Flutter

Introduction à Flutter et Dart

Flutter ?

- **Flutter** est un framework pour le développement d’applications mobiles
  - multi-plateformes (code source unifié)
  - développé par Google
  - open-source
- https://flutter.dev

Multi-plateformes

- Android
- iOS (Mac nécessaire pour compiler)
- Desktop (Windows, macOS, Linux)
- Web
- Appareils embarqués

Dart ?

- **Dart** est le langage de programmation utilisé avec Flutter
  - développé par Google
  - famille des langages C (comme Java, C#, JavaScript…)
  - fortement typé
  - implémente la POO
  - implémente le *Null safety*
- Utilisé à la fois pour la logique métier et pour la description de l'UI
- Compile en code natif (ARM/x64) ou en JavaScript (Web)

DartPad

- Éditeur en ligne pour tester du code Dart
- https://dartpad.dev

```
void main() {
  print('Coucou !');
}
```

```
void main() {
  String name = 'Alice';
  print('Coucou ' + name + ' !'); // Concaténation
  print('Coucou $name !');          // Interpolation
}
```

`var` et `dynamic`

- `var` : inférence de type
  - permet de déclarer une variable sans préciser son type
  - le type est déterminé par Dart à la compilation
  - **cela reste un typage statique**
- `dynamic` : typage dynamique
  - le type est déterminé à l'exécution
  - **la variable peut changer de type**
- En Dart, le typage statique est beaucoup plus utilisé

```dart
void main() {
  var name = 'Alice';
  // name = 42; // Erreur de compilation
  dynamic age = 42;
  print('Coucou $name, tu as $age ans !');
  age = 'quarante-deux';
  print('Coucou $name, tu as $age ans !');
}
```

Déclaration de constantes

- `final` et `const` permettent de déclarer une constante
  - doit être initialisée à la déclaration
  - ne peut pas être modifiée
- Bonne pratique : toujours utiliser des constantes lorsque la valeur de la variable ne change pas
  - meilleures performances
  - pas de risque de modification non souhaitée (le compilateur nous en empêchera)

`final` - constante à l'exécution

- Avec `final`, la valeur est déterminée **au moment de l'exécution**, quand le programme arrive sur la ligne en question
  - le type peut être omis, comme pour `var`
  - `final int age = 42;`
  - `final age = 42;` (idem)

`const` - **constante à la compilation**

- Avec `const`, la valeur est déterminée **au moment de la compilation**, quand l'exécutable est construit
  - **le type ne peut pas être omis**
  - la valeur doit être un **littéral** (valeur « en dur » : `42`, `'toto'`…)
  - `const int age = 42;`
  - `const age = 42; // erreur de compilation`

```dart
void main() {
  final name = 'Alice';  // constante
  // ici 'Alice' étant un littéral,
  // on pourrait aussi utiliser const en indiquant le type :
  // const String name = 'Alice';
  print('Coucou $name !');
  name = 'Bob'; // erreur de compilation (constante !)
}
```

Listes

- Dart supporte les listes génériques
- Syntaxe : `List<Type> variable = [valeur1, valeur2, …];`

```dart
void main() {
  List<String> names = ['Alice', 'Bob', 'Eve'];
  names.add('Dave'); // 4 éléments dans la liste
  names.remove('Dave');
  names.removeAt(1); // supprime Bob (indices commencent à 0)
  print(names);      // [Alice, Eve, Dave]
  // Parcours de la liste
  for (int i = 0; i < names.length; i++) {
    print(names[i]);
  }
}
```

Parcours de liste amélioré

```
void main() {
  List<String> names = ['Alice', 'Bob', 'Eve'];
  // Syntaxe for spécifique pour les listes
  for (String name in names) {
    print(name);
  }
}
```

Fonctions

- La syntaxe est similaire à celle de Java
  - `TypeDeRetour nomFonction(Type1 parametre1, Type2 parametre2) { … }`

```dart
void main() {

  List<String> names = ['Alice', 'Bob', 'Eve'];
  if (containsName(names, 'Bob')) {
    print('Bob est dans la liste');
  } else {
    print('Bob n\'est pas dans la liste');
  }

  bool containsName(String name) {
    return names.contains(name);
  }
}
```

Arrow Functions

- Syntaxe plus concise pour les fonctions à une seule expression
- Utilisation d'une *fat arrow* ⇒, suppression du `return` et des accolades

```dart
void main() {

  List<String> names = ['Alice', 'Bob', 'Eve'];
  if (containsName(names, 'Bob')) {
    print('Bob est dans la liste');
  } else {
    print('Bob n\'est pas dans la liste');
  }

  // Équivalent à la fonction précédente
  bool containsName(String name) => names.contains(name);
}
```

Classes et Objets

- Dart est un langage orienté objet
- Syntaxe similaire à Java
  - `class NomClasse { … }`
  - particularité : `new` n'est pas obligatoire pour instancier un objet (en Flutter son usage est même *deprecated*)

```
void main() {
  IceCream vanilla = IceCream();
  vanilla.flavor = 'Vanilla';
  vanilla.isFruit = true;
  print(vanilla.flavor);
}

class IceCream {
  String flavor = '';
  bool isFruit = false;
}
```

Constructeurs

- La déclaration des constructeurs est similaire à Java
  - Dart supporte également les constructeurs par défaut (exemple précédent)
- Syntaxe : `NomClasse(param1, param2) { … }`

```
void main() {
  IceCream chocolate = IceCream('Chocolate', false);
  print(chocolate.flavor);
}

class IceCream {
  String flavor = '';
  bool isFruit = false;

  IceCream(String flavor, bool isFruit) {
    this.flavor = flavor;
    this.isFruit = isFruit;
  }
}
```

Syntaxe raccourcie

- Dart propose une syntaxe raccourcie pour les constructeurs
  - en spécifiant le mot-clé `this` dans les paramètres du constructeur, il devient inutile de les assigner dans le corps du constructeur

```
void main() {
  IceCream vanilla = IceCream('Vanille', true);
  print(vanilla.flavor);
}

class IceCream {
  String flavor = '';
  bool isFruit = false;

  IceCream(this.flavor, this.isFruit);
}
```

Constructeurs nommés

- Dart ne supporte pas la surcharge de constructeurs
  - ⇒ un seul constructeur a le droit de prendre le nom de la classe
- Pour contourner cela, Dart propose les constructeurs nommés
  - Syntaxe : `NomClasse.nomConstructeur(param1, param2) { … }`

```dart
void main() {
  IceCream vanilla = IceCream('Vanille', true);
  IceCream chocolate = IceCream.withFlavor('Chocolat');
  IceCream strawberry = IceCream.fruitWithFlavor('Fraise');
  print(vanilla);
}

class IceCream {
  String flavor = '';
  bool isFruit = false;

  // Constructeur non nommé (un seul)
  IceCream(this.flavor, this.isFruit);
  // Constructeur nommé
  IceCream.withFlavor(this.flavor);
  // Autre constructeur nommé
  IceCream.fruitWithFlavor(this.flavor) {
    isFruit = true;
  }
}
```

Héritage

- Dart supporte l’héritage simple
- Syntaxe : `class NomClasse extends ClasseParente { … }`
  - le mot-clé `super` est ensuite utilisé pour appeler le constructeur de la classe parente

```
void main() {
  Cone cone = Cone('Chocolat', false, true);
}

class IceCream {
  String flavor = '';
  bool isFruit = false;

  IceCream(this.flavor, this.isFruit);
}

class Cone extends IceCream {
  bool isGlutenFree = false;

  Cone(String flavor, bool isFruit, this.isGlutenFree)
      : super(flavor, isFruit);
}
```

Dart et la POO en bref

- `new` n'est pas nécessaire pour instancier un objet
- Un seul constructeur non nommé maximum
  - syntaxe `this.` dans les paramètres pour initialiser automatiquement les champs
- Autres constructeurs doivent être nommés
  - syntaxe `NomClasse.nomConstructeur(param1, param2) { }`
  - le pattern de nommage `with...` est ici souvent utilisé
- Héritage simple avec le mot-clé `extends`
  - mot-clé `super` pour désigner la classe parente

Null safety

- En POO, les variables `null` sont à éviter au maximum
- Dart supporte plusieurs constructions pour implémenter la *Null safety* (sécurité contre les valeurs `null`)
  - types *nullables*, opérateurs `"??"`, `"?."`, `"??="`, opérateur *Bang* `"!"`
- Le *Null safety* aide à éviter les erreurs de type `null`
  - protection à la compilation
  - et aussi à l'exécution

```
class IceCream {
  // erreurs de compilation : champs 'null' interdits par défaut
  String flavor;
  bool isFruit;
}
```

Null safety : types *nullables*

- Solutions :
  - donner un valeur par défaut, comme auparavant : `String flavor = '';`
  - utiliser le constructeur non nommé avec la syntaxe `this.` en paramètre
  - **utiliser un type *nullable*** : `String? flavor;`
- Un type *nullable* est un type dont les variables peuvent être `null`
  - en Dart, il faut donc spécifier explicitement qu'on autorise une variable à être `null`

```
class IceCream {
  String? flavor;
  bool? isFruit;
}
```

Null safety : pas de conversion implicite

```
main() {
  IceCream iceCream = IceCream();
  iceCream.flavor = 'Vanilla';
  String flavor = iceCream.flavor; // erreur de compilation
}

class IceCream {
  String? flavor;
  bool? isFruit;
}
```

Opérateur *Null coalescing*

- Le *Null coalescing* est un opérateur qui permet de retourner une valeur par défaut si une valeur est `null`
- Syntaxe : `valeur1 ?? valeur2`
  - si `valeur1` est `null`, alors `valeur2` est retournée
  - sinon, `valeur1` est retournée

```dart
main() {
  IceCream iceCream = IceCream();
  iceCream.flavor = 'Vanilla';
  // La valeur 'Inconnue' sera utilisée si flavor est null
  String flavor = iceCream.flavor ?? 'Inconnue';
}

class IceCream {
  String? flavor;
  bool? isFruit;
}
```

Opérateur *Null assertion* ou opérateur *Bang* !

- L'opérateur *Bang* `!` permet de forcer l'accès à une variable qui peut être `null`
  - on l'utilise seulement quand on est certain que la variable n'est pas `null`
- Syntaxe : `variable!`
  - si `variable` est `null`, une erreur est levée
  - sinon, la variable est retournée

```dart
main() {
  IceCream iceCream = IceCream();
  iceCream.flavor = 'Vanilla';
  // Ici le dev sait que flavor n'est pas null
  // Il force donc l'accès
  String flavor = iceCream.flavor!;
}

class IceCream {
  String? flavor;
  bool? isFruit;
}
```

Accès à un champ *nullable*

```
main() {
  IceCream iceCream = IceCream();
  String? flavor = iceCream.flavor; // OK
  print(flavor.length); // Erreur de compilation
}

class IceCream {
  String? flavor;
  bool? isFruit;
}
```

Opérateur *Null-aware*

- L'opérateur *Null-aware* `?.` permet d'accéder à un champ sur un objet qui peut être `null`
- Syntaxe : `objet?.champ`
  - si `objet` est `null`, alors `null` est retourné
  - sinon, la valeur de `champ` est utilisée
- On peut aussi l'utiliser pour un appel de méthode : `objet?.methode()`

```dart
main() {
  IceCream iceCream = IceCream();
  iceCream.flavor = 'Chocolat';
  String? flavor = iceCream.flavor;
  // Utilisation de ?.
  // OK, mais attention : la valeur peut être nulle
  print(flavor?.length);
}

class IceCream {
  String? flavor;
  bool? isFruit;
}
```

Opérateur !.

- Même idée que l'opérateur *Bang* pour accéder à un champ *nullable*
  - ⇒ si vous êtes sûr que le champ n'est pas `null`, vous pouvez utiliser `!.` pour avoir une erreur si c'est le cas
- Syntaxe : `objet!.champ`
  - si `objet` est `null`, une erreur est levée
  - sinon, la valeur de `champ` est utilisée

```
main() {
  IceCream iceCream = IceCream();
  iceCream.flavor = 'Chocolat';
  String? flavor = iceCream.flavor;
  // Utilisation de !.
  // OK, mais si la valeur est nulle => erreur d'exécution
  print(flavor!.length);
}

class IceCream {
  String? flavor;
  bool? isFruit;
}
```

Opérateur d'affectation *Null coalescing*

- L'opérateur d'affectation *Null coalescing* `??=` permet d'assigner une valeur par défaut à une variable si elle est `null`
- Syntaxe : `variable ??= valeur`
  - si `variable` est `null`, alors `valeur` est assignée à `variable`
  - sinon, `variable` reste inchangée

```dart
main() {
  IceCream iceCream = IceCream();
  iceCream.flavor = 'Chocolat';
  // Si flavor est null, on lui affecte 'Vanilla'
  iceCream.flavor ??= 'Vanilla';
  print(iceCream.flavor);
}

class IceCream {
  String? flavor;
  bool? isFruit;
}
```

Combinaison des opérateurs

- Tous ces opérateurs sont souvent combinés pour garantir le *Null safety* tout en gardant une syntaxe concise :
  - `objet?.champ ?? valeurParDefaut`
  - `objet?.champ!.methode() ?? valeurParDefaut`
  - `objet?.champ ??= valeurParDefaut`

Flutter

Installation locale de Flutter

Dépendances

- *Git*
- *Visual Studio Code* (en ce qui nous concerne)
- *Flutter SDK* (*Software Development Kit*)
- Navigateur *Chrome* ou *Edge* pour cibler le web
- *Android Studio* pour cibler Android
- Sur Windows : *Visual Studio* pour cibler Windows Desktop
- Sur macOS : *Xcode* pour cibler iOS et macOS Desktop
- Sur Linux : *GCC* et *Make* pour cibler Linux Desktop

Installation de Flutter SDK

- docs.flutter.dev/get-started/install
  - procédure officielle
- La procédure détaillée ci-après utilise plutôt *VS Code* pour installer *Flutter SDK* sous Windows
  - sous un autre système ⇒ procédure officielle
  - le *Flutter SDK* peut toujours être installé depuis *VS Code*
  - en cas de problème sous *VS Code* ⇒ procédure officielle

Sous VS Code

- Installer l'extension *Flutter* (*Dart Code*)
  - Support de Flutter/Dart sous *VS Code*
- *Palette de commandes* > *flutter* > *New Project*
  - propose l'installation de *Flutter SDK*
  - indiquer le chemin local d'installation (choisir un chemin non synchronisé avec un *cloud storage*)
  - après l'installation, cliquer sur «*Add SDK to PATH*»
  - choisir le template de projet *Application*
  - choisir l'emplacement du projet
  - choisir le nom du projet (minuscules, _ autorisé)

Choix de la cible et test de lancement

- Cliquer sur le outon *No Devices* en bas à droite de VS Code
- Sélectionner *Chrome* ou *Edge* pour cibler le web
- Cliquer sur le bouton *Start Debugging* (flèche en haut à droite)
  - au premier lancement, le *SDK Web* va être téléchargé avant de lancer l'application
- Le navigateur devrait atteindre *localhost* sur un port prédéfini
- L'application consiste en un simple compteur incrémentable à l'aide d'un bouton +

Android Studio

- Pour cibler Android, il faut installer *Android Studio*
  - même si on développe sous *VS Code*, *Android Studio* est nécessaire pour cibler Android
  - *Android Studio* peut aussi être utilisé pour développer des applications Android natives ou des applications Flutter
  - nous utiliserons *VS Code* (plus léger)

Installation d’Android Studio

- developer.android.com/studio
  - au téléchargement, ignorer la création du profil
- Laisser les options par défaut durant l’installation
  - sur la page *License Agreement*, accepter explicitement toutes les demandes

Test de l’émulateur Android

- Nous allons utiliser *Android Studio* que pour configurer un émulateur Android que l’on pourra ensuite utiliser sous *VS Code*
- Créer un nouveau projet vide
- Une fois le projet créé, cliquer sur l’icône *Device Manager* (icône de smartphone à droite)
  - il devrait déjà y avoir un émulateur configuré par défaut
  - sinon, installer un émulateur (préférez un appareil *Pixel* pour le dev)
  - tester l’émulateur (bouton *Play*)

Configuration de l’émulateur

- Dans un terminal, lancer `flutter doctor`
  - permet de vérifier l’installation de Flutter et de ses dépendances
- À ce stade, tout devrait être vert sauf *Android toolchain* et le développement *Windows Apps*
- Installation des outils *CLI Android* :
  - Android Studio : *Settings* > *SDK Manager* > *SDK Tools*
  - cocher *Android SDK Command-line Tools*
- Acceptation des licences Android :
  - terminal : `flutter doctor --android-licenses`
  - accepter toutes les licences
- Un dernier `flutter doctor` ne devrait laisser que *Windows Apps* en rouge

Visual Studio

- Cette étape est **facultative**
- Pour cibler Windows Desktop, il faut installer *Visual Studio*
  - produit Microsoft différent de *Visual Studio Code*
  - IDE riche et complet pour le développement Windows
  - axé principalement sur le développement .NET (framework Microsoft)
- visualstudio.microsoft.com/fr/vs/community
- Pour cibler *Windows Desktop* avec Flutter, lors du choix des modules à installer, **il faut sélectionner le module *Desktop development with C++***
  - aucun autre module n'est nécessaire pour Flutter
- `flutter doctor` pour vérifier l'installation

Problèmes potentiels et solutions

- Problème à l'installation de Flutter ?
  - docs.flutter.dev/get-started/install/help
- Problèmes avec *VS Code* ?
  - code.visualstudio.com/docs/setup/windows
  - code.visualstudio.com/docs/setup/mac
  - code.visualstudio.com/docs/setup/linux
- Problèmes avec *Android Studio* et l'émulateur ?
  - developer.android.com/studio/troubleshoot
- Problèmes avec *Visual Studio* ?
  - learn.microsoft.com/fr-fr/troubleshoot/developer/visualstudio/installation/troubleshoot-installation-issues

Flutter

Développement Mobile

Créer un projet Flutter

- Palette de commandes : *Flutter: New Project > Empty Application*
  - un nouveau répertoire va être créé à l'endroit spécifié
- On peut aussi utiliser Flutter CLI dans le répertoire souhaité
  - `flutter create -e mon_projet`

Structure du projet

- Répertoire *lib* : code source
  - *main.dart* : point d'entrée de l'app
- *pubspec.yaml* : configuration et dépendances
- *android*, *ios*, *windows* : code spécifique à chaque plateforme si besoin
- *web* : code spécifique à la cible web

Widgets

- En Flutter, tout ce qu'on voit à l'écran est un *widget*
  - *Layouts* : *Grid*, *List*, *Row*, *Column*…
  - *Objects* : *Text*, *Image*, *Icon*, *Button*…
  - l'app elle-même (`MaterialApp`) est un *widget*
- Les *widgets* sont :
  - fournis par le framework
  - ou créés par le développeur à partir de *widgets* existants
  - tout écran que vous créerez sera un *widget*

Widgets sans état

StatelessWidget

- Un *StatelessWidget* est un *widget* qui ne peut pas être modifié
- Notre page d'accueil sera un *StatelessWidget*
- Snippet *stless* dans VS Code pour créer rapidement un *StatelessWidget*

Page d’accueil

- Créer fichier *lib/screens/welcome.dart*

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

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

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

Adaptation `main.dart`

- `MaterialApp` est un widget qui agit comme conteneur pour l'app
  - permet de configurer l'app (localisation, navigation, thème, etc.)
- *home* est un paramètre obligatoire de `MaterialApp`
  - doit être un *widget*
  - désigne la route par défaut (page d'accueil)
- On va indiquer le chemin vers notre écran d'accueil
  - il faut importer le fichier welcome_screen.dart_ (avec chemin relatif)
  - et instancier le *widget* `WelcomeScreen` dans *home*

```dart
// main.dart
import 'package:flutter/material.dart';
import 'package:flutter_app/screens/welcome_screen.dart';

void main() {
  runApp(const MainApp());
}

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

  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
      home: WelcomeScreen()
    );
  }
}
```

Widget Scaffold

- *Scaffold* est un *widget* qui implémente la structure de base d'une app
  - *AppBar*, *Drawer*, *BottomNavigationBar*, *FloatingActionButton*...
- *Scaffold* est souvent utilisé comme racine de l'app

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

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

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Accueil')),
      body: const Center(
        child: Text(
          'Du contenu...',
          style: TextStyle(fontSize: 24),
        ),
      ),
    );
  }
}
```

DevTools Widget Inspector

- *Widget Inspector* est un outil de débogage puissant
  - permet de visualiser la hiérarchie des *widgets* d'une application Flutter
  - il faut s'habituer à l'utiliser, notamment quand on ne comprend pas pourquoi un écran plus complexe ne s'affiche pas comme souhaité
- Pour l'activer sous VS Code :
  - `F1 > Flutter: Open DevTools Widget Inspector`
  - ou clic sur l'icône *Open DevTools* dans la barre d'exécution une fois l'application lancée

Les *widgets* de layout

- Les *widgets* de layout permettent de structurer l'interface
- Ils contiennent d'autres *widgets* et définissent comment ils sont affichés
  - par exemple, *Center* centre son contenu horizontalement et verticalement
- Certains possèdent un seul enfant (*child*)
  - ex. : *Center*, *Container*, *Column*, *Row*
- D'autres peuvent en avoir plusieurs (*children*)
  - ex. : *Stack*, *ListView*, *GridView*

Widget - Stack

- *Stack* est un *widget* qui empile ses enfants les uns sur les autres
  - comme un *z-index* en CSS, on empile les *widgets* du fond vers le premier plan
  - les *widgets* de la *Stack* sont spécifiés dans une liste donnée au paramètre *children*
- *Positioned* permet de positionner les *widgets* dans le *Stack*
  - *top*, *bottom*, *left*, *right* pour positionner
  - *width*, *height* pour dimensionner

Implémentation d’une *Stack*

- On va utiliser une *Stack* pour superposer un texte et un bouton sur une image de fond

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

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

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Accueil')),
      body: const Stack(
        children: [
          Center(
            child: Text(
              'Du contenu...',
              style: TextStyle(fontSize: 24),
            ),
          )
        ],
      ),
    );
  }
}
```

Ajout d’une image en fond de Stack

- Plusieurs façons d’inclure des images :
  - *Image.network* : depuis le web
  - *Image.asset* : depuis les ressources du projet
  - *Image.memory* : depuis la mémoire
  - *Image.file* : depuis un fichier local
- On va charger notre image depuis un répertoire dédié du projet
  - on crée un répertoire *assets/images* à la racine du projet

`pubspec.yaml`

- *pubspec.yaml* est utilisé pour déclarer les dépendances et les ressources du projet
- *pubspec* est un fichier de configuration YAML
  - YAML est un format de sérialisation de données (comme JSON)
  - mais plutôt utilisé pour la configuration
  - l'indentation est impoortante : elle spécifie la hiérarchie
  - puis on a des paires `clé: valeur` pour spécifier la configuration

Configuration du chemin des images

- Les resources locales sont spécifiées dans la section *flutter* de *pubspec.yaml*
  - on ajoute le chemin de nos images

```
// pubspec.yaml
name: ...

...

flutter:
  uses-material-design: true

  assets:
    - assets/images/
```

Ajout de l’image sur l’écran d’accueil

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

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

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Accueil')),
      body: const Stack(
        children: [
          Positioned.fill(
            child: Image.asset(
              'assets/images/sea.jpg',
              fit: BoxFit.cover,
            )
          ),
          Center(
            child: Text(
              'Du contenu...',
              style: TextStyle(fontSize: 24),
            ),
          )
        ],
      ),
    );
  }
}
```

Les boutons

- On va ajouter un bouton à la *Stack*
- Flutter propose plusieurs types de boutons, dont :
  - *TextButton* : bouton simple avec texte
  - *IconButton* : bouton avec icône
  - *OutlinedButton* : bouton avec contour
  - *ElevatedButton* : bouton « rempli »;

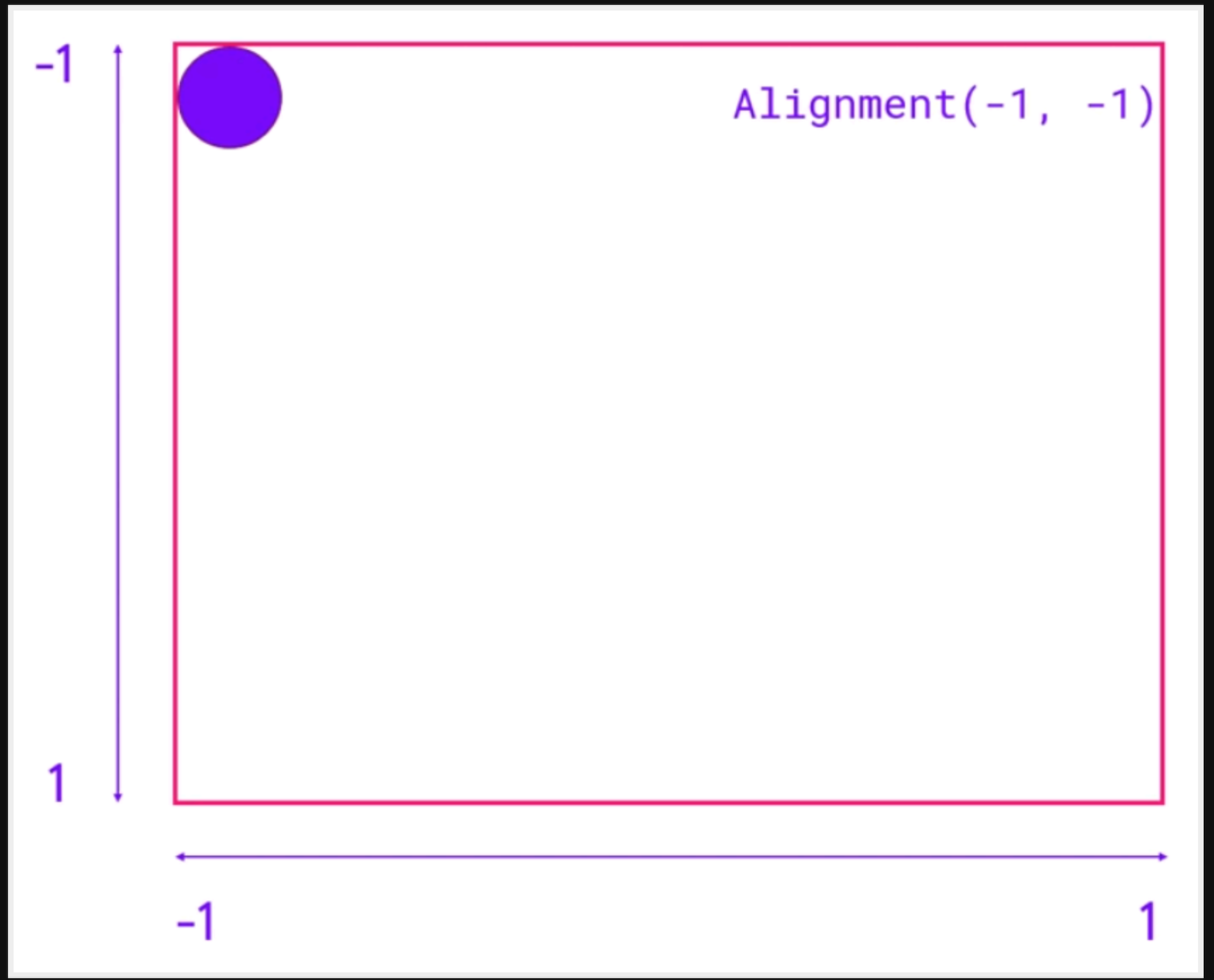
Ajout d’un bouton

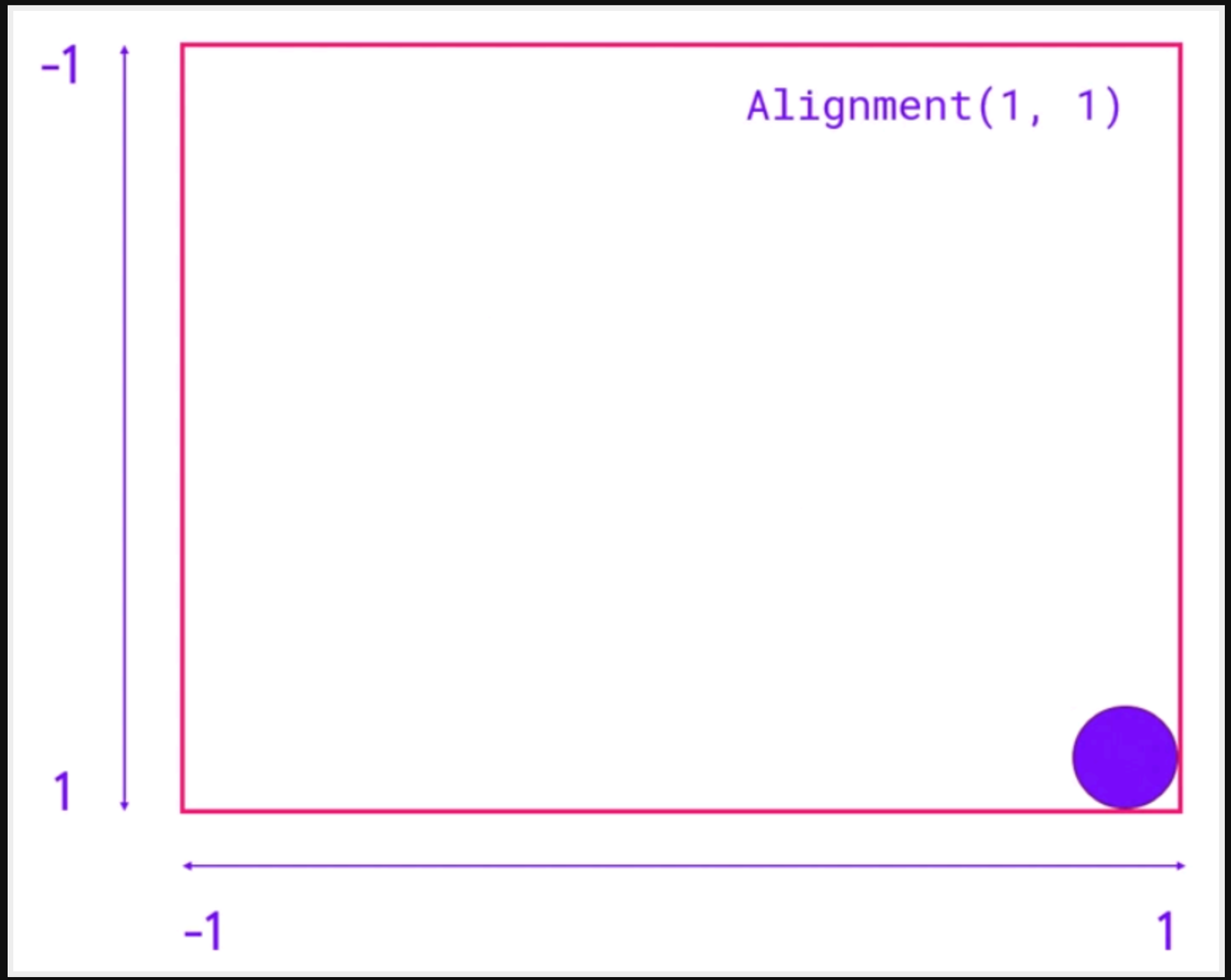
- On va ajouter un *ElevatedButton* en bas de l’écran
- Un *ElevatedButton* requiert :
  - `onPressed` pour définir l’action à effectuer
  - `child` pour définir le contenu du bouton
- On va utiliser le *widget Positioned* pour positionner le bouton
  - les paramètres *top*, *bottom*, *left* et *right* définissent la distance par rapport au bord de l’écran
  - on va « ancrer » le bouton centré en bas, pour cela on utilise juste *bottom*, *left* et *right* avec une valeur identique

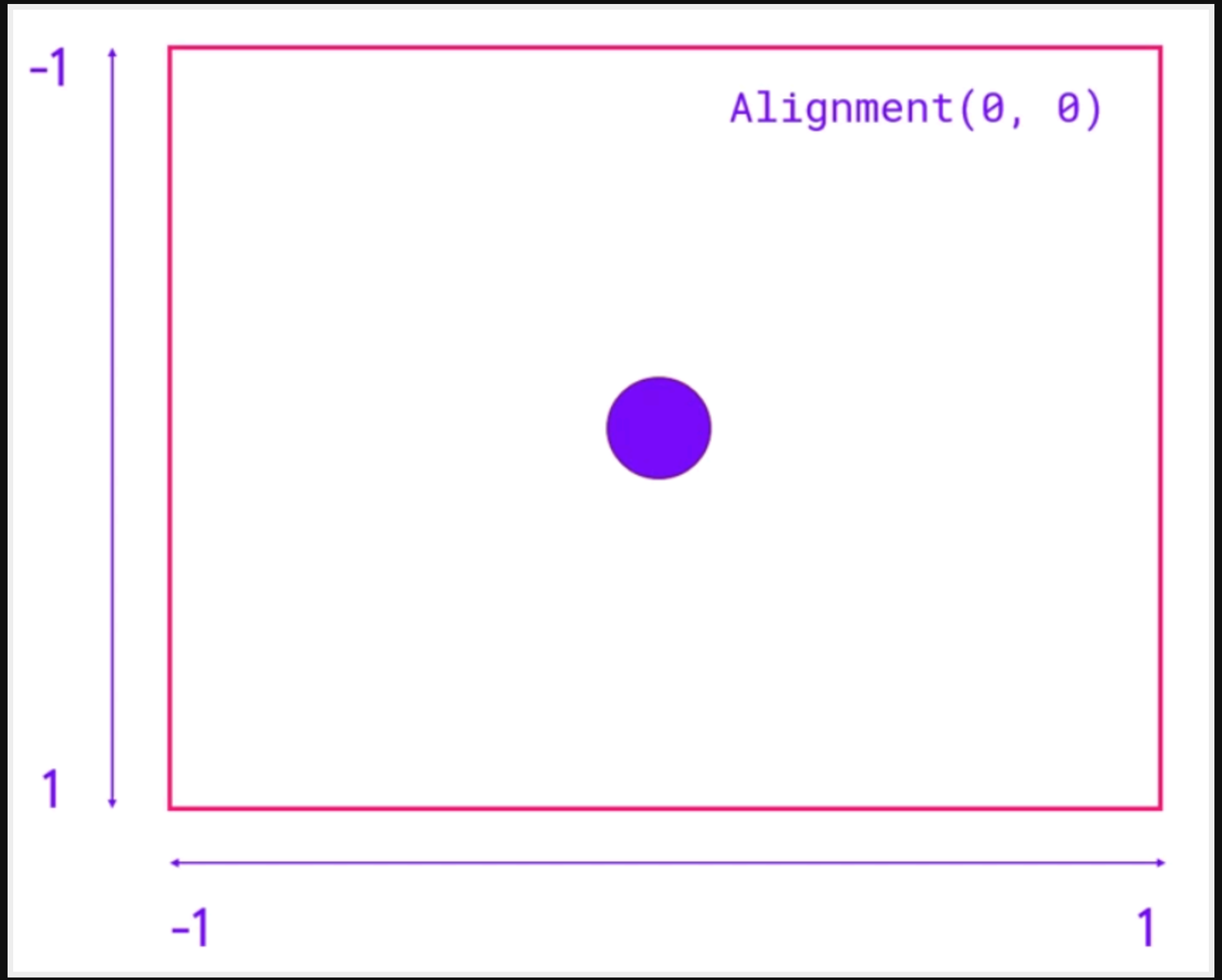
```
...

Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: const Text('Accueil')),
    body: Stack(
      children: [
        Positioned.fill(...),  // image
        Center(...),           // texte
        Positioned(            // bouton
          bottom: 16,
          left: 16,
          right: 16,
          child: ElevatedButton(
            onPressed: () {},
            child: const Text('Cliquez-moi'),
          ),
        )
      ],
    ),
  );
}
```

Positionnement relatif - *Widget* `Align`

- On peut aussi utiliser le *widget Align* pour positionner un *widget* dans le parent
- *Align* prend un paramètre *alignment*
  - *alignment* est un objet de type *Alignment*
  - *Alignment* prend deux paramètres : *x* et *y* (valeurs entre -1 et 1)
  - *x* : -1 = tout à gauche, 1 = tout à droite
  - *y* : -1 = tout en haut, 1 = tout en bas


-1
Alignment(-1, -1)
1
-1
1


-1
Alignment(1, 1)
1
-1
1


-1
Alignment(0, 0)
1
-1
1

Modification du *layout*

- Utilisons des *widgets Align* pour modifier le *layout* de notre *Stack* :
  - texte centré horizontalement et à 25 % de la hauteur
  - bouton centré horizontalement et à 75 % de la hauteur

```
...

Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: const Text('Accueil')),
    body: Stack(
      children: [
        Positioned.fill(...),  // image
        const Align(
          alignment: Alignment(0, -0.5),
          child: Text(...),   // texte
        ),
        Align(
          alignment: Alignment(0, 0.5),
          child: ElevatedButton(...),  // bouton
        )
      ],
    ),
  );
}
```

Améliorations esthétiques

- Modifions le style du texte :
  - couleur blanche (paramètre `color`)
  - ombre portée (paramètre `shadows` - liste)

Récap

- L’interface est décrite en langage Dart
- Tout ce qu’on voit est un *widget*
  - même la structure de l’UI (le *layout*) est contrôlée par des *widgets*
- La racine de l’app est un *widget* `MaterialApp` (*Material Design*)
- Chaque écran est typiquement structuré par un *widget* `Scaffold`
  - *appBar*, *body*,*bottomNavigationBar*…
- Pour structurer le contenu (*body*), on utilise des *widgets* de *layout*
  - *Stack*, *Center*, *Positioned*, *Align*…
  - tous ces *widgets* sont des conteneurs pour un ou plusieurs autres
- Ces autres *widgets* peuvent eux-mêmes être des conteneurs ou bien des objets finaux :
  - *Text*, *Image*, *Icon*, *Button*…
- L’outil *Widget Inspector* permet de visualiser la hiérarchie des *widgets*

Retour - *Widgets Stateless*

- Dans l’écran créé, on a utilisé des *widgets Stateless*
  - tout ce qui est affiché est statique
- Nous allons maintenant voir comment ajouter de l’interactivité

Widgets avec état

Widgets Stateful

- Un *StatefulWidget* est un *widget* qui peut être modifié
- Nécessaire dès que l'interface peut changer
  - écrire du texte
  - agir quand on clique sur un bouton
  - afficher une liste déroulante…
- C'est différent car on doit maintenant gérer un **état** (*state*)
  - quel est le texte affiché ?
  - que fait-on quand on clique sur le bouton ?
  - quelle est la valeur sélectionnée dans la liste déroulante ?

Quelques exemples de *widgets* avec état

- *StatefulWidget* : gérer l'état de l'interface
- *Navigator* : gérer la navigation entre les écrans
- *TextField* : écrire du texte
- *TextEditingController* : gérer le texte saisi dans un *TextField*
- *DropdownButton* : liste déroulante
- *SharedPreferences* : stocker des données de manière persistante

Écran « Paramètres »

- On va illustrer l’utilisation des *widgets* décrits précédemment en développant un écran de paramètres pour l’application
  - nom du l’utilisateur (champ texte)
  - choix de l’image de fond (menu déroulant)
  - bouton pour valider les paramètres
  - persistance entre les lancements de l’application

Implémentation

- Créer un nouvel écran dans un fichier *settings*
  - utiliser un *widget StatefulWidget* (snippet *stful* sous VS Code) nommé *SettingsScreen*
- Différence principale avec *StatelessWidget* : on doit implémenter 2 classes :
  - le *widget* lui-même (*SettingsScreen*)
  - la classe *State* qui gère l'état de l'écran

StatelessWidget vs *StatefulWidget*

- *StatelessWidget* : *widget* qui ne peut pas être modifié
  - classe unique dérive de *StatelessWidget*
  - méthode `build` obligatoire pour décrire l'UI
- *StatefulWidget* : *widget* qui peut être modifié
  - classe dérivée de *StatefulWidget*
    - méthode `createState` obligatoire pour créer une instance de *State*
  - **2ème classe** dérivée de *State* pour gérer l'état
    - méthode `build` obligatoire pour décrire l'UI

La classe *State*

- La classe *State* décrit l'information :
  - définie à la création du *widget* (**initialisation**)
  - et qui peut varier durant le cycle de vie du *widget* (**modification de l'état**)
- Un *StatefulWidget* reste donc **immuable** (ne change pas), comme un *StatelessWidget*
  - **c'est l'objet *State* associé qui change**

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

class SettingsScreen extends StatefulWidget {
  const SettingsScreen({super.key});

  @override
  State<SettingsScreen> createState() => _SettingsScreenState();
}

class _SettingsScreenState extends State<SettingsScreen> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Paramètres')),
      body: const Placeholder(),
    );
  }
}
```

Navigation en Flutter

- Deux façons d’implémenter la navigation :
  - *Navigator* : navigation simple
  - *Router* : navigation plus complexe (gestion des routes, des transitions, etc.)
- On va utiliser *Navigator*

Navigator

- *Navigator* est un *widget* qui gère la navigation entre les écrans
- Fonctionne sur un principe de pile d'écrans :
  - afficher un nouvel écran ? ⇒ on empile un écran sur la pile (*push*)
  - revenir en arrière ? ⇒ on retire un écran de la pile (*pop*)
  - afficher un nouvel écran en supprimant l'actuel ? ⇒ on retire et on empile (*pushAndReplace*)
- *Navigator* est accessible via la méthode `Navigator.of(context)`
  - on appelle ensuite `push`, `pop`… sur cet objet

Navigation - *push*

- Pour afficher un nouvel écran, on utilise la méthode `push`
- On passe un objet `Route` qui décrit l'écran à afficher
  - spécifiquement, on peut utiliser `MaterialPageRoute` pour afficher un écran avec une transition Material prédéfinie

```
Navigator.of(context).push(
  MaterialPageRoute(
    builder: (BuildContext context) => const SettingsScreen()
  ),
);
```

Événement déclencheur

- On veut naviguer vers l'écran *SettingsScreen* quand on clique sur le bouton de la page d'accueil
- On doit donc maintenant ajouter un corps à la méthode `onPressed` du *ElevatedButton* (précédemment vide)
  - c'est à cet endroit qu'on va insérer le code de navigation

```
ElevatedButton(
  onPressed: () {
    Navigator.of(context).push(
      MaterialPageRoute(builder: (context) => const SettingsScreen())
    );
  },
  child: const Text('Paramètres'),
),
```

Implémentation automatique du retour

- Quand on navigue vers un nouvel écran, un bouton « Retour » est automatiquement ajouté dans l'*AppBar*
  - il utilise la méthode `pop` pour revenir à l'écran précédent
- On peut bien sûr personnaliser ce mécanisme

Persistance des paramètres - le package

`shared_preferences`

Packages

- Les packages sont des composants réutilisables que l'on peut inclure dans nos apps
  - implémentés par les équipes Dart/Flutter
  - ou par la communauté Open Source
- https://pub.dev : repository officiel des packages Dart/Flutter
- On souhaite ajouter un package pour gérer la persistance des paramètres de l'application à travers les lancements
  - une solution est d'utiliser le package `shared_preferences`

SharedPreferences - Installation

- Trouver le package sur https://pub.dev
- Regarder la doc d'installation (*Installing*)
- Installer localement
- Vérifier que le package est bien ajouté dans *pubspec.yaml*

SharedPreferences - Présentation

- *SharedPreferences* est une solution de stockage **simple mais limitée**
  - paires clé-valeur
  - nombres (entiers, flottants), booléens, strings, listes de strings
  - ex. : (nom, 'John'), (age, 25), (theme, 'dark')...
- À éviter donc pour :
  - données critiques
  - données volumineuses
  - données complexes (objets, listes complexes...)

SharedPreferences - Utilisation

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

// Récupérer une instance de SharedPreferences
SharedPreferences sp = await SharedPreferences.getInstance();

// Écrire une valeur : clé + valeur
await sp.setInt('age', 25);

// Autres méthodes dispos : setDouble, setBool, setString, setStringList
// Chaque setter a un getter correspondant : getInt, getDouble...

// Lire une valeur
int age = sp.getInt('age');

// Supprimer une valeur
bool estSupprime = await sp.remove('age');
```

Code asynchrone

- Les exemples de code précédents utilisent le mot-clé `await`
- `await` est utilisé pour attendre le résultat d'une **fonction asynchrone**
  - une **fonction asynchrone** est une fonction qui fait un traitement « long » lié à une opération d'entrée-sortie (disque dur, BDD, réseau…)
- Quand on *await* le résultat d'une fonction asynchrone, on doit se trouver dans une fonction elle-même asynchrone (**marquée par le mot-clé *async***)
- **Une fonction *async* renvoie un objet de type *Future***
  - prend en <paramètre de type> le type de retour de la fonction
  - ex. : `Future<int>`, `Future<String>`, `Future<void>`…
- Tout cela est un peu complexe ;)
  - retenez le fondamental : *await* toujours avec *async* dans notre code
  - regardez bien les exemples

Implémentation de la persistance des paramètres

- Fichier *settings_persistence.dart* dans un nouveau répertoire *persistence*
- Classe *SettingsPersistence* avec 2 méthodes :
  - `saveSettings` pour sauvegarder les paramètres
  - `loadSettings` pour charger les paramètres

```dart
// lib/persistence/settings_persistence.dart
import 'package:shared_preferences/shared_preferences.dart';

class SettingsPersistence {
  Future<void> saveSettings(String username, String image) async {
    final prefs = await SharedPreferences.getInstance();
    await prefs.setString('username', username);
    await prefs.setString('backgroundImage', image);
  }

  Future<Map<String, String>> loadSettings() async {
    final prefs = await SharedPreferences.getInstance();
    final username = prefs.getString('username');
    final backgroundImage = prefs.getString('backgroundImage');

    return {
      'username': username ?? '',
      'backgroundImage': backgroundImage ?? '',
    };
  }
}
```

Implémentation de l’UI

- Implémentez l’UI de l’écran *SettingsScreen* :
  - champ texte pour le nom
  - liste déroulante pour l’image de fond
  - ces 2 *widgets* seront dans un conteneur *Column*
    - alignement vertical, les uns en dessous des autres
  - bouton en bas à droite pour valider les paramètres

Implémentez l’UI - TextField

- *TextField* est un *widget* qui permet à l’utilisateur de saisir du texte
- Prend un paramètre `decoration` de type *InputDecoration*
  - paramètre *hintText* = texte affiché quand le champ est vide

```
// Implémentation du TextField
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: const Text('Paramètres')),
    body: const Column(
      children: [
        Text('Nom d\'utilisateur'),
        TextField(
          decoration: InputDecoration(
            hintText: 'Votre nom',
          ),
        ),
      ],
    ),
  );
}
```

Implémentation de l'UI - *DropdownButton*

- *DropdownButton* est un *widget* qui affiche une liste déroulante
  - prend un paramètre `items` de type *List<DropdownMenuItem>*
- chaque élément de la liste est un *DropdownMenuItem*
  - *DropdownMenuItem* prend un paramètre `value` et un paramètre `child`
  - `value` est la valeur de l'élément
  - `child` est le *widget* à afficher dans la liste déroulante

Widget DropdownButton - Fonctionnement

```
DropdownButton<String>(  // String est le type de la valeur
  value: 'image1',       // la valeur sélectionnée
  items: [               // la liste des éléments
    DropdownMenuItem(    // chaque élément a :
      value: 'image1',         // - une valeur (qui sera consultable)
      child: Text('Image 1'), // - un _widget_ (qui est affiché)
    ),
    DropdownMenuItem(
      value: 'image2',
      child: Text('Image 2'),
    ),
  ],
  onChanged: (value) {
    // Code à exécuter quand on sélectionne une nouvelle valeur
    // La valeur sélectionnée est passée en paramètre ('value')
  },
),
```

```
// Implémentation du DropdownButton de l'écran Paramètres
body: Column(
  children: [
    const Text('Nom d\'utilisateur'),
    const TextField(...),
    const Text('Image de fond'),
    DropdownButton<String>(
      value: 'assets/images/sea.jpg',
      items: const [
        DropdownMenuItem(
          value: 'assets/images/country.jpg',
          child: Text('Campagne'),
        ),
        DropdownMenuItem(
          value: 'assets/images/lake.jpg',
          child: Text('Lac'),
        ),
        DropdownMenuItem(
          value: 'assets/images/mountain.jpg',
          child: Text('Montagne'),
        ),
        DropdownMenuItem(
          value: 'assets/images/sea.jpg',
          child: Text('Mer'),
        ),
      ],
      onChanged: (value) {},
    ),
  ],
```

Traquer l’élément sélectionné

- Pour conserver la trace de l’élément sélectionné, on va ajouter à notre classe une variable *selectedImage*
  - utilisée pour stocker la *value* de l’image sélectionnée
  - modifiée dans la méthode onChanged du *DropdownButton*

Mettre à jour l’UI

- Petit problème : si on se contente de mettre à jour la variable *selectedImage* dans la méthode `onChanged`, rien ne se passe à l’écran
  - c’est normal, car l’interface n’est pas mise à jour automatiquement
- **Il faut notifier le framework que l’interface doit changer**
  - pour cela on utilise la méthode `setState`
  - elle indique au framework que l’interface doit être reconstruite
- Une syntaxe permet de passer une fonction anonyme à `setState` pour mettre à jour la variable

```
setState(() {
  // Code qui justifie la mise à jour de l'interface
});
```

```dart
class _SettingsScreenState extends State<SettingsScreen> {
  var selectedImage = 'assets/images/sea.jpg';

  @override
  Widget build(BuildContext context) {
      // ...
      DropdownButton<String>(
        value: selectedImage,
        items: const [
          DropdownMenuItem(
            value: 'assets/images/country.jpg',
            child: Text('Campagne'),
          ),
          DropdownMenuItem(
            value: 'assets/images/lake.jpg',
            child: Text('Lac'),
          ),
          DropdownMenuItem(
            value: 'assets/images/mountain.jpg',
            child: Text('Montagne'),
          ),
          DropdownMenuItem(
            value: 'assets/images/sea.jpg',
            child: Text('Mer'),
          ),
        ],
        onChanged: (value) {
          setState(() {
            selectedImage = value ?? 'assets/images/sea.jpg';
```

Lire/écrire dans un *TextField* depuis le code

- On va avoir besoin de manipuler le *TextField* depuis le code
  - initialisation du champ au moment de l'affichage de l'écran
  - récupération de la valeur du champ au moment de la validation
- Une option courante pour lire/écrire dans un *TextField* en code est d'utiliser un ***TextEditingController***
  - on crée une variable *TextEditingController* pour chaque champ de texte (dans la classe *State*)
  - on l'associe au *TextField* via le paramètre `controller` (dans le *build*)
- On peut ensuite manipuler le *TextField* via le la variable créée
  - accès en lecture/écriture via *leController.text*

```dart
class _SettingsScreenState extends State<SettingsScreen> {
  final usernameController = TextEditingController();

  // ...

  TextField(
    controller: usernameController,
    decoration: InputDecoration(
    hintText: 'Votre nom'),
  ),
}
```

Implémentation du bouton de validation

- Le bouton de validation de l'écran de paramètres sera un *FloatingActionButton*
  - un bouton flottant qui apparaît en bas à droite de l'écran
  - on lui associe une icône (*icon*) et une action (*onPressed*)
- Bonne pratique : extraire la logique de sauvegarde des paramètres dans une méthode à part
- Le code associé utilisera la classe *SettingsPersistence* pour sauvegarder les paramètres

Icônes Material

- Le thème *Material* propose un grand nombre d'icônes prédéfinies
  - https://api.flutter.dev/flutter/material/Icons-class.html
- Listées par catégories sur le site officiel de *Material Design*
  - https://material.io/icons/
- Bien sûr, vous pouvez utiliser d'autres icônes

```
// Implémentation du bouton de validation
FloatingActionButton(
  child: const Icon(Icons.save),
  onPressed: () async {
    saveSettings();  // méthode séparée, à écrire dans la classe
  }
)
```

Méthode `saveSettings`

- On va ajouter la méthode `saveSettings` à la classe *SettingsScreenState*
  - cette méthode va récupérer les valeurs des *TextFields* et du *DropdownButton*
  - puis les sauvegarder dans les *SharedPreferences*

```dart
class _SettingsScreenState extends State<SettingsScreen> {
  final usernameController = TextEditingController();
  var selectedImage = 'assets/images/sea.jpg';

  // ...

  Future<void> saveSettings() async {
    final username = usernameController.text;
    final image = selectedImage;

    final sp = SettingsPersistence();
    await sp.saveSettings(username, image);
  }
}
```

Initialisation de l’écran Paramètres

- Lorsque l’écran Paramètres est affiché, il faut que les paramètres enregistrés soient chargés
- Pour cela, on va écrire une méthode *loadSettings* qui va de nouveau utiliser la classe *SettingsPersistence*
- Cette méthode doit être appelée à chaque fois que l’écran est affiché
  - la méthode prédéfinie *initState*, déclenchée automatiquement au chargement de l’écran, est idéale pour ce besoin

```dart
class _SettingsScreenState extends State<SettingsScreen> {
  final usernameController = TextEditingController();
  var selectedImage = 'assets/images/sea.jpg';

  // initState : appelée automatiquement au chargement de l'écran
  @override
  void initState() {
    super.initState(); // appel à la méthode parente, obligatoire
    loadSettings();    // chargement des paramètres
  }

  // ...

  // Méthode de chargement des paramètres
  // Utilise la classe SettingsPersistence
  Future<void> loadSettings() async {
    final sp = SettingsPersistence();
    final settings = await sp.loadSettings();
    setState(() {
      usernameController.text = settings['username'] ?? '';
      selectedImage = settings['backgroundImage'] ?? 'assets/images/sea.jpg';
    });
  }
}
```

Confirmation de l'enregistrement

- Actuellement, aucun retour (*feedback*) n'est donné à l'utilisateur après l'enregistrement des paramètres
  - Mauvaise UX (expérience utilisateur, *User eXperience*)
- On va ajouter un *SnackBar* pour afficher un message de confirmation
  - un *SnackBar* est un message éphémère qui apparaît en bas de l'écran
- On va utiliser la méthode `showSnackBar` de *ScaffoldMessenger*
  - on lui passe un *SnackBar* avec le message à afficher
- Il faut appeler `showSnackBar` après la sauvegarde des paramètres dans la propriété `onPressed` du bouton de validation

```
floatingActionButton: FloatingActionButton(
  child: const Icon(Icons.save),
  onPressed: () {
    saveSettings();
    ScaffoldMessenger.of(context).showSnackBar(
      const SnackBar(
        content: Text('Paramètres sauvegardés'),
        duration: Duration(seconds: 3),
      ),
    );
  },
),
```

Amélioration visuelle - *Widget Padding*

- Comme en CSS, on peut ajouter des marges autour d'un *widget*
- En Flutter, on utilise le *widget Padding*
  - prend un paramètre `padding` de type *EdgeInsets*
  - *EdgeInsets* est un objet qui définit les marges
- On va envelopper le *widget Column* avec un *Padding* pour ajouter de l'espace autour de notre interface

```
// Ajout de marges autour de l'interface
body: Padding(
  padding: const EdgeInsets.all(16),
  child: Column(
      // ...
    ],
  ),
),
```

Utilisation des paramètres dans l’écran d’accueil

- Deux fonctionnalités à implémenter :
  - afficher « Bienvenue … ! » avec le nom de l’utilisateur dans l’écran d’accueil
  - utiliser l’image de fond choisie

Implémentation

- Problème : on ne peut pas redéfinir *initState* dans la classe *WelcomeScreen*
  - *StalelessWidget* ⇒ pas d'état !
- Solution : convertir le *widget* en *StatefulWidget*
- 2 variables d'instances pour « retenir » le nom et l'image
  - initialisées au chargement (*initState* délègue à une nouvelle méthode *loadSettings*)
  - utilisées dans le *build* pour afficher le nom et l'image

```dart
class WelcomeScreen extends StatefulWidget {
  const WelcomeScreen({super.key});

  @override
  State<WelcomeScreen> createState() => _WelcomeScreenState();
}

class _WelcomeScreenState extends State<WelcomeScreen> {
  String username = '';
  String backgroundImage = 'assets/images/sea.jpg';

  // Redéfinition de initState pour appeler loadSettings
  @override
  void initState() {
    super.initState();
    loadSettings();
  }

  @override
  Widget build(BuildContext context) {
    // ...
    child: Image.asset(
      backgroundImage,  // la variable à la place du chemin en dur
      fit: BoxFit.cover,
    ),
    // ...
    child: Text(
      'Bienvenue, $username !', // idem, ici en $interpolation
    // ...
```

La méthode *dispose*

- La méthode *dispose* d'un *State* est appelée juste avant la destruction du *widget*
  - c'est l'endroit où l'on doit libérer des ressources
  - les ressources non libérées peuvent causer des **fuites mémoire**
- Dans *SettingsScreen*, on a créé un *TextEditingController*
  - c'est une ressource car il est associé à un *listener* qui « écoute » les changements du *TextField*
  - lorsque le *widget* est détruit, la variable *controller* n'est plus accessible par votre code mais reste en mémoire car un *listener* y fait toujours référence !

dispose : quelles ressources ?

- Voici le type de ressources qu'il faut typiquement libérer dans la méthode *dispose* :
  - **animations**
  - **streams** (fichiers…)
  - **controllers** (plusieurs catégories de controllers existent)
  - **sockets** (réseau)

dispose: implémentation

```
class _SettingsScreenState extends State<SettingsScreen> {
  @override
  void dispose() {
    usernameController.dispose(); // libération du controller du TextField
    super.dispose();     // toujours appeler la méthode parente en dernier
  }
}
```

En résumé (1)

- *Stateless* : on implémente la méthode `build` pour décrire l'UI
- *Stateful* : 2 classes, la « viande » est plutôt dans la classe *State*
  - on a des variables d'instance pour les valeurs dont on veut contrôler l'état
  - on redéfinit :
    - `initState` pour initialiser l'état
    - `build` pour décrire l'UI
    - `dispose` pour libérer les ressources
  - on appelle `setState` chaque fois que l'on doit modifier l'UI

En résumé (2)

- *Navigator* pour la navigation entre les écrans
  - `push`, `pop`, `pushAndReplace`
- packages
  - *SharedPreferences* pour la persistance des paramètres
  - pub.dev
  - `flutter pub add...` ⇒ *pubspec.yaml*
- Programmation asynchrone
  - *await*, *async*, *Future*, *then*

Références

- Doc officielle : https://flutter.dev/docs
- Tous les *widgets*, avec des exemples de code : https://flutter.dev/docs/development/ui/widgets
- Communauté Flutter : https://flutter.dev/community
- Chaîne YT *Google Devs Flutter* : https://www.youtube.com/c/flutterdev
- Thème *Material* : https://material.io
- Composants *Cupertino* : https://flutter.dev/docs/development/ui/widgets/cupertino

Flutter

Accéder à une API Web publique

Web API

- De nombreuses applications, surtout mobiles, accèdent à des API Web
- On va voir ici comment faire des requêtes HTTP en Flutter sur une API Web et à gérer correctement les réponses
  - on va utiliser une API Web publique et gratuite
  - vous apprendrez plus tard à créer votre propre API Web

Au menu

- Récupération de donnés (*fetching*)
- Classes « *Model* » pour stocker les données
- *Parsing JSON* : conversion de données JSON en objets Dart
- *FutureBuilder* pour gérer les données reçues et mettre à jour l'UI
- Gestion basique des erreurs (erreurs réseau, réponses invalides…)

L’API Web Zenquotes

- https://zenquotes.io
- API publique qui fournit des citations aléatoires
  - https://zenquotes.io/api/random : exemple de réponse JSON
- Trois champs : *q* (citation), *a* (auteur), *h* (citation en HTML)

Exemple JSON


```
[
  {
    "q": "Often in the real world, it's not the smart that get ahead, but the k
    "a": "Robert Kiyosaki",
    "h": "<blockquote>&ldquo;Often in the real world, it's not the smart that g
  }
]
```

Rappels JSON

- Objets entre {accolades}
  - chaque objet contient des paires *propriété*: *valeur* séparées par des virgules
  - les valeurs peuvent être des chaînes, des nombres, des booléens, des tableaux ou des objets
  - ⇒ les objets peuvent donc être imbriqués
- Tableaux (*arrays*) entre [crochets] : liste d'objets
  - l'exemple précédent est donc un tableau d'un seul objet contenant trois propriétés dont les valeurs sont des strings

Avertissement sur l'utilisation de l'API

- **Zenquotes.io** limite le nombre de requêtes à 5 toutes les 30 secondes
- Si on accède tous à l'API depuis l'IP du lycée, on risque de dépasser cette limite fréquemment
- Il est donc probable que Zenquotes nous renvoie régulièrement le JSON suivant ; ce n'est pas bien grave, on aura quand même une citation et un auteur à afficher !


```
{
  "q": "Too many requests. Obtain an auth key for unlimited access.",
  "a": "zenquotes.io",
  "h": "Too many requests. Obtain an auth key for unlimited access @ zenquotes
}
```

Le package HTTP

- Le package *http* fournit des fonctions simples pour faire des requêtes HTTP et gérer les réponses
- https://pub.dev/packages/http
- À installer

Écran *quote_screen.dart*

- Ce nouvel écran est dédié à l'affichage d'une citation aléatoire récupérée depuis l'API Web
- Dans un premier temps, on va juste se contenter de tester l'API
  - requête et récupération de la citation JSON
  - affichage du résultat **sur la console**
  - on s'occupera de l'UI plus tard

quote_screen.dart - Test API

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

class QuoteScreen extends StatefulWidget {
  const QuoteScreen({super.key});

  @override
  State<QuoteScreen> createState() => _QuoteScreenState();
}

class _QuoteScreenState extends State<QuoteScreen> {
  @override
  void initState() {
    super.initState();
    fetchQuote();  // récupération de la citation
  }

  @override
  Widget build(BuildContext context) {
    return const Placeholder(); // pas d'UI pour l'instant
  }

  // Méthode pour la récupération de la citation
  // Premier jet qui affiche la citation sur la console
  Future<void> fetchQuote() async {
    final url = Uri.parse('https://zenquotes.io/api/random');
    final response = await http.get(url);
    if (response.statusCode == 200) { // 200 = OK
      final quote = response.body;
```

Opérations réseaux asynchrones

- À noter : **toutes les méthodes du package *http* sont asynchrones**
  - l'interaction avec le réseau est naturellement asynchrone
  - en effet, les opérations réseaux prennent un temps indéfini pour s'exécuter complètement
  - il est inadmissible de bloquer complètement l'UI à chaque fois qu'une requête réseau est effectuée
- Il faudra donc utiliser *async/await* et garder en tête que les méthodes retournent des *Future*

De l’API à l’UI

- Plusieurs étapes sont nécessaires pour finalement arriver à la présentation des données sur l’UI
  - récupération en JSON via l’API (déjà fait)
  - conversion du JSON en objet Dart (classes *Model*)
  - utilisation des modèles pour affichage sur l’UI

Classes *Model*

Principe : JSON ⇒ objet Dart

Classes *Model*

- Le format JSON est très souple et pratique pour le stockage et l'échange de données
  - en revanche, il n'est pas pratique pour manipuler ces données dans le code
- les classes *Model* sont des classes qui représentent les données (ici récupérées depuis l'API)
  - un objet modèle va stocker les données (par exemple ici, d'une citation) de manière structurée et les rendre facilement et directement accessibles depuis le code
- On va créer une classe *Quote* pour stocker les citations
  - ⇒ chaque objet de type *Quote* représentera donc une citation

Classe *Quote*

- Créons la classe *Quote* dans un nouveau répertoire *models*
  - citation : propriété *quote* (q dans le JSON)
  - auteur : propriété *author* (a dans le JSON)
- Il est fréquent de ne pas forcément utiliser toutes les données fournies par une API publique
  - ici, on n’aura pas l’usage de la propriété JSON *h* (citation au format HTML)

```dart
class Quote {
  final String text;
  final String author;

  Quote(this.text, this.author);
}
```

Conversion JSON ⇒ objet Dart

- Maintenant on a de quoi stocker les données JSON en mémoire
- Mais comment convertir les données JSON en objets Dart ?
- ⇒ Bibliothèque *dart:convert* (fournie avec Dart)
  - `import 'dart:convert';`
  - méthode *decode* : JSON ⇒ Dart
  - méthode *encode* : Dart ⇒ JSON

Nouveau constructeur *Quote.fromJSON*

- Ajoutons un constructeur nommé *fromJSON* à la classe *Quote*
  - capable de prendre un objet JSON provenant de l'API
  - de le convertir en objet *Quote*
- On se souvient que l'API renvoie un tableau d'un seul objet
  - ⇒ il faudra extraire cet objet unique avant de le traiter

```
Quote.fromJSON(String quoteJSON) {
  final List jsonList = json.decode(quoteJSON); // conversion JSON => Dart
  final quoteMap = jsonList.first; // unique objet de la liste
  text = quoteMap['q'];   // récup du texte par la clé 'q'
  author = quoteMap['a']; // récup de l'auteur par la clé 'a'
}
```

fetchQuote - utiliser le constructeur *fromJSON*

- On va maintenant mettre à jour la méthode *fetchQuote* pour convertir le JSON en objet *Quote* en utilisant ce nouveau constructeur

```
Future fetchQuote() async {
  final url = Uri.parse('https://zenquotes.io/api/random');
  final response = await http.get(url);
  if (response.statusCode == 200) {  // 200 = OK
    final quoteJSON = response.body; // récup du body de la réponse
    final quote = Quote.fromJSON(quoteJSON); //
    print('Citation : ${quote.text} ; Auteur : ${quote.author}');
  } else {
    print('Échec de récupération de la citation');
  }
}
```

fetchQuote - renvoyer le résultat

- Maintenant qu'on a correctement récupéré un objt modèle *Quote*, on va renvoyer ce résultat pour pouvoir l'utiliser plus tard
  - on se débarrasse du *print* de débogage
  - on renvoie l'objet *Quote* à la place
  - on n'oublie pas de changer le type de retour

```dart
Future<Quote> fetchQuote() async {
  final url = Uri.parse('https://zenquotes.io/api/random');
  final response = await http.get(url);
  if (response.statusCode == 200) {
    final quoteJSON = response.body;
    Quote quote = Quote.fromJSON(quoteJSON);
    return quote;  // renvoi de la citation
  } else {
    // En cas d'erreur, on utilise le constructeur non-nommé
    // pour renvoyer l'erreur dans le texte de la citation
    return Quote('Erreur de récupération', '');
  }
}
```

Écran de la citation

- On va enfin s’occuper d’afficher la citation sur l’écran dédié
- Citation centrée sur l’écran (avec l’auteur juste en dessous), horizontalement et verticalement
  - citation en italique
  - auteur en gras (*bold*)
  - léger espace entre les deux
- Padding sur l’élément englobant pour éviter les bords de l’écran

Gestion de la citation

- Ne pas oublier de récupérer la *Quote* depuis *initState*
  - et de la stocker dans une variable d’instance
  - pour pouvoir utiliser ses 2 propriétés dans *build*

Citation du jour

Mistakes are always forgivable, if one has the courage to admit them.

Bruce Lee

```dart
class _QuoteScreenState extends State<QuoteScreen> {
  Quote quote = Quote('', '');

  @override
  void initState() {
    super.initState();
    // utilisation de then
    // permet d'exécuter une action dès que la Quote est récupérée
    fetchQuote().then((value) {
      // on n'oublie pas ssetState car on modifie l'UI
      setState(() {
        quote = value;
      });
    });
  }

  // ...

}
```

```dart
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: const Text('Citation du jour')),
    body: Padding(
      padding: const EdgeInsets.all(20.0),
      child: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            Text(
              quote.text,
              style: const TextStyle(
                fontStyle: FontStyle.italic,
                fontSize: 24,
              ),
              textAlign: TextAlign.center,
            ),
            const SizedBox(height: 16),
            Text(
              quote.author,
              style: const TextStyle(
                fontWeight: FontWeight.bold,
                fontSize: 18,
              ),
              textAlign: TextAlign.center,
            ),
          ],
        ),
```

mainAxisAlignment

- Ici un *widget Center* ne suffirait pas à centrer la *Column* verticalement
  - en fait la *Column* est bien centrée (car elle prend de toute façon tout l'espace disponible)
  - mais ses enfants en le sont pas (ils sont relatifs à la *Column*, et pas au *Center*)
- On utilise la propriété *mainAxisAlignment* de *Column* (aussi sur *Row*) :
  - elle permet de définir comment les enfants de la *Column* vont être alignés verticalement
  - ici, *MainAxisAlignment.center* centre correctement les enfants *à l'intérieur* de la *Column*

Bug - Paramètres manquants

- Notre application a actuellement un bug qui ne concerne que son tout premier lancement après installation (ou après suppression des données de l'application)
- Lorsque l'app va chercher les paramètres pour charger l'écran d'accueil, elle ne trouvera rien dans les *SharedPreferences*
- Ce bug n'a normalement pas été rencontré dans le développement
  - en effet, on a d'abord dev l'écran de paramètres, et donc enregistré des paramètres, avant de les utiliser !
  - on n'a jamais eu à expérimenter l'app, **avec les fonctionnalités de paramètres**, lors d'un premier lancement
- Cela montre l'importance de tester son app dans toutes les situations possibles


Accueil
DEBUG
Bienvenue, !

Correction du bug

- Un moyen simple est de s'assurer, au moment de l'utilisation des variables lors de l'affichage (*build*), que les variables ne sont pas vides
  - on peut ici utiliser l'opérateur ternaire pour afficher un texte par défaut si la variable est vide
- L'implémentation ci-après utilise de nouveau l'interpolation avec accolades (${...}) pour effectuer un traitement légèrement plus complexe pour calculer la valeur à afficher à cet endroit

```
// ...
child: Image.asset(
  backgroundImage.isEmpty          // test : string vide ?
      ? 'assets/images/sea.jpg'    // valeur par défaut
      : backgroundImage,           // valeur normale
  fit: BoxFit.cover,
),

// ...

child: Text(
  'Bienvenue, ${username.isEmpty ? 'utilisateur' : username} !',
  ),
// ...
```

Réimplémenter l’accès aux paramètres

- Actuellement, on a utilisé le bouton d’accès aux paramètres pour tester notre *QuoteScreen*
- On pourrait ajouter un second bouton pour pouvoir naviguer sur les deux écrans
- Mais on va placer les paramètres dans la *appBar* (pattern classique dans les apps mobiles)
  - *appBar* possède une propriété *actions* sur laquelle on peut placer des *IconButton*
  - ceux-ci apparaîtront à droite de la *appBar*
  - utilisé traditionnellement pour la navigation, la recherche…

```dart
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: const Text('Accueil'),
      actions: [
        IconButton(
          onPressed: () async {
            goToSettings(context);
          },
          icon: const Icon(Icons.settings),
        ),
      ],
    ),
    // ...
  )
}

Future<void> goToSettings(BuildContext context) async {
  await Navigator.of(context).push(
    MaterialPageRoute(builder: (context) => const SettingsScreen()),
  );
  loadSettings(); // maj de l'UI après retour
}
```

Extraction de méthode pour la navigation

- On a extrait la méthode *goToSettings* pour la navigation vers l'écran des paramètres
- De la même manière, on va extraire la méthode *goToQuote* pour naviguer vers l'écran de citation
- Pour extraire une méthode dans VS Code :
  - sélectionner le code à extraire
  - clic droit > *Refactor* > *Extract Method…* (ou `Ctrl+.` > *Extract Method…*)

Extraction de *goToQuote*

```
// ...
child: ElevatedButton(
  onPressed: () {
    goToQuote(context);
  },
  child: const Text('Citation du jour'),
),

// ...

Future<void> goToQuote(BuildContext context) async {
  await Navigator.of(context).push(
    MaterialPageRoute(builder: (context) => const QuoteScreen()),
  );
}
```

Refresh de la citation

- Actuellement, pour avoir une nouvelle citation, on doit sortir de l'écran *QuoteScreen* et y revenir pour forcer une nouvelle requête à l'API
- On souhaite faire en sorte de pouvoir récupérer une nouvelle citation tout en restant dans l'écran *QuoteScreen*
- Solution simple et pratique : un bouton permettant de rafraîchir l'UI
  - de nouveau ici, on va choisir de placer cette fonctionnalité dans la *appBar*

```
// appBar de QuoteScreen
appBar: AppBar(
  title: const Text('Citation du jour'),
  actions: [
    IconButton(
      onPressed: () {
        // on vide la citation (force l'affichage du spinner
        //     en attendant la nouvelle citation)
        setState(() {
          quote.text = '';
          quote.author = '';
        });
        fetchQuote().then((value) {
          setState(() {
            quote = value;
          });
        });
      },
      icon: const Icon(Icons.refresh),
    ),
  ],
),
```

FutureBuilder - Principe

- Actuellement, dans le *initState* de *QuoteScreen* :
  - on appelle *fetchQuote*
  - qui fait une opération asynchrone (requête HTTP)
  - on récupère un *Future*
  - on appelle *setState* pour mettre à jour l'UI en fonction de ce résultat
- Ça fonctionne, mais c'est un *pattern* tellement utilisé que Flutter propose un *widget* dédié : *FutureBuilder*
  - cela facilite également la gestion du *feedback* utilisateur pendant le traitement (actuellement on fait ça « à la main »)

FutureBuilder - Fonctionnement

- supprimer le code de *initState*
- englober le *widget* principal du *body* du *Scaffold* dans un *FutureBuilder* (utiliser les *code actions*)
- ajouter la propriété *future* et y appeler *fetchQuote*
- ajouter le *snapshot* comme 2ème paramètre du *builder*
  - le *snapshot* contient les données renvoyées par le *Future* une fois terminé et ainsi que l'état du *Future* (propriété *connectionState*)

FutureBuilder - Snapshot

- L'intérêt du *snapshot* est de pouvoir gérer l'UI en fonction de l'état du *Future*
- Dans le code, on va tester la propriété *connectionState* du *snapshot* (avec des `if`, tout simplement)
- en fonction de l'état du *Future*, on va retourner un *widget* différent
  - *spinner* pour montrer qu'on est en attente de la réponse
  - interface complète une fois la réponse reçue
  - interface d'erreur si la requête a échoué

FutureBuilder - les différents états

FutureBuilder - Implémentation

- Le *snapshot* peut être dans 4 états différents :
  - *ConnectionState.none* : pas encore de *Future* associé
  - *ConnectionState.waitin* : le *Future* est en cours d'exécution, mais n'est pas encore terminé (l'UX voudrait qu'on indique à l'utilisateur que le traitement est en cours)
  - (*ConnectionState.active* : peu utilisé avec *FutureBuilder*)
  - *ConnectionState.done* : le *Future* a terminé son exécution (succès ou erreur)
- On n'a plus besoin d'un *initState*, le *FutureBuilder* va appeler lui-même *fetchQuote* via sa propriété *future*
- Dans le *builder*, on utilise une suite de `if/else` pour renvoyer l'UI correspondante à l'état actuel du *Future*
- Gestion des erreurs :
  - propriété *snapshot.hasError* à *true* si une erreur est survenue
  - propriété *snapshot.error* contient l'erreur
  - on peut utiliser les données contenues dans *snapshot.data* (ou gérer l'erreur)
- Le bouton de rafraichissement n'a plus qu'à appeler *fetchQuote* pour rafraîchir la citation

```dart
class _QuoteScreenState extends State<QuoteScreen> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Citation du jour'),
        actions: [
          IconButton(
            onPressed: () {
              setState(() {
                fetchQuote(); // cet appel suffit maintenant
              });
            },
            icon: const Icon(Icons.refresh),
          ),
        ],
      ),
      body: FutureBuilder(
        future: fetchQuote(),  // l'action gérée par le FutureBuilder
        builder: (context, snapshot) {
          if (snapshot.connectionState == ConnectionState.waiting) {
            return const Center(
              child: CircularProgressIndicator(),
            );
          } else if (snapshot.hasError) {
            return Center(
              child: Text('Erreur : ${snapshot.error}'),
            );
          } else {  // ici on sait que le Future est terminé
```

En résumé (1)

- API Web et format JSON
- Package *http* pour les requêtes HTTP
  - `http.get` pour de simples requêtes GET
  - *status code* des réponses
- Classes *Model* pour stocker les données en Dart
- Conversion JSON ⇒ Dart

En résumé (2)

- *FutureBuilder* pour gérer les traitement asynchrones avec l'UI
- Gestion diverse des erreurs
- *CircularProgressIndicator* pour donner un *feedback* de traitement en cours à l'utilisateur
- *actions* dans la *appBar*

Flutter

ListView

Nouveau besoin

- On peut ajouter des citations en favoris
- Nouvel écran « Citations favorites » qui affiche les citations en liste
- On peut supprimer une citation des favoris
- Dans un premier temps, tout se fera en mémoire (pas de persistance)
- Pour les tests, on pourra construire en dur une liste de citations favorites au chargement de l’app
  - ainsi on pourra tester les fonctionnalités d’affichage et de suppression sans avoir à ajouter des citations à la main à chaque fois

ListView

- *ListView* est un *widget* qui permet d'afficher une liste de données
  - *scrollable* (défilement)
  - *lazy loading* (chargement à la demande)
- On peuple la *ListView* avec des *widgets* enfants
  - typiquement, des *ListTile* pour une liste de données simples

ListTile

- *ListTile* est un *widget* qui affiche **un seul** élément de liste
- Voici quelques-unes de ses propriétés usuelles :
  - *title* (titre)
  - *subtitle* (sous-titre)
  - *leading* (élément à gauche), souvent une icône qui indique l'état ou le type de l'élément
  - *trailing* (élément à droite), souvent une icône qui indique une action possible sur l'élément
  - *onTap* (action au clic)…

Pour notre besoin

- On va utiliser une *ListView* pour afficher les citations favorites
- Chaque élément de la liste sera un *ListTile*
- Ici, on va simplement utiliser :
  - *title* pour afficher la citation
  - *subtitle* pour afficher l'auteur
- Mais il serait possible de créer un *ListTile* personnalisé, en assemblant des *widgets* comme d'habitude

Écran *favorites.dart*

- Il faut toujours se poser la question : *Stateless* ou *Stateful* ?
- Si on n’est pas sûr, on peut partir sur un *Stateless* et changer en *Stateful* si besoin en utilisant les outils de refactorisation automatique de *VS Code*

Suppression d’une citation

- On souhaite supprimer un citation de la liste des favoris en la faisant glisser vers la gauche (*swiping*)
  - la citation doit immédiatement disparaître de la liste affichée
  - elle devra évidemment aussi être supprimée des favoris en mémoire
- Améliorations possibles :
  - *SnackBar* avec un bouton pour annuler la suppression
  - si aucun favori dans la liste, afficher au centre de l’écran un message indiquant « *Pas de favoris* » ainsi que ce que l’utilisateur doit faire pour ajouter des favoris

Évolution possible - Persistance des favoris

- *SharedPreferences* n'est pas bien adapté ici (trop simpliste pour une liste de données complexes, même si ce serait possible)
- Nombreuses solutions :
  - BDD légère NoSQL locale (*Sembast*, *Hive*)
  - BDD légère SQL locale (*SQLite*)
  - SGBD NoSQL (nécessite connexion au serveur)
  - SGBDR (*MySQL*, *PostgreSQL*…, nécessite connexion au serveur)
- Il faut étudier précisément les besoins et choisir une solution adaptée
  - toujours utiliser la solution avec laquelle on est à l'aise techniquement n'est pas toujours un bon calcul