Lecture individuelle n° 2

Flutter

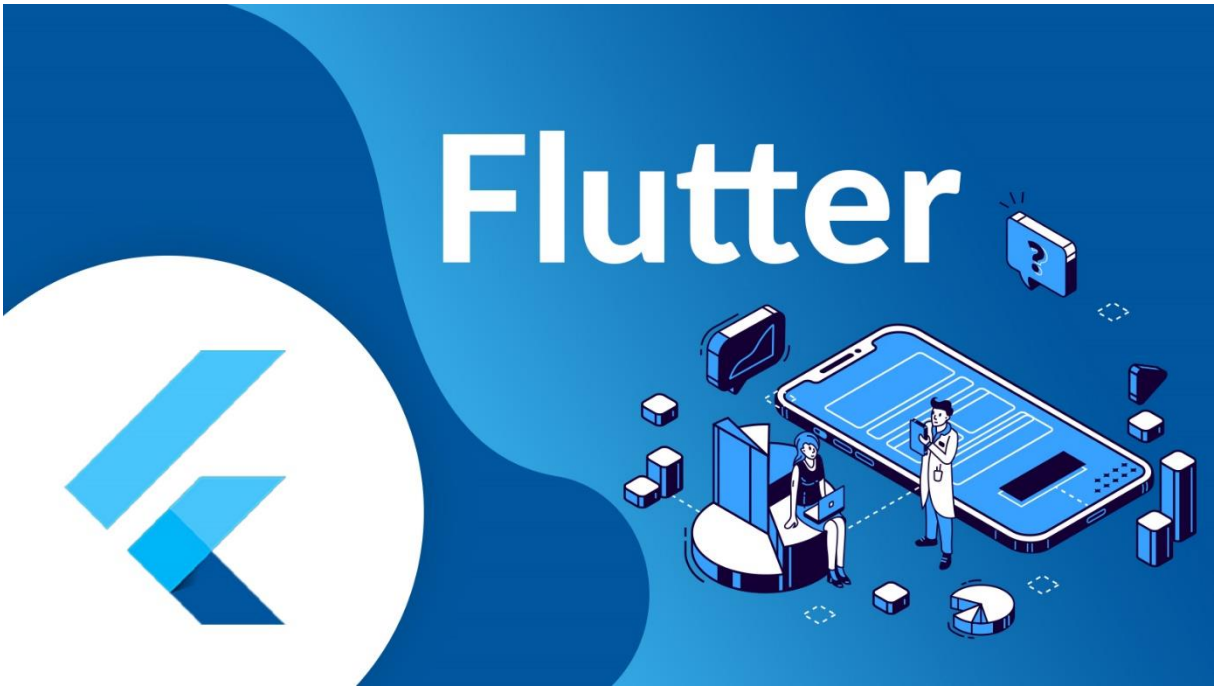


Table des matières

Introduction

Flutter est un SDK (software development kit), ce qui veut dire kit de développement.

Selon Wikipédia, un kit de développement, est un ensemble d'outils logiciels destinés aux développeurs, facilitant le développement d'un logiciel sur une plateforme donnée.

Le SDK Flutter a été conçu par Google pour s'adapter aux deux environnements de développements iOS et Android: Xcode et Android Studio.

Il s'installe sur notre machine en complément de ces deux logiciels.

En résumé, Flutter nous permet de travailler depuis un éditeur de code et de compiler pour les deux plateformes en même temps.

Ce SDK a plusieurs avantages :

- La rapidité de développement
    - Flutter nous offre une grande rapidité de développement. À chaque enregistrement de nos fichiers, grâce à l'extension Flutter, notre application s'actualise sur notre émulateur.

- Son interface utilisateur très flexible
    - Le deuxième point essentiel de Flutter est son interface utilisateur ou vraiment révolutionnaire. Il existe une immense bibliothèque d'animations et d'interactions que Flutter propose pour dynamiser nos applications.

- Des performances natives
    - Les "performances natives" font référence à la vitesse, à l'efficacité et à la capacité d'exécution optimale d'un programme, d'une application ou d'un système sur une plateforme particulière, généralement celle pour laquelle le logiciel a été spécifiquement conçu. Lorsqu'un logiciel est conçu pour tirer pleinement parti des fonctionnalités matérielles et des spécificités d'une plateforme donnée, on dit qu'il offre des performances natives.

Installation

Dans la documentation officielle de Flutter, il y a une marche à suivre de son installation pour les différents systèmes d'exploitation.

Elle se trouve à ce lien :

https://docs.flutter.dev/get-started/install

Dans ce lien, il y a aussi l'explication de la configuration de Flutter dans les différents logiciels de code (Visual Studio Code // Android Studio et Intellij) :

https://docs.flutter.dev/get-started/editor?tab=vscode

Créer une application en partant de zéro

Pour créer une application Flutter, il faut aller dans le terminal, dans le dossier où l'on souhaite créer son application, et exécuter la commande suivante :

Flutter create my_app

my_app peut être remplacé par le nom souhaité pour votre application.

Qu'est-ce que Dart ?

Dart est un langage de programmation, développé par Google, au départ pour combler les défauts du JavaScript et avait pour objectif de le remplacer en tant que langage phare du web.

Cependant ce langage est devenu plus connu en tant que langage de programmation principal pour le développement d'applications avec Flutter.

Aujourd'hui, avec le langage Dart et Flutter, nous pouvons créer des applications mobiles, des sites web et des logiciels de bureau.

Le Dart et les autres langages

Le Dart est un langage orienté objet comme le sont la plupart des langages de développement logiciel (C#, Swift, ou Objective C).

Sur les frameworks JavaScript comme Angular et React, on développe avec les langages web classiques:

HTML: Pour le contenu de notre application (texte, image, vidéo, boutons)
CSS: Pour styliser nos pages et ajouter des animations
JavaScript: Pour dynamiser nos applications et les connecter aux bases de données

Avec Flutter, 98% de notre application sera codée en Dart, qu'il s'agisse de la partie design ou de la connexion à la base de données Firebase.
C'est à la fois un avantage et un inconvénient, car dans le cas des langages web cela facilite l'apprentissage au début.

Le langage HTML par exemple est très facile à prendre en main, et on peut progressivement styliser nos pages avec le CSS puis les dynamiser avec le JS.

Avec Flutter et Dart c'est différent, on doit obligatoirement comprendre et maîtriser le Dart pour afficher du contenu.

L'avantage est qu'une fois le Dart maîtrisé, on peut tout coder avec ce même langage jusqu'à la fin du développement de notre application.

DART

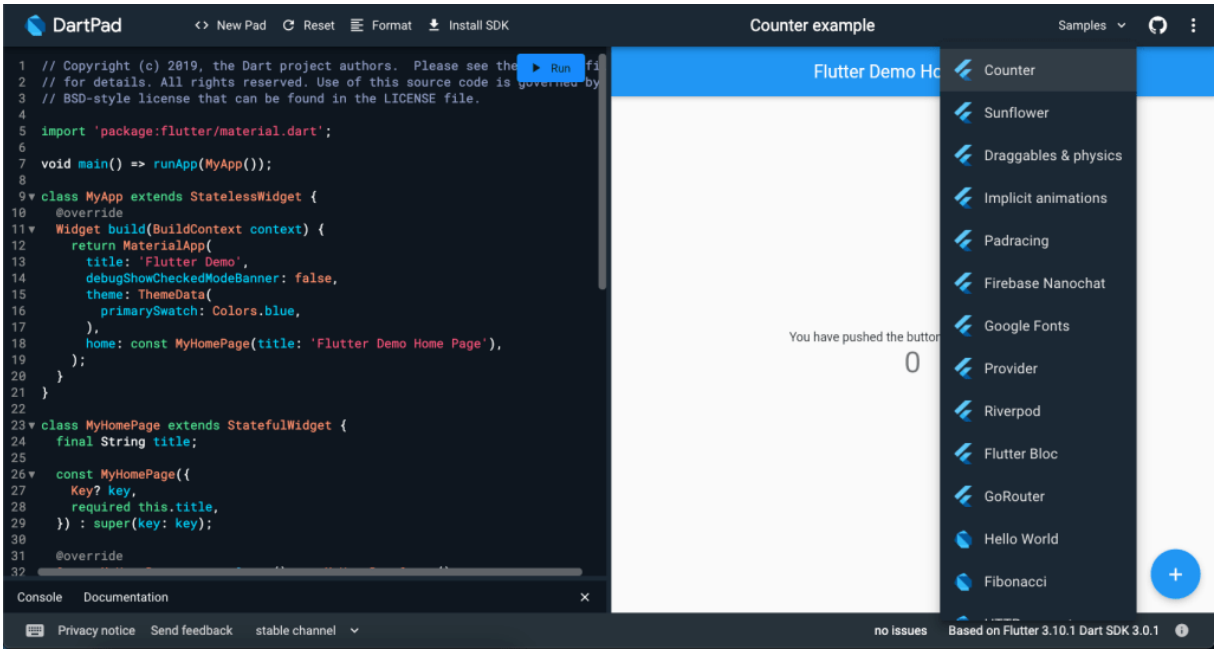
Dart, comme tous les langages informatiques, possède des règles et des notions fondamentales à connaître. Dans cette partie théorique, nous allons explorer les fondements du Dart, qui vous seront indispensables dans votre parcours de développement d'applications mobiles avec Flutter.

Présentation de DartPad

Pour faciliter notre apprentissage, nous utiliserons la plateforme DartPad, un outil développé par Google qui permet d'exécuter du code Dart directement depuis un navigateur. Vous pouvez y accéder via le lien suivant: https://dartpad.dev/

DartPad offre des exemples de scripts déjà codés, ce qui vous permet d'explorer différents aspects du langage.

Il propose également des exemples d'applications Flutter complètes. Dans l'onglet "Samples" en haut à droite, vous trouverez une variété d'exemples avec des icônes Dart ou Flutter correspondantes.



Documentation Dart : https://dart.dev/tools/dart-run

La fonction main()

Dans Dart, la fonction main() joue un rôle essentiel, car c'est le point d'entrée de votre programme. Tout le code que vous souhaitez exécuter dans votre application Dart ou Flutter doit être placé à l'intérieur de cette fonction.

Voici un exemple simple :

```dart
void main() {
  print('Hello, World!');
}
```

Ici, la fonction main() affichera le message « Hello, World! » dans la console lorsqu'elle est exécutée. C'est un moyen simple de vérifier si votre programme fonctionne correctement.

Dans la plupart des cas, vous n'écrirez pas tout votre code à l'intérieur de la fonction main().

Au lieu de cela, vous créerez des fonctions supplémentaires pour effectuer des tâches spécifiques et vous appellerez ces fonctions depuis la fonction main().

Voici un exemple:

```Dart
void printInteger(int aNumber) {
  print('The number is $aNumber.');
}

void main() {
  var number = 42;
  printInteger(number);
}
```

Cela permet d'organiser votre code en morceaux logiques et facilite la compréhension et la maintenance de votre application.

Les classes Dart

Une classe est un concept fondamental de la POO. Elle regroupe des méthodes et des propriétés qui permettent d'interagir avec les données. À partir d'une classe, vous pouvez créer plusieurs objets qui partagent les mêmes caractéristiques et comportements. Voici un exemple de classe Dart qui représente un entrepreneur.

```Dart
class Startuper {
  String name = '';
  String company = '';

  String createSentence() {
    return '$name a créé $company';
  }
}
```

Objet et Instance

Un objet est une structure informatique qui regroupe un ensemble de données ou de champs. Lorsqu'un objet est créé **à partir d'une classe**, on parle d'une **instance** de cette classe. Par exemple, pour créer une instance de la classe Startuper, nous utilisons la syntaxe suivante:

```Dart
Startuper person = Startuper(); // Création d'une instance
```

Voilà un exemple de **code Dart** complet pour afficher dans la console la phrase (« Elon Musk a créé SpaceX »):

```Dart
void main() {
  Startuper person = Startuper();
  person.name = 'Elon Musk';
  person.company = 'SpaceX';
  print(person.createSentence());
```

```
}

class Startuper {
 String name = '';
 String company = '';

 String createSentence() {
  return '$name a créé $company';
 }
}
```

Les différents types de variables Dart

En Dart, on retrouve les types de variables **couramment utilisés** dans la plupart des langages de programmation:

1. **Chaîne de caractères**: pour stocker du texte ou d'autres caractères.
2. **Nombre**: pour stocker des valeurs numériques entières ou décimales.
3. **Booléen**: pour stocker des valeurs booléennes (vraie ou fausse).
4. **Liste**: pour stocker une séquence ordonnée d'éléments.
5. **Map**: pour stocker une collection d'associations clé-valeur.

Dans le contexte du développement d'applications avec **Flutter**, nous utiliserons tous ces types de variables, chacun ayant une **utilité spécifique**.

Pour déclarer une variable **en Dart**, vous pouvez utiliser le **mot-clé** var, ou nous pouvons directement spécifier le type de variable.

Voici des exemples :

Chaînes de caractères :

```
Dart
var chaine1 = 'Une chaîne de caractères avec deux apostrophes';
String chaine1 = 'Une chaîne de caractères avec deux apostrophes';
```

Les nombres

```
Dart
var nombre1 = 1;
int nombre2 = 42;
double nombre4 = 1.2;
```

Les booléens

```
Dart
var booleen1 = true;
bool booleen2 = false;
```

Les listes

```
Dart
var startups = ['Apple', 'Google', 'Facebook', 'Amazon'];
List entrepreneurs = ['Elon Musk', 'Steve Jobs', 'Bill Gates'];
```

Une particularité des listes en Dart est qu'elles peuvent stocker des données de différents types :

```
Dart
List muskData = ['Elon Musk', 49, 104.6e9, 'Los Angeles'];
```

Vous pouvez ajouter des éléments à la liste à l'aide de la fonction add() :

```Dart
growableList.add('Amazon');
```

Les maps

Les maps en Dart sont similaires aux objets en JavaScript. Ils permettent de regrouper différents champs dans une même variable, chaque champ étant associé à une clé.

Pour déclarer une map en Dart, vous pouvez utiliser les accolades {} et spécifier les paires clé-valeur:

```Dart
var map = {
  'clé1': 'valeur 1',
  'clé2': 1,
};
```

Vous pouvez également utiliser le mot-clé Map pour déclarer une nouvelle map:

```Dart
Map userData = {
  'pseudo': 'Elon Musk',
  'age': 49
};
```

Nous pouvons afficher le **contenu complet** de la map ou accéder à des champs spécifiques et les afficher dans la **console**:

```Dart
print(userData);
print('${userData['pseudo']} a ${userData['age']} ans');
```

Les opérateurs de conditions (if & else)

Ils permettent de vérifier le contenu des variables et d'exécuter du code en fonction de certaines conditions.

Par exemple, si nous avons une variable d'âge qui contient l'âge d'un utilisateur, nous pouvons vérifier si cet utilisateur est majeur en comparant son âge à 18 :

```Dart
int age = 25;
if (age >= 18) {
  print('Il est majeur');
} else {
  print('Il est mineur');
}
```

Les opérateurs de conditions sont essentiels dans le développement mobile pour effectuer des actions en fonction de certaines conditions sur les variables.

Les boucles Dart

Une boucle permet de répéter une opération en respectant certaines conditions ou en parcourant une liste, par exemple.

On peut spécifier les paramètres de la boucle directement dans la fonction for(). Par exemple, pour afficher les mois de l'année, on peut utiliser:

```Dart
for (int month = 1; month <= 12; month++) {
  print(month);
}
```

Les boucles for sont également utiles pour parcourir des listes de données. On peut utiliser l'instruction for...in pour itérer sur chaque élément de la liste:

```Dart
List<String> startups = ['Apple', 'Google', 'Facebook', 'Amazon'];
for (String name in startups) {
  print(name);
}
```

Une autre option est d'utiliser la fonction forEach() pour parcourir et afficher les éléments d'une liste:

```Dart
startups.forEach((name) => print(name));
```

Si vous utilisez une liste de nombres, vous pouvez préciser le type de données attendu pour chaque élément:

```Dart
List<int> numbers = [1, 2, 3, 4, 5];
numbers.forEach((int i) => print(i));
```

Il existe d'autres opérateurs de boucle moins couramment utilisés, tels que while et do-while.

Les différentes fonctions Dart

Les fonctions sont **indispensables** dans la programmation, car elles permettent d'exécuter rapidement du code en les appelant. En Dart, il existe des fonctions préconstruites telles que la fonction **print()** qui affiche du contenu dans la console. Mais vous pouvez également créer **vos propres fonctions** pour structurer votre code et séparer les différentes actions.

Les fonctions de haut niveau

Il s'agit de la fonction principale main(). Cette fonction est le point d'entrée de votre code Dart, à partir duquel tout le reste du code sera exécuté.

Les fonctions anonymes

Une fonction anonyme est **incluse dans une autre fonction** et **ne retourne aucun résultat**. Elle est souvent utilisée pour exécuter un code spécifique sans avoir à le répéter. Par exemple :

```Dart
Function helloWorld = () {
  print('Hello');
};};
```

Les fonctions de premier ordre

Les fonctions de premier ordre **renvoient des données** qui peuvent être utilisées ou stockées. Par exemple :

```Dart
int add(int n1, int n2) {
 int result = 0;
 result = n1 + n2;
 return result;
}

void main() {
 int number = add(10, 23);
 print(number);
}
 print(sentence);
}
}
```

En résumé, les fonctions sont un **élément essentiel** du développement d'applications Dart et Flutter. Elles permettent de structurer votre code, de séparer les **différentes actions** et d'améliorer sa réutilisabilité

Les différentes librairies Dart

L'installation du SDK Dart inclut des **bibliothèques Dart** que vous pouvez importer dans vos fichiers Dart.

Utiliser les librairies de Dart

Pour **importer une bibliothèque** en Dart, utilisez la syntaxe suivante:

```Dart
import 'dart:math';
```

Par exemple, une fois la bibliothèque dart:math importée, vous pouvez utiliser sa fonction de calcul de la **racine carrée** comme ceci :

```Dart
void main() {
  print("La racine carrée de 36 est: ${sqrt(36)}");
}
```

Chaque bibliothèque Dart possède sa propre documentation, où vous pouvez trouver des informations sur la façon de l'importer et les différentes fonctions qu'elle propose.

Voici quelques-unes des bibliothèques Dart **les plus connues**:

Librairies	**Description**
dart:io	Permet la prise en charge des fichiers HTTP pour les applications serveur. Cette bibliothèque est importée par défaut.
dart:core	Donne accès aux fonctionnalités de base pour les programmes Dart. Cette bibliothèque est également importée par défaut.

Librairies	Description
dart:math	Donne accès aux constantes et fonctions mathématiques, ainsi qu'à un générateur de nombres aléatoires.
dart:convert	Permet de convertir différents types de données, par exemple entre JSON et UTF-8.

La fonction print() que nous utilisons régulièrement est accessible via la **bibliothèque** dart:core.

Il est également possible d'utiliser des **alias** pour importer des bibliothèques et utiliser leurs fonctions. Cela permet de bien identifier **l'appartenance d'une fonction** à une bibliothèque, par exemple. Pour la bibliothèque dart:math, nous pouvons utiliser l'alias Math :

```Dart
import 'dart:math' as Math;
```

Ensuite, nous pouvons **appeler ses fonctions** avec la syntaxe suivante:

```Dart
print(Math.min(1, 2));
```

Flutter

Les fichiers importants d'un projet Flutter

Lorsque nous créons un projet Flutter, il contient dans son arborescence, plusieurs **fichiers et dossiers importants**. Voici quelques-uns des fichiers clés à manipuler:

1. **lib/main.dart**: Ce fichier contient le point d'entrée de votre application Flutter. C'est ici que vous pouvez définir le widget racine de votre application et démarrer l'exécution.
2. **pubspec.yaml**: Ce fichier est utilisé pour déclarer les **dépendances** de votre application Flutter, telles que les packages externes. Vous pouvez ajouter, supprimer ou mettre à jour les dépendances de votre projet en modifiant ce fichier. Il contient également des **informations** sur le nom de l'application, la version, les ressources, etc.
3. **android/**: Ce dossier contient les fichiers spécifiques à la **plate-forme Android**. Vous pouvez y trouver des fichiers tels que AndroidManifest.xml pour la configuration de l'application Android, des fichiers de configuration Gradle, etc.
4. **ios/**: Ce dossier contient les fichiers spécifiques à la plate-forme iOS. Vous pouvez y trouver des fichiers tels que Info.plist pour la configuration de l'application iOS, des fichiers de configuration Xcode, etc.
5. **test/**: Ce dossier est utilisé pour **écrire des tests** automatisés pour votre application. Vous pouvez y trouver par exemple le fichier de test widget_test.dart pour tester les widgets.

Explorez ces fichiers et dossiers pour mieux comprendre la structure de votre projet Flutter. Ouvrez les fichiers et examinez leur contenu pour voir comment ils sont organisés et comment ils fonctionnent.

Créer une application vierge

Ce code représente une **application Flutter simple** qui affiche un écran avec le texte « Hello World » au centre:

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

void main() {
 runApp(
  MaterialApp(
   title: 'My App',
   home: Scaffold(
    appBar: AppBar(
     backgroundColor: Colors.blue,
     title: const Text('Hello World'),
    ),
    body: const Center(
     child: Text('Hello World'),
    ),
   ),
  ),
 );
}
```

Voici une explication détaillé du code :

Importation de packages

Le package flutter/material.dart est importé pour accéder aux widgets et aux fonctionnalités de Flutter.

Les fonctions main() et runApp()

La fonction main() est le **point d'entrée** de l'application. Elle est appelée lors du **démarrage** de l'application.
La fonction runApp() est utilisée pour **exécuter l'application** Flutter et passe MaterialApp comme widget racine de l'application.

Le widget MaterialApp

Le widget MaterialApp est utilisé pour configurer les **propriétés de l'application**:

- Le paramètre title définit le **titre** de l'application.
- Le paramètre home définit le **widget principal** de l'application.

Le widget Scaffold

Le widget Scaffold est utilisé pour définir la structure de base de la page, y compris la barre d'applications et le corps de la page.

Le widget AppBar est utilisé pour définir la barre d'applications:

1. Le paramètre backgroundColor définit la couleur de fond de la barre d'applications.
2. Le paramètre title définit le titre affiché dans la barre d'applications.

Le widget body est utilisé pour définir le contenu principal de la page.

1. Le widget Center est utilisé pour centrer le contenu à l'intérieur de la page.
2. Le widget Text est utilisé pour afficher le texte « Hello World ».

Lorsque vous exécutez cette application, elle affiche une **interface utilisateur simple** avec une barre d'applications bleue contenant le titre « Hello World » et le texte « Hello World » centré à l'écran.
Ce code vous donne un aperçu des concepts de base de Flutter, tels que les widgets, la hiérarchie des widgets et la mise en page. Vous pouvez le personnaliser en modifiant les couleurs, le texte et le contenu du widget body pour créer des applications plus complexes.

Prendre en main les StatelessWidgets

Un Stateless Widget (widget sans état) est une classe immuable qui hérite de la classe StatelessWidget de Flutter. Il est utilisé pour représenter des parties de l'interface utilisateur qui ne changent pas en fonction de l'état de l'application.

La syntaxe d'un StatelessWidget

Voici la syntaxe d'un Stateless Widget:

```
Dart
class NomDuWidget extends StatelessWidget {
 const NomDuWidget({Key key}) : super(key: key);

 @override
 Widget build(BuildContext context) {
  return WidgetPrincipal();
 }
}
```

Voici un exemple avec le widget HomePage qui renvoie un Scaffold:

```
Dart
class HomePage extends StatelessWidget {
 const HomePage({Key key}) : super(key: key);

 @override
```

```
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        backgroundColor: Colors.blue,
        title: const Text('Hello World'),
      ),
      body: const Center(
        child: Text('Hello World'),
      ),
    );
  }
}
```

- La **classe** HomePage hérite de StatelessWidget.
- Le **constructeur** HomePage est déclaré avec une clé optionnelle Key.
- La **méthode** build() est substituée pour retourner un widget principal Scaffold avec une barre d'applications contenant un texte « Hello World » et un corps centré avec le même texte.

Ajouter des paramètres

```
Dart
final String title;
```

Ici, on déclare une variable title de type String, le mot-clé final indique simplement que la variable est une **constante** en **lecture seule**.
On définit alors dans le **constructeur** pour la classe HomePage qui accepte un paramètre nommé title de type String.

```
Dart
const HomePage({Key? key, required this.title});
```

Le mot-clé required indique que ce paramètre est **obligatoire** lors de la création d'une instance de la classe HomePage. Le paramètre key est un paramètre facultatif de type Key hérité de la classe Widget.

Dans cet exemple, la méthode build retourne un widget Scaffold avec une barre d'applications (appBar) contenant le **titre passé en paramètre**.

```
Dart
class HomePage extends StatelessWidget {
  const HomePage({super.key, required this.title});

  final String title;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        backgroundColor: Colors.blue,
        title: Text(title),
      ),
      body: const Center(
        child: Text('Hello World'),
      ),
    );
  }
}
```

On doit également **transmettre ce paramètre** title depuis l'appel du widget HomePage:

```
Dart
class MyApp extends StatelessWidget {
```

```
 const MyApp({super.key});

 @override
 Widget build(BuildContext context) {
  return const MaterialApp(
    title: 'My App',
    debugShowCheckedModeBanner: false,
    home: HomePage(title: 'Hello World'),
  );
 }
}
```

Manipuler les StatefulWidgets

Voici l'exemple du code d'une application de comptage, que nous allons analyser :

```dart
class HomePage extends StatefulWidget {
 const HomePage({super.key, required this.title});

 final String title;

 @override
 State<HomePage> createState() => _HomePageState();
}

class _HomePageState extends State<HomePage> {
 int _counter = 0;

 incrementCounter() {
  setState(() {
   _counter++;
  });
 }

 @override
 Widget build(BuildContext context) {
  return Scaffold(
   appBar: AppBar(
    backgroundColor: Colors.blue,
    title: Text(widget.title),
   ),
   body: Center(
    child: Text('$_counter'),
   ),
   floatingActionButton: FloatingActionButton(
    onPressed: incrementCounter,
    child: const Icon(Icons.add),
   ),
  );
 }
}
```

```dart
@override
State<HomePage> createState() => _HomePageState();
```

La méthode createState est une méthode obligatoire pour tout widget de type StatefulWidget. Elle est responsable de la création de l'état associé au widget. Dans cet exemple, la méthode createState retourne une instance de la classe _HomePageState qui est l'état associé à HomePage.

```dart
class _HomePageState extends State<HomePage> {
```

La ligne ci-dessus déclare une classe _HomePageState qui hérite de State et est associée à HomePage.

```dart
int _counter = 0;
```

Cette ligne déclare une variable _counter de type int qui est l'état interne de HomePage.

```dart
incrementCounter() {
 setState(() {
  _counter++;
 });
}
```

La méthode incrementCounter est définie pour incrémenter la valeur de _counter à chaque fois qu'elle est appelée. La méthode setState est utilisée pour indiquer à Flutter que l'état a été modifié, ce qui déclenchera la reconstruction de l'interface utilisateur associée.

```dart
@override
Widget build(BuildContext context) {
 return Scaffold(
  appBar: AppBar(
   backgroundColor: Colors.blue,
   title: Text(widget.title),
  ),
  body: Center(
   child: Text('$_counter'),
  ),
  floatingActionButton: FloatingActionButton(
   onPressed: incrementCounter,
   child: const Icon(Icons.add),
  ),
 );
}
```

La méthode build est une méthode obligatoire pour tout widget de type State. Elle est appelée lorsqu'il est nécessaire de construire l'interface utilisateur du widget.

Dans cet exemple, la méthode build retourne un widget Scaffold avec une barre d'applications (appBar) contenant le titre passé en paramètre du widget HomePage. Le corps (body) du Scaffold contient un widget Center avec un texte qui affiche la valeur actuelle de _counter. Un bouton flottant (floatingActionButton) est également présent pour incrémenter _counter lorsqu'il est pressé.

Prise en main des packages Flutter

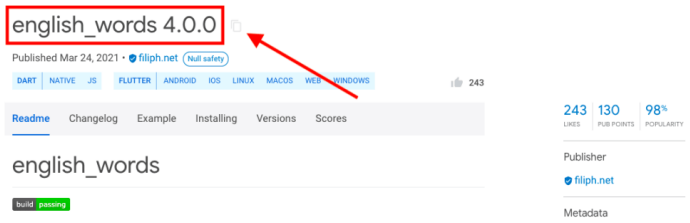
Les packages Flutter sont des **outils essentiels** pour le développement d'applications mobiles avec Flutter. Ils fournissent un accès facile à de nombreuses **fonctionnalités** et simplifient le processus de développement. Dans ce tutoriel, nous allons explorer les étapes pour installer et utiliser des packages Flutter dans votre application.

Installation d'un package Flutter

La première étape consiste à trouver et installer le package souhaité. Pour cela, vous pouvez visiter le site pub.dev, qui répertorie tous les packages Dart et Flutter disponibles.

Recherchez un package souhaité en utilisant la **barre de recherche** ou en explorant les catégories proposées. Une fois que vous avez trouvé le **package souhaité**, ouvrez sa page pour consulter les exemples d'utilisation et les étapes d'installation.

Voici un exemple d'important du package english_words :



Dans le fichier pubspec.yaml de votre projet Flutter, ajoutez la **dépendance du package** sous la section dependencies :

```yaml
dependencies:
  flutter:
    sdk: flutter
  cupertino_icons: ^1.0.5
  english_words: ^4.0.0 # version spécifique du package
```

Enregistrez le fichier pubspec.yaml et exécutez la **commande suivante** dans le terminal pour télécharger et installer les packages:

```shell
flutter pub get
```

Une fois que la commande est terminée, vous pouvez **importer le package** dans vos fichiers Dart :

```dart
import 'package:english_words/english_words.dart';
```

Utilisation d'un package Flutter

Maintenant que vous avez installé le package, vous pouvez **l'utiliser dans votre application** Flutter. Par exemple, le package english_words fournit la classe WordPair, qui permet d'associer **deux mots anglais aléatoires**. Vous pouvez utiliser la méthode random() pour obtenir une paire de mots aléatoire:

```dart
final wordPair = WordPair.random();
```

Autres packages Flutter

Voici quelques exemples de **packages populaires** :

- google_fonts: Permet d'utiliser facilement des **polices** de caractères personnalisées dans votre application.
- animate_do: Fournit des **animations** préconstruites pour ajouter du mouvement et de l'interactivité à votre interface utilisateur.
- http: Offre des fonctionnalités de **requêtes HTTP** pour interagir avec des API.

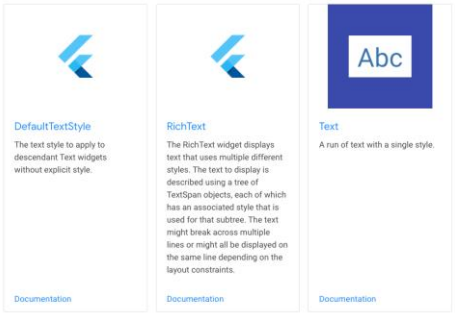
N'hésitez pas à consulter la **documentation** de chaque package pour en savoir plus sur leur utilisation spécifique.
Les packages Flutter sont des **ressources précieuses** pour les développeurs d'applications mobiles. Ils offrent une multitude de fonctionnalités prêtes à l'emploi et simplifient le développement.

Les différents widgets

On passe maintenant à la découverte plus approfondie des **widgets de base** de Flutter.
Il s'agira ici de construire de véritables applications Flutter, avec les fondements d'une **interface utilisateur**.

Le widget Text

Il existe trois widgets Text dans Flutter qui nous permettent d'affecter ou non les mêmes propriétés de style à tout son contenu.

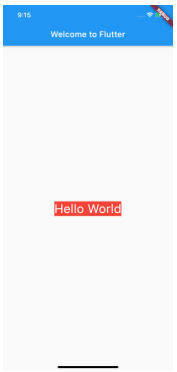


On utilisera en premier le **widget le plus simple** Text() qui nous permet déjà de créer un bon design.
On lui appliquera la **classe** TextStyle() pour lui modifier toutes ses propriétés visuelles.
Par exemple, pour augmenter la **taille de la police** de mon texte, je n'ai qu'à ajouter la classe TextStyle avec la propriété fontSize.
Je peux également lui ajouter un **fond** de couleur **rouge** et mettre le **texte** en **blanc**.

```Dart
Text(
  'Hello World',
  style: TextStyle(
    fontSize: 30,
    color: Colors.white,
    backgroundColor: Colors.red,
  ),
),
```

Vous retrouverez dans la documentation toutes les **propriétés** existantes pour le widget Text().

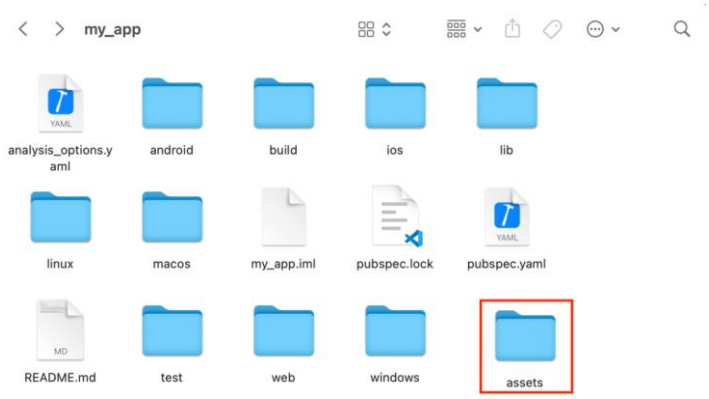
Voilà le résultat de mon **application de démarrage** avec ces propriétés appliquées à mon texte centré:



Accéder aux widgets Image

Ce widget nous permet d'afficher des **images ou logos** pour nos applications Flutter.

Pour commencer, il faut créer un dossier pour contenir toutes les images de votre application.
Vous pouvez l'appeler « assets » et y placer tout simplement le sous-dossier « images » dans le cas où il contiendra des images:



Une fois ces nouveaux dossiers créés, je vous invite à y **placer une photo** de votre choix.
Une fois que votre nouvelle image a été ajoutée, vous pouvez aller dans votre **fichier** pubspec.yaml, afin de **déclarer notre image** pour pouvoir l'utiliser.

Vous retrouverez **vers la fin de votre fichier** pubspec la **section** assets qui vous permettra d'indiquer les ressources de votre application. Vous pouvez ajouter la **référence de votre image** avec la syntaxe suivante:

```yaml
assets:
 - assets/images/bebe_yoda.jpeg
 - assets/images/
```

Une fois notre référence ajoutée, nous pouvons utiliser le widget Image avec le champ image et la classe AssetImage():

```dart
body: const Center(
  child: Image(
    image: AssetImage('assets/images/bebe_yoda.jpeg'),
  ),
),
```

C'est à cette classe AssetImage() que nous indiquons **l'adresse de notre image** dans notre application Flutter.

On peut également directement utiliser la **méthode** asset() pour indiquer l'adresse de l'image:

```dart
body: Center(
  child: Image.asset('images/bebe_yoda.jpeg'),
),
```

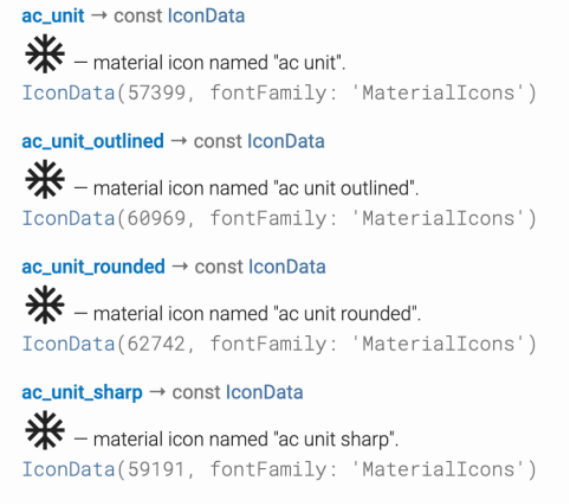
Il existe une variante de notre widget Image qui consiste à utiliser des photos issues **d'internet**.
On utilise pour cela la fonction network() pour indiquer **l'adresse web** de notre fichier image que l'on souhaite afficher.

Voici un exemple :

```dart
Image.network(
  'https://drissas.com/wp-content/uploads/2021/08/bebe_yoda.gif'),
```

Découvrez le widget Icon

Flutter propose un **widget dédié** pour afficher et **personnaliser** les icônes de la bibliothèque Material. Voici un lien pour accéder à tous les icônes disponibles : https://api.flutter.dev/flutter/material/Icons-class.html



Avant d'utiliser le widget Icon() de Flutter, vous devez vous assurer que la police des icônes Material est bien présente dans votre fichier pubspec.yaml:

```yaml
name: my_awesome_application
flutter:
  uses-material-design: true
```

Flutter propose désormais par défaut cette classe dans la section dédié de votre fichier pubspec.yaml.

Nous pouvons donc utiliser le widget Icon() dans notre application pour afficher une **première icône** très simple:

```dart
body: const Center(
  child: Icon(Icons.chat),
),
```

Ici, on précise le **nom** de notre icône avec la **constante** dédié, par exemple Icons.chat pour afficher le symbole correspondant:

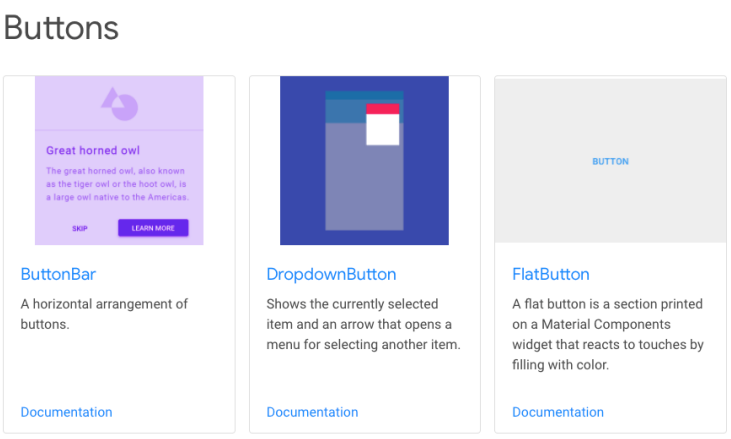
On peut modifier les icônes en, ajoutant par exemple, une **couleur** à notre icône avec le champ color et une **taille** plus grande avec le champ size:

```dart
body: const Center(
  child: Icon(
    Icons.favorite,
    color: Colors.pink,
    size: 100,
  ),
),
```

Exploiter tous les widgets Buttons

Les boutons sont souvent utilisés pour les **formulaires** d'envoi de données, mais également pour **valider** certaines actions.
Un bouton aura la capacité de faire **exécuter du code** une fois que l'utilisateur aura cliqué dessus.

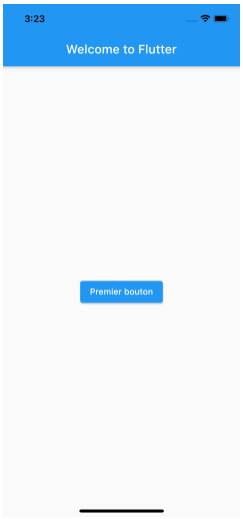
Il existe dans Flutter, différents types de boutons, voici les différentes **variantes possibles**: https://flutter.dev/docs/development/ui/widgets/material#Buttons



Nous allons utiliser dans cette partie essentiellement le classe de bouton ElevatedButton.

Ce widget va nous permettre de créer des boutons vraiment les **plus simples possibles**, avec toutes les propriétés de style.

Voilà par exemple le code de notre **premier bouton Flutter**, avec du **texte** et un message qui s'affiche dans la **console** lorsqu'on clique dessus:



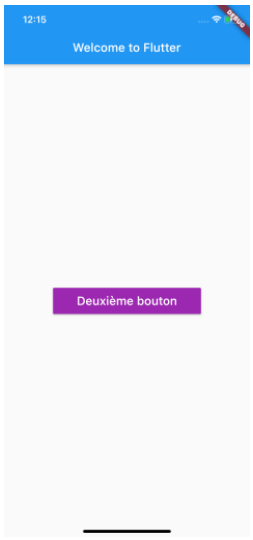

```Dart
ElevatedButton(
  onPressed: () {
    debugPrint('code de mon bouton');
  },
  child: const Text('Premier bouton'),
),
```

Il est évidemment possible d'ajouter des **propriétés de style** à notre bouton pour lui donner l'apparence souhaitée.
Nous pouvons utiliser par exemple des champs comme backgroundColor et textStyle pour colorer le fond et changer la taille du texte.
Avec le champ padding, je peux détailler les **marges intérieures** de mon bouton avec la méthode fromLTRB().
Et enfin, je peux modifier la **taille du texte** de mon bouton avec la classe TextStyle que nous avons abordé précédemment.

Voilà donc le style de notre **deuxième bouton** avec de la couleur et une taille plus grande:

```Dart
final ButtonStyle style = ElevatedButton.styleFrom(
  backgroundColor: Colors.purple,
  padding: const EdgeInsets.fromLTRB(40, 10, 40, 10),
  textStyle: const TextStyle(fontSize: 20),
);
```

Que je peux **appliquer** à mon ElevatedButton grâce à son champ style:




```Dart
ElevatedButton(
  style: style,
  child: const Text('Deuxième bouton'),
  onPressed: () {
    debugPrint('Code de mon bouton');
  },
),
```

Interface utilisateur

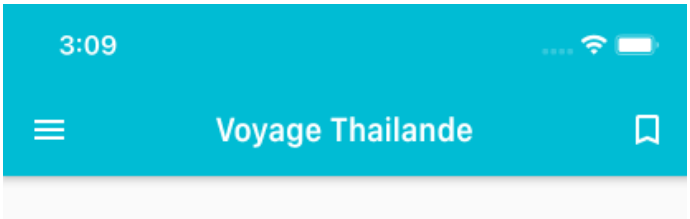
Introduction aux widgets de mise en page de Flutter (Column et Row)

Dans ce chapitre dédié à **l'interface utilisateur** avec Flutter, nous allons voir pas à pas la **création d'un design** d'application.

Nous allons commencer par créer une application sur le thème du tourisme en Thaïlande.
Je vous fournis également le code source de cette application, que vous pouvez télécharger ci-dessous:
Télécharger le code source

Le design de la AppBar

Dans cette partie, nous allons nous concentrer sur la conception de la **partie supérieure** de notre page Flutter, où nous souhaitons ajouter des **boutons**.



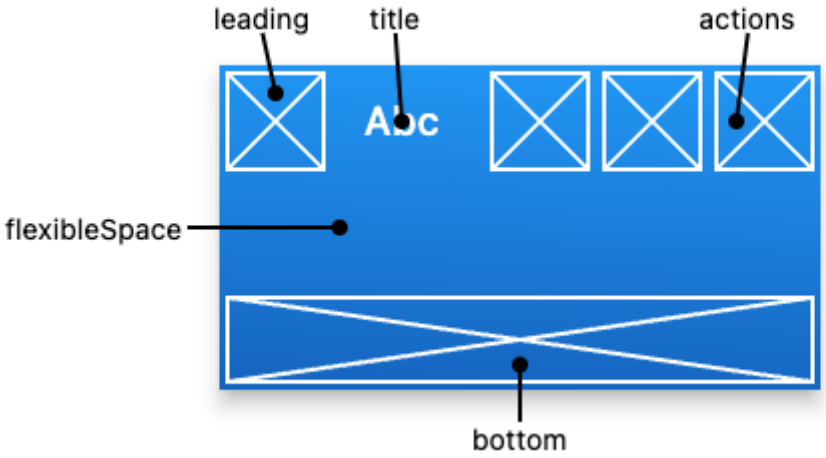
Pour commencer, vous pouvez utiliser le code Dart suivant pour afficher une application Flutter avec un titre simple:

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

void main() {
```

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

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

 @override
 Widget build(BuildContext context) {
  return const MaterialApp(
   title: 'Voyage',
   debugShowCheckedModeBanner: false,
   home: HomePage(),
  );
 }
}

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

 @override
 Widget build(BuildContext context) {
  return Scaffold(
   appBar: AppBar(
    backgroundColor: Colors.cyan,
    title: const Text('Voyage Thailande'),
   ),
  );
 }
}
```

Maintenant, nous allons ajouter **deux boutons** à notre AppBar en utilisant les champs distincts leading et actions.

Le schéma ci-dessous, tiré de la documentation Flutter, illustre parfaitement le **positionnement** des champs « leading » et « actions« :



En ce qui concerne le champ actions, vous avez la possibilité d'y ajouter **plusieurs boutons**. Commençons par le premier champ leading, qui est souvent associé au **bouton du menu**.

Voici le code pour notre **bouton de gauche** avec l'icône du menu, qui pour le moment n'effectue aucune action:

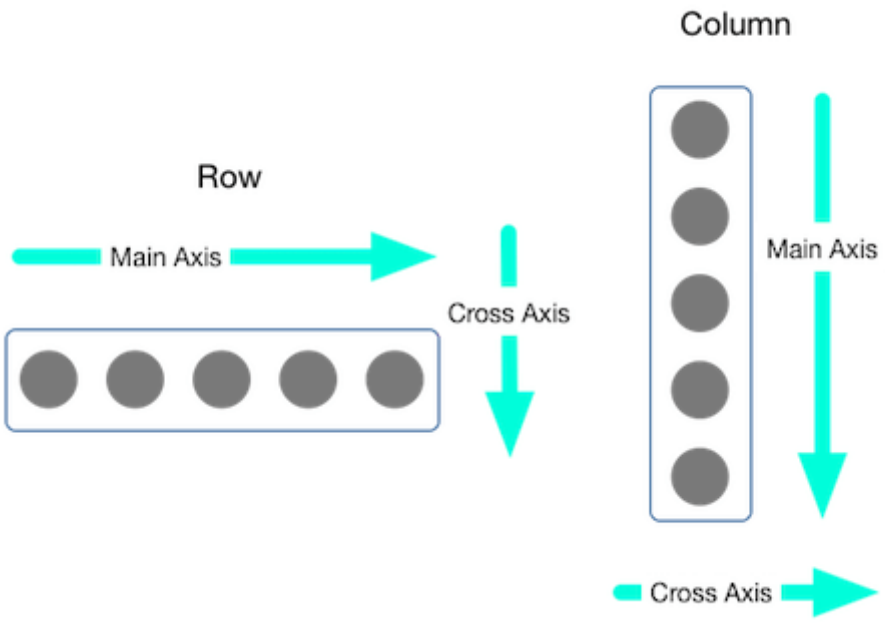
```Dart
leading: IconButton(
  icon: const Icon(Icons.menu),
  onPressed: () {},
),
```

Pour le champ actions, nous allons créer une liste de widgets IconButton. Cependant, dans cet exemple, nous n'aurons **qu'un seul bouton** dans cette liste.
Voici donc le **code Dart** pour le champ actions avec sa liste de widgets:

```Dart
actions: [
  IconButton(
    icon: const Icon(Icons.bookmark_border),
    onPressed: () {},
  ),
],
```

Les Widgets de structure (Row et Colum)

Nous poursuivons avec le contenu de notre page, où nous aborderons la notion de widget Row et Column.
Voici un **schéma** provenant de la **documentation Flutter**, les expliquant :



Une Row nous permet d'aligner du contenu **horizontalement**, tandis qu'une Column permet un alignement vertical.

Ces deux widgets sont largement utilisés et sont souvent utilisés en tandem pour créer des mises en page complexes.

Dans les deux cas, nous utilisons le champ children pour fournir plusieurs widgets à l'intérieur de notre Row() ou Column().

Ce sont ces widgets qui seront alignés **verticalement** ou **horizontalement** en fonction de leur parent.
Par exemple, je peux **aligner verticalement** différents widgets Text() et les centrer dans notre page :

```Dart
Column(
 mainAxisAlignment: MainAxisAlignment.center,
 children: const [
  Text('Hello 1'),
  Text('Hello 2'),
  Text('Hello 3'),
  Text('Hello 4'),
  Text('Hello 5'),
 ],
),
```

Voilà le **résultat** de nos différents widgets Text() centrés à la verticale:

Ajouter du contenu à notre page

Maintenant, nous passons à la création du contenu de notre **application touristique**.
Pour obtenir le résultat souhaité, nous utiliserons différentes **combinaisons** de Row et Column.
Pour la **partie supérieure** de notre page, nous utiliserons simplement le widget Image comme premier contenu.
Vous pouvez donc déclarer votre widget Image séparément et le stocker dans une **classe** appelée ImageSection:

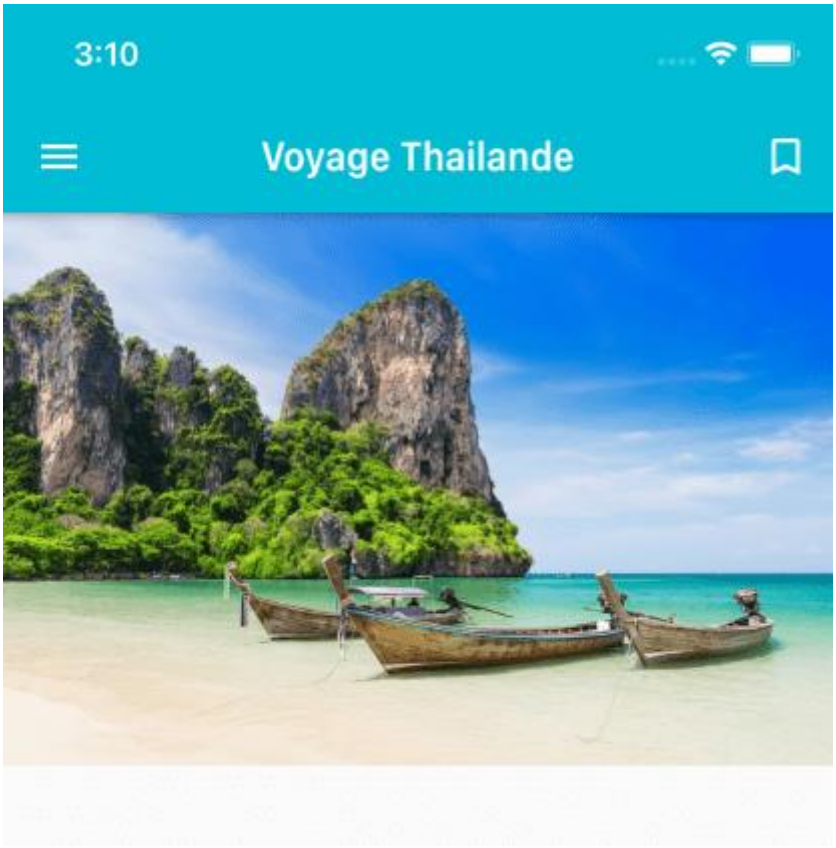
```Dart
class ImageSection extends StatelessWidget {
 const ImageSection({super.key});

 @override
 Widget build(BuildContext context) {
  return Image.network(
    'https://drissas.com/wp-content/uploads/2021/08/photo_thailande.jpeg');
 }
}
```

On créera un **widget** pour **chaque section** de notre page, pour faciliter la lecture de notre code par la suite.

Vous pouvez ensuite reprendre votre ImageSection et **l'ajouter** à votre widget Column dans le corps de votre page:

```
Dart
body: SingleChildScrollView(
  child: Column(
    children: const [
      ImageSection(),
    ],
  ),
),
```

Ici, tout le contenu de notre page est placé dans le widget SingleChildScrollView() pour pouvoir **scroller le contenu** qui dépassera.
Voilà le **résultat** pour l'instant de notre page Flutter:



On continue avec une section qui contiendra le **titre de notre page**, avec un **sous-titre** pour préciser les informations.
On utilisera pour cela le widget Text() avec quelques **propriétés de style**, par exemple pour notre **titre**:

```
Dart
Text(
  'Bienvenue au paradis',
  style: TextStyle(fontSize: 25, fontWeight: FontWeight.w800),
),
```

Et pour notre **sous-titre** ici:

```
Dart
Text(
  'Réservez votre séjour en Thailande',
  style: TextStyle(fontSize: 17, fontWeight: FontWeight.w500),
```

```
),
```

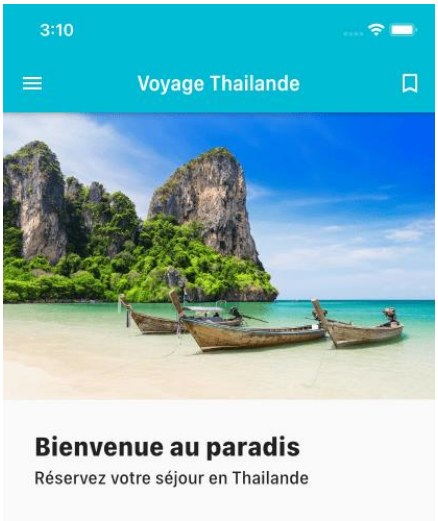
Avec ces deux widgets Text je vais pouvoir créer une section dédiée, appelé TitleSection avec une disposition en **colonne**:

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

 @override
 Widget build(BuildContext context) {
  return Container(
    width: double.infinity,
    padding: const EdgeInsets.all(20),
    child: Column(
     crossAxisAlignment: CrossAxisAlignment.start,
     children: const [
      Text(
       'Bienvenue au paradis',
       style: TextStyle(fontSize: 25, fontWeight: FontWeight.w800),
      ),
      Text(
       'Réservez votre séjour en Thailande',
       style: TextStyle(fontSize: 17, fontWeight: FontWeight.w500),
      ),
     ],
    ),
  );
 }
}
```

Vous pouvez donc ajouter vos **deux widgets** Text à en tant qu'enfants de votre widget Column.
N'oubliez pas d'ajouter ensuite la **nouvelle section** à votre page pour pouvoir visualiser son contenu:

```dart
body: SingleChildScrollView(
 child: Column(
  children: const [
   ImageSection(),
   TitleSection(),
  ],
 ),
),
```

Voilà le **résultat** de notre nouvelle section avec un affichage sur iOS:



On termine le début de notre page avec une **dernière section** de type **texte**.
Nous allons à nouveau utiliser le **widget** de type **texte**, mais cette fois pour afficher un **paragraphe** complet.
J'ai notamment récupéré une **petite introduction** sur la Thaïlande que je vous propose de reprendre dans cette nouvelle section:

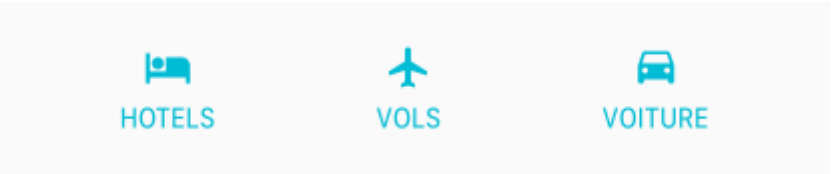
```dart
class TextSection extends StatelessWidget {
 const TextSection({super.key});

 @override
 Widget build(BuildContext context) {
  return Container(
    padding: const EdgeInsets.symmetric(horizontal: 20),
    child: const Text(
      "La Thaïlande, en forme longue le Royaume de Thaïlande, est un pays d'Asie du Sud-Est dont le territoire couvre 514 000 km2. Avant 1939, il s'appelait le Royaume de Siam. Il est bordé à l'ouest par la Birmanie, au sud par la Malaisie, à l'est par le Cambodge et au nord-est par le Laos. C'est une monarchie constitutionnelle depuis 1932. Sa capitale est Krung Thep, anciennement appelée Bangkok. La langue officielle est le thaï et la monnaie le baht."),
  );
 }
}
```

Ajoutez ensuite cette TextSection à votre page pour pouvoir **visualiser** le résultat suivant:

On en a donc fini avec le début de notre page, qui consistait à utiliser des **widgets de bases**.
On passe maintenant à une nouvelle partie où on utilise des **combinaisons plus avancées** de widgets.

Créer des combinaisons de widgets plus avancées

On commence cette nouvelle partie avec l'affichage de nos icônes Flutter sous forme de bouton.
Voilà le résultat que nous souhaitons obtenir entre nos icônes et le texte qui leur est associé:



Nous allons commencer par créer à nouveau une section dédiée à nos icônes, qui sera constituée pour commencer d'une Row:

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

 @override
 Widget build(BuildContext context) {
  return Container(
   padding: const EdgeInsets.only(bottom: 10),
   child: Row(
    mainAxisAlignment: MainAxisAlignment.spaceEvenly,
    children: [...],
   ),
  );
 }
}
```

Plutôt que de directement ajouter nos icônes Flutter en tant qu'enfants, nous allons créer des boîtes plus élaborées.
On crée pour chaque icône une boîte Container dédiée dans laquelle nous allons contenir une icône et du texte.
On utilise la disposition en colonne pour que le contenu s'aligne à la verticale:

```Dart
Container(
 padding: const EdgeInsets.all(20),
 child: Column(
  children: [
   const Icon(Icons.hotel, color: Colors.cyan),
   const SizedBox(height: 5),
   Text(
    'Hotels'.toUpperCase(),
    style: const TextStyle(color: Colors.cyan),
   )
  ],
 ),
),
```

On utilise également le nouveau widget SizedBox() pour créer une **espace** de séparation entre l'icône et son texte.
Ensuite, pour les widgets Text et Icon, on utilise les propriétés de style classique pour leur donner la couleur Cyan.
Voilà donc le **code final** de votre section consacrée aux icônes Flutter:

```Dart
class IconSection extends StatelessWidget {
```

```
const IconSection({super.key});

@override
Widget build(BuildContext context) {
 return Container(
  padding: const EdgeInsets.only(bottom: 10),
  child: Row(
   mainAxisAlignment: MainAxisAlignment.spaceEvenly,
   children: [
    Container(
     padding: const EdgeInsets.all(20),
     child: Column(
      children: [
       const Icon(Icons.hotel, color: Colors.cyan),
       const SizedBox(height: 5),
       Text(
        'Hotels'.toUpperCase(),
        style: const TextStyle(color: Colors.cyan),
       )
      ],
     ),
    ),
    Container(
     padding: const EdgeInsets.all(20),
     child: Column(
      children: [
       const Icon(Icons.airplanemode_active, color: Colors.cyan),
       const SizedBox(height: 5),
       Text(
        'Vols'.toUpperCase(),
        style: const TextStyle(color: Colors.cyan),
       )
      ],
     ),
    ),
    Container(
     padding: const EdgeInsets.all(20),
     child: Column(
      children: [
       const Icon(Icons.drive_eta, color: Colors.cyan),
       const SizedBox(height: 5),
       Text(
        'Voiture'.toUpperCase(),
        style: const TextStyle(color: Colors.cyan),
       )
      ],
     ),
    ),
   ],
  ),
 );
 }
}
```

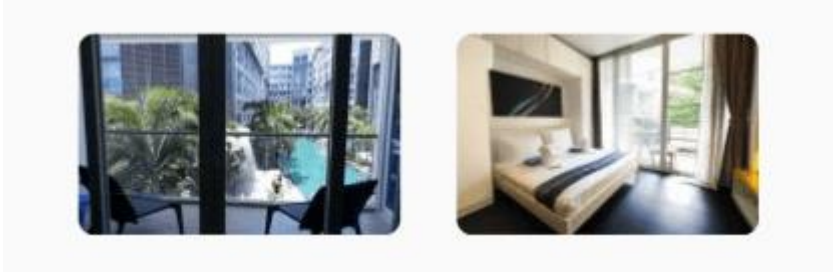
Cela représente aussi un code assez volumineux, mais gardez en tête que nous aurions pu déclarer **un seul widget** Container et indiquer différents **paramètres** à chaque fois.

Voilà le **résultat** de notre page actuellement avec cette **nouvelle section** d'icônes:

On passe maintenant au **contenu final** de notre page avec des images et un bouton classique.

Proposer une grille d'image aux angles arrondis

On termine donc avec la fin de notre page qui sera composé d'images et d'un bouton.
C'est l'occasion de vous montrer comment créer des images avec des bords arrondis:



Pour réaliser cette fonctionnalité, on utilisera tout simplement le widget ClipRRect qui permet de masquer le contenu qui dépasse.
En fait, il s'agit d'une boîte qui a la propriété de cacher (overflow: hidden) tous les widgets qui dépasseront de son champ.
En lui donnant des bords arrondis, l'image placée à l'intérieur donnera l'impression de les avoir également.
Voilà le code de notre widget ClipRRect avec à l'intérieur une image issue d'internet:

```Dart
ClipRRect(
  borderRadius: BorderRadius.circular(8),
  child: Image.network(
```

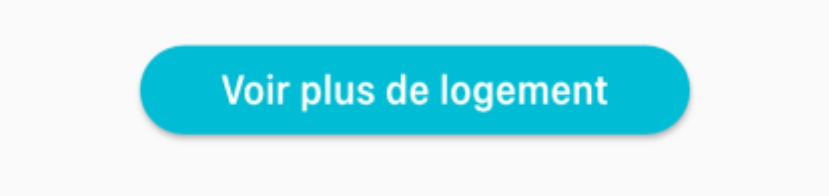
```
    'https://drissas.com/wp-content/uploads/2021/08/photo_thailande_1.jpeg'),
),
```

Et voilà le **code complet** de notre section dédiée aux images de logement:

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

 @override
 Widget build(BuildContext context) {
  return Container(
   padding: const EdgeInsets.symmetric(horizontal: 20),
   child: Row(
    mainAxisAlignment: MainAxisAlignment.spaceEvenly,
    children: [
     ClipRRect(
      borderRadius: BorderRadius.circular(8),
      child: Image.network(
       'https://drissas.com/wp-content/uploads/2021/08/photo_thailande_1.jpeg'),
     ),
     ClipRRect(
      borderRadius: BorderRadius.circular(8),
      child: Image.network(
       'https://drissas.com/wp-content/uploads/2021/08/photo_thailande_2.jpeg'),
     ),
    ],
   ),
  );
 }
}
```

L'affichage des images se fera sous forme de **grille**, elles prendront un peu moins de **50%** de l'espace.
Vous pouvez également gérer la taille de vos images en précisant les **marges** de vos widgets.
Voilà le **résultat** à ce stade:

Il ne nous reste plus qu'à ajouter un **bouton** à notre bas de page pour explorer potentiellement d'autres photos:



Pour cela, on utilise tout simplement le widget ElevatedButton pour créer un bouton arrondi avec simplement du texte:

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

 @override
 Widget build(BuildContext context) {
  return Container(
    margin: const EdgeInsets.symmetric(vertical: 30),
    child: ElevatedButton(
      style: ElevatedButton.styleFrom(
        backgroundColor: Colors.cyan,
        textStyle: const TextStyle(fontSize: 20),
        padding: const EdgeInsets.symmetric(horizontal: 40, vertical: 10),
        shape: const RoundedRectangleBorder(
          borderRadius: BorderRadius.all(Radius.circular(30)))),
      ),
      child: const Text('Voir plus de logements'),
      onPressed: () {},
    ),
  );
 }
}
```

Et voilà le **résultat** final de notre page Flutter:

Différents exemples de code

Dans la suite du cours, il y a d'autres exemples de comment créer différents designs. A travers ces exemples, cela permet d'aborder plusieurs concepts de l'interface utilisateur.

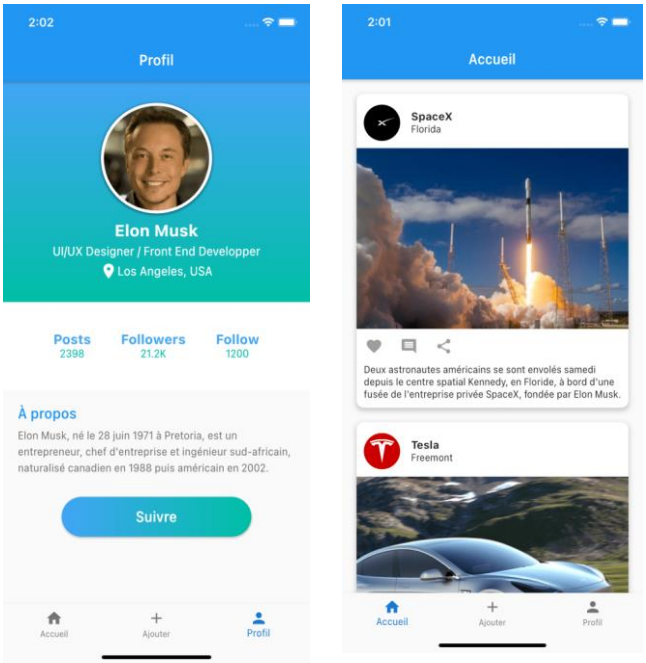
J'ai mis dans le dossier de la lecture individuelle, les différents codes, car je ne trouve pas pertinent de tout mettre les différentes explications dans cette lecture individuelle. En analysant le code, cela vous permettra de comprendre les différents possibilités en termes d'UI dans flutter.

Exemple d'un code de page de login avec navigation entre deux pages.

Voici à code ressemblera le résultat final de ce code :

Exemple d'une application à interface sociale avec une barre de navigation

Voici à code ressemblera le résultat final de ce code :



Créer des animations d'images lors de transition de page avec le Widget Hero

Dans ce chapitre nous allons aborder les animations sur Flutter. Plus précisément, nous allons voir comment prendre en main un type d'animation pendant la navigation deux pages.
Pour cela, nous allons utiliser les widgets Hero().

Ce code s'est basé sur un modèle **d'application de musique** pour mettre en scène cette animation:

Enregistrement
2023-11-15 225622.m

Ce genre d'animations Flutter fait vraiment la **différence** dans le rendu de nos applications mobiles. Cela permet vraiment de donner une **fluidité** et une expérience utilisateur au-delà des applications concurrentes.

Comment faire une animation de héros dans Flutter?

Voici la **documentation flutter** de l'animation hero :
https://flutter.dev/docs/development/ui/animations/hero-animations

Le but de ce code est de créer une simple image cliquable et centrée dans notre écran.
Lorsqu'on clique dessus, on ouvre une deuxième page qui affiche cette image en grand.
Pendant la navigation, l'image donne l'impression de se déplacer d'une page à l'autre et de changer de forme.

On peut donc commencer avec notre premier **code Dart**, pour simplement créer une **page vierge** dans notre application Flutter:

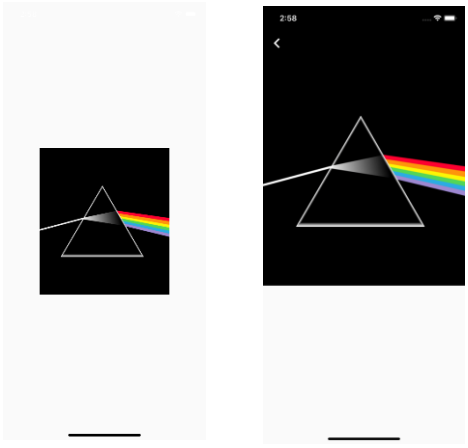
```dart
import 'package:flutter/material.dart';

void main() => runApp(const MyApp());

class MyApp extends StatelessWidget {
 const MyApp({Key? key}) : super(key: key);
 @override
 Widget build(BuildContext context) {
  return const MaterialApp(
   debugShowCheckedModeBanner: false,
   title: 'Hero Animation Flutter',
   home: FirstPage(),
  );
 }
}
```

Je crée une *StatelessWidget* pour créer le contenu de ma page avec à l'intérieur notre image centrée et cliquable.
En fait on utilise le widget GestureDetector() associé à un Container() pour gérer la navigation entre nos pages.
On ouvre à ce moment-là la deuxième page appelé pour le moment SecondPage() que nous allons déclarer ensuite.

Mais surtout on contient notre **image** dans un widget *Hero()* qui possède le champ *tag* qui permet de l'identifier:

```dart
class FirstPage extends StatelessWidget {
  const FirstPage({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: Container(
          height: 300,
          width: 300,
          alignment: Alignment.center,
          child: GestureDetector(
            onTap: () {
              Navigator.push(
                context,
                MaterialPageRoute(builder: (context) => const SecondPage()),
              );
            },
            child: Hero(
              tag: "album-image",
              child: Image.network(
                "https://img.ohmymag.com/article/musique/pochette-de-l-album-des-pink-floyd-dark-of-the-moon_52798d7562d1aa0907e1c57e5ed4216c397ba79e.jpg",
              ),
            ),
          ),
        ),
      ),
    );
  }
}
```

Nous devrons faire **référence** à ce **même tag** dans notre deuxième page pour reproduire ce **retour en arrière** dynamique.
Nous devons donc maintenant créer cette **deuxième page** pour afficher notre image en grand.



Pour cela nous créons également un *StatelessWidget()* pour notre *SecondPage()* qui contient uniquement notre image.

Mais à nouveau celle-ci est contenue dans un widget *Hero()* qui possède le même *tag* que la première page, ici '**album-image**':

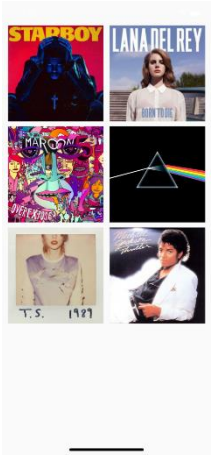
```Dart
class SecondPage extends StatelessWidget {
 const SecondPage({Key? key}) : super(key: key);
 @override
 Widget build(BuildContext context) {
  return Scaffold(
   appBar: AppBar(
    backgroundColor: Colors.black,
   ),
   body: Hero(
    tag: "album-image",
    child: Image.network(
     "https://img.ohmymag.com/article/musique/pochette-de-l-album-des-pink-floyd-dark-of-the-moon
_52798d7562d1aa0907e1c57e5ed4216c397ba79e.jpg",
    ),
   ),
  );
 }
}
```

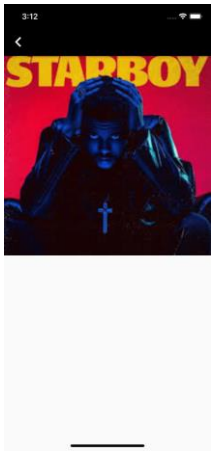
Grâce à ce widget *Hero()*, nous pouvons très facilement créer une animation entre nos pages.

Mais quand est-il lorsque nous gérons des **données dynamiques**, comme avec **Firebase** plus tard ? Nous allons prendre un exemple avec **plusieurs images**, mais des classes qui prennent en paramètres notamment le champ *tag*.

Comment gérer plusieurs animations en parallèle ?

On continue donc avec un **exemple** un peu **plus avancé** et **complémentaire** de notre fonctionnalité précédente.
En effet, pour l'instant on est capable de **créer une animation** pour une transition de page.
Mais qu'en est-il si le **contenu de nos pages** est **différent** ?
Notamment ici notre **deuxième page** comportera les informations de **l'album** sur lequel on vient de cliquer.
Voilà donc le rendu de notre nouveau code Dart avec **différentes images** et les **informations transmises** à la deuxième page:





Pour afficher ma **grille d'image**, j'utiliserai le **package** *responsive_grid* dont la documentation se trouve à ce lien: https://pub.dev/packages/responsive_grid

```
import 'package:responsive_grid/responsive_grid.dart';
```

Nous allons stocker le corps de notre page dans un Widget séparé, que nous allons appeler GridSection():

```
Dart
class FirstPage extends StatelessWidget {
 const FirstPage({Key? key}) : super(key: key);
 @override
 Widget build(BuildContext context) {
  return const Scaffold(
   body: Center(
    child: GridSection(), // nouvelle section
   ),
  );
 }
}
```

Nous allons créer une **grille de contenu** pour afficher nos couvertures d'album.
Pour le design des albums, j'ai créé un **widget séparé** que j'ai appelé *AlbumCover()* et qui prendra différents paramètres.
Notamment pour le design plus avancé nous lui demanderont de prendre en **paramètre** le **nom** de l'album, sa **photo**, **l'artiste** et surtout le **tag**.
Voilà donc pour le moment le code de ma section **grille**, qui affiche plusieurs widgets **AlbumCover** avec différentes **images** et **tags** en paramètres:

```
Dart
class GridSection extends StatelessWidget {
 const GridSection({Key? key}) : super(key: key);
 @override
 Widget build(BuildContext context) {
  return ResponsiveGridList(
   desiredItemWidth: 150,
   minSpacing: 10,
   children: const [
    AlbumCover(
     albumUrl:
      'https://encrypted-tbn0.gstatic.com/images?q=tbn%3AANd9GcRr4Q776RctXur78Z4NaMRrdh7-_2CZ7wDdBg&usqp=CAU',
     albumTag: 'Starboy',
    ),
    AlbumCover(
     albumUrl:
      'https://i.pinimg.com/236x/26/eb/e9/26ebe9788b358c734a2851048d05b12c--pop-albums-music-albums.jpg',
     albumTag: 'Born to die',
    ),
    AlbumCover(
     albumUrl:
      'https://i.pinimg.com/originals/6a/65/a1/6a65a167095e5f930b5569b276818213.jpg',
     albumTag: 'Overexposed',
    ),
    AlbumCover(
     albumUrl:
      'https://images-na.ssl-images-amazon.com/images/I/31tZr4Nr5vL._AC_SY450_.jpg',
     albumTag: 'Dark Side',
```

```
      ),
      AlbumCover(
        albumUrl:
            'https://static.billboard.com/files/media/Taylor-Swift-1989-album-covers-billboard-1000x1000-compressed.jpg',
        albumTag: '1989',
      ),
      AlbumCover(
        albumUrl:
            'https://img.huffingtonpost.com/asset/5badb5be200000e500ff1775.jpeg?ops=scalefit_630_noupscale',
        albumTag: 'Thriller',
      ),
    ],
  );
 }
}
```

Le code du widget *AlbumCover()* est un peu **nouveau** par rapport à tout ce que nous avons vu par le passé.
Il permet d'intégrer des **paramètres** à un **StatelessWidget** et leur **clé** associée.

Nous pouvons ainsi préciser dans le champ *Image.Network()* la variable *albumUrl* qui contient l'adresse de notre image.

On utilise donc à nouveau le widget *Hero()* avec également le champ tag qui prend cette fois-ci le paramètre *albumTag*.

```Dart
class AlbumCover extends StatelessWidget {
 final String albumUrl;
 final String albumTag;
 const AlbumCover({Key? key, required this.albumUrl, required this.albumTag})
    : super(key: key);
 @override
 Widget build(BuildContext context) {
  return GestureDetector(
    onTap: () {
      Navigator.push(
        context,
        MaterialPageRoute(
          builder: (context) => AlbumPage(
            albumUrl: albumUrl,
            albumTag: albumTag,
          ),
        ),
      );
    },
    child: Hero(
      tag: albumTag,
      child: Image.network(albumUrl),
    ),
  );
 }
}
```

Enfin on crée une **nouvelle page** que nous appelons **AlbumPage()** et qui contiendra les informations de notre album.

On utilise la même syntaxe pour prendre en **paramètre** les **informations de notre album**, pour le moment la photo et le tag.

```dart
class AlbumPage extends StatelessWidget {
 final String albumUrl;
 final String albumTag;
 const AlbumPage({Key? key, required this.albumUrl, required this.albumTag})
    : super(key: key);
 @override
 Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      backgroundColor: Colors.black,
    ),
    body: Hero(
      tag: albumTag,
      child: Container(
        width: double.infinity,
        height: 400.0,
        alignment: Alignment.topCenter,
        decoration: BoxDecoration(
          image: DecorationImage(
            image: NetworkImage(
              albumUrl,
            ),
            fit: BoxFit.cover),
        ),
      ),
    ),
  );
 }
}
```

Vous avez maintenant une **grille d'image** sur lesquelles vous pouvez cliquer pour l'ouvrir en grand. Cette fonctionnalité peut être **pratique** dans de nombreuses situations pour nos applications.

Créer des animations avec la classe Animator et d'autres packages utiles

Il existe plusieurs packages qui permettent de faire des animations sur Flutter, un des plus connus est Animator.
Nous allons dans ce chapitre, voir un exemple d'un cas d'utilisation.

La class Animator pour créer des animations simples

Comme d'habitude on aborde en premier lieu la **fonctionnalité** de la manière la plus **brute** possible.
On commencera donc par faire **varier** la **taille** ou **l'opacité** d'un widget Flutter:

Enregistrement 2023-11-16 000253.m

Enregistrement 2023-11-16 000323.m

On utilisera également le widget Animator() qui est accessible via le package suivant : https://pub.dev/packages/animator

C'est ce widget *Animator()* qui nous permettra de faire **varier** certaines **données** de notre contenu, comme sa taille ou son opacité.

La structure de base du widget *Animator()* est assez simple, elle propose le champ *tween* (*entre* en français) pour préciser deux valeurs limites.

Ensuite, il va faire **varier** cette valeur entre ces **deux extrêmes** et l'utiliser dans un autre widget pour déterminer sa taille par exemple:

```Dart
Animator<double>(
 tween: Tween<double>(begin: 50, end: 200), // nos deux valeurs extrèmes
 cycles: 0,
 builder: (context, animatorState, child) => Center(
  child: Container(
   height: animatorState.value, // le premier champ qui varie
   width: animatorState.value, // le deuxième champ qui varie
   child: FlutterLogo(),
  ),
 ),
)
```

On peut par exemple créer une **classe** pour contenir notre logo Flutter avec son animation, dans un *StatelessWidget* que nous déclarons:

```Dart
class AnimatedLogo extends StatelessWidget {
 const AnimatedLogo({Key? key}) : super(key: key);
 @override
 Widget build(BuildContext context) {
  return Animator<double>(
   tween: Tween<double>(begin: 50, end: 200),
   cycles: 0,
   duration: const Duration(seconds: 1),
   builder: (context, animatorState, child) => Center(
    child: SizedBox(
     height: animatorState.value,
     width: animatorState.value,
     child: const FlutterLogo(),
    ),
   ),
  );
 }
}
```

Ici on propose la valeur qui varie (animatorState.value) entre 50 et 200 dans les champs height et width de notre Container ou SizedBox.

En fait à chaque nouvelle valeur de notre donnée, le **widget** va être **reconstruit** avec ces nouvelles valeurs.

Le rendu dans notre application est que nous voyons notre **logo s'agrandir** puis revenir à sa taille initiale.

Le widget *Animator()* propose également le champ *cycles* pour préciser le **nombre de fois** où vous souhaitez que l'animation se produise. Avec la valeur **0**, l'animation se reproduit **sans arrêt** et la valeur varie entre le maximum et le minimum.

Voici un **deuxième exemple** avec la gestion de **l'opacité** :

On utilise pour cela le widget Opacity() pour contenir notre logo Flutter et faire varier son opacité:

```Dart
class AnimatedLogo extends StatelessWidget {
 const AnimatedLogo({Key? key}) : super(key: key);
 @override
 Widget build(BuildContext context) {
  return Animator<double>(
   tween: Tween<double>(begin: 0, end: 1000),
   cycles: 0,
   duration: const Duration(seconds: 1),
   builder: (context, animatorState, child) => Center(
    child: Opacity(
     opacity: animatorState.value / 1000,
     child: const FlutterLogo(
      size: 200,
     ),
    ),
   ),
  );
 }
}
```

Ici je fais varier le champ tween de **0** à **1000** pour que l'opacité varie plus lentement que directement entre 0 et 1.
Pour le champ *opacity*, je divise cette valeur par 1000 pour obtenir un nombre décimal compris entre **0 et 1**.
Vous pouvez également préciser la **durée de votre animation** avec le champ *duration*.

Animer plusieurs widgets en parallèle

Dans cette partie nous allons apprendre à gérer plusieurs animations en parallèle.
Dans cette exemple, on crée une rangée d'icônes qui viennent s'agrandir progressivement de la gauche vers la droite.

Enregistrement
2023-11-16 001012.m

On commence par créer un StatelessWidget pour contenir le code de nos icônes de manière générique.

Pour le moment ce Widget ne prend en **paramètre** que deux informations: le **nom de l'icône** et sa **couleur**.

On contient cette icône dans la classe *Animator()* qui propose une valeur qui varie entre **0** et **50** pour agrandir la taille d'un *Container()*:

```Dart
class IconLogo extends StatelessWidget {
 final IconData iconName;
 final Color iconColor;
 const IconLogo({Key? key, required this.iconName, required this.iconColor})
   : super(key: key);
 @override
```

```
 Widget build(BuildContext context) {
  return Animator<double>(
   tween: Tween<double>(begin: 0, end: 50),
   cycles: 1,
   duration: const Duration(seconds: 1),
   builder: (context, animatorState, child) => Center(
    child: Container(
     width: animatorState.value,
     height: animatorState.value,
     decoration: BoxDecoration(
      color: iconColor,
      borderRadius: BorderRadius.circular(50),
      boxShadow: [
       BoxShadow(
        color: Colors.white.withOpacity(0.1),
        spreadRadius: 3,
        blurRadius: 15,
        offset: const Offset(0, 3),
       ),
      ],
     ),
     child: Icon(
      iconName,
      color: Colors.white,
      size: animatorState.value / 2,
     ),
    ),
   ),
  );
 }
}
```

Je peux ainsi créer une **nouvelle section** pour contenir toutes mes icônes, que j'appelle *topIcons*.

Je contiens mes différents *IconLogo()* dans un widget *Row()* pour les contenir en rangée.
Je donne également à chaque widget *IconLogo()* ses deux paramètres uniques, à savoir le nom de **l'icône** et sa **couleur**:

```
Dart
Widget topIcons = Container(
 height: 130,
 padding: const EdgeInsets.all(30),
 margin: const EdgeInsets.fromLTRB(0, 0, 0, 20),
 child: Row(
  mainAxisAlignment: MainAxisAlignment.spaceBetween,
  children: [
   IconLogo(
     iconName: Icons.directions_walk,
     iconColor: Colors.tealAccent.shade700),
   const IconLogo(
     iconName: Icons.directions_run, iconColor: Colors.lightBlue),
   const IconLogo(iconName: Icons.directions_bike, iconColor: Colors.purple),
   const IconLogo(iconName: Icons.favorite, iconColor: Colors.pink),
   IconLogo(iconName: Icons.more_vert, iconColor: Colors.grey.shade800),
  ],
 ),
);
```

Vous pouvez alors appeler cette section depuis votre widget principal **MyApp**:

```Dart
class MyApp extends StatelessWidget {
 const MyApp({Key? key}) : super(key: key);
 @override
 Widget build(BuildContext context) {
  return MaterialApp(
   debugShowCheckedModeBanner: false,
   title: 'Flutter Animator',
   home: Scaffold(
    body: Center(
     child: topIcons,
    ),
   ),
  );
 }
}
```

De cette manière, vous devriez voir votre **rangée d'icônes s'agrandir** toutes en même temps.
Vous pouvez jouer avec la taille du *Container()* pour reproduire un effet différent.

Nous allons maintenant ajouter un **nouveau paramètre** pour améliorer encore à nouveau notre animation.

Il s'agit de la durée de chaque animation, que nous pouvons faire varier avec le paramètre duration en lui précisant des valeurs différentes.

J'ajoute donc le paramètre *animationDuration* (de type *Duration*) à mon widget IconLogo() et je l'intègre au champ *duration* de la classe *Animator()*:

```Dart
class IconLogo extends StatelessWidget {
 final IconData iconName;
 final Color iconColor;
 final Duration animationDuration;
 const IconLogo(
    {Key? key, required this.iconName, required this.iconColor, required this.animationDuration})
    : super(key: key);
 @override
 Widget build(BuildContext context) {
  return Animator<double>(
   tween: Tween<double>(begin: 0, end: 50),
   cycles: 1,
   duration: animationDuration,
   builder: (context, animatorState, child) => Center(
    child: Container(
     width: animatorState.value,
     height: animatorState.value,
     decoration: BoxDecoration(
      color: iconColor,
      borderRadius: BorderRadius.circular(50),
      boxShadow: [
       BoxShadow(
        color: Colors.white.withOpacity(0.1),
        spreadRadius: 3,
        blurRadius: 15,
        offset: const Offset(0, 3),
       ),
```

```
          ],
        ),
        child: Icon(
          iconName,
          color: Colors.white,
          size: animatorState.value / 2,
        ),
      ),
    ),
  );
 }
}
```

De cette manière, je peux reprendre ma section *topIcons* et ajouter à tous mes widgets *IconLogo()* le nouveau paramètre *animationDuration*.
Ici je vais travailler avec des **millisecondes** pour proposer une différence assez légère entre mes icônes.
On décale en fait chaque icône d'une durée de **100 millisecondes**, avec la syntaxe *Duration(milliseconds: 500)*:

```
Dart
Widget topIcons = Container(
 height: 130,
 padding: const EdgeInsets.all(30),
 margin: const EdgeInsets.fromLTRB(0, 0, 0, 20),
 child: Row(
  mainAxisAlignment: MainAxisAlignment.spaceBetween,
  children: [
   IconLogo(
    iconName: Icons.directions_walk,
    iconColor: Colors.tealAccent.shade700,
    animationDuration: const Duration(milliseconds: 500),
   ),
   const IconLogo(
    iconName: Icons.directions_run,
    iconColor: Colors.lightBlue,
    animationDuration: Duration(milliseconds: 600),
   ),
   const IconLogo(
    iconName: Icons.directions_bike,
    iconColor: Colors.purple,
    animationDuration: Duration(milliseconds: 700),
   ),
   const IconLogo(
    iconName: Icons.favorite,
    iconColor: Colors.pink,
    animationDuration: Duration(milliseconds: 800),
   ),
   IconLogo(
    iconName: Icons.more_vert,
    iconColor: Colors.grey.shade800,
    animationDuration: const Duration(milliseconds: 900),
   ),
  ],
 ),
);
```

Voilà donc pour ce deuxième exemple dans lequel nous vu comment **animer plusieurs Widgets** en parallèle avec la classe Animator().

Bibliographie

DRISS AS (sans date). Flutter révolution

Consulté le 5 novembre 2023, à l'adresse https://drissas.com/courses/flutter-revolution/