

BUT2 RA Développement mobile - cours 3 :

Idéalement, utiliser VSCode et Linux pour ce cours. Sinon, utilisez DartPad : <https://dartpad.dev/>

- **Initialisation du projet**
 - Créer l'application
 - Lancer l'application
 - Structure d'un projet Flutter
 - Le minimum pour afficher quelque chose
 - Afficher proprement une interface
 - Sans Scaffold / Avec Scaffold
 - Les widgets
 - La documentation officielle
 - Créer un widget custom (1)
 - Créer un widget custom (2)
 - Connaitre les paramètres d'un widget
 - Attention aux anti-patterns
- **Elaboration d'une page**
 - Le dark mode
 - Widget : Container

- Widget : Row
- Widget : Column
- Ajouter des paramètres à notre widget
- Générer du contenu dans une Column
- Widget : TextField
- Erreurs liées aux height et width
- Erreurs liées à l'overflow et au scroll
- **Suite au prochain cours**

Initialisation du projet

Créer l'application

Dans votre terminal, écrivez :

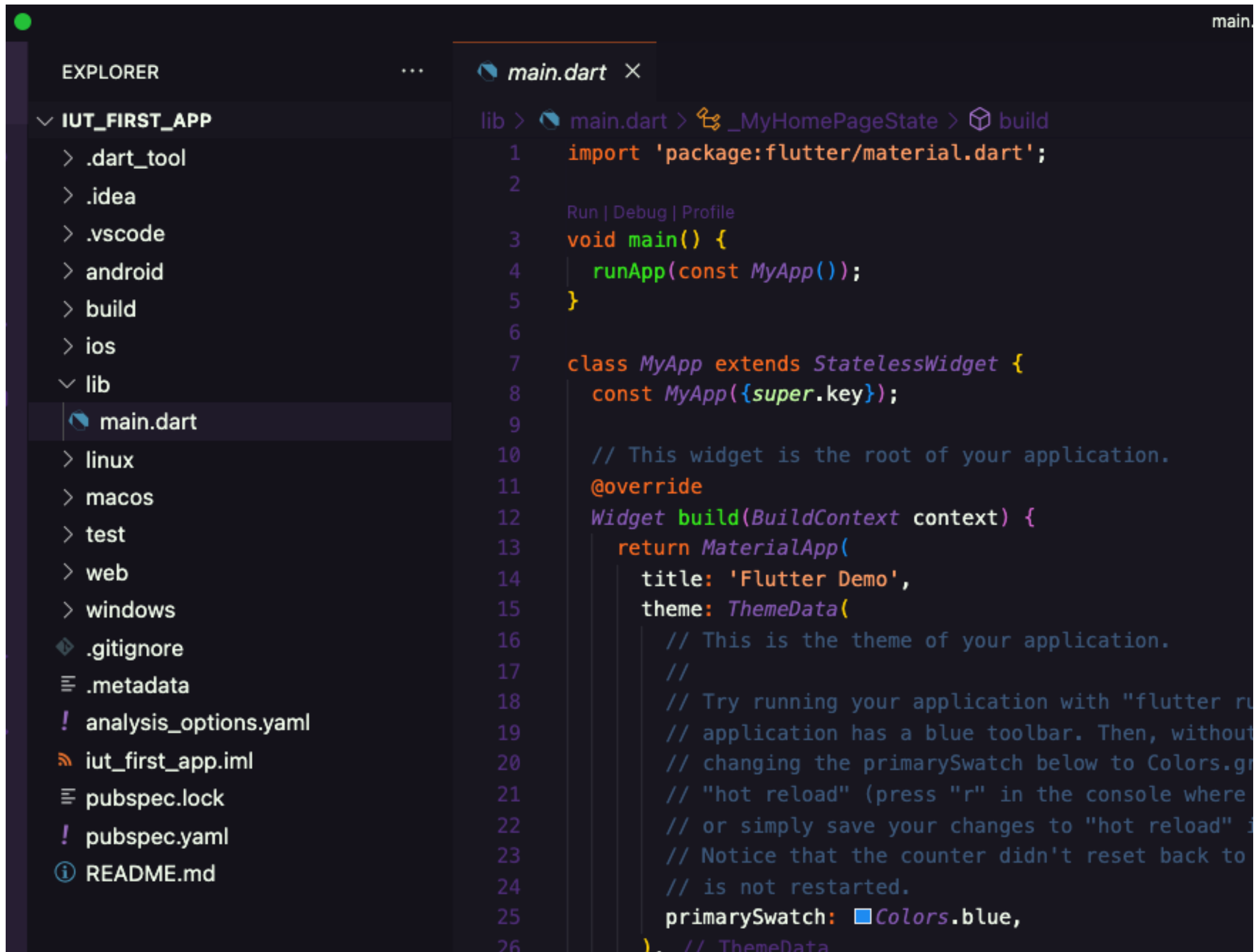
```
$ flutter create <NOM_DU_PROJET>  
$ cd ./<NOM_DU_PROJET>
```

Lancer l'application

Avant de lancer, vous aurez besoin d'un émulateur. Vous pouvez aussi lancer l'application sur votre téléphone, en activant le mode développeur de votre téléphone.

```
flutter run
```

La commande va générer la structure suivante :



The screenshot shows an IDE with a dark theme. On the left, the 'EXPLORER' sidebar displays the project structure for 'IUT_FIRST_APP'. The 'lib' directory is expanded, showing 'main.dart' as the selected file. The main editor area displays the content of 'main.dart', which includes an import statement for 'package:flutter/material.dart', a 'void main()' function that calls 'runApp(const MyApp())', and a 'class MyApp' that extends 'StatelessWidget'. The 'build' method of 'MyApp' returns a 'MaterialApp' with a title of 'Flutter Demo' and a blue primary color. The code is partially visible, ending with a closing brace for the 'build' method.

```
lib > main.dart > _MyHomePageState > build
1  import 'package:flutter/material.dart';
2
3  void main() {
4    runApp(const MyApp());
5  }
6
7  class MyApp extends StatelessWidget {
8    const MyApp({super.key});
9
10   // This widget is the root of your application.
11   @override
12   Widget build(BuildContext context) {
13     return MaterialApp(
14       title: 'Flutter Demo',
15       theme: ThemeData(
16         // This is the theme of your application.
17         //
18         // Try running your application with "flutter run"
19         // application has a blue toolbar. Then, without
20         // changing the primarySwatch below to Colors.green,
21         // "hot reload" (press "r" in the console where
22         // or simply save your changes to "hot reload" i
23         // Notice that the counter didn't reset back to
24         // is not restarted.
25         primarySwatch: Colors.blue,
26         // ThemeData
```

Structure d'un projet Flutter

The image shows a screenshot of an IDE (Visual Studio Code) displaying the structure of a Flutter project named `IUT_FIRST_APP`. The Explorer view on the left lists the project's files and folders, while the main editor shows the `main.dart` file. Annotations with arrows point to specific files and folders, explaining their purpose.

Annotations:

- android**: Dossier natif d'android qui contient par exemple le build.gradle
- ios**: Dossier natif d'ios, inutile si vous n'avez pas de Mac et d'iphone
- main.dart**: Point d'entrée de l'appli
- test**: Dossier des tests unitaires
- analysis_options.yaml**: Fichier yaml du linter (= règles de syntaxes du code)
- pubspec.yaml**: Ajout de packages/librairies

Code Snippet (main.dart):

```
lib > main.dart > _MyHomePageState > build
1 import 'package:flutter/material.dart';
2
3 void main() {
4   // This widget is the root of your application.
5   @override
6   Widget build(BuildContext context) {
7     MaterialApp(
8       title: 'Flutter Demo',
9       theme: ThemeData(
10        // This is the theme of your application.
11
12        // your application with "flutter run" has a blue toolbar. Then, without
13        // changing the primarySwatch below to Colors.green,
14        // "hot reload" (press "r" in the console where
15        // simply save your changes to "hot reload" i
16        // Notice that the counter didn't reset back to
17        // is not restarted.
18        primarySwatch: Colors.blue,
19      ),
20    ),
21  },
22 }
```

Le minimum pour afficher quelque chose

Trois points d'entrée sont nécessaires :

- `void main()` est le point d'entrée de l'appli, cette fonction est appelée automatiquement
- `runApp()` sert à attacher un premier "widget" à l'écran.
- `MaterialApp` a plusieurs rôles :
 - indique à Flutter dans quel sens l'UI doit être lue
 - définir une "home" (ainsi que les autres routes/pages de l'application)
 - gérer le thème global de l'application (couleurs, tailles de polices, etc)

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

void main() => runApp(
  const MaterialApp(
    home: Center(
      child: Text('Hello World'),
    ),
  ),
);
```

Afficher proprement une interface

- Ici on ajoute un `Scaffold` afin de ne pas avoir un écran noir
- C'est un widget qui donne une structure de base pour chaque page : appbar, body, floating button, etc.
- Il contient un thème par défaut (couleurs, taille de polices, etc)

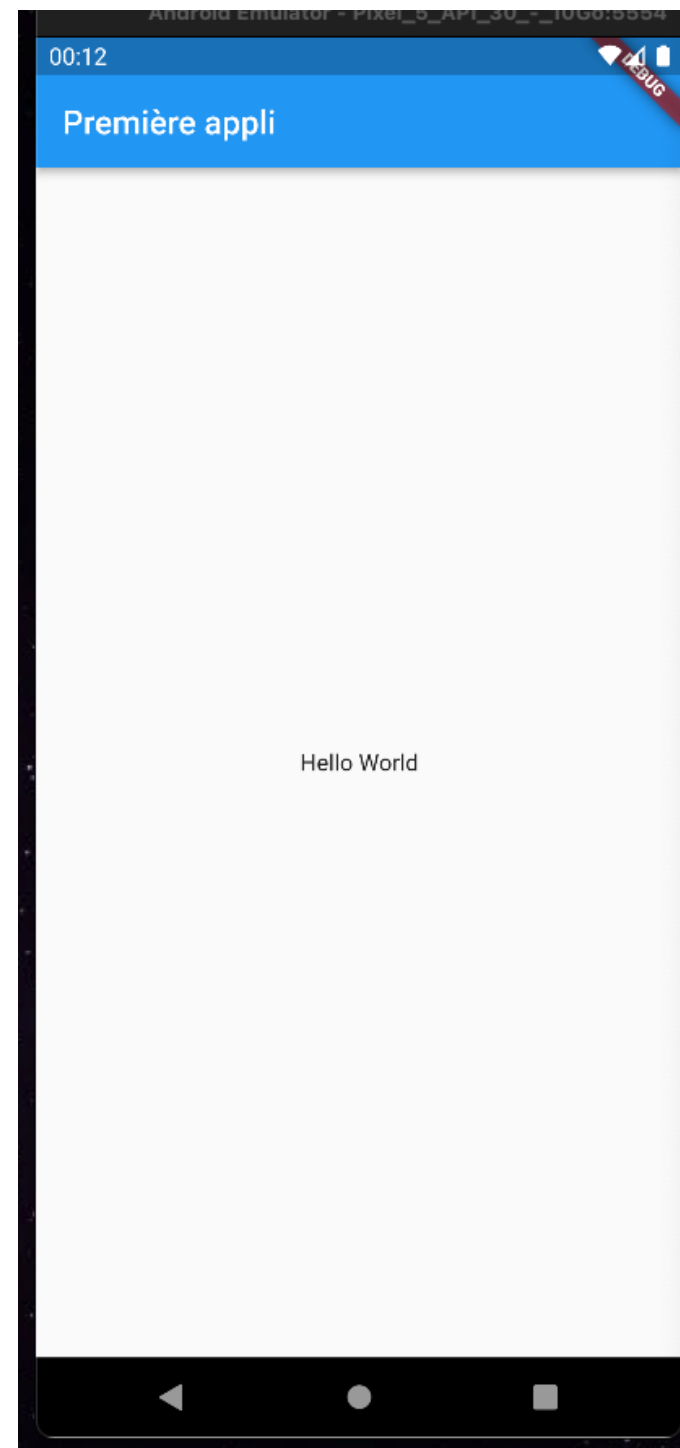
```
void main() => runApp(  
  MaterialApp(  
    home: Scaffold(  
      appBar: AppBar(title: const Text('Première appli')),  
      body: const Center(  
        child: Text('Hello World'),  
      ),  
    ),  
  ),  
);
```

Sans Scaffold / Avec Scaffold

Sans Scaffold



Avec Scaffold



Les widgets

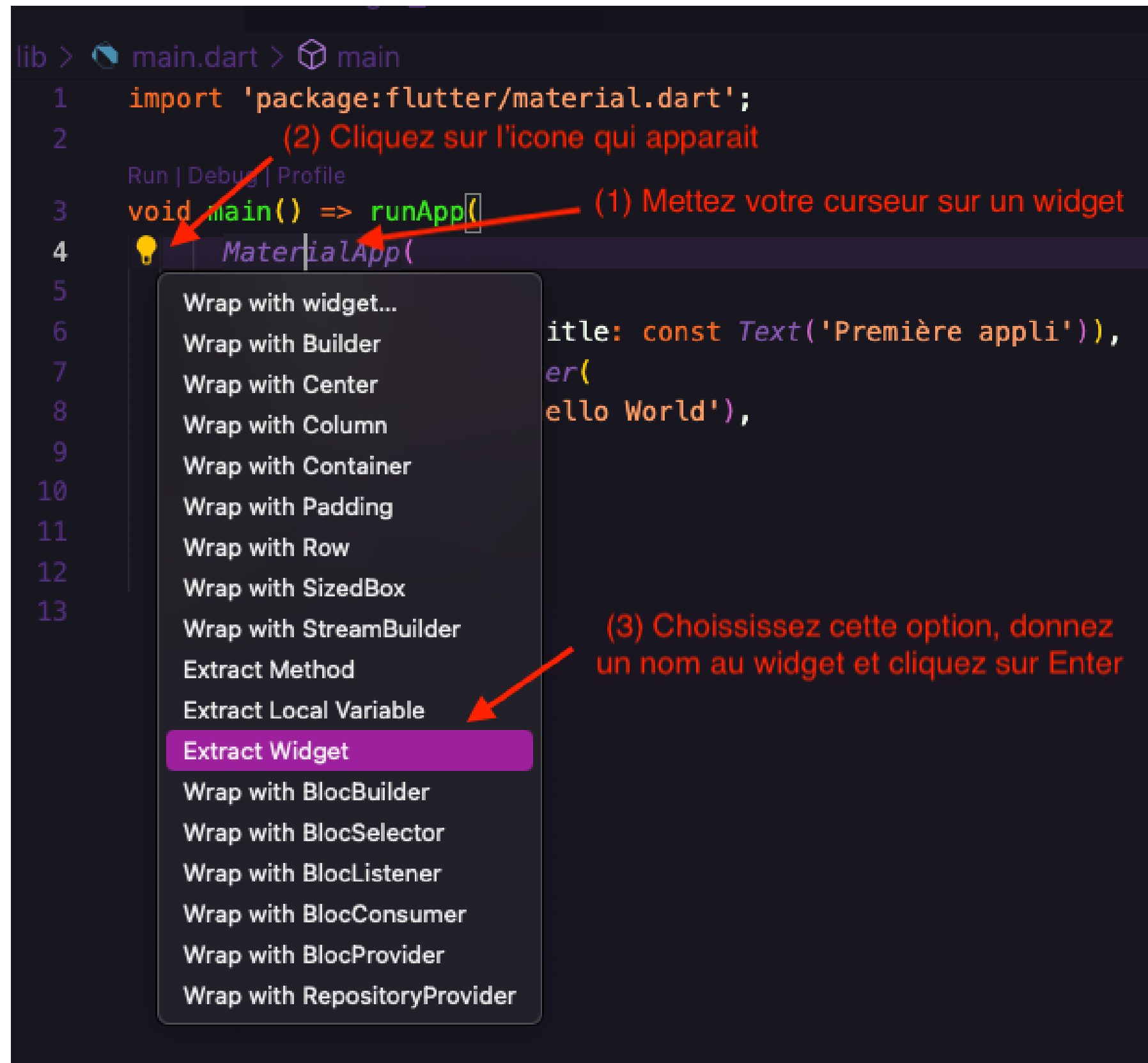
- Les widgets sont des composants qui permettent d'afficher une interface graphique.
- Tout est un widget : les pages, les textes, les columns, les paddings, le fait de centrer quelque chose, etc.
- On essaye au maximum de les rendre lisible en finissant chaque widget par une virgule, ainsi que chaque paramètre du widget.
- Ils peuvent être `Stateless` ou `Stateful`

La documentation officielle

La documentation officielle est très bien faite, pensez à vous en inspirer.

- Liste complète : <https://docs.flutter.dev/development/ui/widgets>
- La base : <https://docs.flutter.dev/development/ui/widgets/basics>
- Les widgets de placement : <https://docs.flutter.dev/development/ui/widgets/layout>
- Exemple pour les `Textfields` : <https://api.flutter.dev/flutter/material/TextField-class.html>

Créer un widget custom (1)



Créer un widget custom (2)

- Notre `class MyApp` va hériter d'un widget avec `extends StatelessWidget`
 - (On verra plus tard les `StatefulWidget`)
- La méthode `build` est `override` car provient de ce `StatelessWidget`, et gèrent l'affichage du rendu graphique.
- VSCode affichera très souvent des warnings comme-ci dessous pour indiquer que le constructeur doit etre constant et contenir une key. Cela donne : `const MyApp({super.key});`
 - `const` sert à dire que le widget ne changera plus.
 - `{super.key}` permet à Flutter d'identifier les widgets dans l'arboressence. Il est toujours en paramètre optionnel, et vous n'aurez quasiment jamais besoin de l'utiliser comme paramètre.



The screenshot shows a code editor with a Dart file named `main.dart` in the `package:iut_first_app`. The code defines a `MyApp` class that extends `StatelessWidget`. A warning message is displayed over the `MyApp` class, stating: "Constructors for public widgets should have a named 'key' parameter. Try adding a named parameter to the constructor. dart([use_key_in_widget_constructors](#))". Below the warning, there are two buttons: "View Problem" and "Quick Fix... (⌘.)". The code in the background is as follows:

```
lib > main.d package:iut_first_app/main.dart
1 import
2
3 void m
4
5 class MyApp extends StatelessWidget {
6
7   @override
8   Widget build(BuildContext context) {
9     return MaterialApp(
10      home: Scaffold(
```

Connaitre les paramètres d'un widget

```
(const) Text Text(  
  String data, {  
    Key? key,  
    TextStyle? style,  
    StrutStyle? strutStyle,  
    TextAlign? textAlign,  
    TextDirection? textDirection,  
    Locale? locale,  
    bool? softWrap,  
    TextOverflow? overflow,  
    double? textScaleFactor,  
    int? maxLines,  
    String? semanticsLabel,  
    TextWidthBasis? textWidthBasis.  
  })  
child: Text('Hello World'),  
// Center  
Scaffold
```

Passer la souris sur un widget pour en
connaître les attributs

Attention aux anti-patterns

- Pour rendre votre code lisible, découper le en widgets et **NE FAITES PAS DE FONCTIONS POUR RETOURNER DES WIDGETS**. Ne choisissez PAS l'option "Extract Method". C'est un anti-pattern de Flutter, c'est à dire que Flutter n'est pas fait pour fonctionner avec des méthodes quand ils génèrent l'interface graphique. Ca marchera, mais vous provoquerez des grosses pertes de performances :
 - l'appli sera plus longue à se lancer
 - les pages chargeront plus lentement
 - l'appli utilisera plus de ressources et cela va drainer la batterie du téléphone inutilement

Donc faites des widgets (Text), dans des widgets (Center), dans des widgets (Column), dans des widgets (Padding), dans des widgets (Scaffold/page), etc...

Elaboration d'une page

On commence par isoler la page Home en transformant le `Scaffold` en un widget qu'on appelle `HomePage` :

```
class HomePage extends StatelessWidget {  
  const HomePage({super.key});  
  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: AppBar(  
        title: const Text('Hola Mundo'),  
      ),  
      body: const Center(  
        child: Text('Hello World'),  
      ),  
    );  
  }  
}
```

Le dark mode

Dans le premier widget MyApp que l'on a généré, on définit deux thèmes et on choisit le dark :

```
class MyApp extends StatelessWidget {  
  const MyApp({super.key});  
  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      theme: ThemeData( // <== définit le thème Light  
        brightness: Brightness.light,  
      ),  
      darkTheme: ThemeData( // <== définit le thème Dark  
        brightness: Brightness.dark,  
      ),  
      themeMode: ThemeMode.dark, // <== on choisit le Dark  
      home: const HomePage(),  
    );  
  }  
}
```

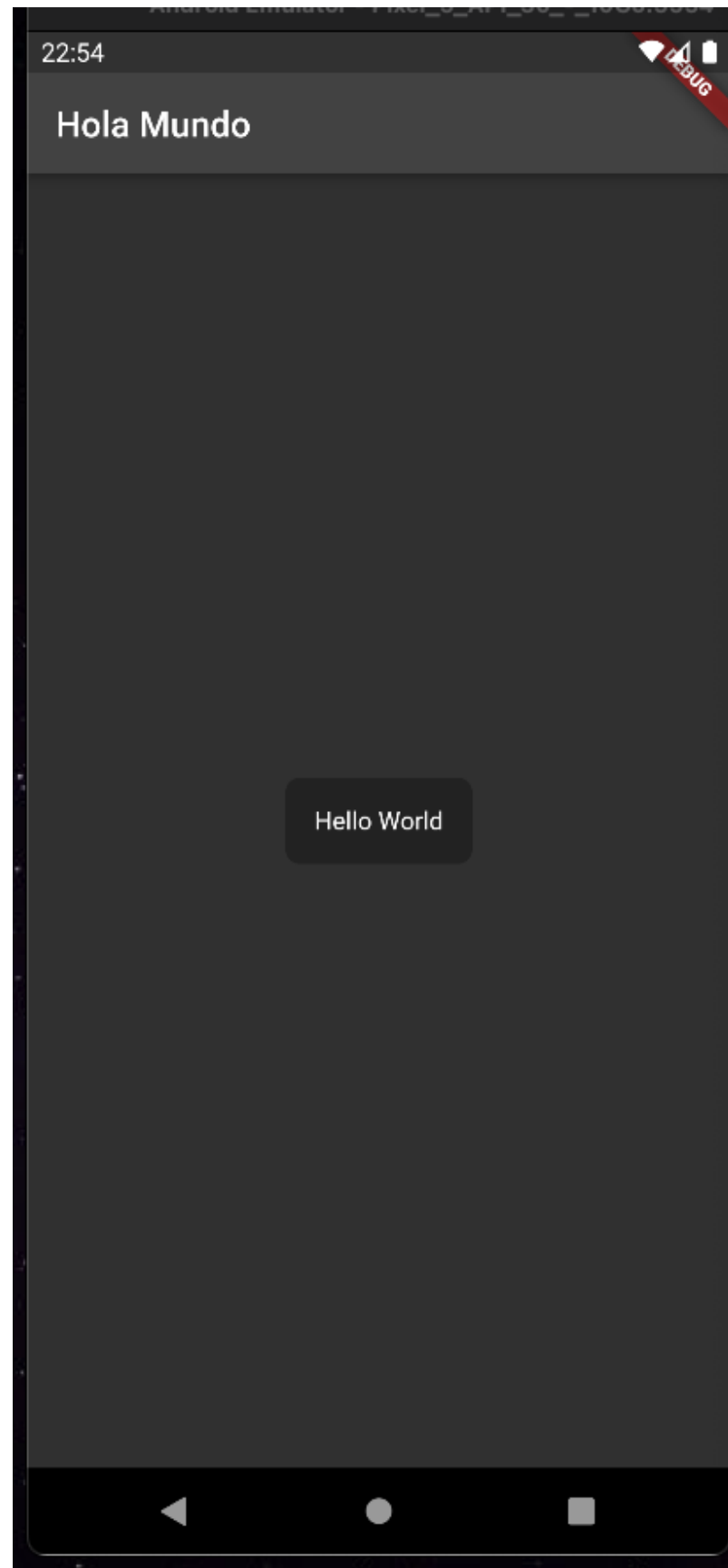

Widget : Container

- Le **Container** est le widget le plus basic pour définit une longueur, une largeur, un padding interne, un couleur de fond, un border radius (des coins rond), etc.
- Autour de notre **Text** Hello World, utilisez ce **Container**

```
Container(  
  decoration: BoxDecoration(  
    color: Colors.black.withOpacity(0.3),  
    // <cherchez comment appliquer un border radius >  
  ),  
  padding: const EdgeInsets.symmetric(  
    horizontal: 16,  
    vertical: 12,  
  ),  
  child: const Text('Hello World'),  
),
```

- Ajoutez un paramètre de **BorderRadius** au **Container**
- Ajoutez un widget **Padding** de 16px et centrer le au milieu de la page

Vous devez obtenir le résultat suivant :



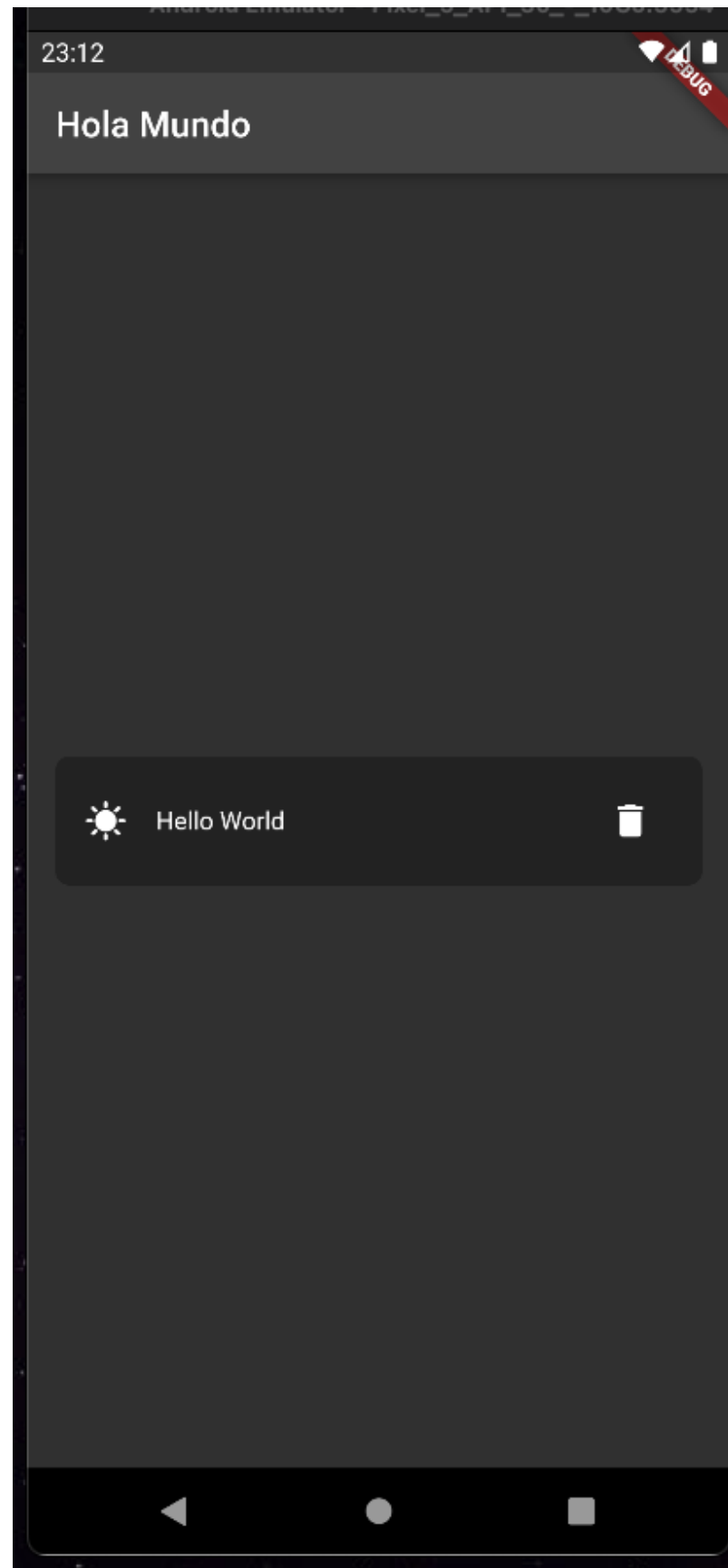
Widget : Row

- Dans le child du **Container**, on ajoute un widget **Row** afin de placer un widget **Icon** à côté du widget **Text**.
- On place également un widget bouton, qui pour l'instant n'affichera qu'un **print**

```
Row(  
  children: [  
    const Icon(Icons.sunny),  
    const Text('Hello World'),  
    IconButton(  
      icon: const Icon(Icons.delete),  
      onPressed: () => print('Hallo Welt'),  
    ),  
  ],  
)
```

- Ajoutez un widget pour espacer l'icone et le texte de 16px. Cela peut être fait de différentes manières.
- Chercher un widget à ajouter autour du **Text** pour qu'il prenne toute la largeur disponible.

Vous devez obtenir le résultat suivant :



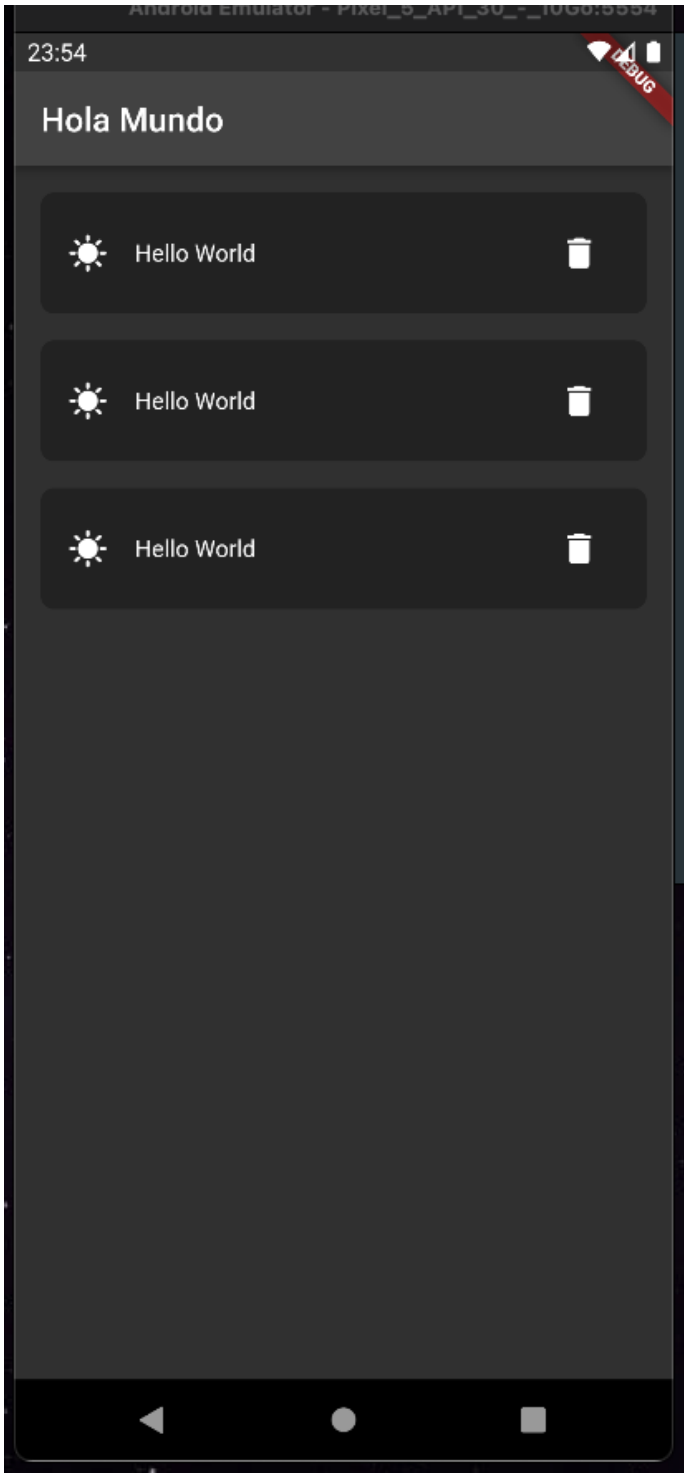
Widget : Column

- On commence par extraire notre composant à partir du `Container`, on l'appelle `TheAmazingRow`.
- On place notre widget `TheAmazingRow` dans un widget `Column`, et on duplique 3 fois notre widget

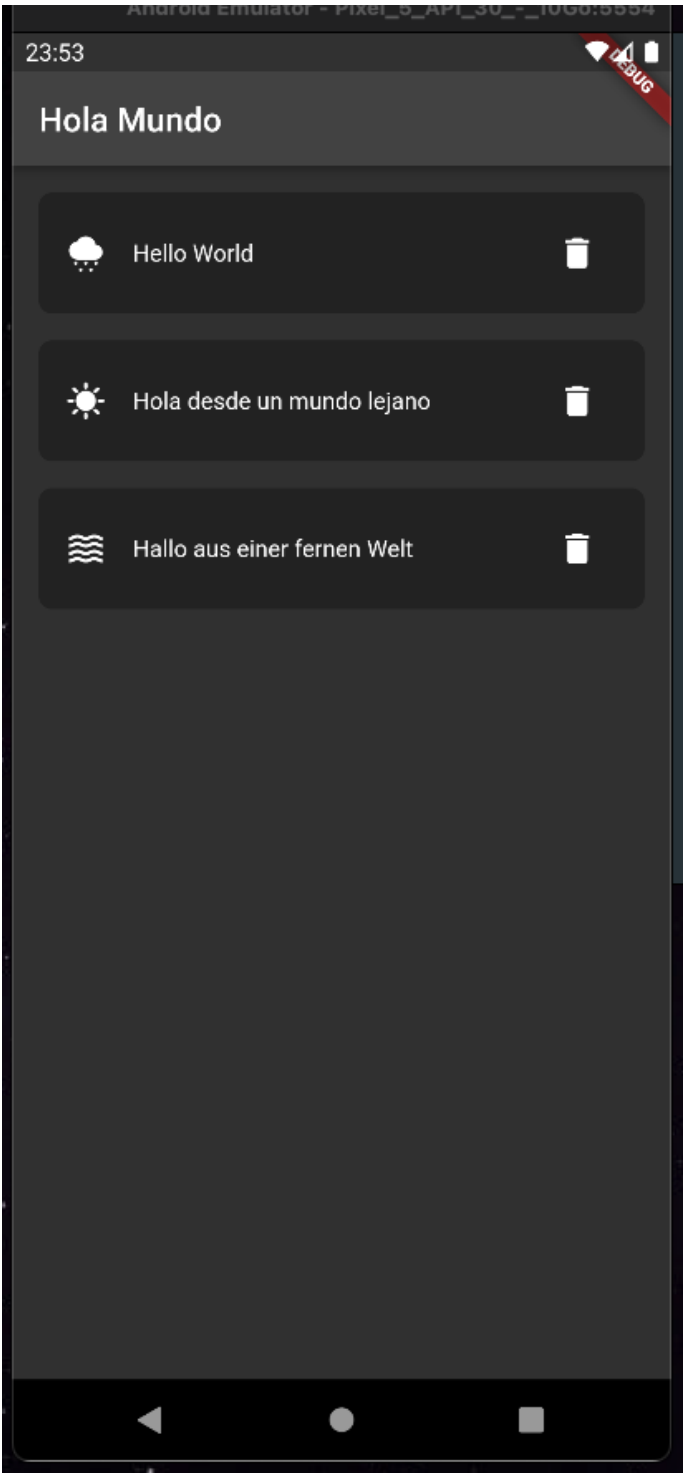
```
Column(  
  children: const [  
    TheAmazingRow(),  
    SizedBox(height: 16),  
    TheAmazingRow(),  
    SizedBox(height: 16),  
    TheAmazingRow(),  
  ],  
)
```

Vous devez obtenir le résultat suivant :

Sans parametres



Avec label et icon en parametres



Ajouter des paramètres à notre widget

- Dans notre widget `TheAmazingRow`, on ajoute les éléments suivant (entre le constructeur et la méthode `build`) :

```
final IconData icon;  
final String label;
```

On ajoute ces paramètres dans le constructeur en tant que paramètres nommés obligatoire (voir le cours précédant si besoin). Puis dans la méthode `build`, on fait référence à ces paramètres en remplaçant l'icone et le texte :

```
Row(  
  children: [  
    Icon(icon),  
    ...  
    Expanded(child: Text(label)),  
    ...  
  ],  
)
```

On peut alors écrire :

```
Column(  
  children: const [  
    TheAmazingRow(  
      label: 'Hello World',  
      icon: Icons.snowing,  
    ),  
    SizedBox(height: 16),  
    TheAmazingRow(  
      label: 'Hola desde un mundo lejano',  
      icon: Icons.sunny,  
    ),  
    SizedBox(height: 16),  
    TheAmazingRow(  
      label: 'Hallo aus einer fernen Welt',  
      icon: Icons.sunny,  
    ),  
  ],  
)
```


Générer du contenu dans une Column

Pour générer du contenu, il nous faut une liste appelée "pokedex" qui contient des objets **Pokemon**. Vous devez donc créer une classe **Pokemon** (et non un widget) qui peut contenir un texte, une icone, et un getter qui renvoie le texte en lettres capitales. Cette liste est placée dans notre widget **HomePage**.

```
class HomePage extends StatelessWidget {  
  const HomePage({super.key});  
  
  final pokedex = const <Pokemon>[ // <== liste finale, contenu constant  
    Pokemon('Artikodin', Icons.ac_unit),  
    Pokemon('Sulfura', Icons.sunny),  
    Pokemon('Electhor', Icons.thunderstorm),  
    Pokemon('Mewtwo', Icons.remove_red_eye),  
  ];  
  
  @override  
  Widget build(BuildContext context) {  
  }
```

On peut aussi écrire `final List<Pokemon> pokedex = const [...];`

La liste est `final` car elle restera toujours une `List`, et contiendra toujours des `Pokemon`. Elle ne deviendra pas un `int` ou une `Map` ou une `String`. Cependant, son contenu peut changer. On pourra enlever/ajouter des items dedans.

La mention `const`, signifie que chaque élément de la liste est constant. Les noms et les icones sont définis en dur, et ne changeront pas.

Si un `Pokemon` prend en paramètre une variable, il n'est plus constant. Idem, le contenu de la liste n'est plus exclusivement constant. On l'ajoute donc devant chaque `Pokemon` constant. Cela permet au langage de savoir ce qui ne changera jamais, et donc de l'afficher plus vite, en utilisant moins de ressources. On gagne ainsi en performance.

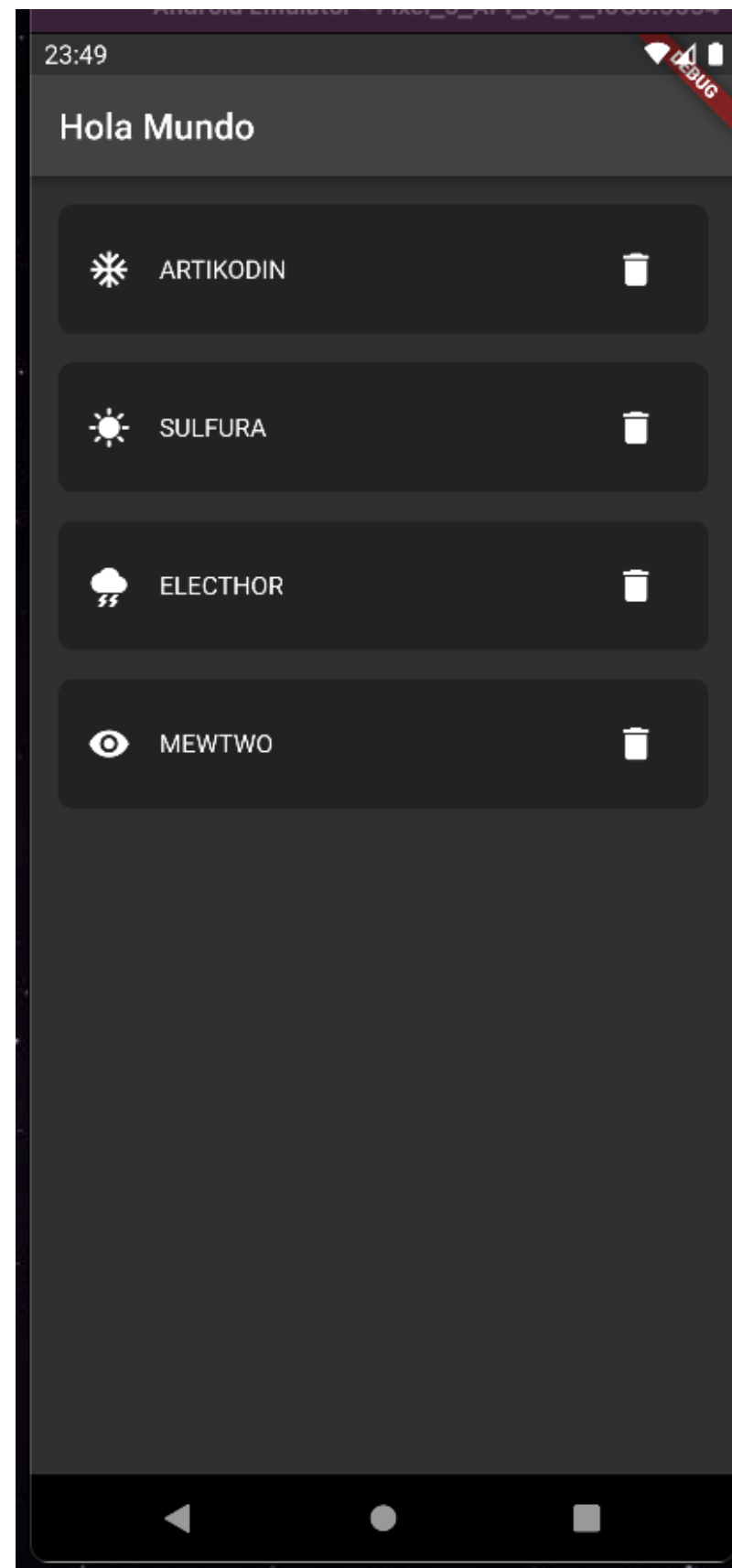
```
IconData snowflakeIcon = Icons.ac_unit;

final pokedex = <Pokemon>[
  Pokemon('Artikodin', snowflakeIcon),
  const Pokemon('Sulfura', Icons.sunny),
  const Pokemon('Electhor', Icons.thunderstorm),
  const Pokemon('Mewtwo', Icons.remove_red_eye),
];
```

Cette `List` nous permet de générer du contenu avec une boucle `for` ou une `map` :

```
Column(  
    children: [  
        for (final Pokemon item in pokedex)  
            TheAmazingRow(  
                label: item.upperCaseText,  
                icon: item.illustration,  
            ),  
    ],  
)  
// ----- OU -----  
Column(  
    children: pokedex  
        .map(  
            (Pokemon item) => TheAmazingRow(  
                label: item.upperCaseText,  
                icon: item.illustration,  
            ),  
        )  
        .toList(),  
)
```

Vous devez obtenir le résultat suivant :

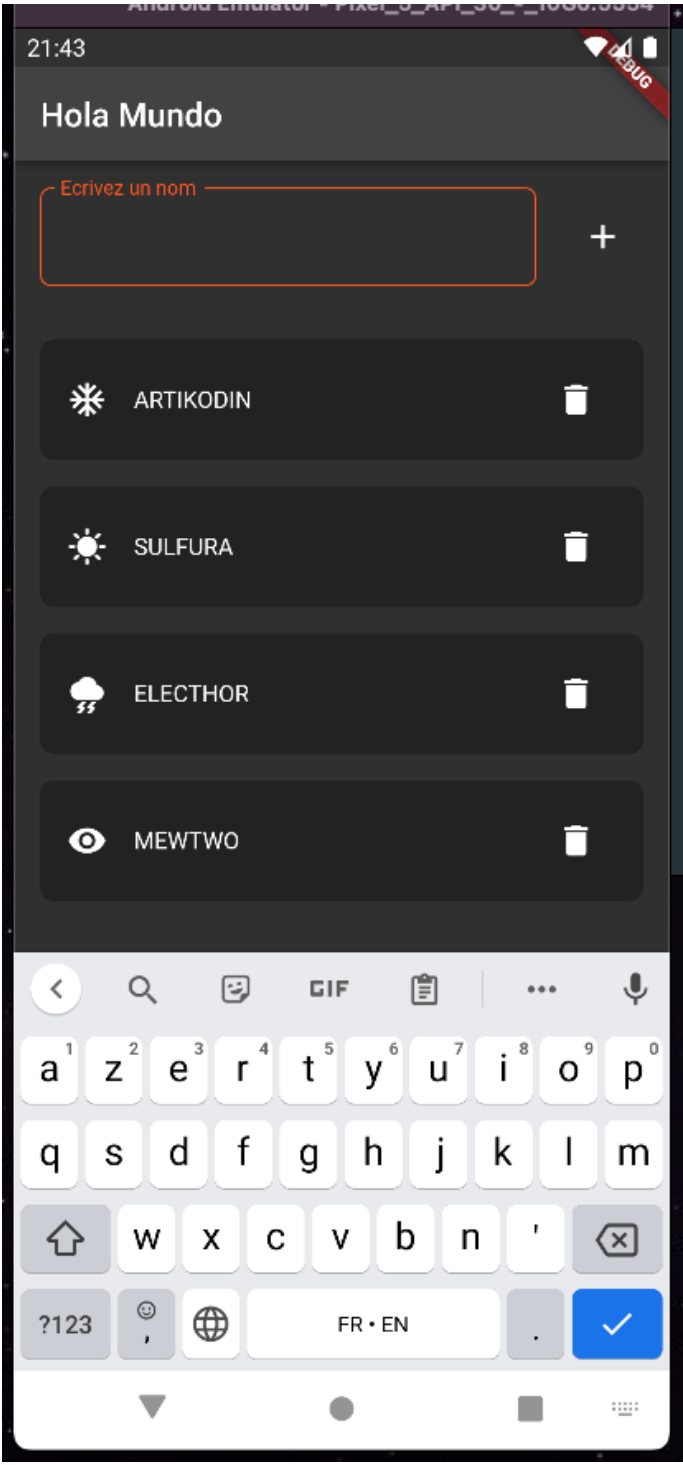
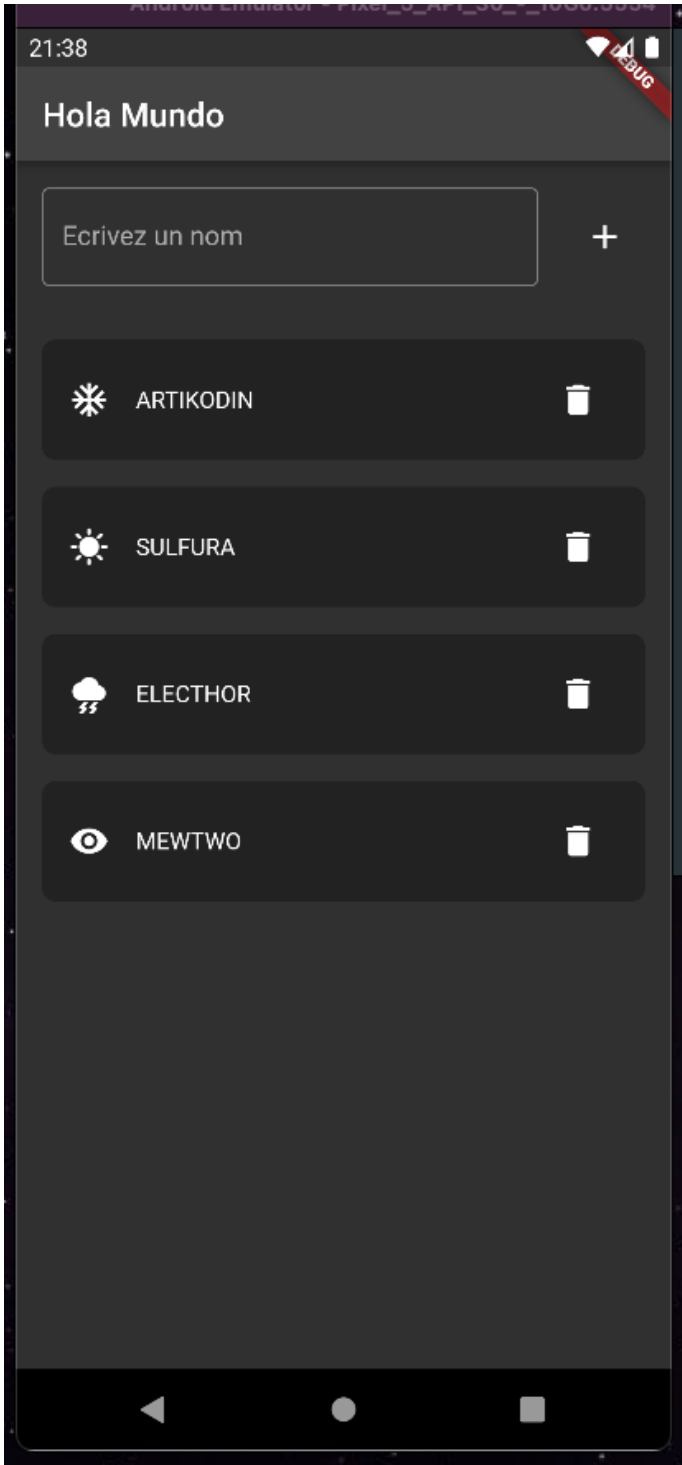


Widget : TextField

- Ajouter un `TextField`, puis modifiez son aspect. Ajoutez :
 - une bordure
 - un border radius,
 - un label et une couleur de label
 - une couleur lorsque le Textfield est focused (quand l'utilisateur a cliqué dessus et que le clavier s'ouvre)
- Vous pouvez faire cela en modifiant le `darkTheme` du `MaterialApp`, ou au niveau du `Textfield` directement

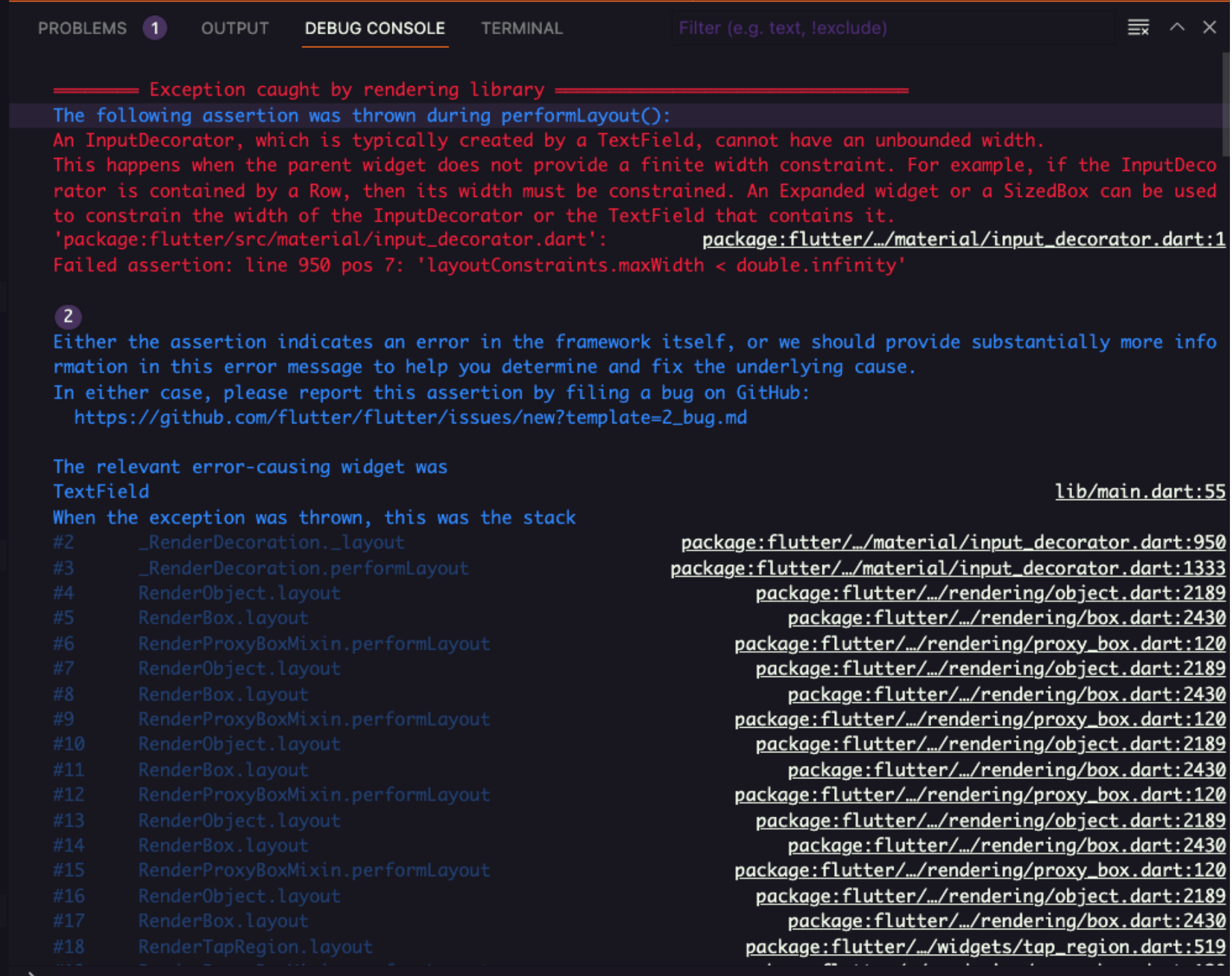
```
TextField(  
  decoration: InputDecoration(  
    border: ... , // <== complétez  
    focusedBorder: ... , // <== complétez  
    ... , // <== complétez  
    labelText: 'Ecrivez un nom',  
  ),  
)
```

Vous devez obtenir le résultat suivant :



Erreurs liées aux height et width

- Placer le `TextField` dans un `Row`, et mettre un `IconButton` à droite.
- Vous allez rencontrer beaucoup d'erreur de ce type :



```
PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL Filter (e.g. text, !exclude)

===== Exception caught by rendering library =====
The following assertion was thrown during performLayout():
An InputDecorator, which is typically created by a TextField, cannot have an unbounded width.
This happens when the parent widget does not provide a finite width constraint. For example, if the InputDecorator is contained by a Row, then its width must be constrained. An Expanded widget or a SizedBox can be used to constrain the width of the InputDecorator or the TextField that contains it.
'package:flutter/src/material/input_decorator.dart': package:flutter/.../material/input_decorator.dart:1
Failed assertion: line 950 pos 7: 'layoutConstraints.maxWidth < double.infinity'

2
Either the assertion indicates an error in the framework itself, or we should provide substantially more information in this error message to help you determine and fix the underlying cause.
In either case, please report this assertion by filing a bug on GitHub:
https://github.com/flutter/flutter/issues/new?template=2_bug.md

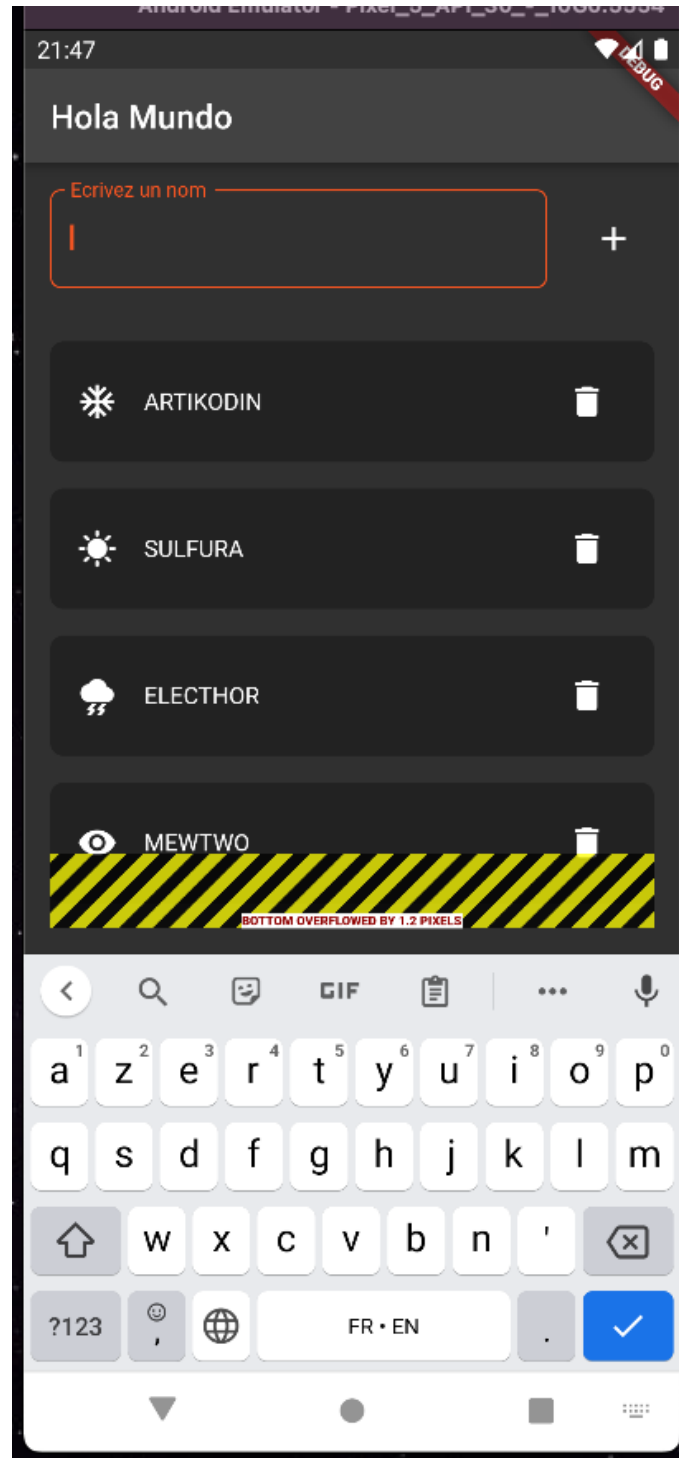
The relevant error-causing widget was
TextField lib/main.dart:55
When the exception was thrown, this was the stack
#2      _RenderDecoration._layout package:flutter/.../material/input_decorator.dart:950
#3      _RenderDecoration.performLayout package:flutter/.../material/input_decorator.dart:1333
#4      RenderObject.layout package:flutter/.../rendering/object.dart:2189
#5      RenderBox.layout package:flutter/.../rendering/box.dart:2430
#6      RenderProxyBoxMixin.performLayout package:flutter/.../rendering/proxy_box.dart:120
#7      RenderObject.layout package:flutter/.../rendering/object.dart:2189
#8      RenderBox.layout package:flutter/.../rendering/box.dart:2430
#9      RenderProxyBoxMixin.performLayout package:flutter/.../rendering/proxy_box.dart:120
#10     RenderObject.layout package:flutter/.../rendering/object.dart:2189
#11     RenderBox.layout package:flutter/.../rendering/box.dart:2430
#12     RenderProxyBoxMixin.performLayout package:flutter/.../rendering/proxy_box.dart:120
#13     RenderObject.layout package:flutter/.../rendering/object.dart:2189
#14     RenderBox.layout package:flutter/.../rendering/box.dart:2430
#15     RenderProxyBoxMixin.performLayout package:flutter/.../rendering/proxy_box.dart:120
#16     RenderObject.layout package:flutter/.../rendering/object.dart:2189
#17     RenderBox.layout package:flutter/.../rendering/box.dart:2430
#18     RenderTapRegion.layout package:flutter/.../widgets/tap_region.dart:519
```

- L'erreur de la page précédente est causée par la nécessité de certains widgets de connaître quelle height/width occuper. Pour régler cela, vous devez :
 - utiliser `mainAxisSize: MainAxisSize.min`, pour indiquer aux `Column/Row` de prendre le moins de place possible.
 - ajouter des `Expanded` et `Flexible` autour de certains widgets enfants des `Column` et `Row`.

```
Column(  
  mainAxisSize: MainAxisSize.min,  
  children: [  
    Row(  
      mainAxisSize: MainAxisSize.min,  
      children: const [  
        ... ,  
      ],  
    ),  
  ],  
)
```


Erreurs liées à l'overflow et au scroll

Vous verrez des bandes jaunes et noirs apparaissent en cas d'overflow :



Pour fixer cette erreur, chercher un widget qui permet d'ajouter un scroll au niveau de toute la page.

Suite au prochain cours

Au prochain cours, on transformera certains `StatelessWidget` en `StatefulWidget`.

Pour ceux qui veulent prendre de l'avance, vous pouvez déjà tenter de faire fonctionner les éléments de cet écran :

- les boutons de suppression de chaque ligne
- le bouton "+" qui ajoute le contenu du `TextField` dans la liste (avec une icône random)
- Ajouter une liste dropdown pour choisir l'icône à afficher.

Bon courage !