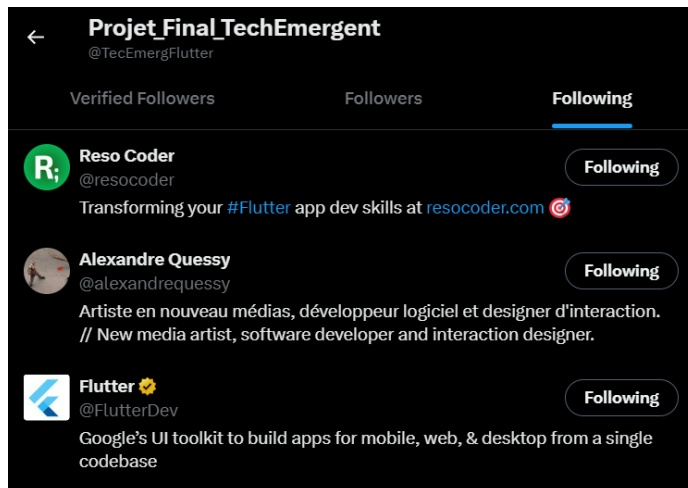
InfoMinder

[Add Page\(s\)](#) [Delete Checked](#) [Share Checked](#) View List All Total Minders: 5

	Edit	Page Location (URL)	Description	Changes	Changed On
<input type="checkbox"/>	Edit	www.itworldcanada.com/	Technology News for IT Professionals in Canada IT World Canada	0	Never Changed
<input type="checkbox"/>	Edit	www.techno-science.net/	Actualité technologique et scientifique	0	Never Changed
<input type="checkbox"/>	Edit	www.meetup.com/fluttermontreal...	Flutter Montréal @ Meetup	0	Never Changed
<input type="checkbox"/>	Edit	fluttercanada.ca/	Flutter Canada	0	Never Changed
<input type="checkbox"/>	Edit	www.appstudio.ca/flutter-app-d...	Flutter App Development au Canada	0	Never Changed

La creation d'un compte sur X (Twitter) et l'abonnement à trois comptes.

The screenshot shows the Twitter (X) app interface. On the left is a navigation menu with icons for Home, Explore, Notifications, Messages, Grok, Communities, Premium, Profile, and More. Below the menu is a 'Post' button and a user profile snippet for 'Projet_Final_TechEmergent' (@TecEmergFlutter). The main content area shows the profile of 'Projet_Final_TechEmergent' (@TecEmergFlutter), which is marked as 'Get verified'. The profile bio is empty, and it shows '3 Following' and '0 Followers'. Below the profile information are tabs for 'Posts', 'Replies', 'Highlights', 'Articles', 'Media', and 'Likes'. The 'Posts' tab is selected, and the content area displays a 'Let's get you set up' section with four colorful buttons: 'Complete your profile', 'Follow 5 accounts', 'Follow 3 topics', and 'Turn on notifications'.



Reso Coder : Développeur Flutter populaire, créateur de tutoriels et vidéos YouTube sur Flutter, Dart, et des bonnes pratiques de développement mobile

Alexandre Quessy : Développeur Flutter québécois travaille chez Art Plus Code .

Flutter : Le compte officiel de Flutter, avec des annonces sur les nouvelles versions, des événements à venir et des ressources pour les développeurs Flutter.