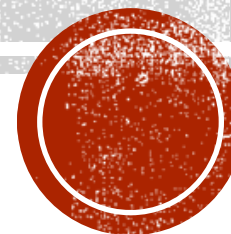


FLUTTER



PROGRAMME

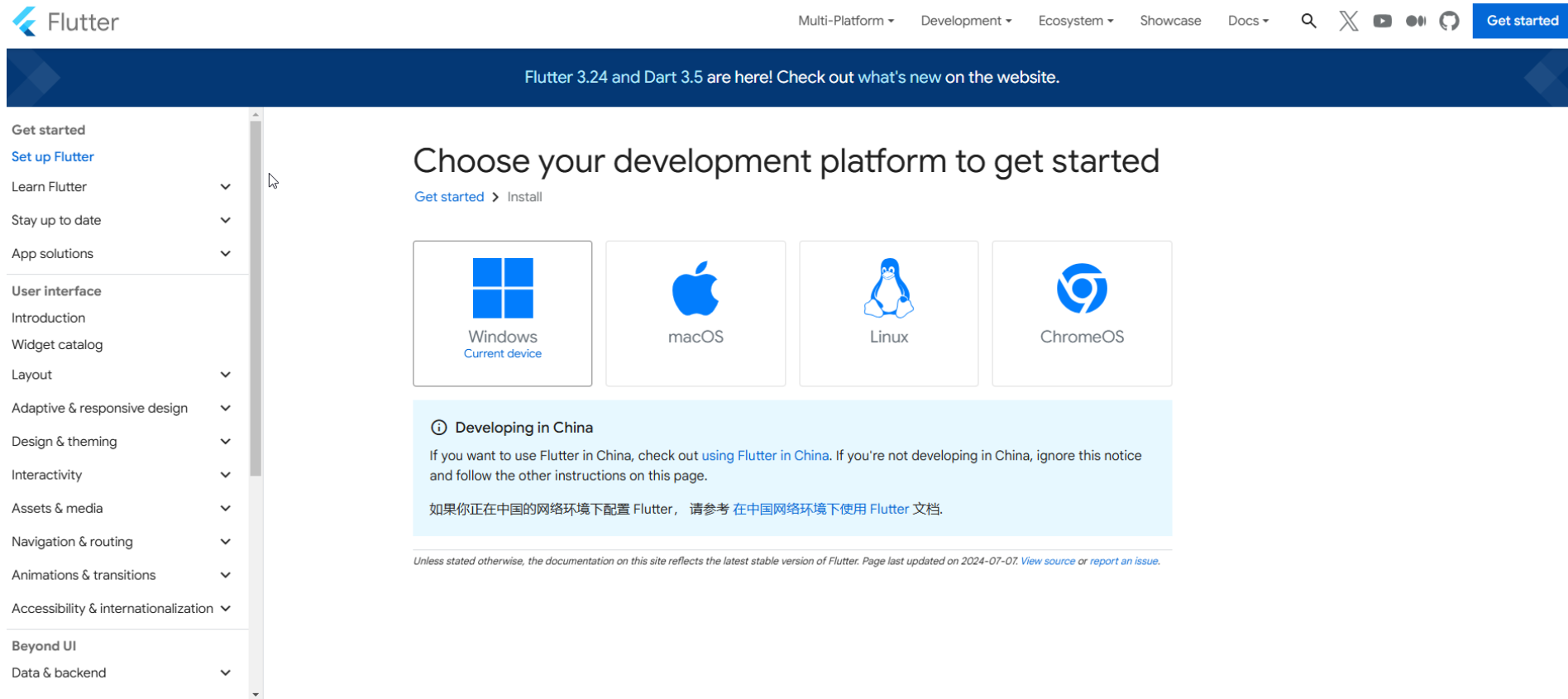
- 1. Introduction à Flutter
- 2. Les bases de Dart pour Flutter
- 3. Introduction aux widgets
- 4. Widgets de Layout
- 5. Introduction à Material Design
- 6. Stateless, Stateful, Card, Ink et InkWell
- 7. Widgets relatifs aux listes
- 8. Date, BottomNavigationBar et cycle des widgets
- 9. Approfondissement sur le fonctionnement de Flutter
- 10. La navigation
- 11. Utilisation du protocole HTTP
- 12. Les formulaires



INSTALLATION

- Pour installer flutter il faut suivre les étapes de la documentation officiel

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



The screenshot shows the Flutter documentation website. At the top, there's a navigation bar with the Flutter logo, a search icon, and links for Multi-Platform, Development, Ecosystem, Showcase, Docs, and a 'Get started' button. Below the navigation bar is a blue banner announcing 'Flutter 3.24 and Dart 3.5 are here! Check out what's new on the website.' The main content area is titled 'Choose your development platform to get started' with a breadcrumb 'Get started > Install'. There are four large buttons for different platforms: Windows (labeled 'Current device'), macOS, Linux, and ChromeOS. Below these buttons is a light blue box with a notice titled 'Developing in China' in Chinese and English. At the bottom, there's a small disclaimer: 'Unless stated otherwise, the documentation on this site reflects the latest stable version of Flutter. Page last updated on 2024-07-07. View source or report an issue.'

Flutter

Multi-Platform ▾ Development ▾ Ecosystem ▾ Showcase Docs ▾ 🔍 X YouTube GitHub

Get started

Flutter 3.24 and Dart 3.5 are here! Check out what's new on the website.

Get started
Set up Flutter
Learn Flutter ▾
Stay up to date ▾
App solutions ▾
User interface
Introduction
Widget catalog
Layout ▾
Adaptive & responsive design ▾
Design & theming ▾
Interactivity ▾
Assets & media ▾
Navigation & routing ▾
Animations & transitions ▾
Accessibility & internationalization ▾
Beyond UI
Data & backend ▾

Choose your development platform to get started

Get started > Install

Windows
Current device

macOS

Linux

ChromeOS

📌 Developing in China
If you want to use Flutter in China, check out [using Flutter in China](#). If you're not developing in China, ignore this notice and follow the other instructions on this page.
如果你正在中国的网络环境下配置 Flutter, 请参考 [在中国网络环境下使用 Flutter](#) 文档.

Unless stated otherwise, the documentation on this site reflects the latest stable version of Flutter. Page last updated on 2024-07-07. [View source](#) or [report an issue](#).



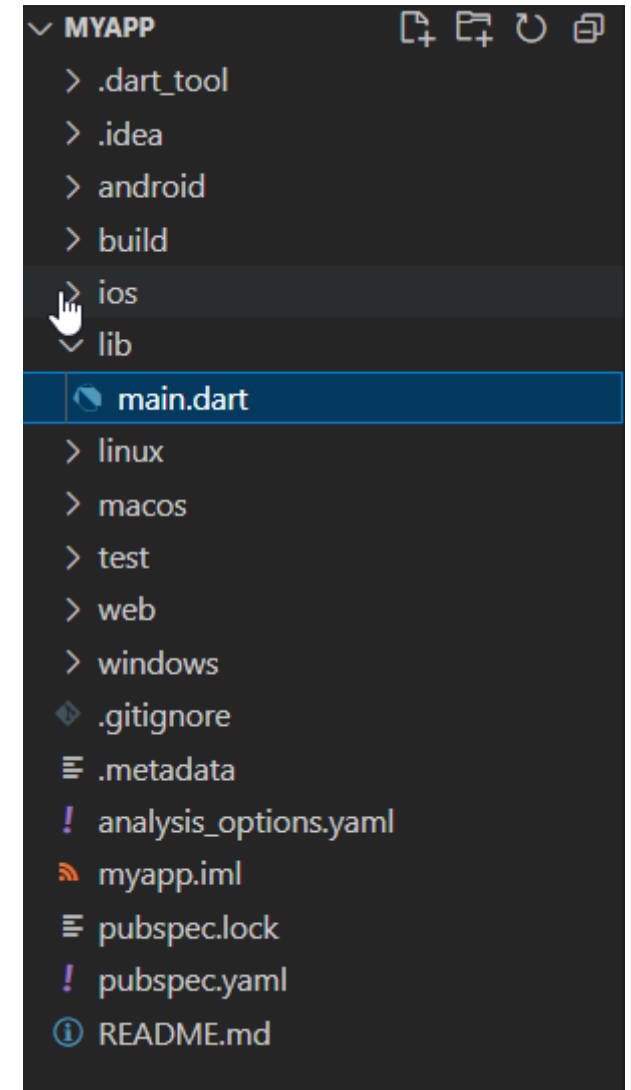
CRÉATION D'UN PROJET

- Une fois visual studio installer on lance la cmd suivante pour créer un nouveau projet
- `Flutter create NomProjet`



STRUCTURE DES DOSSIERS

- `.idea` : contient une configuration de android studio
- `android` : contient un projet android complet, c'est le projet que flutter sdk utilisera pour fusionner avec le code flutter
- `Build`: contient la sortie de l'application
- `Lib` : c'est le dossier ou on fait tout le travail
- `pubspec.yaml`: Fichier de configuration pour les dépendances, les assets et les métadonnées du projet.



WIDGETS

- Un **widget** est l'élément de base de toute application Flutter. Imaginez les widgets comme des **briques de Lego** : chaque widget est une pièce individuelle que vous assemblez pour construire l'interface de votre application. Tout, de la mise en page au texte en passant par les boutons, est un widget dans Flutter.
- Pour explorer les différents types de widgets disponibles, visitez le **catalogue officiel des widgets** de Flutter :
[Flutter Widget Catalog](#).



NOTRE PREMIER WIDGET

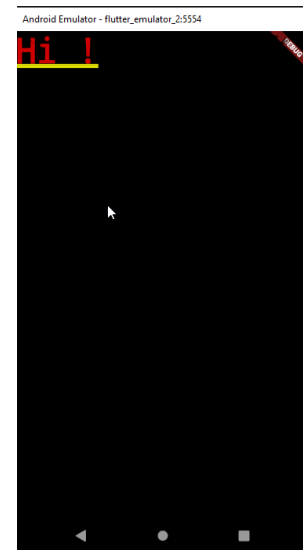
- La fonction `main()` :
 - Rôle : C'est le point d'entrée de votre application Flutter.
 - `runApp()` : Cette fonction lance l'application en affichant le widget passé en argument (ici, `MyApp`).
- La classe `MyApp` :
 - Pourquoi une classe ? En Flutter, toute l'interface utilisateur est construite à l'aide de widgets, qui sont des classes. Nous créons ici un widget personnalisé appelé `MyApp`.
 - Héritage : `MyApp` hérite de `StatelessWidget`, un type de widget utilisé pour afficher des interfaces statiques.
- L'annotation `@override` :
 - C'est quoi ? L'annotation `@override` indique que nous modifions ou "redéfinissons" une méthode existante de la classe parent (`StatelessWidget`).
 - Pourquoi ? La méthode `build` est déjà définie dans la classe `StatelessWidget`. En la redéfinissant dans `MyApp`, nous spécifions ce que ce widget doit afficher.
- La méthode `build()` :
 - Rôle : Obligatoire pour tous les widgets. Elle retourne l'interface utilisateur du widget.
 - Paramètre `BuildContext` : Fournit des informations sur l'environnement dans lequel le widget est construit. Vous n'avez pas besoin de le modifier pour l'instant. Retour d'un widget : Ici, elle retourne un `MaterialApp`.
- Le widget `MaterialApp` :
 - Rôle : Structure de base de l'application Flutter. Il configure des éléments comme le design, les thèmes, et la navigation.
 - Argument `home` : Définit le widget affiché au démarrage. Ici, nous affichons un simple widget `Text`.
- Le widget `Text` :
 - Rôle : Affiche du texte à l'écran. Dans cet exemple, le texte affiché est "Hi!".

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

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

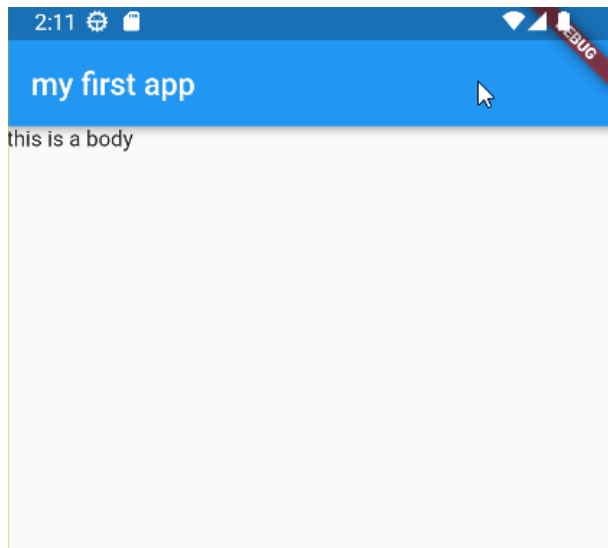
class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
      home: Text('Hi !'),
    );
  }
}
```



SCAFFOLD

- Pour changer l'apparence du texte, il va nous falloir un autre widget,
- Il s'agit du widget « Scaffold », le Scaffold permet de structurer un écran (l'équivalent d'une page web), par exemple d'avoir une barre en haut, un bouton d'action en bas à droite et le contenu au milieu.
- Scaffold possède des arguments nommés
- Pour notre exemple on utilise l'argument appBar pour notre barre de haut et body pour notre contenu
- Les deux arguments ont comme valeur le widget « Text » pour afficher du texte



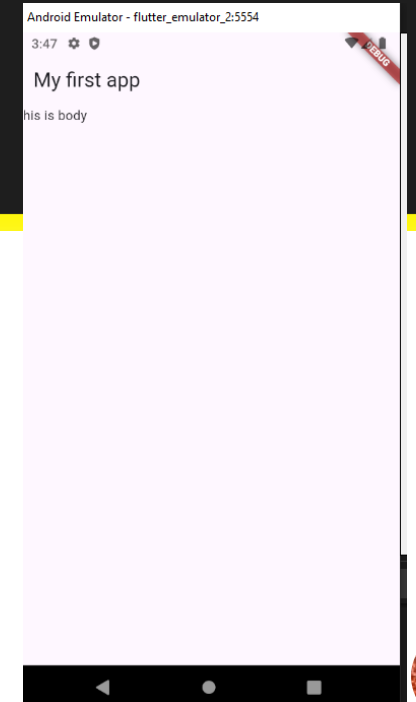
```
class MyApp extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      home: Scaffold(  
        appBar: AppBar(  
          title: Text('my first app'),  
        ),  
        body: Text('this is a body'),  
      ),  
    );  
  }  
}
```



SCAFFOLD

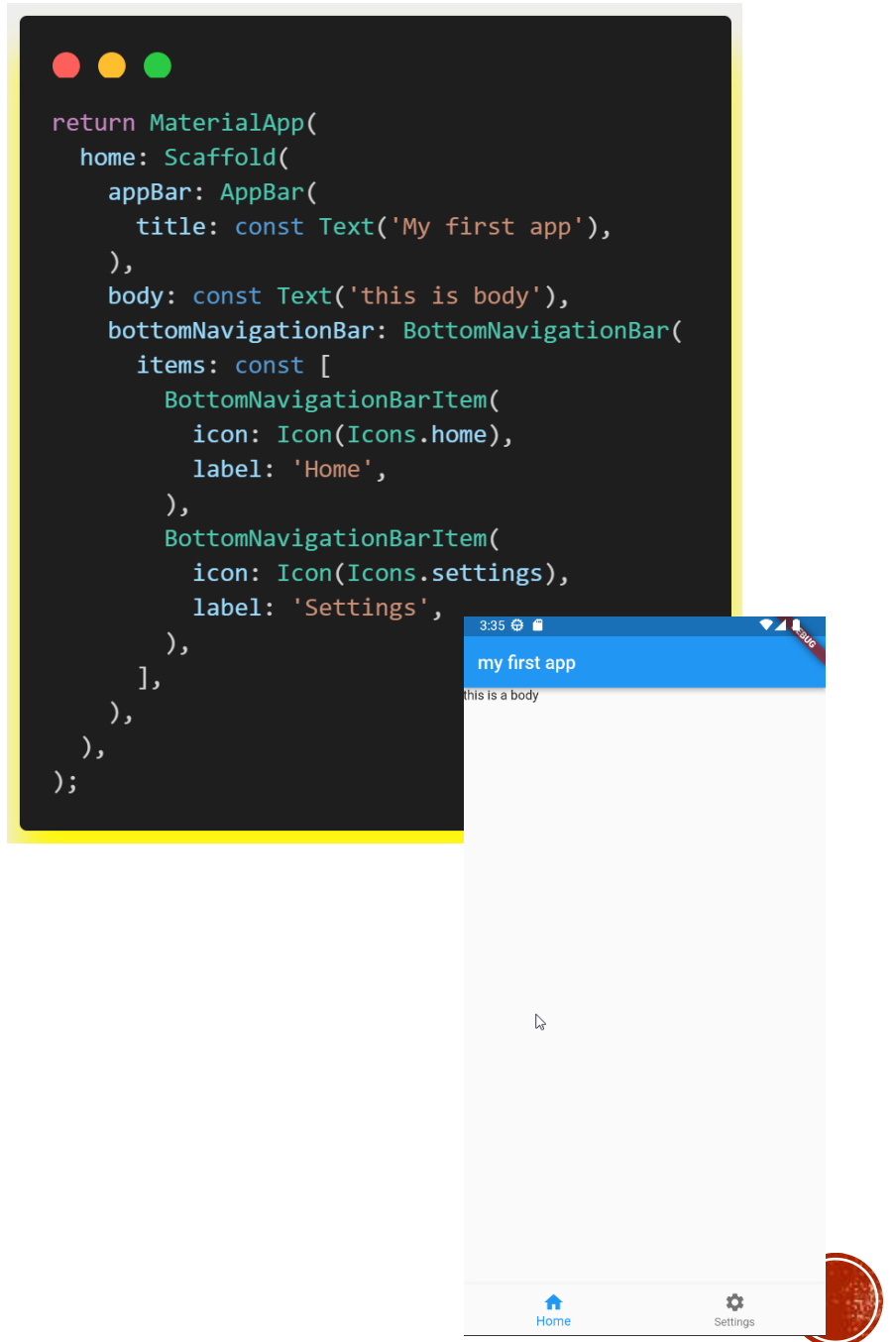
- Pourquoi Scaffold ?
 - Par défaut, le texte affiché dans MaterialApp apparaît sur un fond noir et peut ne pas être lisible.
 - Le widget Scaffold est utilisé pour structurer un écran complet, un peu comme une "page web". Il permet d'ajouter : Une barre en haut (AppBar). Du contenu principal au milieu (body).
 - Et d'autres éléments comme des boutons d'action en bas à droite.
- Les arguments principaux de Scaffold :
 - appBar : Définit la barre en haut de l'écran (comme un en-tête).
 - Dans cet exemple, nous utilisons un widget AppBar avec un titre (Text('My first app')).
 - body : Contient le contenu principal de la page. Ici, nous utilisons un widget Text pour afficher le message "This is body".

```
class MyApp extends StatelessWidget {  
  const MyApp({super.key});  
  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      home: Scaffold(  
        appBar: AppBar(  
          title: const Text('My first app'),  
        ),  
        body: const Text('this is body'),  
      ),  
    );  
  }  
}
```



BUTTONNAVIGATIONBAR

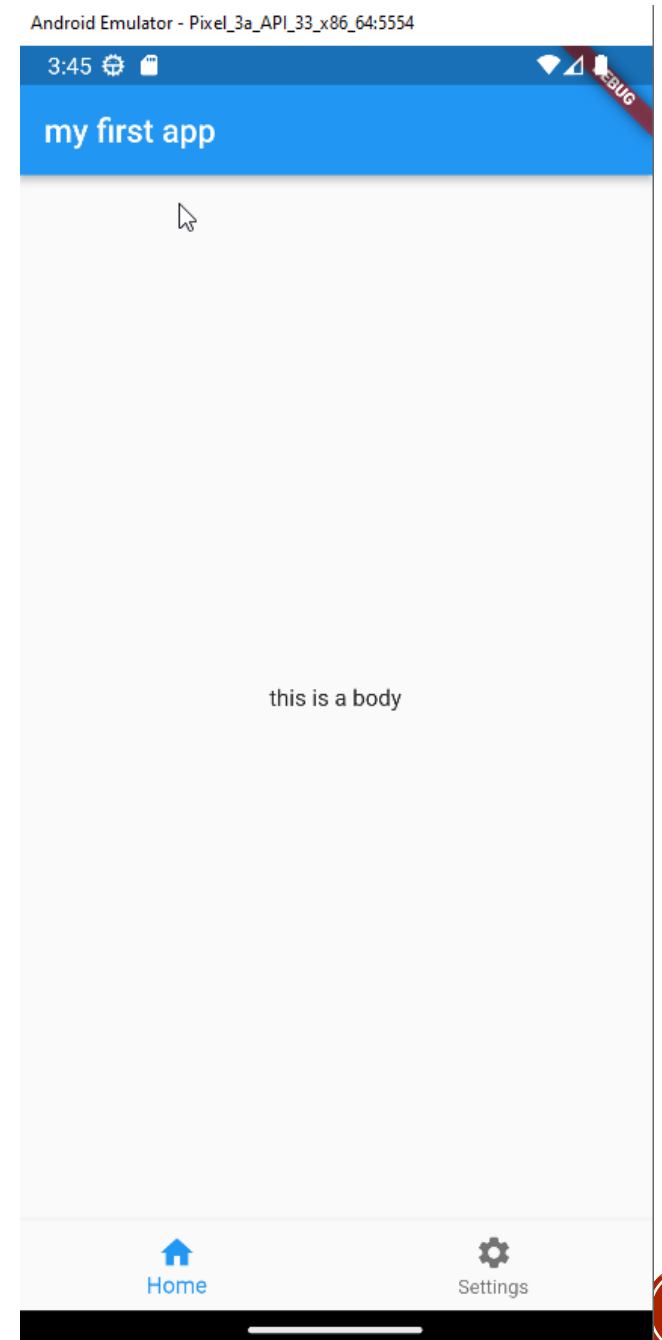
- La barre de navigation en bas permet de naviguer entre différentes sections ou pages de l'application. C'est une structure courante dans les applications modernes.
- Comment l'ajouter ?
 - On utilise l'argument nommé `bottomNavigationBar` dans le widget `Scaffold`.
 - Sa valeur est un widget `BottomNavigationBar`, qui est conçu spécialement pour afficher une barre de navigation en bas.
- Le widget `BottomNavigationBar` :
 - `items` : Cet argument contient une liste de widgets `BottomNavigationBarItem`.
 - Chaque `BottomNavigationBarItem` représente un bouton dans la barre.
 - Il doit inclure :
 - `icon` : Un widget `Icon` pour l'icône.
 - `label` : Une chaîne de caractères pour le texte.
- Dans l'exemple :
 - La barre de navigation contient deux éléments :
 - Une icône "Home" avec le label "Home".
 - Une icône "Settings" avec le label "Settings".
- Vous pouvez ajouter d'autres éléments si nécessaire, mais au moins deux sont obligatoires.



CENTRER LE BODY

- On utilise le widget center pour Center le body


```
return MaterialApp(  
  home: Scaffold(  
    appBar: AppBar(  
      title: const Text('my first app'),  
    ), // AppBar  
    body: const Center(child: Text('this is a body')),  
    bottomNavigationBar: BottomNavigationBar(  
      items: const [  
        BottomNavigationBarItem(  
          label: 'Home',  
          icon: Icon(Icons.home),  
        ), // BottomNavigationBarItem  
        BottomNavigationBarItem(  
          label: 'Settings',  
          icon: Icon(Icons.settings),  
        ), // BottomNavigationBarItem  
      ],  
    ), // BottomNavigationBar  
  ), // Scaffold  
); // MaterialApp
```





- Pour remanier plus rapidement clique droit sur le widget (refactor en anglais)

Atteindre la définition	F12
Atteindre la définition du type	
Atteindre les implémentations	Ctrl+F12
Atteindre les références	Maj+F12
Aperçu	>
Rechercher toutes les références Maj+Alt+F12	
Rechercher toutes les implémentations	
Renommer le symbole F2	
Modifier toutes les occurrences Ctrl+F2	
Mettre en forme le document avec...	
Mettre le document en forme Maj+Alt+F	
Remanier...	Ctrl+Maj+R
Action de la source	
Couper	Ctrl+X
Copier	Ctrl+C
Coller	Ctrl+V
Ajouter un point d'arrêt Inline Maj+F9	
Palette de commandes... Ctrl+Maj+P	


Extraire...


 Extract Method


 Extract Local Variable


 Extract Widget


Plus d'actions...


 Wrap with widget...


 Wrap with Builder


 **Wrap with Center**


 Wrap with Column

 Wrap with Container

 Wrap with Padding

 Wrap with Row

 Wrap with SizedBox

 Wrap with StreamBuilder

Enter à A



MATERIAL DESIGN

- Material Design :
 - Un système de design moderne et intuitif créé par Google.
 - Flutter propose des widgets prêts à l'emploi basés sur Material Design.
- Widgets Importants :
 - MaterialApp : Point de départ pour suivre Material Design.
 - AppBar : Barre supérieure.
 - Scaffold : Structure de la page.
 - FloatingActionButton : Bouton d'action flottant.
 - BottomNavigationBar : Barre de navigation en bas.
- Personnalisation :
 - Utilisez ThemeData pour définir des couleurs et des styles cohérents dans toute l'application.

```
return MaterialApp(  
  theme: ThemeData(  
    primarySwatch: Colors.teal, // C  
    couleur principale de l'application  
    appBarTheme: const AppBarTheme(  
      backgroundColor: Colors.teal,  
      // Couleur de l'AppBar  
      foregroundColor: Colors.white,  
      // Couleur du texte dans l'AppBar  
    ),  
  ),  
  home: Scaffold(  

```



BUTTON

- Flutter fournit plusieurs widgets pour créer des **boutons** interactifs. Voici 3 types de boutons couramment utilisés et comment les implémenter.
 - ElevatedButton :
 - Bouton avec un fond surélevé (effet d'ombre). Utilisé pour des actions importantes sur une page.
 - TextButton
 - Bouton simple avec un texte, sans fond.
 - Utilisé pour des actions secondaires ou dans un contexte où un style minimaliste est suffisant.
 - IconButton
 - Bouton avec une icône uniquement (pas de texte).
 - Souvent utilisé pour des actions spécifiques, comme la recherche, les paramètres, ou la navigation.

Pour affiché des message sur la consol on utilise la methode print

A light purple rounded rectangle button with a subtle shadow and the text "Elevated Button" in a dark purple font.

Elevated Button

```
body: Center(  
  child: ElevatedButton(  
    onPressed: () {  
      print('ElevatedButton Pressed');  
    },  
    child: Text('Elevated Button'),  
  ),  
)
```

A light purple rounded rectangle button with the text "Text Button" in a dark purple font.

Text Button

```
body: Center(  
  child: TextButton(  
    onPressed: () {  
      print('TextButton Pressed');  
    },  
    child: const Text('Text Button'),  
  ),  
)
```



```
body: Center(  
  child: IconButton(  
    onPressed: () {  
      print('IconButton Pressed');  
    },  
    icon: const Icon(Icons.thumb_up),  
  ),  
)
```



CRÉER UNE VARIABLE

Dans ce code, nous avons une variable `buttonName` qui contient le texte affiché sur le bouton : "click me". Cependant, si vous cliquez sur ce bouton, **rien ne se passera en apparence**, car nous utilisons un **StatelessWidget**.

- Avec un **StatelessWidget**, Flutter ne détecte pas les changements de la variable `buttonName`, car un **StatelessWidget** ne peut pas "reconstruire" l'interface utilisateur dynamiquement.
- Si vous voulez que cliquer sur le bouton modifie le texte affiché, vous devrez utiliser un **StatefulWidget**, qui permet de gérer des variables modifiables et force la reconstruction du widget.
- En Flutter, le mot-clé `const` est utilisé pour indiquer que l'instance d'un widget (ou tout autre objet) ne changera jamais.
- Cependant, comme nous avons ajouté une **variable mutable** (`buttonName`) dans `MyApp`, nous avons dû supprimer `const`. Pourquoi ? Parce que l'utilisation de `const` est incompatible avec des variables qui peuvent être modifiées.

```
class MyApp extends StatelessWidget {  
  MyApp({super.key});  
  String buttonName = 'click me';  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
  
      home: Scaffold(  
        appBar: AppBar(  
          title: const Text('My first app'),  
        ),  
        body: Center(  
          child: ElevatedButton(  
            onPressed: () {  
              buttonName = 'clicked';  
            },  
            child: Text(buttonName),  
          ),  
        ),  
      ),  
    );  
  }  
}
```

C'est une bonne pratique d'exposer la possibilité de fournir une clé lors de la création de widgets publics.

- **const MyApp** : Crée une instance immuable de `MyApp`.
- **{super.key}** : Transmet une clé unique à la classe parent pour une gestion optimale des widgets.
- **Pourquoi c'est important ?**
 - Optimise les performances de Flutter en évitant les reconstructions inutiles.
 - Permet à Flutter d'identifier précisément chaque widget dans l'arbre.



STATEFULWIDGET OU STATELESSWIDGET

- **Qu'est-ce qu'un StatelessWidget ?**
- Un StatelessWidget est utilisé lorsque l'interface utilisateur ne change pas (pas de mise à jour dynamique à l'écran).
- Exemple : Un écran affichant uniquement du texte ou des boutons statiques.
- **Pourquoi l'utiliser ?**
 - C'est plus léger pour les ressources de votre application. Si un élément de l'interface n'a pas besoin de changer, utilisez toujours un StatelessWidget.
- **Qu'est-ce qu'un StatefulWidget ?**
- Un StatefulWidget est utilisé lorsque l'interface doit être mise à jour dynamiquement.
 - Exemple : Si vous voulez changer la couleur ou le texte en cliquant sur un bouton.
- **Comment fonctionne-t-il ?**
 - Un StatefulWidget fonctionne avec un objet **State** qui gère les changements d'état (par exemple, la couleur ou le texte modifié).
- **Rôle de la Méthode build :**
- La méthode build est présente dans les deux types de widgets (StatelessWidget et StatefulWidget).
- Elle est appelée pour **construire l'interface utilisateur** et afficher ce que le widget doit rendre à l'écran.
- **Pour un StatefulWidget :** La méthode build est appelée à chaque fois que l'état change, ce qui permet de mettre à jour l'écran en fonction des nouvelles valeurs.
- **Quand choisir StatelessWidget ou StatefulWidget ?**
- **StatelessWidget :** Pour les éléments fixes et statiques qui ne changent jamais (par exemple, un titre ou un logo).
- **StatefulWidget :** Pour les éléments interactifs ou qui doivent réagir aux actions de l'utilisateur (par exemple, un compteur qui s'incrémente lorsqu'on clique sur un bouton).



- Le but maintenant est de utiliser statefullW pour changer la variable « `buttonName` »
- **Conversion d'un StatelessWidget en StatefulWidget :**
- Faites un clic droit sur la classe, puis sélectionnez **Refactor → Convert to StatefulWidget.**
- Cela génère automatiquement deux classes :
 - **MyApp** : Le point d'entrée fixe.
 - **_MyAppState** : Gère les changements d'état et la méthode build
- **StatefulWidget** : Séparé en deux classes :
 - Une classe fixe (le point d'entrée).
 - Une classe dynamique (pour gérer les états et changements).
- **createState()** : Associe un widget (MyApp) à sa classe d'état (_MyAppState).
- **setState()** : Rafraîchit l'interface utilisateur lorsque l'état change.

```
class MyApp extends StatefulWidget {  
  const MyApp({super.key});  
  
  @override  
  State<MyApp> createState() => _MyAppState();  
}  
  
class _MyAppState extends State<MyApp> {  
  String buttonName = 'click me';  
  
  @override  
  Widget build(BuildContext context) {
```

```
child: ElevatedButton(  
  onPressed: () {  
    setState(() {  
      buttonName = 'clicked';  
    });  
  },
```

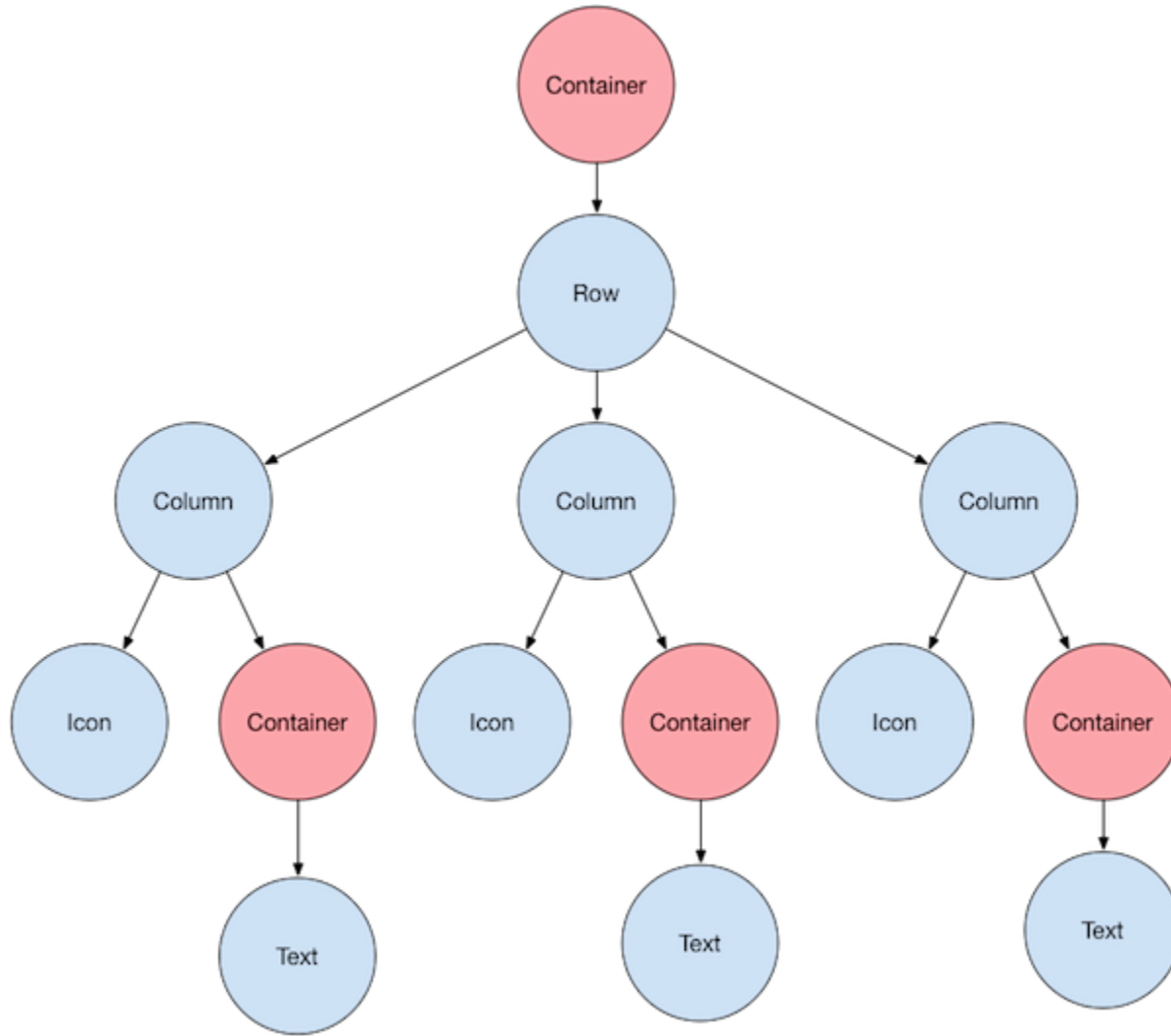


WIDGETS DE LAYOUT - MISE EN PAGE

- Les widgets sont au cœur du mécanisme de mise en page de Flutter. Dans Flutter, presque tout est un widget, même les modèles de mise en page sont des widgets. Les images, les icônes et le texte que vous voyez dans une application Flutter sont tous des widgets. Mais les éléments que vous ne voyez pas sont également des widgets, tels que les lignes, les colonnes et les grilles qui organisent, contraignent et alignent les widgets visibles
- Vous créez une mise en page en composant des widgets pour créer des widgets plus complexes. Par exemple, la première capture d'écran ci-dessous montre 3 icônes avec une étiquette sous chacune .:



- Voici un schéma de l'arborescence des widgets pour cette interface utilisateur :



- Dans cet exemple, chaque Textwidget est placé dans un Container pour ajouter des marges. L'ensemble Row est également placé dans un Container pour ajouter un rembourrage autour de la ligne.



DISPOSEZ PLUSIEURS WIDGETS VERTICALEMENT ET HORIZONTALEMENT

- L'un des modèles de mise en page les plus courants consiste à organiser les widgets verticalement ou horizontalement. Vous pouvez utiliser un Row widget pour organiser les widgets horizontalement et un Column widget pour organiser les widgets verticalement.
- Row et Column sont deux des modèles de mise en page les plus couramment utilisés.
- Row et Column chacun prend une liste de widgets enfants.
- Un widget enfant peut lui-même être un Row, Column ou un autre widget complexe.
- Vous pouvez spécifier comment un Row ou Column aligne ses enfants, à la fois verticalement et horizontalement.
- Vous pouvez étirer ou contraindre des widgets enfants spécifiques.
- Vous pouvez spécifier comment les widgets enfants utilisent l' espace disponible de Row ou Column



Cette mise en page est organisée sous la forme d'un fichier Row. La ligne contient deux enfants : une colonne à gauche et une image à droite :

L'arborescence des widgets de la colonne de gauche imbrique les lignes et les colonnes.

Row

2 children

child: new Column

child: new Image

Column

4 children

Strawberry Pavlova

Pavlova is a meringue-based dessert named after the Russian ballerine Anna Pavlova. Pavlova features a crisp crust and soft, light inside, topped with fruit and whipped cream.

★★★★★

170 Reviews

🕒

PREP:

25 min

🕒


COOK:

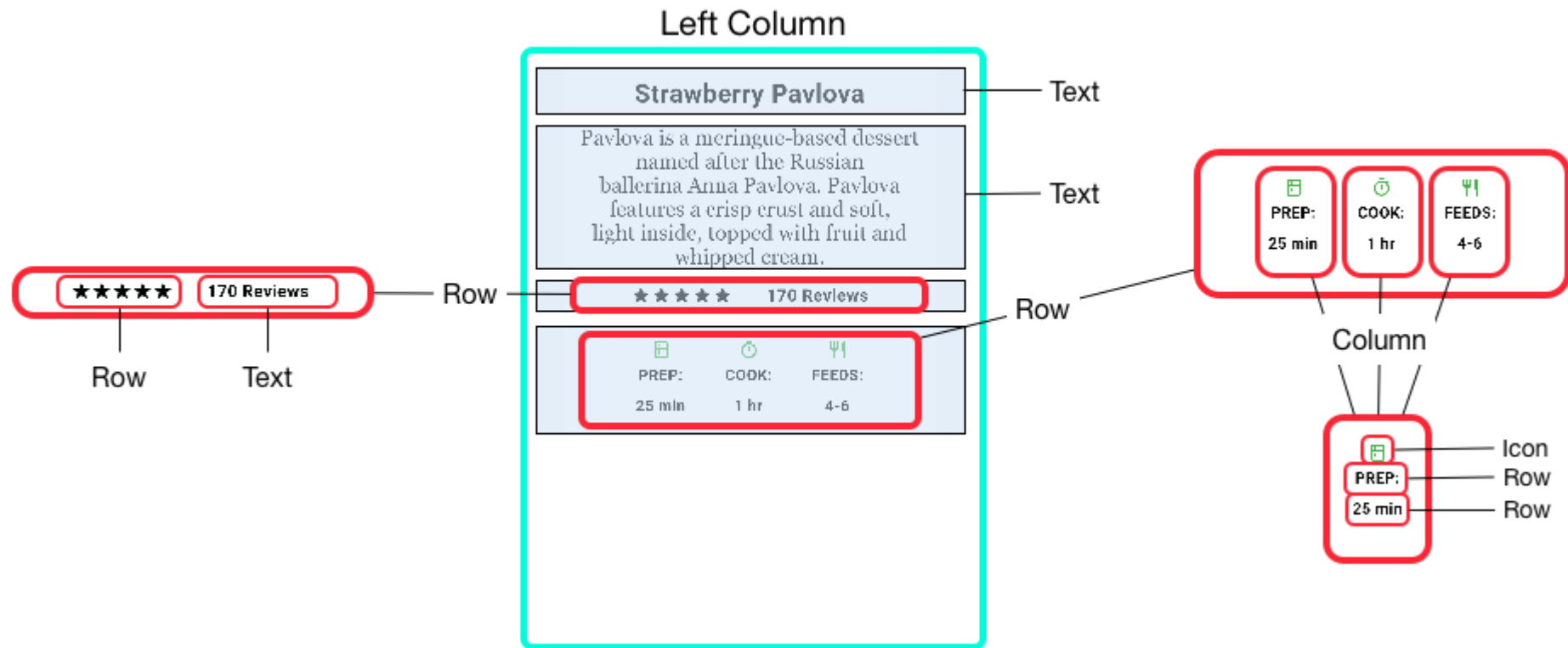
1 hr

🍴

FEEDS:

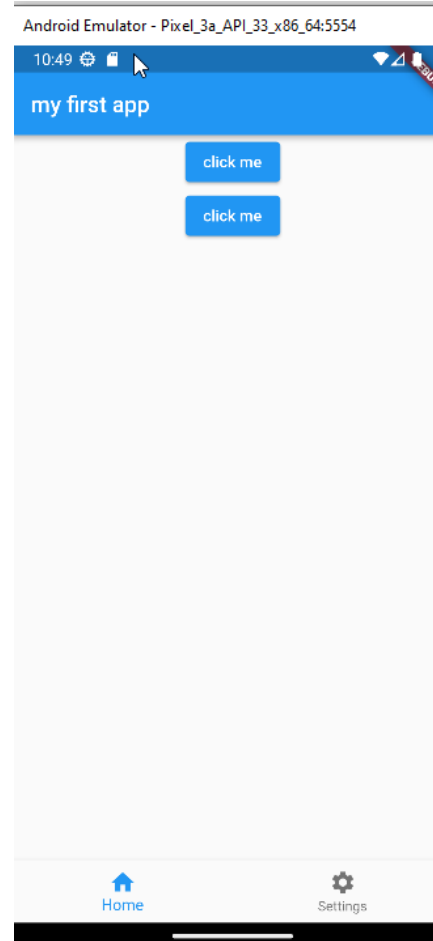
4-6





COLUMN

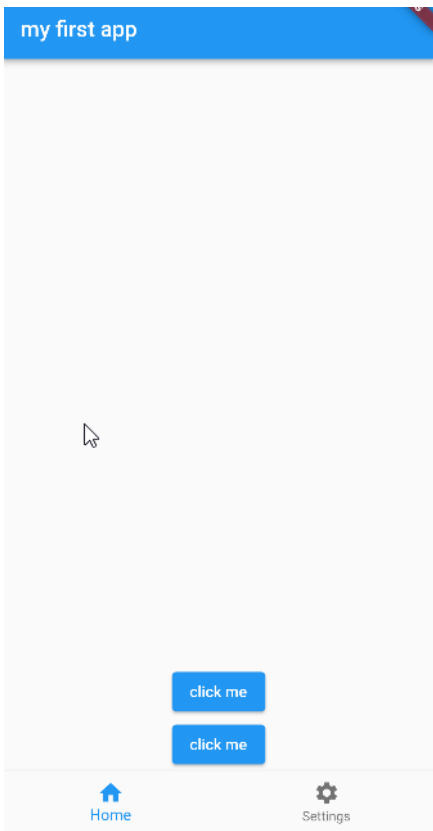
```
body: Center(  
  child: Column(  
    children: [  
      ElevatedButton(  
        onPressed: () {  
          setState(() {  
            buttonName = "clicked";  
          });  
        },  
        child: Text(buttonName),  
      ),  
      ElevatedButton(  
        onPressed: () {  
          setState(() {  
            buttonName = "clicked";  
          });  
        },  
        child: Text(buttonName),  
      ),  
    ],  
  ),  
)
```



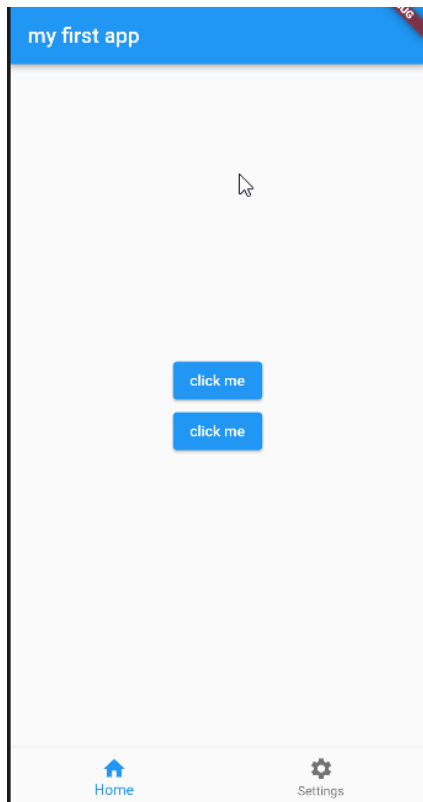
MAINAXISALIGNMENT

```
child: Column(  
  mainAxisAlignment: MainAxisAlignment.valeur, ...
```

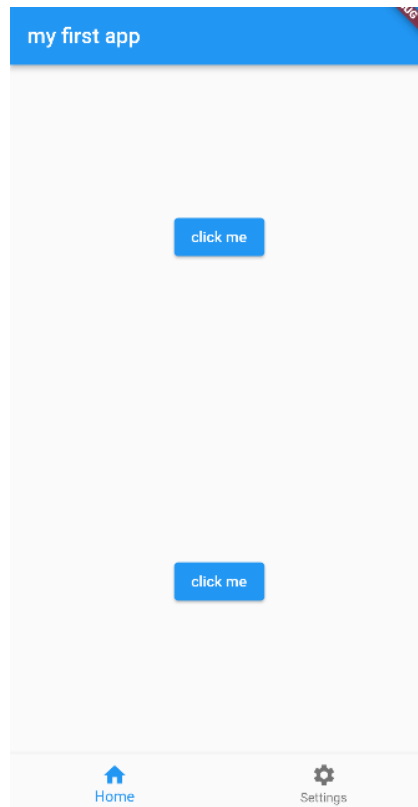
`MainAxisAlignment.end,`



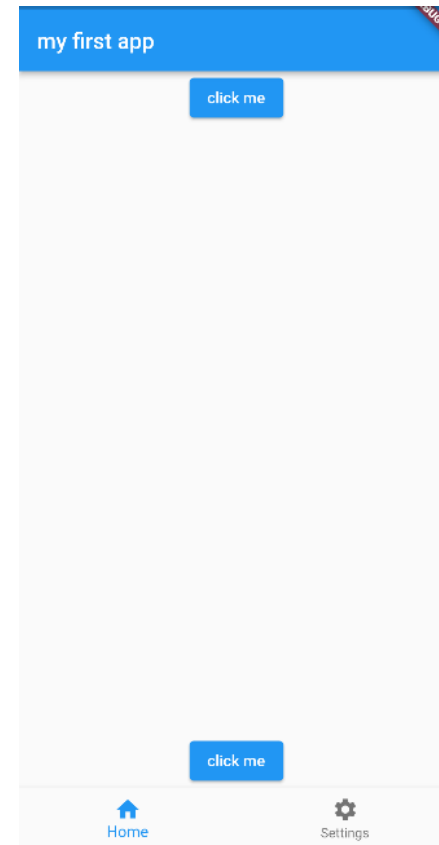
`MainAxisAlignment.center,`



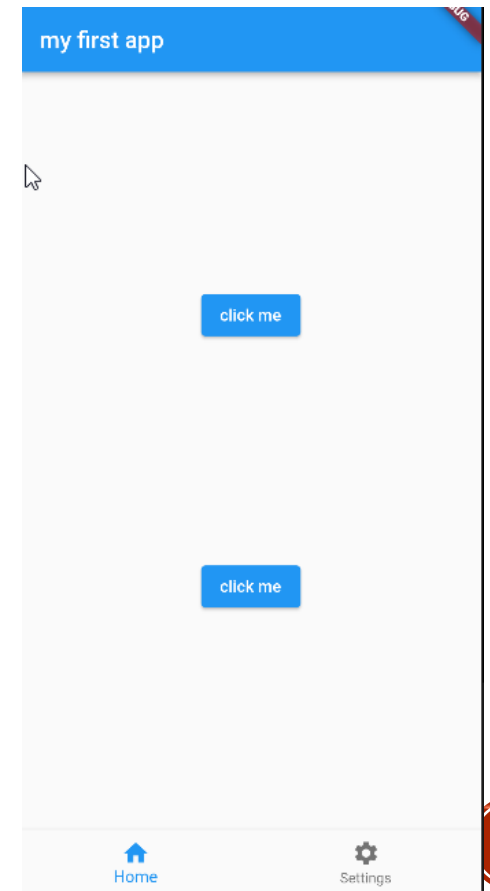
`MainAxisAlignment.spacearound,`



`MainAxisAlignment.spacebetween,`



`MainAxisAlignment.spaceEvenly,`



CROSSAXISALIGNMENT

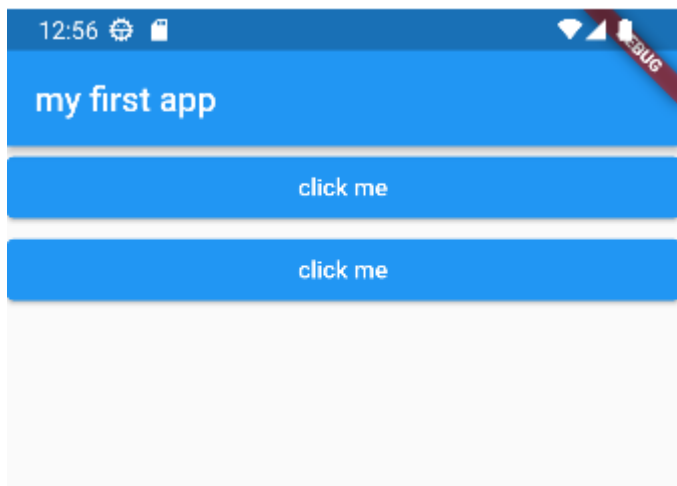
- Etant donnée que notre column n'occupe pas l'intergralité de la largeur on utilise le widget SizedBox

```
body: Center(  
  child: SizedBox(  
    width: double.infinity,  
    child: Column(  
      crossAxisAlignment: CrossAxisAlignment.valeur,  
      children: [  
        ElevatedButton(  

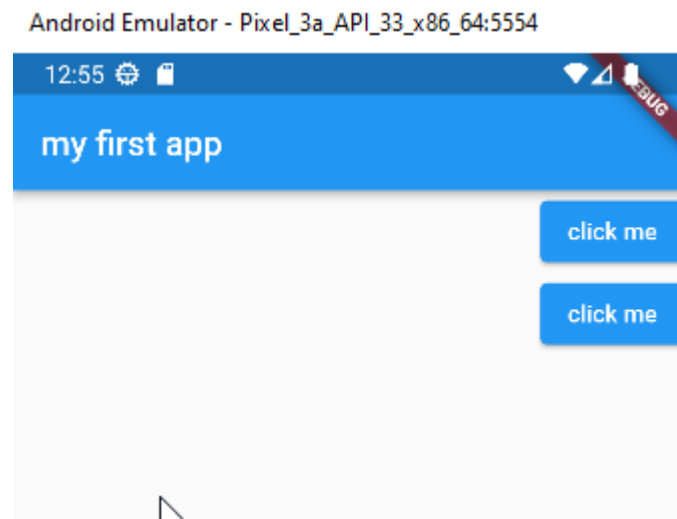
```



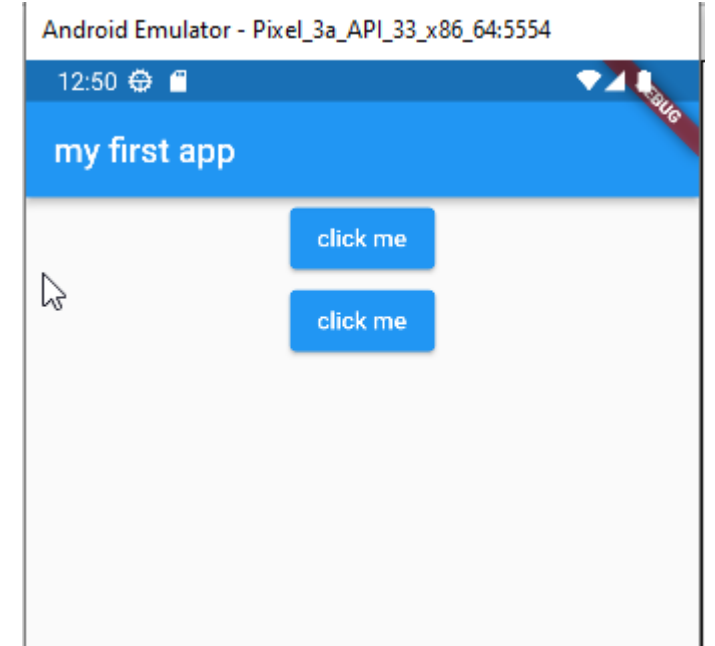
```
crossAxisAlignment: CrossAxisAlignment.stretch,
```



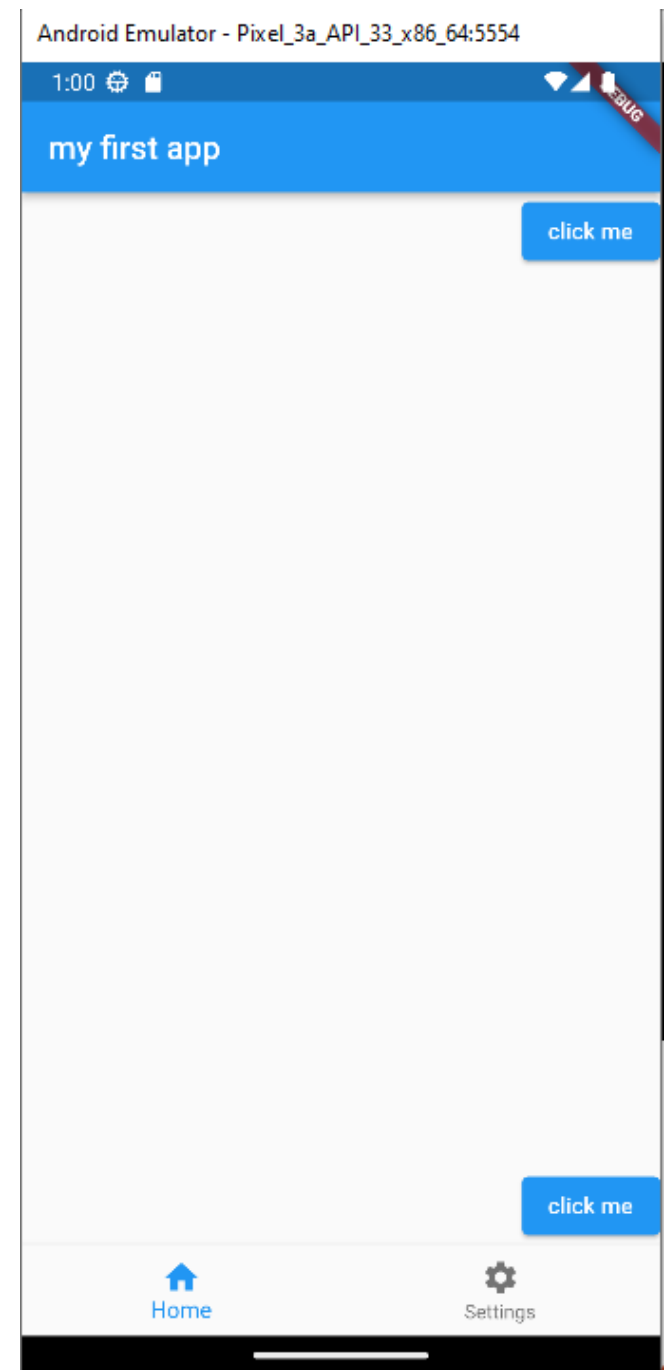
```
crossAxisAlignment: CrossAxisAlignment.end,
```



```
crossAxisAlignment: CrossAxisAlignment.center,
```



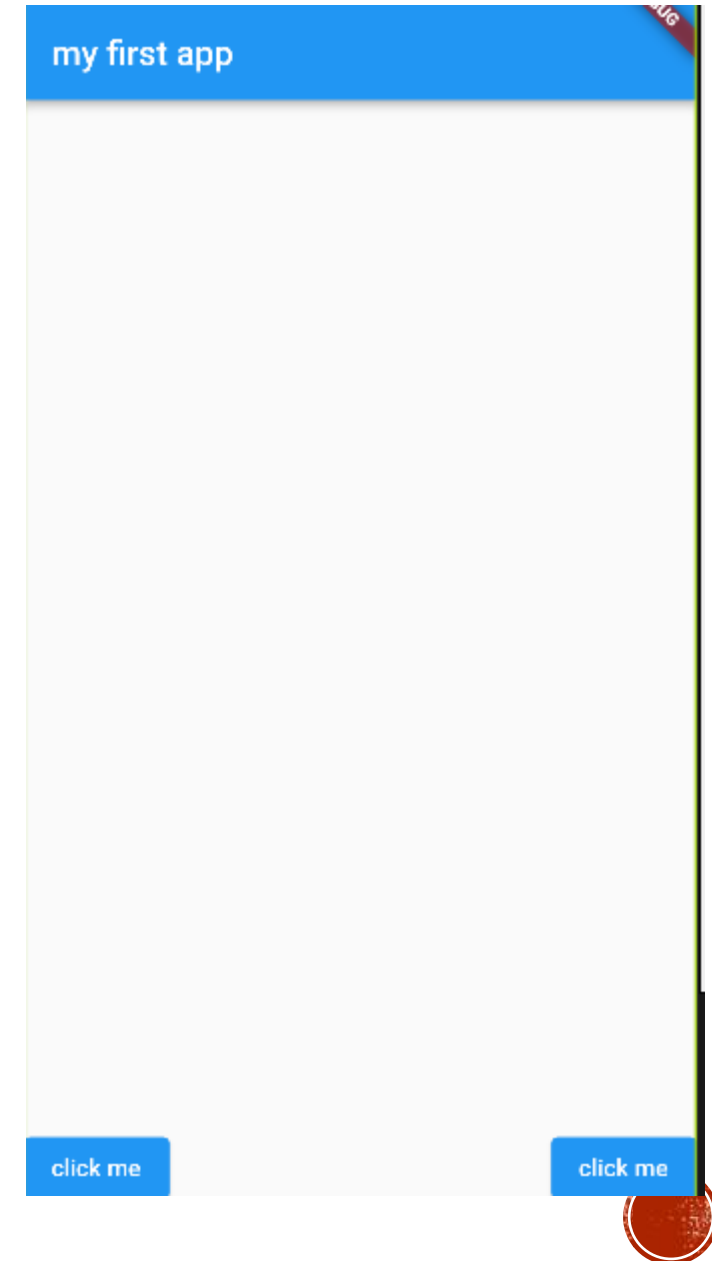
```
mainAxisAlignment: MainAxisAlignment.spaceBetween,  
crossAxisAlignment: CrossAxisAlignment.end,
```



ROW

```
child: SizedBox(  
  width: double.infinity,  
  height: double.infinity,  
  child: Row(  
    mainAxisAlignment: MainAxisAlignment.spaceBetween,  
    crossAxisAlignment: CrossAxisAlignment.end,  
    children: [  
      ElevatedButton(  
        onPressed: () {
```

- Contrairement à column, row affiche les éléments horizontalement



ELEVATEDBUTTON & TEXT STYLE

- On utilise la methode static styleFrom pour modifier les style des button
- Avec vs code si on tape ctrl+espace une liste re propriété s'affiche
- On utilise la methode TextStyle pour changer le style du widget Text



```
ElevatedButton(  
  onPressed: () {  
    setState(() {  
      buttonName = "clicked";  
    });  
  },  
  style: ElevatedButton.styleFrom(  
    backgroundColor: Colors.cyan,  
    foregroundColor: Colors.indigoAccent,  
  ),  
  child: Text(buttonName),  
),
```



```
const Text(  
  'bienvenu',  
  style: TextStyle(  
    backgroundColor: Colors.deepOrange,  
  ),  
),
```



CONTAINER

- Le widget Container est l'un des widgets les plus utilisés et polyvalents dans Flutter. Il sert à décorer, positionner, et dimensionner son widget enfant. Voici ce que vous devez savoir sur le Container et comment l'utiliser efficacement.
- Contrairement à sizebox, container possède plusieurs propriétés hormis la largeur et la hauteur

Le Container peut être considéré comme une "boîte" qui peut contenir un widget enfant. Il fournit plusieurs options pour gérer le positionnement et le stylisme de son contenu ou de lui-même :

1.Décoration : Permet d'ajouter des couleurs, des bordures, des dégradés, et des images.

2.Padding et Margin : Gère l'espace à l'intérieur (padding) et autour (margin) du Container.

3.Dimensionnement : Contrôle la taille du Container, y compris sa largeur et sa hauteur.

4.Alignement : Positionne le widget enfant à l'intérieur du Container selon les axes vertical et horizontal.

```
body: Center(  
  child: Container(  
    width: 200,  
    height: 200,  
    decoration: BoxDecoration(  
      color: Colors.blue,  
      border: Border.all(color: Colors.red, width: 3),  
      borderRadius: BorderRadius.circular(10),  
      boxShadow: [  
        BoxShadow(  
          color: Colors.black.withOpacity(0.5),  
          spreadRadius: 4,  
          blurRadius: 10,  
          offset: Offset(0, 4),  
        )  
      ],  
    ),  
    padding: EdgeInsets.all(  
      20), // Padding: Espace à l'intérieur du Container  
    margin: EdgeInsets.all(20), // Margin: Espace autour du Container  
    child: Text(  
      'Hello, Container!',  
      style: TextStyle(color: Colors.white, fontSize: 20),  
    ),  
  ),  
),
```



WIDGETS DE LISTE

- Les listes sont des composants essentiels dans de nombreuses applications mobiles, permettant de présenter des ensembles de données de manière organisée. Flutter offre plusieurs widgets pour rendre ces listes à la fois belles et fonctionnelles.

Qu'est-ce qu'un Card ?

- Un Card est un conteneur rectangulaire avec des coins arrondis et une ombre, utilisé pour regrouper des informations liées dans un bloc visuel distinct.

Utilisation dans les listes :

- Les Card sont parfaits pour présenter des éléments de liste comme des produits, des articles, ou des profils d'utilisateur, offrant une séparation claire entre les éléments.



```
body: Card(  
  child: Padding(  
    padding: const EdgeInsets.all(16.0),  
    child: Text('Element de la liste'),  
  ),  
),
```

Element de la liste





```
Card(  
  child: InkWell(  
    onLongPress: () => print('Carte touchée'),  
    child: Ink(  
      color: Colors.blue,  
      child: Padding(  
        padding: const EdgeInsets.all(20.0),  
        child: Text('Touchez-moi'),  
      ),  
    ),  
  ),  
)
```

3. Utilisation des Widgets Ink et InkWell

Ink :

- Le widget Ink est utilisé pour ajouter des effets visuels à un Material, comme des dégradés ou des images, tout en conservant les effets d'encre comme le splash color lorsqu'un élément est touché.

InkWell :

- InkWell permet d'ajouter des interactions tactiles aux éléments d'une liste. Il peut être utilisé pour capturer des tapotements, des doubles tapotements, des pressions longues, etc on l'utilise généralement pour pour voir ajouter une interaction a une zone .
- **Interaction** : L'attribut onLongPress est utilisé pour définir une action qui se déclenche lors d'une pression longue par l'utilisateur. Dans ce cas, il affiche un message dans la console.





```
body: ListView.builder(  
  itemCount: 10,  
  itemBuilder: (BuildContext context, int index) {  
    return Card(  
      margin: EdgeInsets.all(10),  
      child: InkWell(  
        onTap: () => print('Carte $index touchée'),  
        child: Padding(  
          padding: const EdgeInsets.all(20.0),  
          child: Row(  
            children: <Widget>[  
              Icon(Icons.star,  
                color: Colors.yellow[700]), // Icône pour embellir  
              SizedBox(width: 20), // Espace entre l'icône et le texte  
              Text('Carte $index',  
                style: TextStyle(fontSize: 16)), // Texte avec style  
            ],  
          ),  
        ),  
      ),  
    );  
  },  
);
```

My first app

★ Carte 0

★ Carte 1

★ Carte 2

★ Carte 3

★ Carte 4

★ Carte 5

★ Carte 6

Home

Settings



NAVIGATION AVEC NAMED ROUTES DANS FLUTTER

- La navigation avec des routes nommées est une approche robuste et facile à gérer pour naviguer entre les écrans dans une application Flutter. Cela permet de définir des chemins par des noms, ce qui facilite la référence aux écrans à travers toute l'application.

Avantages des Routes Nomées

- 1.Centralisation de la Configuration de Navigation** : Toutes les routes sont définies dans un seul endroit, généralement dans le fichier principal de l'application (main.dart).
- 2.Facilité de Maintenance** : Modifier la navigation de l'application devient plus simple car les modifications ne nécessitent des ajustements que dans la configuration des routes.
- 3.Navigation Claire** : Chaque écran peut être appelé par un nom prédéfini, ce qui évite les erreurs de navigation directe.



EXEMPLE DE NAVIGATION AVEC ROUTING NOMINAL DANS FLUTTER


- Dans cet exemple, nous allons créer une application Flutter qui utilise le "named routing" pour naviguer entre deux écrans : un écran d'accueil (HomeScreen) et un écran de paramètres (SettingsScreen). L'écran d'accueil aura une BottomNavigationBar et l'écran de paramètres aura un bouton pour revenir à l'écran d'accueil.
- **Structure des Fichiers**
 - 1.**home_screen.dart** : Contient l'écran d'accueil.
 - 2.**settings_screen.dart** : Contient l'écran de paramètres.
 - 3.**main.dart** : Configure l'application et le routing.





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

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Home"),
      ),
      body: const Center(
        child: Text("Bienvenue sur la page d'accueil"),
      ),
      bottomNavigationBar: BottomNavigationBar(
        items: [
          BottomNavigationBarItem(icon: Icon(Icons.home), label: 'Home'),
          BottomNavigationBarItem(
            icon: Icon(Icons.settings),
            label: 'Settings',
          ),
        ],
        onTap: (index) {
          if (index == 0) {
            Navigator.pushNamed(context, '/settings');
          }
        },
      ),
    );
  }
}
```



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

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

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text("Settings")),
      body: Center(
        child: ElevatedButton(
          onPressed: () => Navigator.pop(context),
          child: const Text("Retourner"),
        ),
      ),
    );
  }
}
```



```
void main() {  
  runApp(MyApp());  
}  
  
class MyApp extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      title: 'Navigation Example',  
      initialRoute: '/',  
      routes: {  
        '/': (context) => HomeScreen(),  
        '/settings': (context) => SettingsScreen(),  
      },  
    );  
  }  
}
```

•Routing Nominal :

- Dans main.dart, les routes sont définies avec initialRoute et un map routes. Chaque route est mappée à une fonction qui construit l'écran correspondant.

•HomeScreen :

- Contient une BottomNavigationBar avec deux items. Lorsque l'item "Settings" est tapé, cela déclenche une navigation vers l'écran de paramètres (SettingsScreen) via Navigator.pushNamed.

•SettingsScreen :

- Contient un bouton qui, lorsqu'il est tapé, retourne à l'écran précédent avec Navigator.pop, ce qui est utile pour gérer le retour à l'écran d'accueil.

