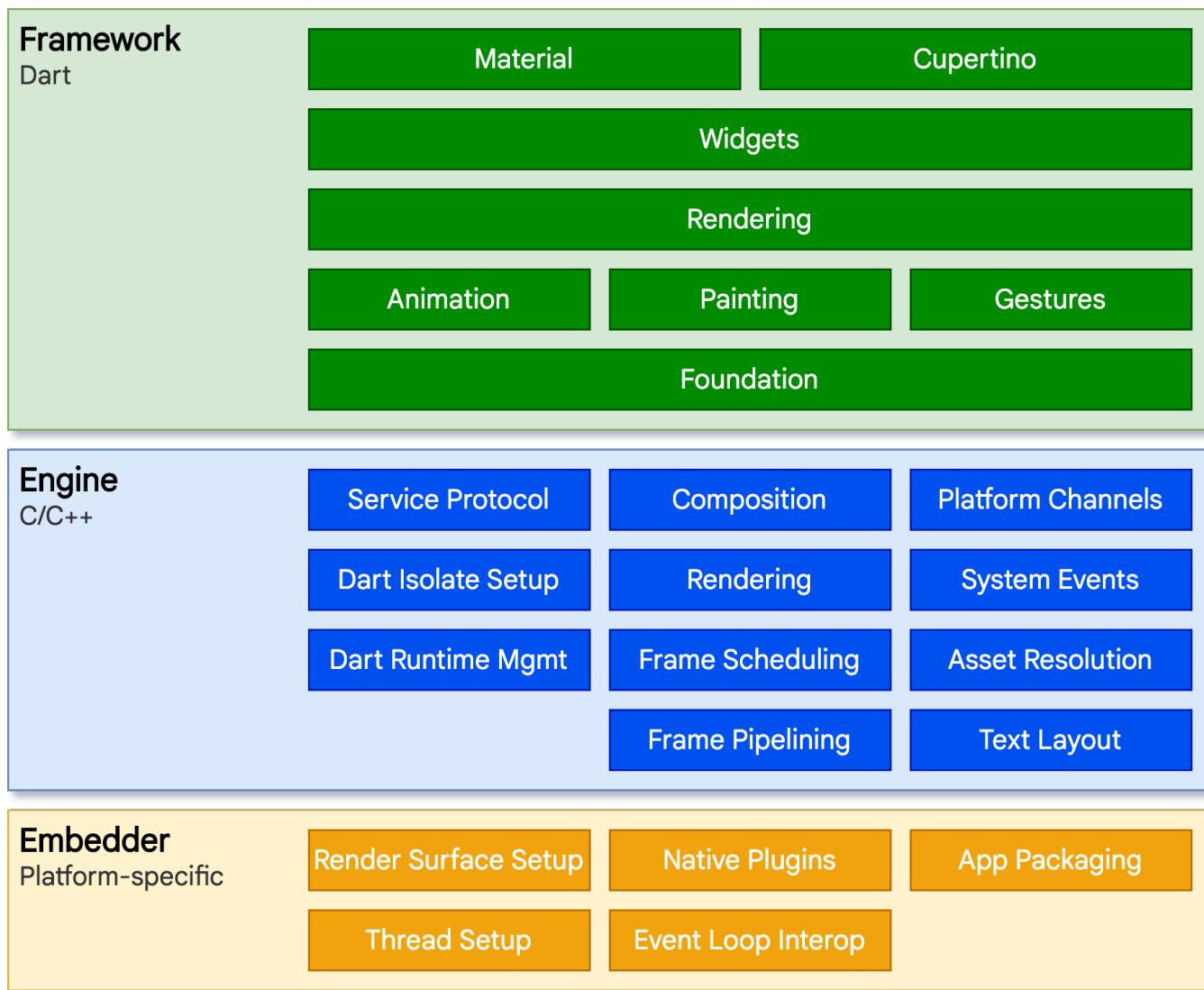


Les bases de Flutter

Aperçu de Flutter

Le Framework Flutter

Flutter est un **kit de développement logiciel (SDK)** open-source pour le développement d'interfaces utilisateur créé par Google. Il est utilisé pour développer des applications pour Android, iOS, Windows, Mac, Linux, et le web à partir d'une base de code unique. L'objectif principal de Flutter est de permettre aux développeurs de fournir des applications à haute performance et à haute fidélité qui semblent naturelles sur différentes plateformes.



Les aspects clés de Flutter incluent son cycle de développement rapide et ses interfaces utilisateur expressives. La fonctionnalité de *hot reload* de Flutter permet aux développeurs de voir presque instantanément les modifications apportées dans le code directement dans l'application, sans perdre l'état actuel de l'application. Cela accélère considérablement le processus de développement.

Flutter met également l'accent sur des interfaces utilisateur personnalisables et complexes. Il est livré avec un riche ensemble de widgets pré-conçus qui suivent des **langages de design** spécifiques, comme **Material Design** (Google) et **Cupertino** (Apple). Cependant, Flutter va au-delà d'être juste un framework; c'est un SDK

complet, fournissant **ses propres widgets et moteur de rendu**, ce qui le rend indépendant des composants UI natifs de la plateforme ciblée.

Une autre caractéristique distinctive de Flutter est son utilisation de **Dart** comme langage de programmation. Dart a été choisi pour ses excellentes caractéristiques de performance (*grâce à la compilation JIT et AOT*) et ses fonctionnalités conviviales pour les développeurs, comme un framework réactif et une riche bibliothèque standard.

Le Moteur de Rendu de Flutter

Le moteur de rendu de Flutter est un composant essentiel, responsable du dessin de l'interface utilisateur des applications. Contrairement à d'autres frameworks qui s'appuient sur les composants natifs de la plateforme ciblée, Flutter utilise son propre moteur de rendu de haute performance pour dessiner les widgets.

Le moteur est principalement écrit en C++ et fournit un support de rendu de bas niveau en utilisant la bibliothèque graphique Skia de Google. Cela signifie que, au lieu d'être simplement une enveloppe autour des composants natifs, Flutter dessine ses propres composants d'interface utilisateur. Cette approche permet des interfaces utilisateur multiplateformes plus flexibles et cohérentes, car Flutter peut rendre son interface utilisateur sur pratiquement n'importe quelle plateforme avec une précision parfaite.

Le moteur de rendu est également responsable des effets tels que les ombres, les dégradés et les transformations, permettant une expérience d'interface utilisateur riche et personnalisable. Il peut gérer des scénarios d'interface utilisateur complexes où plusieurs composants d'interface utilisateur se chevauchent et interagissent, tout en maintenant une performance fluide et une cohérence de rendu.

Le Moteur Graphique Skia

La bibliothèque graphique Skia est une bibliothèque graphique 2D open-source qui sert de cœur au moteur de rendu de Flutter. Skia est utilisée pour le dessin à l'écran et offre toutes les capacités de dessin de bas niveau. Cela inclut la gestion des formes, du texte, des images et plus encore.

Skia est connue pour ses performances élevées et est utilisée dans plusieurs produits Google, y compris Chrome, Chrome OS, Android et maintenant Flutter. En utilisant Skia, Flutter garantit qu'il peut rendre ses widgets de manière cohérente et rapide sur toutes les plateformes.

L'utilisation de Skia apporte également un autre avantage significatif : puisque Skia opère au niveau graphique, Flutter peut repousser les limites de la conception d'interface utilisateur. Il n'est pas limité par l'ensemble des composants fournis par la plateforme native. Cette capacité permet aux concepteurs et développeurs de créer des interfaces utilisateur hautement personnalisées et innovantes qui se démarquent des applications natives typiques.

En résumé, le moteur graphique Skia donne à Flutter la flexibilité de fournir une expérience d'interface utilisateur riche, personnalisée et cohérente sur plusieurs plateformes, ce qui est l'une des raisons clés de la popularité croissante de Flutter parmi les développeurs et les concepteurs.

Configuration et Installation

Installation de Flutter SDK

Pour commencer avec Flutter, la première étape est d'installer la SDK Flutter. La SDK de Flutter contient tout ce dont vous avez besoin pour développer des applications multiplateformes, y compris le moteur, les bibliothèques de widgets, les outils, et plus encore. Le processus d'installation varie légèrement en fonction du système d'exploitation, mais les étapes générales sont les suivantes :

1. **Télécharger le SDK Flutter** : Visitez le [site officiel](#) de Flutter et téléchargez le SDK Flutter pour votre système d'exploitation (Windows, macOS ou Linux).
2. **Extraire le SDK** : Une fois téléchargé, extrayez le fichier ZIP à l'emplacement souhaité sur votre système. Cet emplacement sera votre chemin SDK Flutter.
3. **Mettre à jour votre chemin** : Ajoutez le répertoire `bin/` de Flutter à votre variable d'environnement `PATH`. Cette étape est cruciale pour accéder aux commandes Flutter depuis n'importe quelle fenêtre de terminal.
4. **Vérifier l'installation** : Ouvrez une fenêtre de terminal et exécutez `flutter doctor`. Cette commande vérifie votre environnement et affiche un rapport dans la fenêtre du terminal. La commande recherche des problèmes avec votre installation et fournit des conseils pour les résoudre.

Configuration d'un éditeur (VS Code) ou d'un IDE (Android Studio)

Flutter prend en charge plusieurs éditeurs de code, mais Visual Studio Code (VS Code) et Android Studio sont les plus populaires:

1. **VS Code** : léger et rapide, avec une extension Flutter pour la programmation Dart, qui offre un ensemble riche d'outils pour éditer, déboguer et exécuter vos applications Flutter.
2. **Android Studio** : offre une expérience IDE complète et intégrée pour Flutter. Il comprend un ensemble riche d'outils pour développer des applications Android et Flutter. Il suffit d'installer les plugins Flutter et Dart depuis le marché des plugins.

Après avoir configuré votre environnement, assurez-vous que les plugins/extensions Flutter et Dart sont installés. Ces derniers fournissent des fonctionnalités essentielles comme l'achèvement du code, l'édition de widgets, le support d'exécution et de débogage, et plus encore.

Les Outils en Ligne de Commande Flutter

Le SDK Flutter est livré avec des outils en ligne de commande essentiels pour le développement Flutter. Ces outils sont essentiels pour la gestion de projets, la gestion des dépendances, et l'exécution et le débogage d'applications. Les commandes clés incluent :

1. `flutter create` : crée un nouveau projet Flutter.
2. `flutter run` : exécute une application Flutter.
3. `flutter doctor` : vérifie votre système pour d'éventuels problèmes.
4. `flutter pub get` : obtient tous les packages et dépendances pour un projet Flutter.

Créer Votre Premier Projet Flutter

En Ligne de Commandes

1. Ouvrez votre terminal et naviguez jusqu'au répertoire où vous souhaitez créer le projet.
2. Exécutez `flutter create nom_du_projet` en ligne de commandes en remplaçant `nom_du_projet` par le nom du projet souhaité. Cette commande crée un nouveau projet Flutter avec tous les fichiers et

dossiers nécessaires.

3. Ouvrez le projet dans l'éditeur choisi.
4. Exécutez `flutter run` dans le terminal tout en étant dans le répertoire de votre projet. Cela démarrera le serveur de développement Flutter et lancera votre application sur un émulateur ou un appareil connecté.

Dans un IDE (Android Studio)

1. Démarrez Android Studio et sélectionner **Commencer un nouveau projet Flutter**.
2. Sélectionnez **Application Flutter** comme type de projet et cliquez sur **Suivant**.
3. Si vous n'avez pas déjà configuré le chemin du SDK Flutter dans Android Studio, vous devrez le fournir ici.
4. Entrez le nom de votre projet et assurez-vous que l'emplacement du projet correct est sélectionné, puis cliquer sur **Terminer**.
5. Android Studio configurera votre projet avec les fichiers et dossiers nécessaires. Une fois terminé, vous pouvez voir la structure de votre projet dans le volet gauche.

Structure d'un Projet Flutter

Aperçu

```
flutter_project/
├── android/          # Fichiers spécifiques à la plateforme Android
├── ios/              # Fichiers spécifiques à la plateforme iOS
└── lib/               # Le cœur du projet Flutter
    ├── main.dart       # Point d'entrée de l'application Flutter
    ├── screens/        # Écrans/pages individuels de l'application
    │   └── home_screen.dart # Exemple de fichier d'écran
    ├── widgets/         # Widgets réutilisables personnalisés
    │   └── custom_button.dart # Exemple de fichier widget
    ├── models/          # Modèles de données
    │   └── user.dart     # Exemple de fichier modèle
    └── services/        # Services tels que les appels API, interaction avec la
      base de données
        └── api_service.dart # Exemple de fichier de service
    └── assets/           # Ressources (images, polices, etc.)
    └── pubspec.yaml     # Configuration et dépendances du projet
    └── test/             # Fichiers de test pour l'application
        └── widget_test.dart # Exemple de fichier de test
```

Comprendre le dossier `lib/`

Le dossier `lib/` est le cœur de tout projet Flutter. C'est là que vous passerez la majorité de votre temps à écrire le code Dart de votre application. Ce dossier contient le code source de l'application Flutter.

Typiquement, il inclut :

1. `main.dart` : le point d'entrée de votre application Flutter.
2. **Widgets et Écrans** : widgets personnalisés et différents écrans de votre application.

3. **Services et Utilitaires** : classes et fonctions pour la gestion des données, les requêtes réseau et les fonctions utilitaires.
4. **Modèles** : modèles de données qui représentent la structure de vos objets.

Organiser ce dossier efficacement est essentiel pour maintenir une base de code évolutive et lisible. Il est courant de diviser le dossier en sous-répertoires pour une meilleure organisation du code.

Le Fichier `pubspec.yaml` : Configuration et Dépendances

Le fichier `pubspec.yaml` est crucial dans un projet Flutter. C'est là que vous définissez le nom de votre application, la description, la version, les dépendances et d'autres métadonnées importantes. C'est l'équivalent du fichier `package.json` pour les projets `Node.js`. Les sections clés comprennent :

1. **Dépendances** : liste tous les packages externes sur lesquels votre application s'appuie, tels que le SDK Flutter et les packages tiers.
2. **Dépendances de Développement** : comprend des packages qui aident au développement mais ne sont pas inclus dans la construction de l'application, comme les frameworks de test.
3. **Configurations spécifiques à Flutter** : telles que les ressources (*comme les images et les polices*) et les paramètres spécifiques à la plateforme.

Comprendre `main.dart`

`main.dart` est le point de départ d'une application Flutter. Lorsque vous exécutez une application Flutter, elle recherche la fonction `main()` dans ce fichier. Les composants clés de `main.dart` comprennent généralement :

1. **Fonction `void main()`** : Le point d'entrée de l'application. Elle appelle `runApp()`, qui gonfle le widget donné et l'attache à l'écran.
2. **Widgets** : La racine de l'arborescence des widgets de votre application, généralement un widget `MaterialApp` ou `CupertinoApp`, qui définit la scène pour toute l'application.
3. **Thème et Navigation** : Définitions pour le thème de l'application et la structure de navigation.

Les Widgets Flutter : Les Bases

Que sont les Widgets ?

Les widgets dans Flutter sont les éléments fondamentaux de l'interface utilisateur de l'application. Chaque widget est essentiellement un élément qui dicte la configuration pour un aspect de l'UI. Ils incluent tout, des **éléments structurels** (*comme les boutons et les menus*) aux **éléments stylistiques** (*comme les couleurs et les thèmes*). Les widgets dans Flutter sont uniques car ils suivent un style de programmation **réactif** - chaque widget peut potentiellement impacter la mise en page et l'apparence de l'UI.

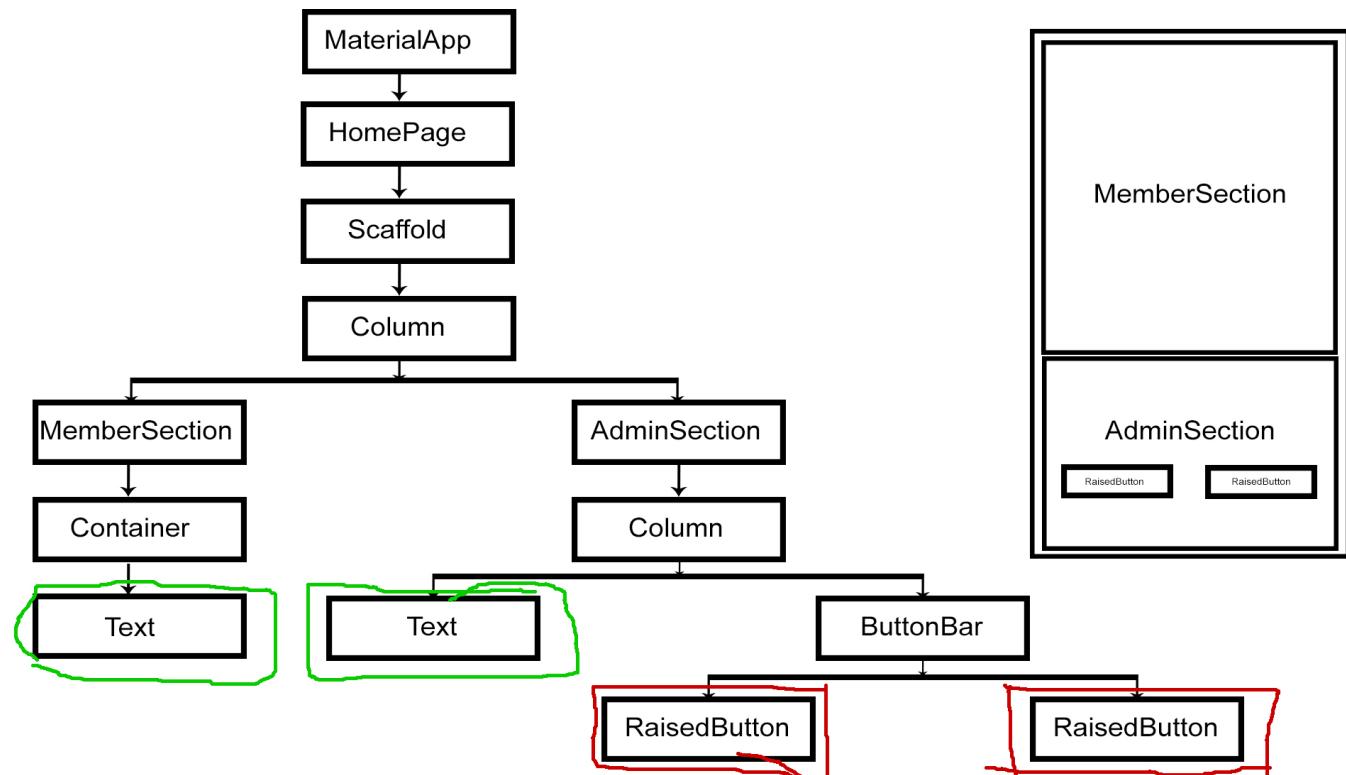
Dans Flutter, **tout est un widget** - d'un simple objet texte à des mises en page complexes. Ce modèle cohérent permet un processus de développement unifié, car vous utilisez la même structure et les mêmes modèles pour créer des composants UI simples et complexes.

Comprendre l'Arborescence des Widgets

L'Arborescence des Widgets dans Flutter est une organisation hiérarchique des widgets. Chaque application commence avec un **widget racine** qui se ramifie en un arbre de widgets. Cette structure arborescente est

essentielle pour le rendu de l'UI et la gestion du flux de données. Chaque widget se niche à l'intérieur de son parent, créant une hiérarchie qui représente toute la mise en page de l'UI.

Comprendre l'Arbre des Widgets est crucial pour un développement Flutter efficace. Il aide à visualiser les relations parent-enfant des composants UI et comment ils sont structurés. Ce cadre conceptuel est la clé pour gérer efficacement l'état et la mise en page.



Widgets Sans État (*Stateless*) vs. Widgets Avec État (*Stateful*)

Flutter dispose de deux classes principales de widgets : Widgets sans état et Widgets avec état.

- Widgets sans état** : statiques et ne changent pas avec le temps. Ils sont redessinés uniquement lorsque des données externes changent. Des exemples incluent **Text** et **Icon**. Les widgets *Stateless* nécessitent moins de gestion et sont efficaces pour des éléments UI simples.
- Widgets avec état** : dynamiques et peuvent changer pendant l'exécution. Ils maintiennent un état qui peut changer en réponse aux interactions des utilisateurs ou à d'autres facteurs. Des exemples incluent **Checkbox**, **Slider** et les **champs de formulaire**. Les widgets *Stateful* sont essentiels pour le contenu interactif et changeant.

Stateless vs Stateful

– A Side by Side Comparison

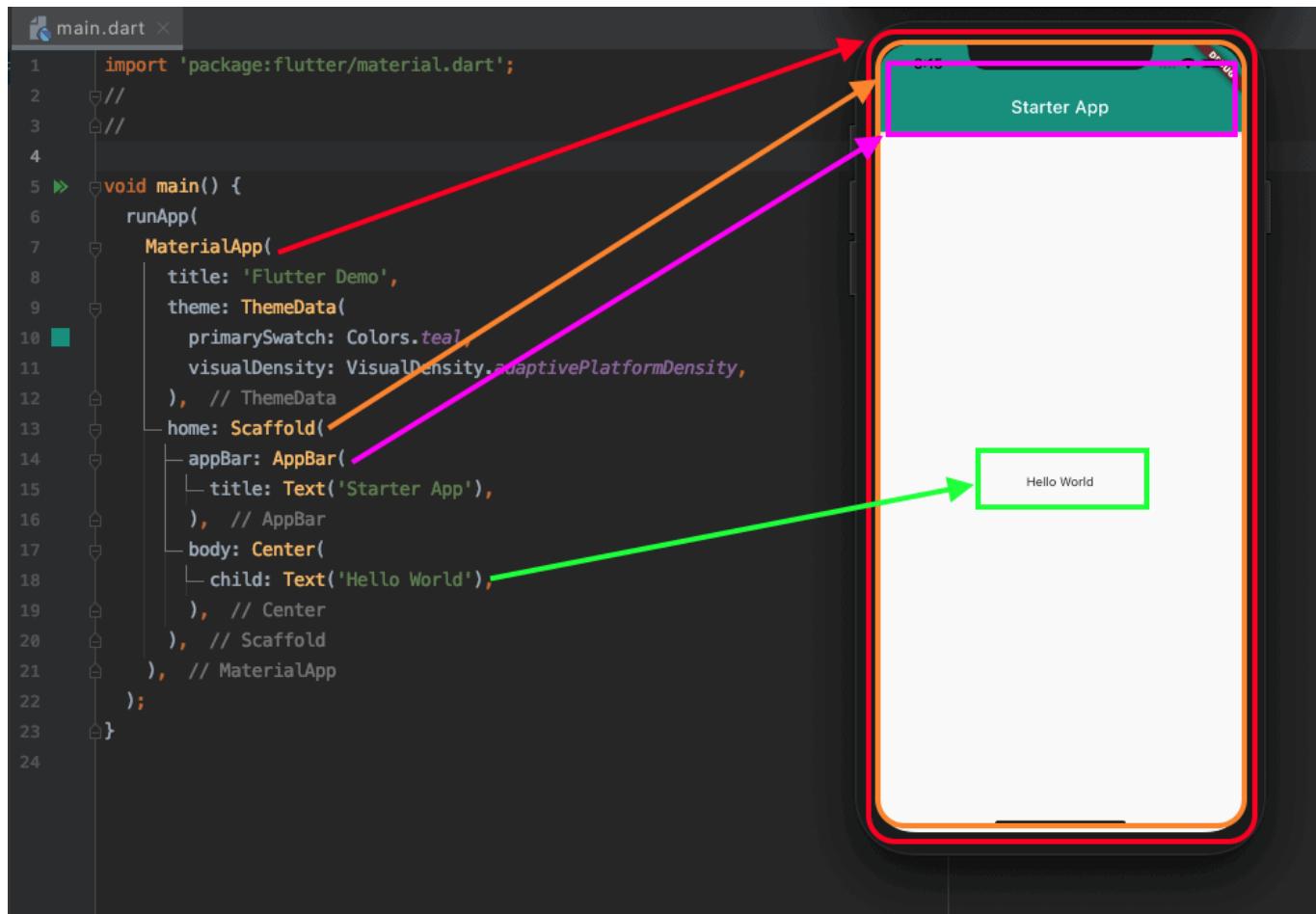


Attribute	Stateless Widgets	Stateful Widgets
Mutable State	Does not have mutable state	Can have mutable state
Dynamic Behavior	Static behavior, does not respond to external events	Can change behavior based on user interactions or data changes
Rebuild Triggering	Does not trigger a rebuild on state change	Triggers a rebuild to reflect updated state
Use Cases	Displaying static content, such as text, icons, or images	Handling dynamic UI elements, interactive components
Performance	Highly efficient and lightweight	May have slightly higher overhead due to state management
Code Complexity	Simple and straightforward with minimal boilerplate	Requires managing state and implementing state change logic
Usage Flexibility	Suitable for components that do not require frequent updates	Essential for components that change frequently
Example Widget	Text, Icon, Image	Checkbox, TextField, Slider

Le Widget MaterialApp

MaterialApp est un widget Flutter prédéfini qui enveloppe plusieurs fonctionnalités nécessaires pour les applications adoptant le **Material Design**. Il prépare le terrain pour toute l'application et se trouve généralement à la racine de votre arbre de widgets. **MaterialApp** fournit une variété de fonctionnalités, y compris la navigation, les thèmes et une barre de titre.

Ce widget est particulièrement important pour les applications qui adhèrent aux directives de **Material Design**. Il assure la cohérence et la fluidité dans les aspects visuels et fonctionnels de l'application.



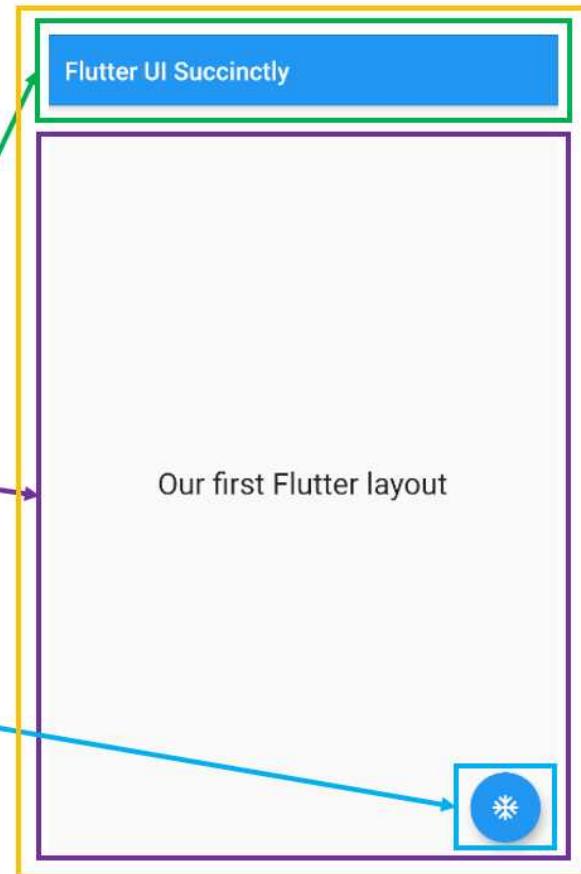
Scaffold et Structure de l'Application

Scaffold est un widget Flutter prédéfini qui fournit une structure de mise en page de base adhérent à **Material Design**. Il offre aux développeurs un cadre cohérent pour construire l'UI. **Scaffold** comprend un certain nombre d'éléments UI prédéfinis comme les **AppBars**, les **Barres de Navigation Inférieures**, les **Boutons d'Action Flottants** et les **Tiroirs**.

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

```
Run | Debug  
void main() => runApp(MyApp());
```

```
class MyApp extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      debugShowCheckedModeBanner: false,  
      home: Scaffold(  
        appBar: AppBar(  
          title: Text('Flutter UI Succinctly'),  
        ), // AppBar  
        body: Center(  
          child: Text(  
            'Our first Flutter layout',  
            style: TextStyle(fontSize: 24),  
          ), // Text  
        ), // Center  
        floatingActionButton: FloatingActionButton(  
          child: Icon(Icons.ac_unit),  
          onPressed: () {  
            print('Oh, it is cold outside...');  
          },  
        ), // FloatingActionButton  
      ), // Scaffold  
    ); // MaterialApp  
  }  
}
```



Utiliser **Scaffold** simplifie le processus de création d'une apparence et d'une sensation standardisées. Il est particulièrement utile pour les nouveaux développeurs ou ceux qui cherchent à prototyper rapidement une application.

Widgets Communs : **Text**, **Row**, **Column**, **Container**, etc.

1. **Widget Text** ([Documentation](#)) : utilisé pour afficher du texte. Il peut être personnalisé avec divers styles, polices et couleurs.
2. **Widgets Row** ([Documentation](#)) et **Column** ([Documentation](#)) : ce sont des widgets flexibles pour créer des mises en page linéaires horizontales (**Row**) ou verticales (**Column**). Ils sont polyvalents et utilisés pour organiser d'autres widgets linéairement.
3. **Widget Container** ([Documentation](#)) : un widget polyvalent utilisé pour le style, le dimensionnement et le positionnement. C'est comme un `div` en HTML et peut être personnalisé avec des marges, du padding, des bordures et une couleur de fond.

Container	Column/Row
Takes exactly one child widget.	Takes multiple (unlimited) child widgets.
Rich alignment and styling options available.	Alignment options available, but there are no styling options.
Flexible width (e.g. child width, available width,).	Always takes full available height (column) /width (row).
Perfect for custom styling and alignment.	Must-use if widgets sit next to/above each other.

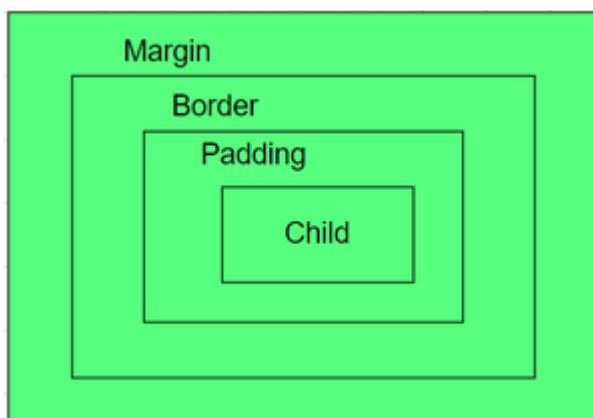
Construction de Mises en Page

Le Modèle de Boîte : Marges, Rembourrage (*Padding*) et Contraintes

Le Modèle de Boîte en Flutter est fondamental pour la conception des UIs et implique les marges, le rembourrage et les contraintes. Ces éléments contrôlent la répartition de l'espace et la taille et le positionnement des widgets.

1. **Les Marges** sont l'espace à l'extérieur de la bordure d'un widget. Elles déterminent l'espace entre un widget et ses éléments environnants.
2. **Le Rembourrage (*Padding*)** est l'espace à l'intérieur d'un widget, entre le contenu et la bordure. Il affecte l'espace au sein du widget, fournissant de la place entre la bordure et ses éléments enfants.
3. **Les Contraintes** sont les règles que les parents imposent aux widgets enfants, dictant leur largeur et hauteur minimales et maximales. Dans Flutter, les contraintes se propagent vers le bas de l'arbre des widgets, tandis que les tailles remontent, ce qui signifie que les enfants ne peuvent pas dépasser les contraintes données par leur parent.

Comprendre le Modèle de Boîte est crucial pour créer des mises en page visuellement attrayantes et fonctionnelles. Il permet aux développeurs de contrôler l'espacement, l'alignement et la taille des widgets dans une application.



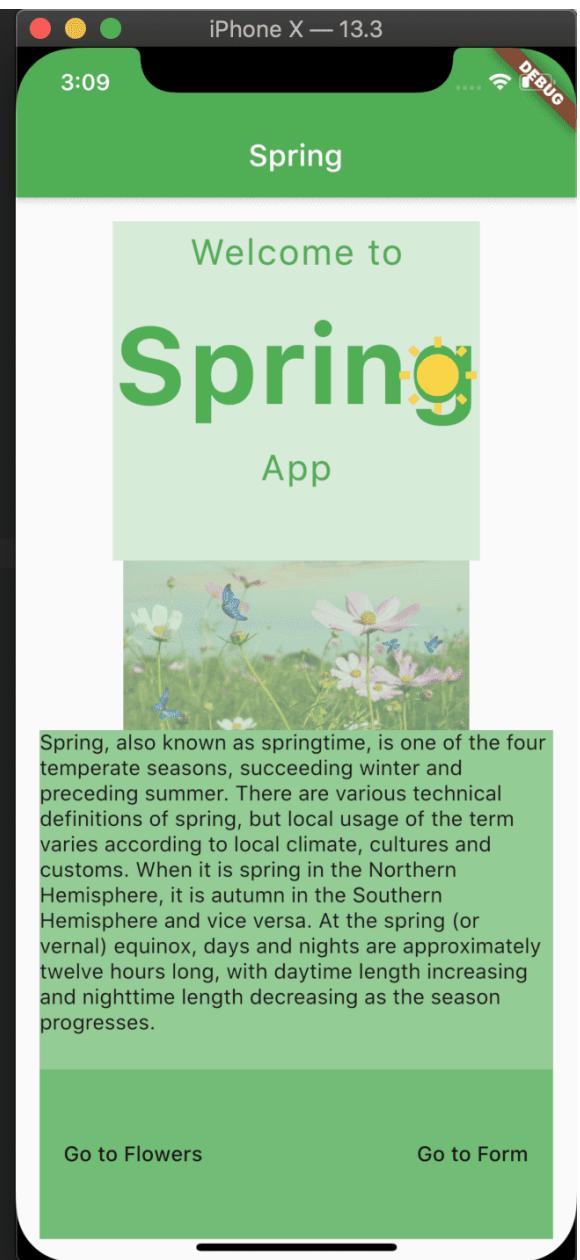
Widgets de Mise en Page : **Stack**, **Positioned**, **ListView**, **GridView**, etc.

Les widgets de mise en page sont les éléments structurels qui aident à organiser et à disposer d'autres widgets sur l'écran.

1. **Stack** ([Documentation](#)) : permet la superposition de widgets les uns sur les autres. Il est utile pour créer des mises en page personnalisées où les widgets doivent se chevaucher ou être positionnés de manière non linéaire.
2. **Positioned** ([Documentation](#)) : un widget utilisé avec **Stack** pour positionner de manière précise ses enfants. Il permet de contrôler la position exacte des widgets enfants en définissant leurs coordonnées x, y, ainsi que leur largeur et hauteur. **Positioned** est idéal pour ajuster la position d'un widget au sein d'une **Stack**.
3. **ListView** ([Documentation](#)) : un widget de liste déroulante. Il affiche ses enfants les uns après les autres dans le sens du défilement (*verticalement par défaut*). **ListView** est idéal pour les longues listes d'éléments où le nombre total d'éléments peut ne pas être connu.
4. **GridView** ([Documentation](#)) : similaire à **ListView** mais affiche les éléments dans un tableau 2D. Les grilles sont utiles pour afficher plusieurs éléments dans un format de grille structuré, comme des galeries d'images ou des listes de produits.

```

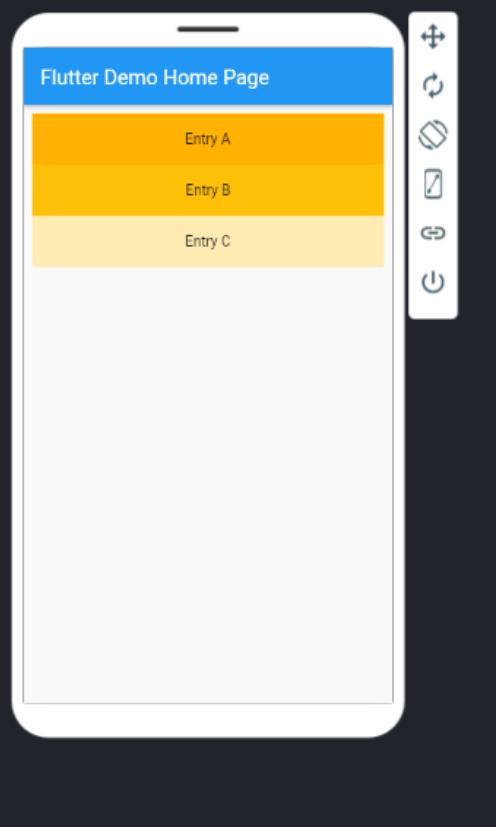
children: <Widget>[
  Expanded(
    flex: 2,
    child: Container(
      color: Colors.green.withOpacity(0.2),
      child: Stack(
        children: <Widget>[
          RichText(
            text: TextSpan(
              children: [
                TextSpan(text: 'Welcome to\n'),
                TextSpan(
                  text: 'Spring',
                  style: TextStyle(...), // TextStyle
                ),
                TextSpan(text: '\nApp'),
              ],
              style: TextStyle(...), // TextStyle
            ),
            textAlign: TextAlign.center,
          ), // RichText
          Positioned(
            top: 76.0,
            right: 0.0,
            child: Icon(
              Icons.wb_sunny,
              color: Colors.yellow[600],
              size: 56.0,
            ), // Icon
          ) // Positioned
        ], // <Widget>[]
      ), // Stack
    ), // Container
  ), // Expanded
  Expanded(
    child: Container(
      color: Colors.green.withOpacity(0.4),
      child: Image(
        image: AssetImage('assets/images/spring.png'),
      ), // Image
    ), // Container
  ), // Expanded
]
  
```



```

@Override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text(title),
    ), // AppBar
    body: ListView(
      padding: const EdgeInsets.all(8),
      children: <Widget>[
        Container(
          height: 50,
          color: Colors.amber[600],
          child: const Center(child: Text('Entry A')),
        ), // Container
        Container(
          height: 50,
          color: Colors.amber[500],
          child: const Center(child: Text('Entry B')),
        ), // Container
        Container(
          height: 50,
          color: Colors.amber[100],
          child: const Center(child: Text('Entry C')),
        ), // Container
      ],
    ), // ListView
  ); // Scaffold
}

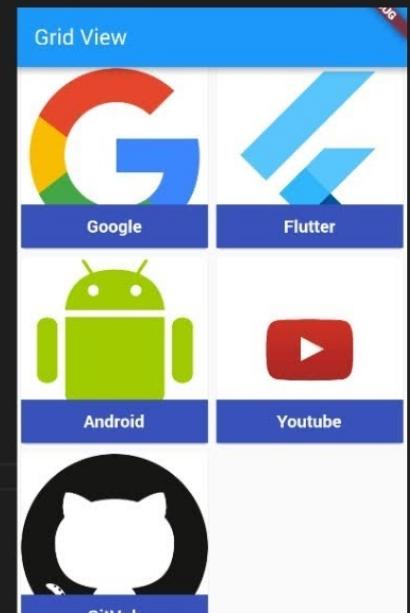
```



```

body: GridView.builder(gridDelegate: SliverGridDelegateWithFixedCrossAxisCount(crossAxisCount: 2),
itemCount: storeItems.length,
itemBuilder: (BuildContext context, int index) {
  return GestureDetector(
    onTap: (){
      //your action
    },
    child: Card(
      child: Stack(
        alignment: FractionalOffset.bottomCenter,
        children: <Widget>[
          Container(
            decoration: BoxDecoration(
              image: DecorationImage(
                image: NetworkImage(storeItems[index].itemImage)
              ) // DecorationImage
            ), // BoxDecoration
          ), // Container
          Container(
            alignment: Alignment.center,
            height: 40.0,
            color: Colors.indigo,
            child: Text(storeItems[index].imageName),
          ) // Container
        ], // <Widget>[]
      ), // Stack
    ), // Card
}

```



Responsive Design dans Flutter : MediaQuery et LayoutBuilder

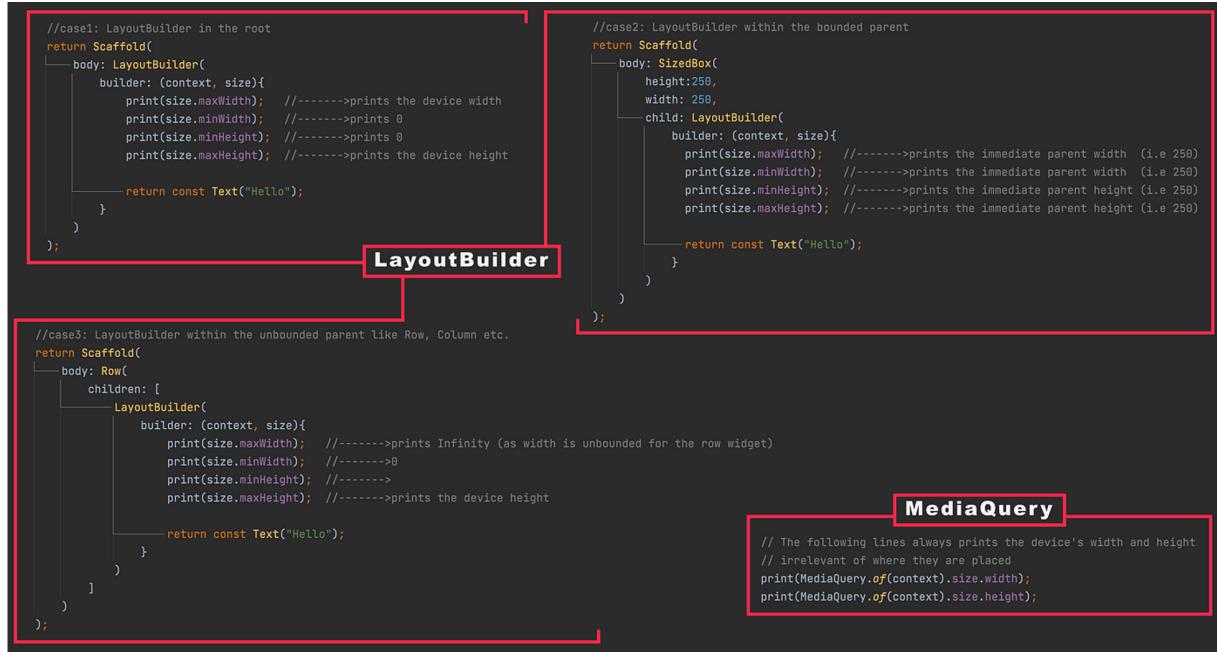
Le design réactif est essentiel pour que les applications Flutter soient attrayantes sur différentes tailles et orientations d'appareils. [MediaQuery](#) et [LayoutBuilder](#) sont deux widgets qui aident à créer des designs réactifs.

1. [MediaQuery](#) ([Documentation](#)) : fournit des informations sur l'appareil, telles que la taille de l'écran, l'orientation et les paramètres d'accessibilité de l'utilisateur. [MediaQuery](#) est souvent utilisé pour

prendre des décisions concernant les tailles, l'espacement et les mises en page spécifiques à l'orientation.

2. **LayoutBuilder** ([Documentation](#)) : Détermine la mise en page du widget en fonction des contraintes de son parent. Il est utile pour créer des widgets qui doivent adapter leur taille en fonction de la taille du widget parent, pour éviter le débordement sur les petits écrans ou s'étendre pour remplir l'espace disponible sur les écrans plus grands.

Ces outils sont indispensables pour créer des mises en page qui s'adaptent à différentes tailles d'écran, garantissant une expérience cohérente et conviviale sur tous les appareils.



The image shows three views of a Flutter Web course page:

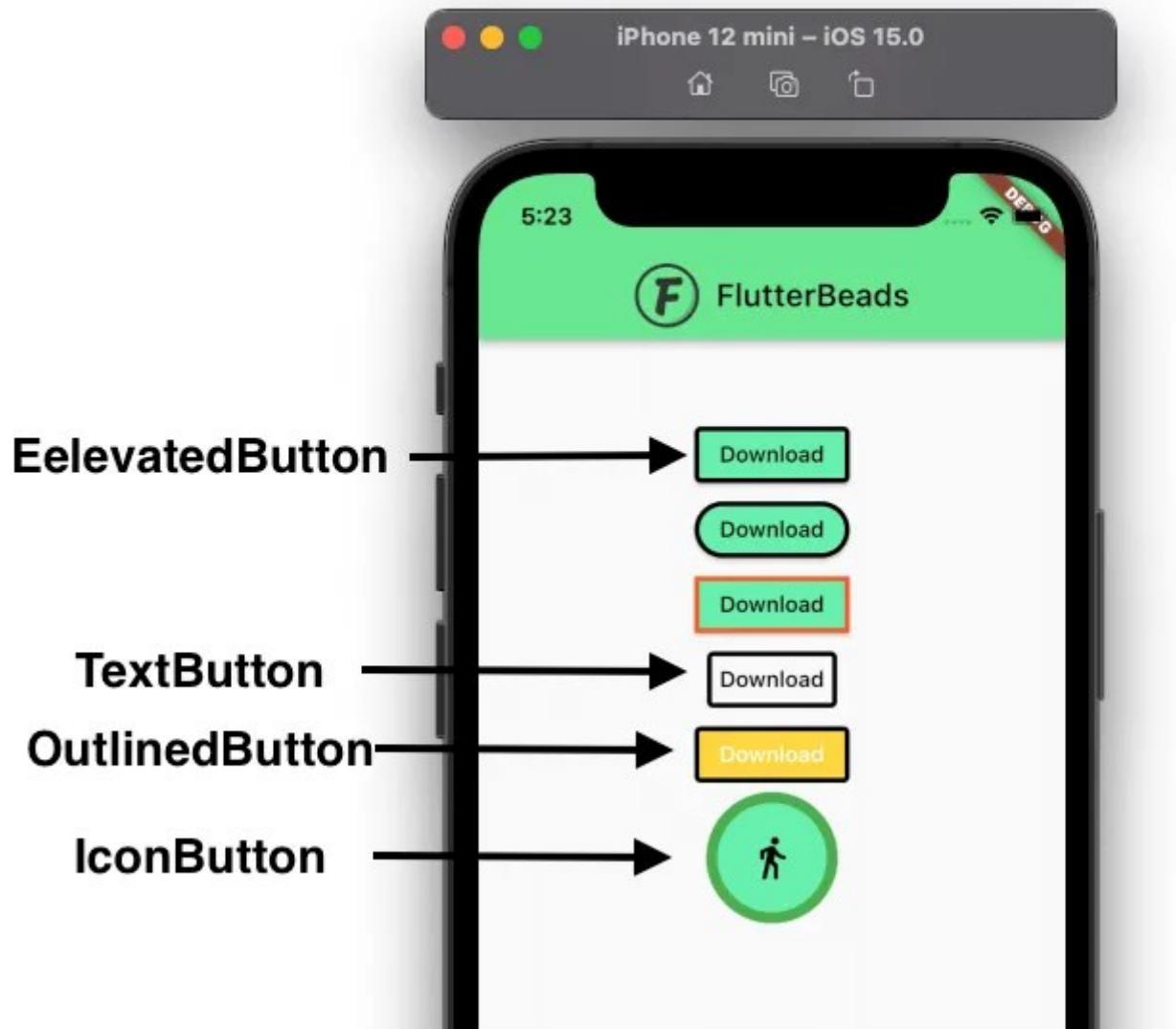
- DESKTOP**: Shows a simple card with the title "FLUTTER WEB. THE BASICS" and a "join course" button.
- TABLET**: Shows a more detailed card with the title, a brief description, and a "join course" button.
- MOBILE**: Shows a mobile-optimized view with a sidebar containing navigation links like "Episodes", "About", and "Skill Up Now".

Interactions Utilisateur

Boutons et Widgets Tactiles

Les boutons et les widgets tactiles sont essentiels pour l'interaction utilisateur dans toute application. Dans Flutter, il existe plusieurs types de boutons et de widgets tactiles, chacun ayant des objectifs différents.

1. [TextButton \(Documentation\)](#) : un widget bouton qui est généralement utilisé pour des actions moins prioritaires ou pour des boutons qui n'ont pas besoin de se démarquer avec un arrière-plan.
`TextButton` est idéal pour les interfaces utilisateur simples et minimalistes.
2. [ElevatedButton \(Documentation\)](#) : ce bouton a un aspect légèrement en relief, lui donnant une présence plus marquée sur l'écran. Il est utilisé pour des actions plus prioritaires ou pour attirer l'attention de l'utilisateur, comme soumettre un formulaire ou confirmer une action importante.
3. [OutlinedButton \(Documentation\)](#) : similaire à `TextButton`, mais avec un contour qui le rend légèrement plus perceptible. C'est un bon choix pour les actions secondaires dans les interfaces utilisateur.
4. [IconButton \(Documentation\)](#) : utilisé pour les actions qui nécessitent des icônes au lieu de texte. C'est parfait pour les barres d'outils, les menus, ou les actions rapides où l'espace est limité ou où une icône peut communiquer l'action plus efficacement qu'un texte.
5. [GestureDetector \(Documentation\)](#) : Un widget polyvalent qui détecte une large gamme de gestes. Il fournit des callbacks pour différents types d'interactions comme taper, double taper, glisser et pincer.
`GestureDetector` est utile lorsque vous devez détecter des gestes sur un widget qui ne support pas d'interaction par défaut. Bien qu'il n'ait pas de représentation visuelle, il est crucial pour les interactions personnalisées.
6. [InkWell \(Documentation\)](#) : Similaire à `GestureDetector` mais spécialement conçu pour les applications **Material Design**. Il fournit un retour visuel (*effet de vague*) lorsque l'utilisateur touche la surface. `InkWell` est généralement utilisé avec des boutons et des éléments de liste en Material Design.



Feature	GestureDetector	Inkwell
Gesture Recognition	Provides a way to recognize various touch gestures such as tap, double tap, long press, drag, and scale	Provides the same gesture recognition capabilities as GestureDetector, including tap, double tap, etc.
Visual Feedback	Does not provide any built-in visual feedback to indicate the user interaction	Provides visual feedback through the ink splash effect when the user interacts with the widget.
Material Design Support	Does not automatically adhere to the Material Design guidelines for touch feedback	Adheres to the Material Design guidelines for touch feedback by default
Ripple Animation	Does not provide built-in ripple animation when the user interacts with the widget	Provides a built-in ripple animation effect to indicate the touch interaction.

Ces widgets sont utilisés pour répondre aux entrées des utilisateurs et initier des actions. Ils sont personnalisables avec différentes propriétés pour la taille, la couleur et la fonctionnalité, permettant une expérience utilisateur sur mesure.

Gestion de l'État et du Contexte

La gestion de l'état est un concept central en Flutter, essentielle pour mettre à jour l'UI basée sur les interactions des utilisateurs ou les changements de données.

- Widgets Stateless:** utilisés pour les widgets statiques (qui ne nécessitent pas de changements d'état). Ils sont redessinés uniquement lorsque des données externes changent.
- Widgets Stateful:** nécessaires pour les widgets qui doivent maintenir un état qui change au fil du temps, comme une case à cocher ou un champ de texte.
- Contexte:** en Flutter, le contexte est un handle vers l'emplacement d'un widget dans l'arbre des widgets. Il est utilisé pour accéder aux données des widgets parents ou pour gérer la navigation et les thèmes.

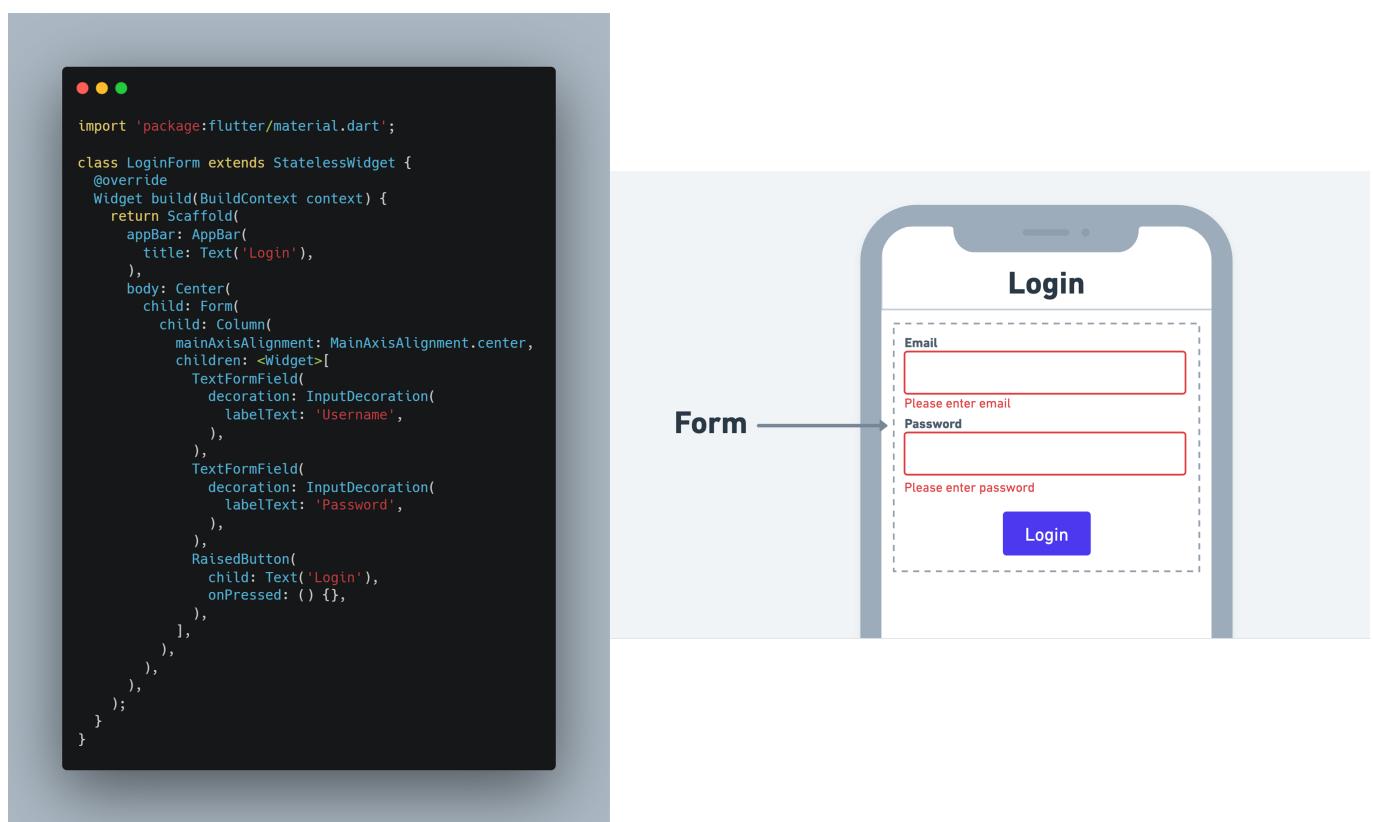
Une gestion efficace de l'état garantit que l'UI reflète l'état actuel de l'application en tout temps, offrant une expérience utilisateur dynamique et réactive.

Entrée et Validation de Formulaire

Les formulaires sont un composant vital de l'interaction utilisateur, permettant aux utilisateurs de saisir et de soumettre des données. Flutter fournit des widgets pour créer et valider des formulaires.

1. **TextField** ([Documentation](#)) : Un widget qui permet aux utilisateurs de saisir du texte. Il peut être utilisé avec un widget **Form** pour la validation.
2. **Form** ([Documentation](#)) : Un widget qui agit comme un conteneur pour plusieurs widgets **FormField**. Il offre un moyen de valider plusieurs champs à la fois.
3. **Validation**: La validation dans les formulaires est cruciale pour garantir l'intégrité de l'entrée utilisateur. Flutter offre des fonctionnalités de validation intégrées, permettant aux développeurs de fournir des retours sur les erreurs d'entrée utilisateur.

Mettre en œuvre des formulaires avec une validation appropriée est essentiel pour obtenir efficacement les données des utilisateurs et maintenir la qualité des données soumises.



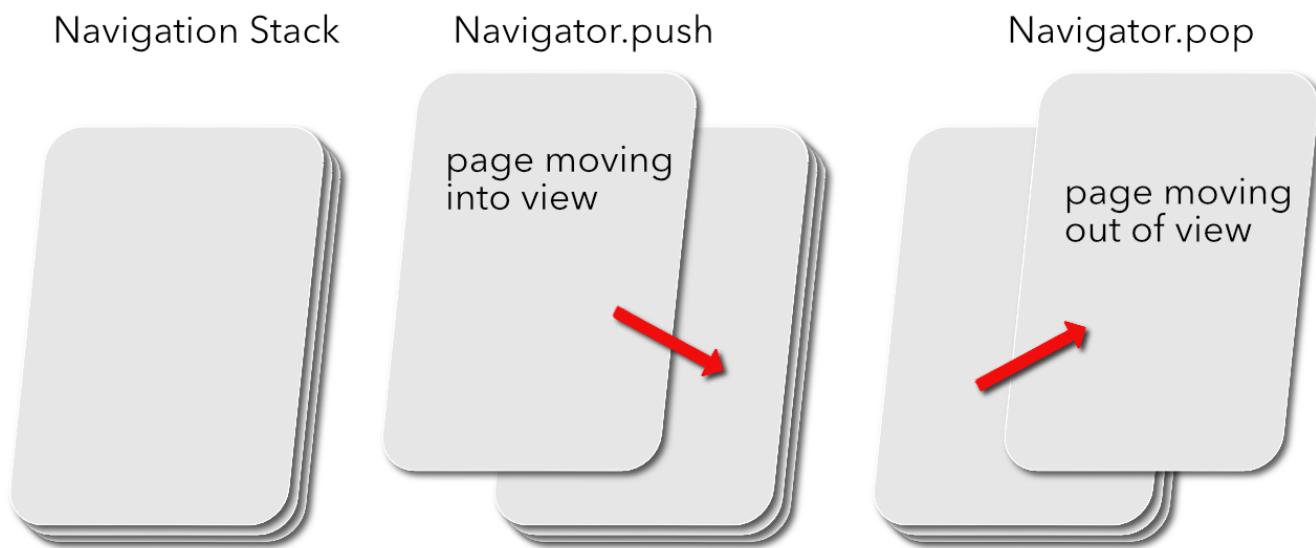
Navigation

La classe **Navigation**

La classe **Navigator** en Flutter est un composant fondamental pour gérer la navigation entre les écrans (ou pages) dans une application. Elle maintient une **pile de routes** (ou écrans) et fournit des méthodes pour gérer cette pile, telles que **Navigator.push()** et **Navigator.pop()**. Lorsqu'un nouvel écran est ouvert, il est empilé sur la pile du **Navigator** et devient la **vue active**. Pour revenir à l'écran précédent, la route actuelle est retirée de la pile, révélant la route sous-jacente. Le Navigator travaille en étroite collaboration avec **BuildContext** pour identifier quelle partie de l'arbre de widgets afficher.

Comprendre le **Navigator** est crucial pour créer des applications multi-pages et pour gérer le parcours de l'utilisateur à travers l'application.

Pages in the navigation stack



Routes et Écrans

Les routes dans Flutter représentent un **écran** ou une **page**. Flutter offre plusieurs façons de définir des routes :

- Routes Nommées** : définies dans le widget `MaterialApp`, les routes nommées rendent la navigation plus intuitive et gérable. Vous attribuez un identifiant de chaîne à chaque route et utilisez ces identifiants pour naviguer.
- Routes Dynamiques** : pour des scénarios de navigation plus complexes, les routes dynamiques permettent de passer des données et de générer des routes à la volée. Les routes sont généralement définies à l'aide de `MaterialPageRoute`, qui s'intègre parfaitement avec **Material Design**, offrant des animations de transition d'écran standard.

Organiser la structure de navigation à l'aide de routes et d'écrans est vital pour une application bien structurée et navigable.

Named Routes

```

1 // main.dart
2 import 'package:flutter/material.dart';
3 import 'my_home_page.dart';
4 import 'about_page.dart';
5
6 void main() {
7   runApp(MyApp());
8 }
9
10 class MyApp extends StatelessWidget {
11   @override
12   Widget build(BuildContext context) {
13     return MaterialApp(
14       title: 'Flutter Demo',
15       initialRoute: '/',
16       routes: {
17         '/': (context) => MyHomePage(),
18         '/about': (context) => AboutPage(),
19       },
20     );
21   }
22 }
// Inside any widget
23 ElevatedButton(
24   onPressed: () {
25     Navigator.pushNamed(context, '/about');
26   },
27   child: Text('Go to About Page'),
28 )

```

Dynamic Routes

```

1 // Inside any widget
2 ElevatedButton(
3   onPressed: () {
4     Navigator.push(
5       context,
6       MaterialPageRoute(
7         builder: (context) => DetailPage(item: 'Some data'),
8       ),
9     );
10   },
11   child: Text('Go to Detail Page with Data'),
12 )
// detail_page.dart
13 import 'package:flutter/material.dart';
14
15 class DetailPage extends StatelessWidget {
16   final String item;
17
18   DetailPage({required this.item});
19
20   @override
21   Widget build(BuildContext context) {
22     return Scaffold(
23       appBar: AppBar(
24         title: Text('Detail Page'),
25       ),
26       body: Center(
27         child: Text(item),
28       ),
29     );
30   }

```

Passage de Données Entre Écrans

Le passage de données entre les écrans est un aspect important de la navigation dans une application. Flutter offre des moyens flexibles de passer des données :

1. Lors de la navigation vers un nouvel écran, vous pouvez passer des données en tant qu'**arguments de constructeur au widget du nouvel écran** ou en les passant à `Navigator.push()`.
2. Pour renvoyer des données depuis un écran, vous pouvez utiliser la méthode `Navigator.pop()`, en passant **éventuellement des données en retour**.
3. Utiliser des solutions de **gestion d'état global** comme `Provider` ou `Riverpod` peut également faciliter le partage de données entre différents écrans sans les coupler étroitement.

Passer efficacement des données entre les écrans est essentiel pour maintenir la continuité et le contexte de l'expérience utilisateur.

Passing Data between Screens through Navigator

```

@Override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: totalResults > 0 ? Text('News ($totalResults)') : Text('News')
    ), // AppBar
    body:
      ListView.separated(itemBuilder: (BuildContext context, int index){
        return Card(
          child: InkWell(
            onTap: () {
              Navigator.push(
                context,
                MaterialPageRoute(builder: (context) => WebViewPage(),
                  settings: RouteSettings(
                    arguments: {
                      'name' : articles[index]['source']['name'],
                      'url' : articles[index]['url'],
                    }
                  ) // RouteSettings
                ) // MaterialPageRoute
              );
            },
            child: Column(

```

● ● ●

```

          override
          Widget build(BuildContext context) {
            return Container(
              RaisedButton(
                onPressed: () {
                  Navigator.pop(context);
                },
              ),
            );
          }

```

Passing Data between Screens using the destination's constructor

```

import 'package:flutter/material.dart';
import 'package:pass_data_demo/second_view.dart';

class FirstView extends StatelessWidget {
  FirstView({Key? key}) : super(key: key);
  final TextEditingController _textEditingController = TextEditingController();
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      resizeToAvoidBottomInset: false,
      appBar: AppBar(
        title: const Text('First Screen'),
      ), // AppBar
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            Padding(
              padding: const EdgeInsets.symmetric(horizontal: 40.0),
              child: TextField(
                controller: _textEditingController,
              ),
            ), // Padding
            ElevatedButton(
              onPressed: () {
                Navigator.of(context).push(MaterialPageRoute(
                  builder: (context) => SecondView(data: _textEditingController.text),
                )); // MaterialPageRoute
              },
              child: const Text('Send Back Data'), // ElevatedButton
            ),
          ],
        ), // Column
      ), // Center
    ); // Scaffold
  }
}

```

Workflow

Concevoir une Interface Utilisateur

Concevoir l'interface utilisateur est la première étape de la création d'une application Flutter. Commencez par considérer la mise en page et les widgets nécessaires pour votre conception.

- Choisir une Mise en Page** : commencez avec une structure de mise en page basique. Pour une application simple, les widgets **Column** ou **Row** peuvent organiser les éléments verticalement ou horizontalement. Pour des structures plus complexes, envisagez d'utiliser **GridView** ou **Stack**.
- Sélectionner des Widgets** : déterminez les widgets dont vous avez besoin. Pour le texte, utilisez des widgets **Text**. Pour les images, utilisez **Image**. Pour les éléments interactifs, considérez les boutons ou les champs de saisie.
- Stylisation** : Appliquez des styles à vos widgets. Personnalisez les couleurs, les polices et les tailles en utilisant les propriétés de chaque widget. Utilisez des widgets **Container** pour un style plus avancé, y compris le padding, les marges et les bordures.

Une interface utilisateur de base doit être intuitive, fonctionnelle et visuellement attrayante. Commencez simplement et affinez au fur et à mesure.

Ajouter de l'Interactivité

L'interactivité rend votre application engageante. Ajoutez des fonctionnalités aux widgets pour répondre aux actions des utilisateurs.

1. **Boutons et Gestes** : utilisez des boutons pour des interactions de base. `TextButton` ou `ElevatedButton` peuvent être utilisés pour des actions telles que soumettre un formulaire ou naviguer vers un nouvel écran. Pour des gestes plus personnalisés, utilisez `GestureDetector`.
2. **Gestion de l'État** : Gérez l'état de vos widgets. Utilisez `setState()` dans les widgets *stateful* pour mettre à jour l'UI en réponse aux événements.
3. **Retours aux Utilisateurs** : Fournissez un retour visuel ou auditif pour les interactions. Utilisez des animations, des messages snackbars ou de simples changements de texte pour informer les utilisateurs du résultat de leurs actions.

Testez vos interactions de manière approfondie pour vous assurer qu'elles sont fluides et intuitives.

Implémenter la Navigation

La navigation entre différentes écrans/pages d'une application est cruciale pour une bonne expérience utilisateur.

1. **Configuration des Routes** : définissez les routes de votre application. Utilisez des routes nommées pour plus de clarté et de facilité de gestion. Configurez les routes dans le widget `MaterialApp`.
2. **Navigation Entre les Écrans** : utilisez `Navigator.push()` pour passer à un nouvel écran et `Navigator.pop()` pour revenir à l'écran précédent.
3. **Transfert de Données Entre les Écrans** : passez des données entre les écrans en utilisant des arguments de constructeur ou des outils de gestion d'état global comme `Provider` ou `Riverpod`.

Implémenter une navigation claire et efficace est essentiel pour une application conviviale. Assurez-vous que les transitions entre les écrans sont fluides et logiques.

Exercice : Application Simple des Notes

Cet exemple est une application simple de prise de notes où les utilisateurs peuvent ajouter et voir des notes. Voici les fichiers introduits :

1. `main.dart` : c'est le point d'entrée de l'application. Il configure le `MaterialApp` avec notre écran d'accueil.
2. `note_list_screen.dart` : l'écran d'accueil affichant une liste de notes. Chaque note est un `ListTile` dans un `ListView.builder`. Il a aussi un `FloatingActionButton` pour ajouter de nouvelles notes.
3. `note_detail_screen.dart` : un écran pour voir et ajouter des notes. Il utilise un `Form` avec `TextField` pour l'entrée et la validation du formulaire.

```
// lib/main.dart
import 'package:flutter/material.dart';
import 'screens/note_list_screen.dart';

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Notes App',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: NoteListScreen(),
    );
  }
}
```

```
        title: 'Application Simple de Notes',
        theme: ThemeData(
            primarySwatch: Colors.blue,
        ),
        home: NoteListScreen(), // Point de départ de l'application
    );
}
}
```

```
// lib/screens/note_list_screen.dart
import 'package:flutter/material.dart';
import 'note_detail_screen.dart';

class NoteListScreen extends StatelessWidget {
    @override
    Widget build(BuildContext context) {
        return Scaffold(
            appBar: AppBar(
                title: Text('Notes'),
            ),
            body: ListView.builder(
                itemCount: 10, // Nombre exemple de notes
                itemBuilder: (context, index) {
                    return ListTile(
                        title: Text('Note $index'),
                        onTap: () {
                            // Naviguer vers l'écran de détail lorsqu'une note est tapée
                            Navigator.push(
                                context,
                                MaterialPageRoute(builder: (context) => NoteDetailScreen()),
                            );
                        },
                    );
                },
            ),
            floatingActionButton: FloatingActionButton(
                onPressed: () {
                    // Naviguer vers l'écran de création de note
                    Navigator.push(
                        context,
                        MaterialPageRoute(builder: (context) => NoteDetailScreen()),
                    );
                },
                child: Icon(Icons.add),
            ),
        );
    }
}
```

```
// lib/screens/note_detail_screen.dart
import 'package:flutter/material.dart';

class NoteDetailScreen extends StatefulWidget {
    @override
    _NoteDetailScreenState createState() => _NoteDetailScreenState();
}

class _NoteDetailScreenState extends State<NoteDetailScreen> {
    final GlobalKey<FormState> _formKey = GlobalKey<FormState>();
    String _noteContent = '';

    void _saveNote() {
        if (_formKey.currentState!.validate()) {
            // Sauvegarder la note ici (par exemple, dans une base de données ou une
            solution de gestion d'état)
            Navigator.pop(context); // Retourner à l'écran précédent
        }
    }

    @override
    Widget build(BuildContext context) {
        return Scaffold(
            appBar: AppBar(
                title: Text('Détail de la Note'),
            ),
            body: Form(
                key: _formKey,
                child: Padding(
                    padding: EdgeInsets.all(16.0),
                    child: Column(
                        crossAxisAlignment: CrossAxisAlignment.stretch,
                        children: <Widget>[
                            TextFormField(
                                decoration: InputDecoration(
                                    labelText: 'Note',
                                    border: OutlineInputBorder(),
                                ),
                                onSaved: (value) {
                                    _noteContent = value ?? '';
                                },
                                validator: (value) {
                                    if (value == null || value.isEmpty) {
                                        return 'Veuillez entrer du texte';
                                    }
                                    return null;
                                },
                            ),
                            ElevatedButton(
                                onPressed: _saveNote,
                                child: Text('Sauvegarder'),
                            ),
                        ],
                    ),
                ),
            ),
        );
    }
}
```

```
        ),  
        ),  
        ),  
    );  
}  
}
```



Note 0

Note 1

Note 2

Note 3

Note 4

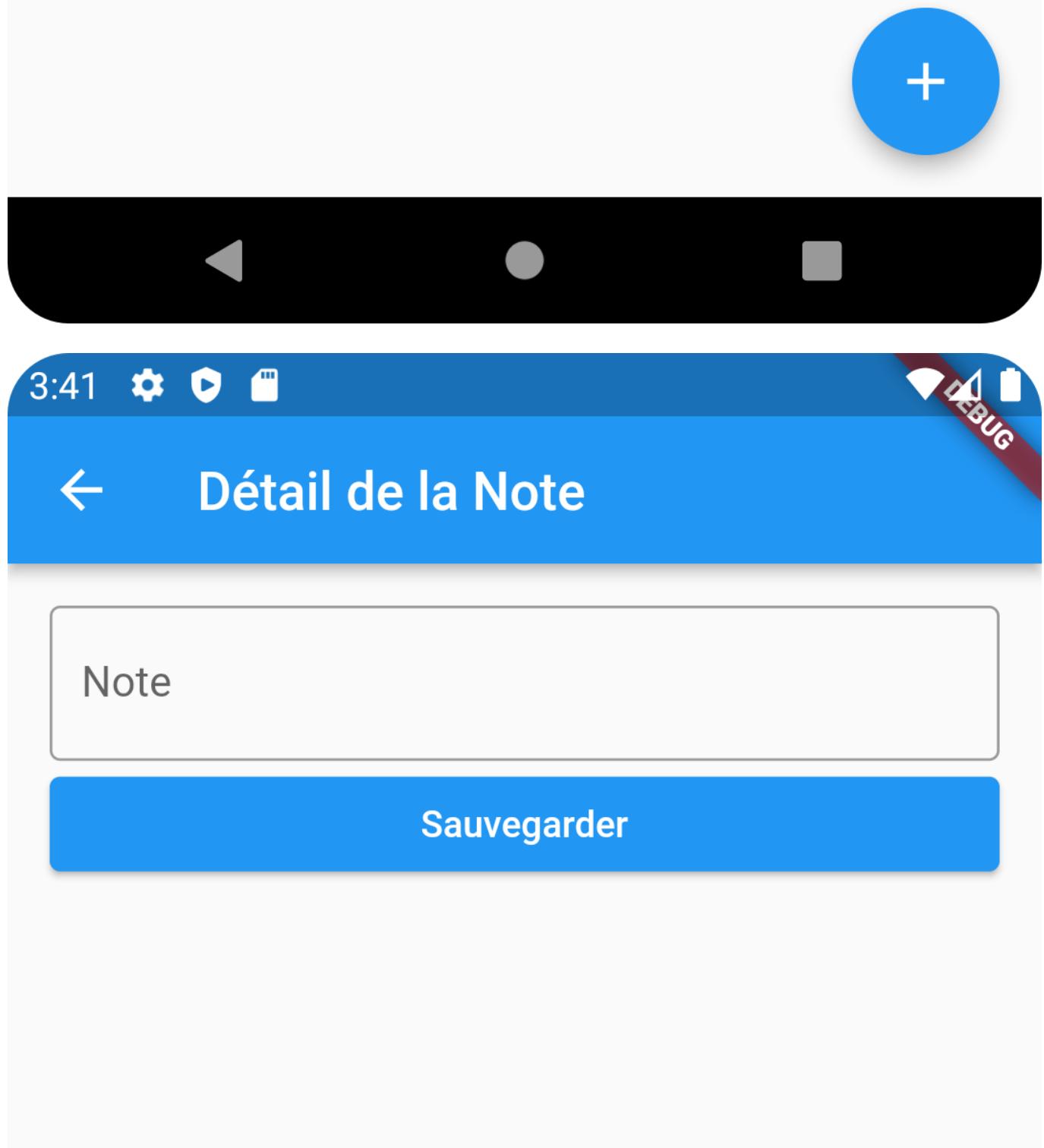
Note 5

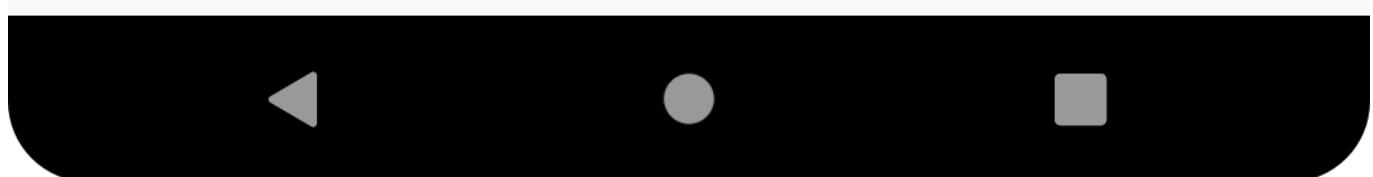
Note 6

Note 7

Note 8

Note 9





Liens Utiles

1. [Widget catalog](#)
2. [Layouts in Flutter](#)
3. [Add interactivity to your Flutter app](#)
4. [Understanding constraints](#)
5. [Work with long lists](#)
6. [Create a grid list](#)

7. [Creating responsive and adaptive apps](#)
8. [Taps, drags, and other gestures](#)
9. [Build a form with validation](#)
10. [Create and style a text field](#)
11. [Navigation and Routing](#)
12. [Navigate with named routes](#)
13. [Send data to a new screen](#)