

« Mobile & Image »

« Développement d'une application mobile »

TP dirigé 1 « Miles ou kilomètres »

Par Alexis GUILLOTEAU & Didier JUGE-HUBERT

Sommaire



| | | |
|-----|---|----|
| 1 | Introduction..... | 2 |
| 2 | Présentation du projet | 3 |
| 3 | Etapes de création de l'application « convertisseur » | 4 |
| 3.1 | Création du nouveau projet | 4 |
| 3.2 | Création de la page d'accueil | 4 |
| 3.3 | Ajout de la logique métier | 19 |
| 3.4 | Challenge | 21 |

1 Introduction

Mesures telles que la distance, la vitesse, le poids, le volume et le changement de température en fonction de votre lieu de résidence est différent. En fait, il existe aujourd'hui deux principaux systèmes de mesure : le système impérial, qui est principalement utilisé aux États-Unis; et le système métrique, qui est utilisé dans la plupart des autres pays.

Dans cette application, vous allez créer une application de conversion de mesures, dans laquelle les mesures de distance et de poids seront converties de l'impériale au métrique, et vice versa.







Ce TP permettra d'aborder les points suivants :

-  Compréhension des widgets *stateful* et *stateless*
-  Création d'une application convertisseur de mesures

Remarque : Pour suivre ce TP, vous devez avoir installé et configuré les logiciels SDK flutter, Android Studio (pour l'émulateur), et optionnellement Visual Studio Code (pour l'édition).

À la fin de ce TP, vous saurez comment tirer parti de la classe *State* à l'aide de widgets tels que *TextFields* pour interagir avec les utilisateurs et rendre vos applications interactives.

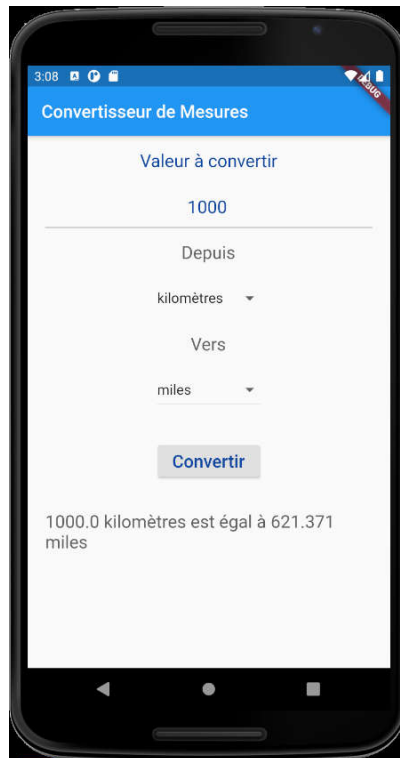
Ce faisant, vous rencontrerez plusieurs concepts fondamentaux dans Flutter, et en particulier, les suivants:

-  Comment vous devez utiliser des widgets sans état ou avec état.
-  Comment vous allez travailler avec les états dans Flutter,
 - Comment utiliser des widgets avec état,
 - Comment mettre à jour l'état dans votre application avec les événements *onChanged* et *OnSubmitted* dans un *TextField*.
-  Comment utiliser le widget de saisie utilisateur,
-  Comment gérer un bouton avec *DropDownButton*,
-  Comment utiliser une liste déroulante avec *DropDownItems*,
-  Comment commencer à séparer la logique de votre application de l'interface utilisateur (UI), et vous obtiendrez quelques conseils sur la façon de créer la structure de votre application.

2 Présentation du projet

L'application de conversion de mesures permettra aux utilisateurs de sélectionner une mesure - métrique ou impériale - et de la convertir en une autre mesure. Par exemple, ils pourront convertir une distance en miles en une distance en kilomètres ou un poids en kilogrammes en un poids en livres.

A la fin du TP, vous devriez obtenir une application ressemblant à cela :

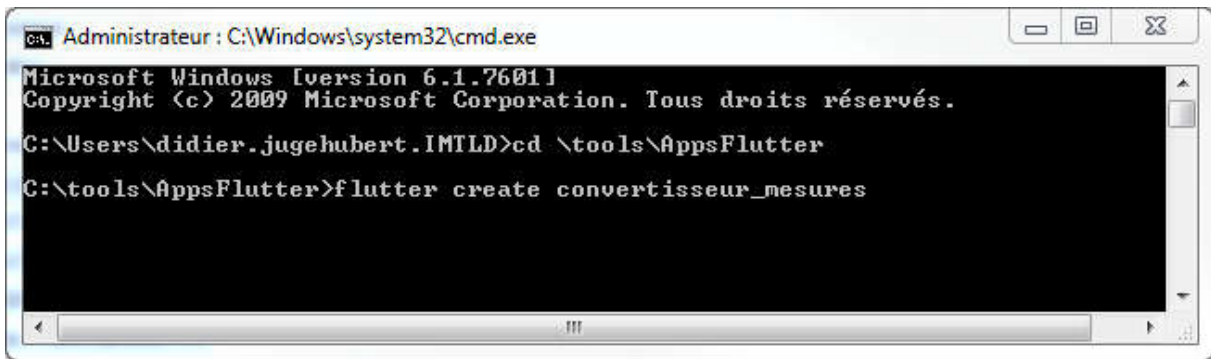


Comme vous pouvez le voir, il s'agit d'un formulaire plutôt standard avec des widgets de *Material Design*, qui devraient être très faciles à compiler et à utiliser pour les utilisateurs. Vous pouvez l'utiliser comme point de départ pour toutes vos applications futures.

3 Etapes de création de l'application « convertisseur »

3.1 Création du nouveau projet

- Je vous propose de créer un répertoire « AppsFlutter » sous le répertoire « tools » qui a été défini lors de l'installation.
- Démarrer votre émulateur avant la création du projet afin qu'il soit reconnu pour l'exécution.
- Pour créer le projet via la ligne de commande, saisissez la commande ci-dessous :



```
Administrator: C:\Windows\system32\cmd.exe
Microsoft Windows [version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Tous droits réservés.
C:\Users\didier.jugehubert.IMTLD>cd \tools\AppsFlutter
C:\tools\AppsFlutter>flutter create convertisseur_mesures
```

- OU Pour créer le projet via Visual Studio Code,
 - Dans le menu « **Affichage => Palette de commandes...** ».
 - Saisissez dans le champ de recherche « > » le texte « *flutter* ».
 - Choisissez « *Flutter : New Project* » puis saisissez le nom de votre nouveau projet « convertisseur_mesures » puis validez avec « **Entrée** ».

3.2 Création de la page d'accueil

3.2.1 Création d'un widget *stateless*

N'ayant aucun a priori au début, je propose d'essayer de créer une application avec un widget *stateless* permettant d'obtenir le résultat ci-contre :

Pour cela vous devez cascader les widgets suivants :

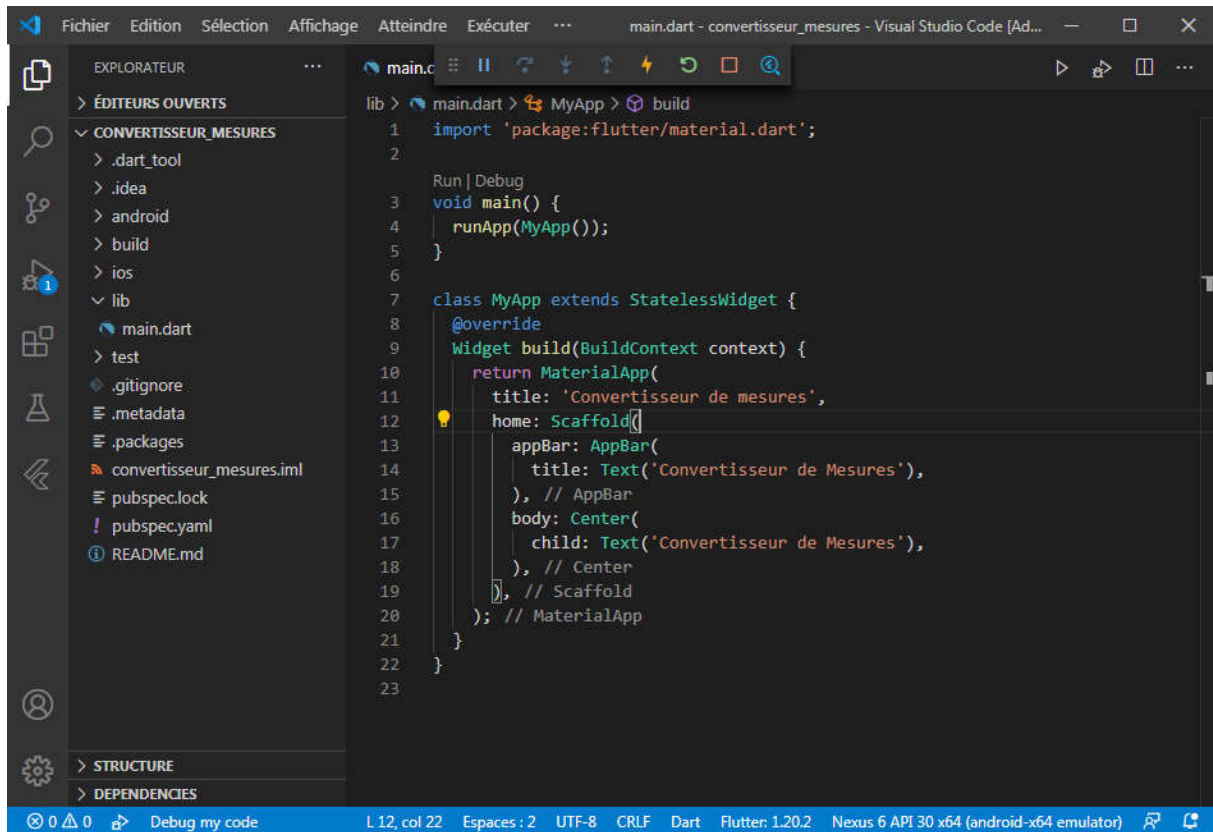
- *Stateless*
- *MaterialApp*
- *Scaffold*
- *AppBar*

Et utiliser le widget *Text* pour afficher les chaînes de caractères.

⇒ Solution sur la page suivante.



TP dirigé 1 « Miles ou kilomètres »



```
lib > main.dart > MyApp > build
1 import 'package:flutter/material.dart';
2
3 Run | Debug
4 void main() {
5   runApp(MyApp());
6 }
7
8 class MyApp extends StatelessWidget {
9   @override
10  Widget build(BuildContext context) {
11    return MaterialApp(
12      title: 'Convertisseur de mesures',
13      home: Scaffold(
14        appBar: AppBar(
15          title: Text('Convertisseur de Mesures'),
16        ), // AppBar
17        body: Center(
18          child: Text('Convertisseur de Mesures'),
19        ), // Center
20      ), // Scaffold
21    ); // MaterialApp
22  }
23 }
```

Mais une fois construit, le widget *stateless* ne change pas, ce qui est gênant 😞.

3.2.2 Modification en un widget *stateful*

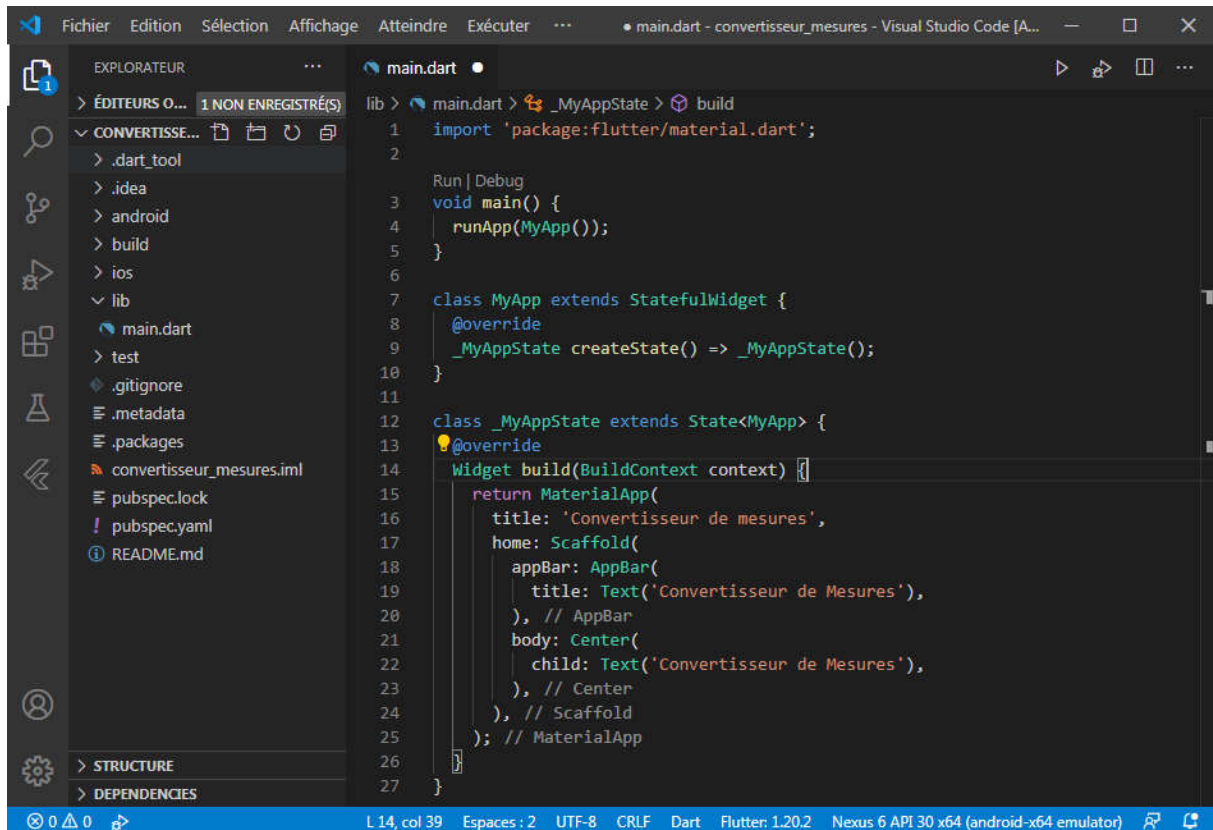
- Remplacer le widget *stateless* par un widget *stateful*.
- Création d'une gestion de l'état du nouveau widget *stateful* à l'aide de la classe *State*.

Vous devriez obtenir l'affichage ci-contre (identique au précédent théoriquement).

⇒ Solution sur la page suivante.



TP dirigé 1 « Miles ou kilomètres »



3.2.3 Lecture des entrées utilisateur depuis TextField

- a) Dans la classe `State`, ajoutez un membre appelé `_nombreSaisi`. Il s'agit d'une valeur qui changera en fonction de l'entrée de l'utilisateur :

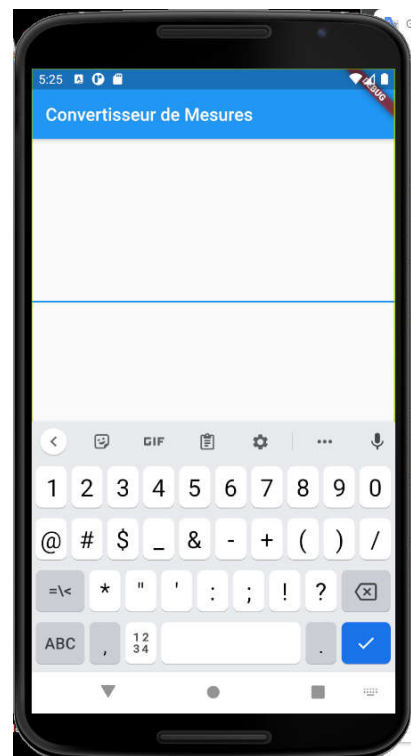
```
double _nombreSaisi;
```

- b) Ensuite, dans le corps de la méthode `build()`, supprimez le widget `Text` et ajoutez le code suivant pour placer un champ de saisie texte

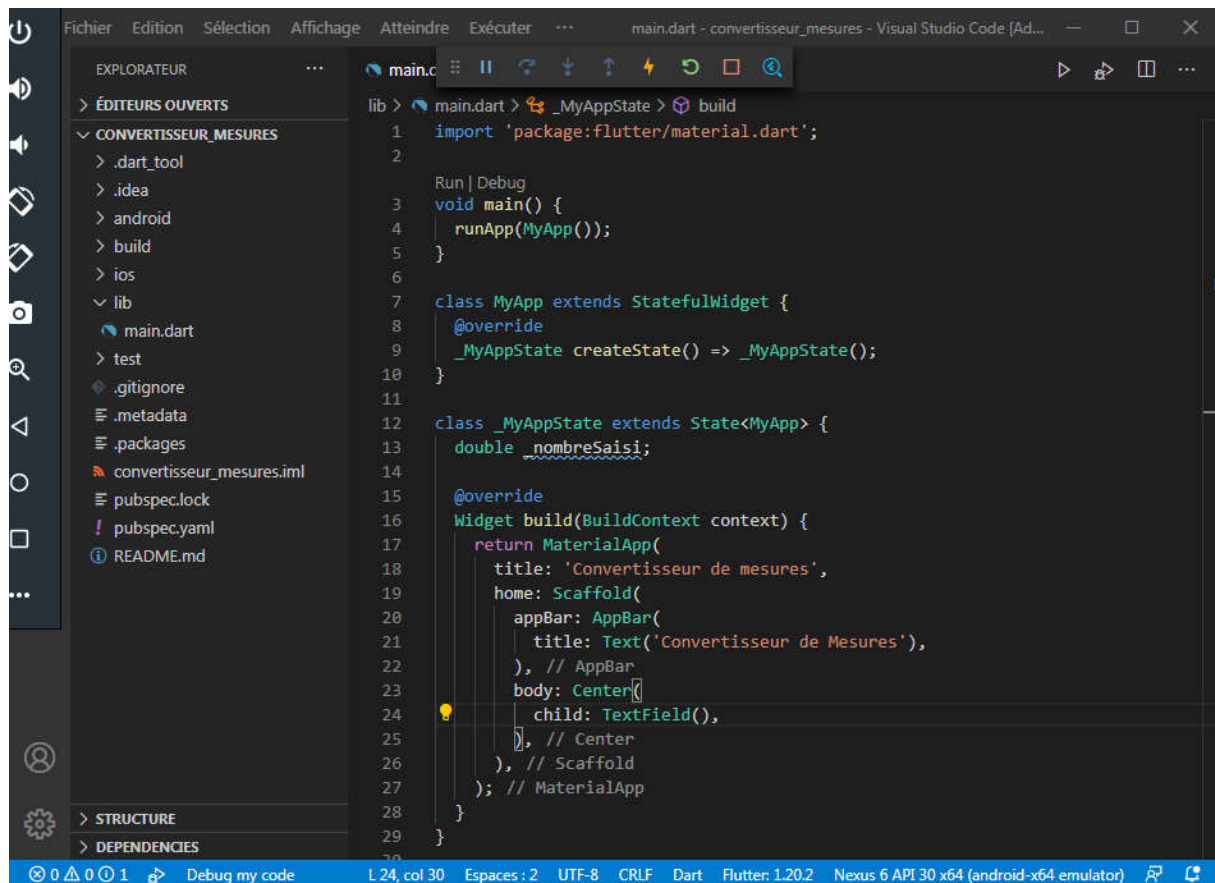
```
body: Center (
  child: TextField(),
),
```

- c) Utilisez généralement `TextField` lorsque vous souhaitez obtenir des informations de l'utilisateur. Comme vous pouvez le voir ci-contre, il y a maintenant `TextField` au centre de votre application, et vous pouvez y écrire quelque chose dedans.

⇒ Solution sur la page suivante.



TP dirigé 1 « Miles ou kilomètres »

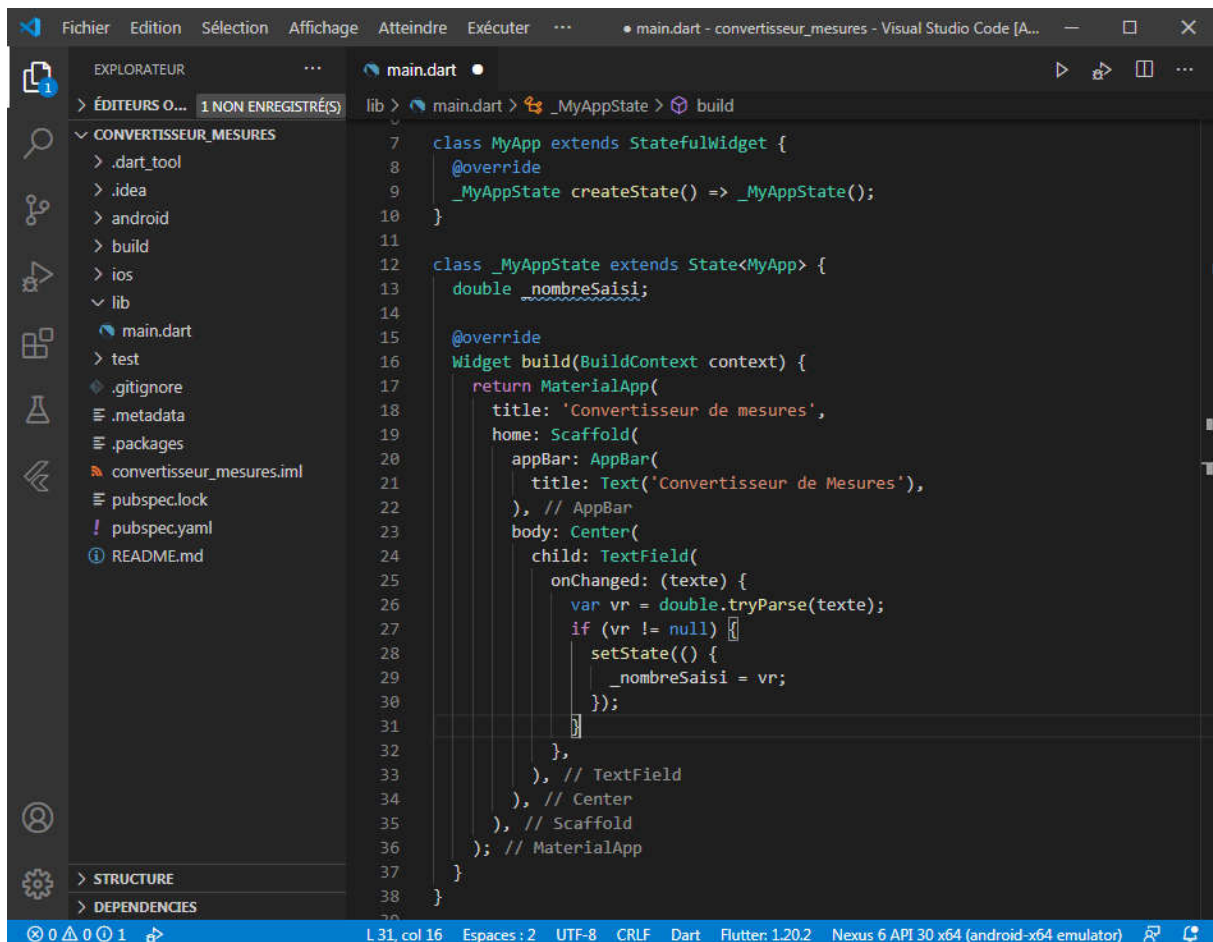


Pour le moment, *TextField* ne fait rien, donc la première chose que vous devez faire est de lire la valeur entrée par l'utilisateur.

- d) Bien qu'il existe différentes façons de lire à partir de *TextField*, pour ce projet, je vous propose de répondre à chaque modification du contenu de *TextField* via la méthode *onChanged*, puis de mettre à jour l'état. Afin de mettre à jour l'état, vous devez appeler la méthode *setState* ().

⇒ Solution sur la page suivante.

TP dirigé 1 « Miles ou kilomètres »



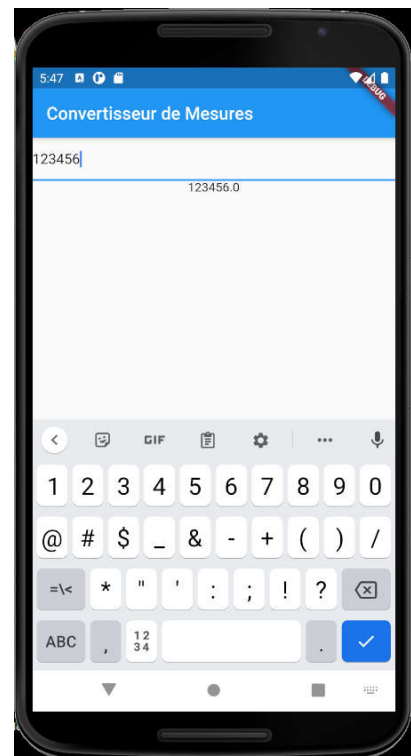
```
7 class MyApp extends StatefulWidget {
8   @override
9   _MyAppState createState() => _MyAppState();
10 }
11
12 class _MyAppState extends State<MyApp> {
13   double _nombreSaisi;
14
15   @override
16   Widget build(BuildContext context) {
17     return MaterialApp(
18       title: 'Convertisseur de mesures',
19       home: Scaffold(
20         appBar: AppBar(
21           title: Text('Convertisseur de Mesures'),
22         ), // AppBar
23         body: Center(
24           child: TextField(
25             onChanged: (texte) {
26               var vr = double.tryParse(texte);
27               if (vr != null) {
28                 setState(() {
29                   _nombreSaisi = vr;
30                 });
31               }
32             },
33           ), // TextField
34         ), // Center
35       ), // Scaffold
36     ); // MaterialApp
37   }
38 }
```

Dans le code précédent, à chaque fois que la valeur de *TextField* change (*onChanged*), on vérifie si la valeur qui a été tapée est un nombre (*tryParse*). S'il s'agit d'un nombre, nous modifions la valeur du membre *_nombreSaisi*. De cette façon, nous avons en fait mis à jour l'état. En d'autres termes, lorsque vous appelez la méthode *setState()* pour mettre à jour un membre de classe, vous mettez également à jour l'état de la classe.

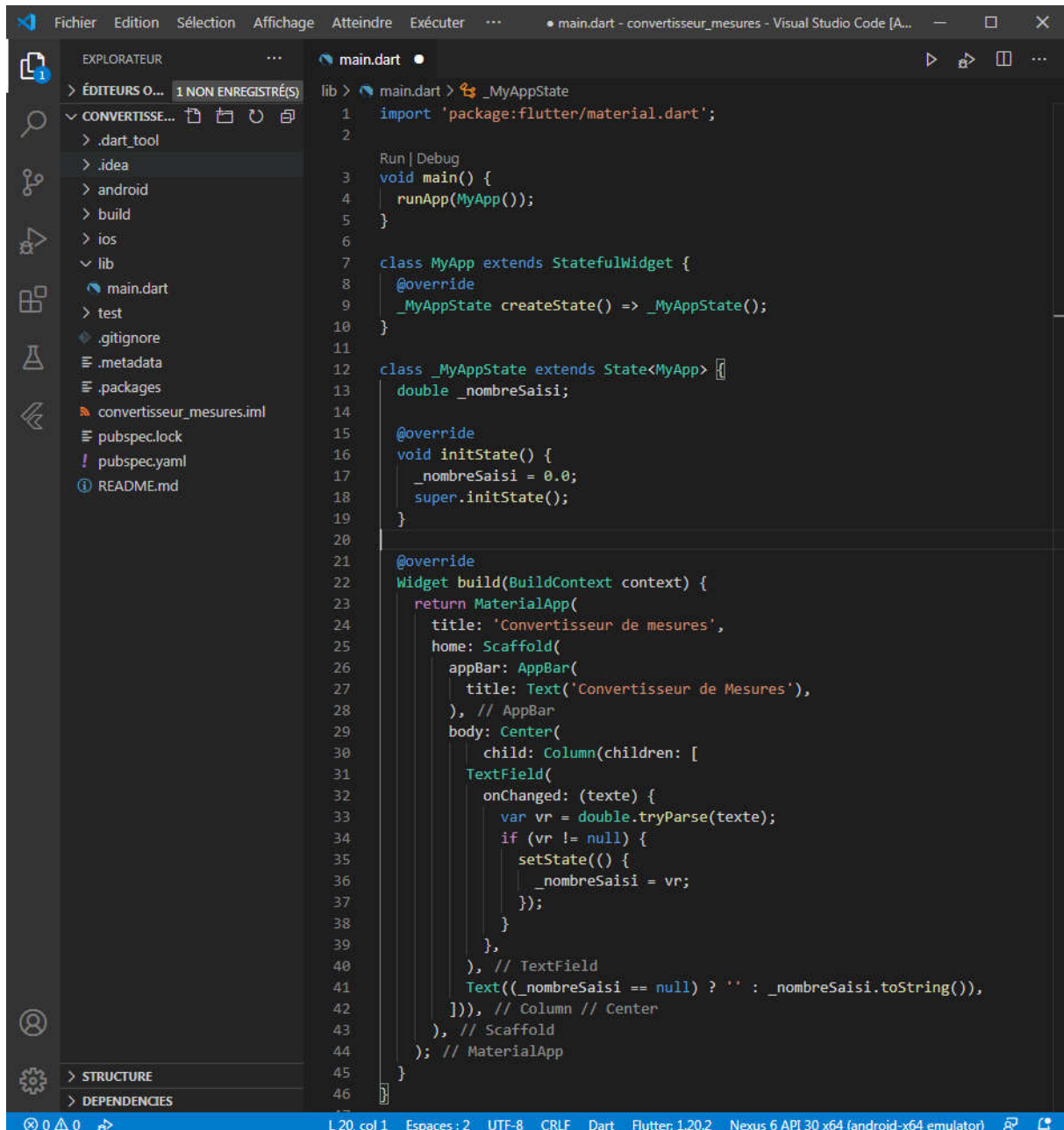
Vous ne donnez aucun retour à l'utilisateur, donc à moins d'utiliser les outils de débogage de notre éditeur, nous ne pouvons pas réellement vérifier si cette mise à jour s'est réellement produite. Afin de résoudre cela.

- e) Ajoutez un widget *Text* qui affichera le contenu du widget *TextEdit*, puis enveloppons les deux widgets dans un widget *Column*.
- f) Avant d'essayer l'application, ajoutez une autre méthode à la classe *MyAppState* permettant d'initialiser la valeur de *_nombreSaisi*.

⇒ Solution sur la page suivante.



TP dirigé 1 « Miles ou kilomètres »



```
lib > main.dart > _MyAppState
1 import 'package:flutter/material.dart';
2
3 Run | Debug
4 void main() {
5   runApp(MyApp());
6 }
7
8 class MyApp extends StatefulWidget {
9   @override
10  _MyAppState createState() => _MyAppState();
11 }
12
13 class _MyAppState extends State<MyApp> {
14   double _nombreSaisi;
15
16   @override
17   void initState() {
18     _nombreSaisi = 0.0;
19     super.initState();
20   }
21
22   @override
23   Widget build(BuildContext context) {
24     return MaterialApp(
25       title: 'Convertisseur de mesures',
26       home: Scaffold(
27         appBar: AppBar(
28           title: Text('Convertisseur de Mesures'),
29         ), // AppBar
30         body: Center(
31           child: Column(children: [
32             TextField(
33               onChanged: (texte) {
34                 var vr = double.tryParse(texte);
35                 if (vr != null) {
36                   setState(() {
37                     _nombreSaisi = vr;
38                   });
39                 }
40             ), // TextField
41             Text((_nombreSaisi == null) ? '' : _nombreSaisi.toString()),
42           ]), // Column // Center
43         ), // Scaffold
44       ); // MaterialApp
45   }
46 }
```

La méthode `initState()` est appelée une fois pour chaque objet `State` lorsque l'état est construit. C'est dans cette méthode que vous mettez généralement les valeurs initiales dont vous pourriez avoir besoin lorsque vous construisez vos classes. Dans ce cas, vous définissez la valeur initiale `_nombreSaisi`. Notez également que vous devez toujours appeler `super.initState()` à la fin de la méthode `initState()`.

Voici un diagramme qui met en évidence les étapes décrites précédemment avec quelques variations, vous utiliserez un modèle similaire chaque fois que vous utilisez des widgets avec état dans vos applications :



Pour résumer, l'appel de `setState()` fait ce qui suit:

- ✚ Avertit le framework que l'état interne de cet objet a changé
- ✚ Appelle la méthode `build()` et redessine ses widgets enfants avec l'objet `State` mis à jour

3.2.4 Utilisation du widget `DropDownButton`

`DropDownButton` est un widget qui permet aux utilisateurs de sélectionner une valeur dans une liste d'éléments. `DropDownButton` affiche l'élément actuellement sélectionné, ainsi qu'un petit triangle qui ouvre une liste pour sélectionner un autre élément.

Pour ajouter un `DropDownButton` à vos applications, vous devez utiliser les étapes suivantes :

- 1) Créez une instance de `DropDownButton`, en spécifiant le type de données qui seront incluses dans la liste.
- 2) Ajoutez une propriété qui contiendra la liste des éléments qui seront affichés à l'utilisateur. ATTENTION : La propriété `items` nécessite une liste de widgets `DropDownMenuItem`. Par conséquent, vous devez mapper chaque valeur que vous souhaitez afficher dans `DropDownMenuItem`.
- 3) Répondez aux actions de l'utilisateur en spécifiant un événement. Généralement, pour `DropDownButton`, vous appellerez une fonction dans la propriété `onChanged`.

`DropDownButton` est un générique, car il est construit comme `DropDownButton<T>`, où le type générique est le type d'élément dans votre widget `DropDownButton`.

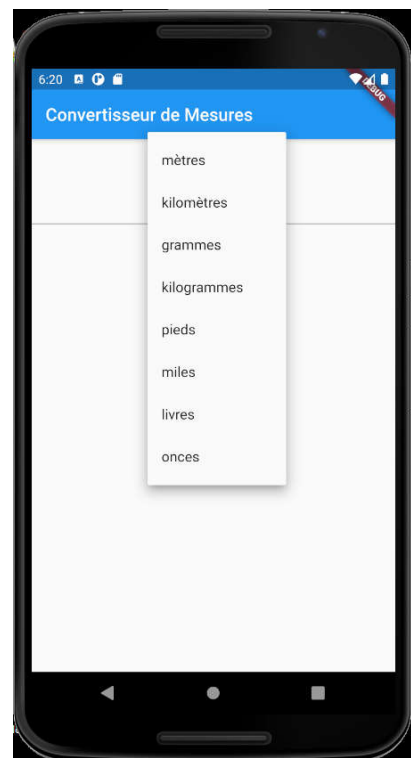
La méthode `map()` parcourt toutes les valeurs du tableau et exécute une fonction sur chaque valeur de la liste. La fonction à l'intérieur de la méthode `map()` renvoie un widget `DropDownMenuItem`, qui a une propriété `value` et une propriété `enfant`. L'enfant est ce que l'utilisateur verra. La valeur est celle que vous utiliserez pour récupérer l'élément sélectionné dans la liste. La méthode `map()` renvoie un itérable, qui est une collection de valeurs accessibles séquentiellement.

Au-dessus de cela, vous appelez la méthode `toList()`, qui crée une liste qui contient les éléments qui doivent être retournés. Ceci est requis par la propriété `items`.

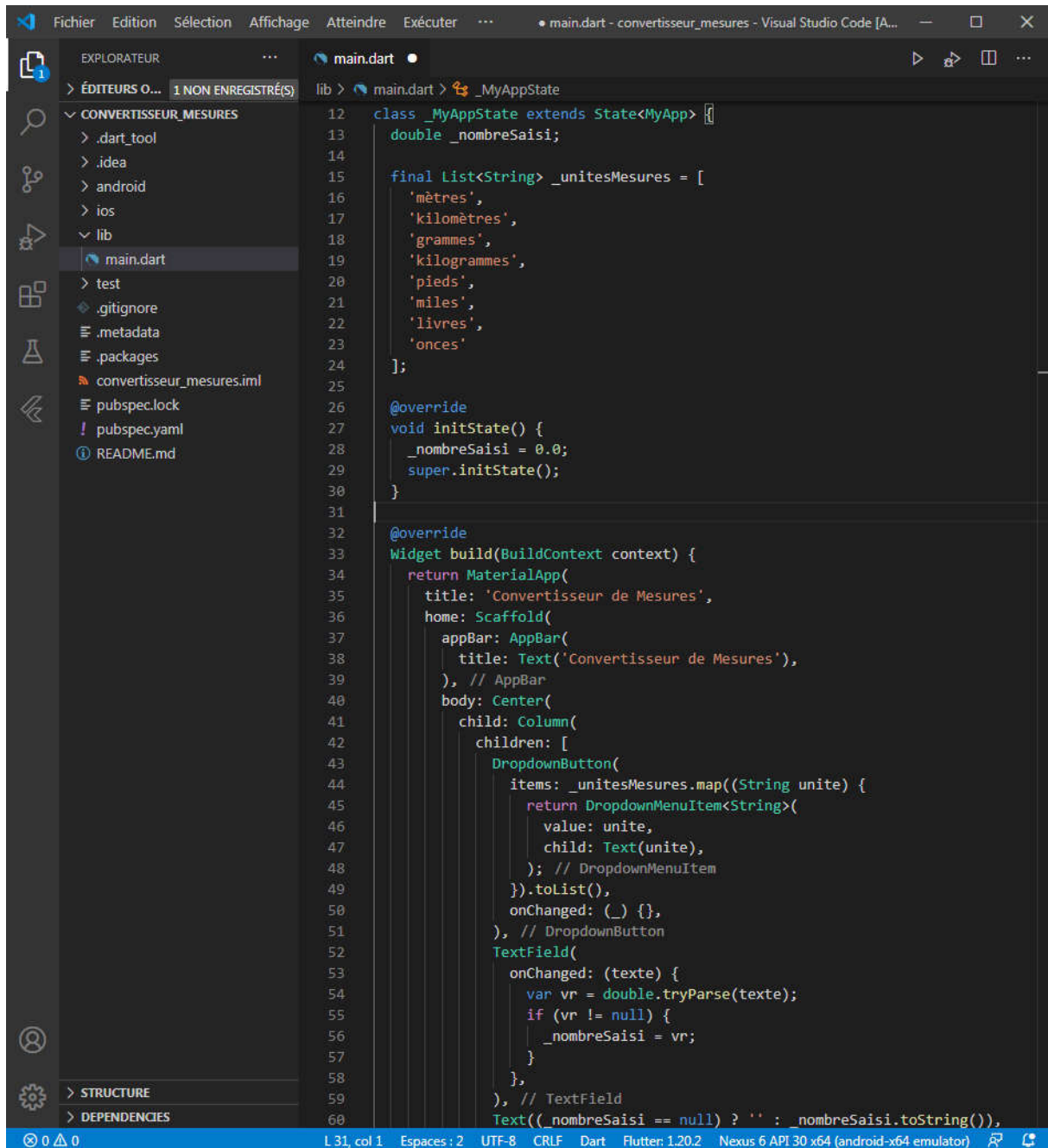
3.2.4.1 Création

- a) Vous allez créer une liste non modifiable de chaîne de caractères au début de la classe `State` que vous nommerez `_unitesMesures` avec les unités suivantes : 'mètres', 'kilomètres', 'grammes', 'kilogrammes', 'pieds', 'miles', 'livres', 'onces'.
- b) Vous créez ensuite deux widgets `DropDownButton`, un pour l'unité de départ et un pour l'unité convertie en utilisant la liste précédente et en prévoyant de répondre à l'événement `onChanged`.

⇒ Solution sur la page suivante.



TP dirigé 1 « Miles ou kilomètres »

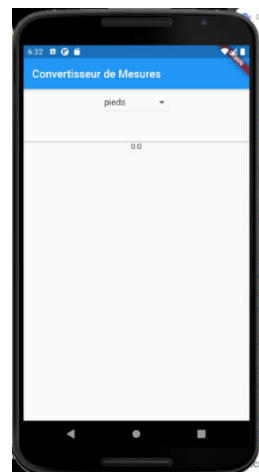


```
12 class _MyAppState extends State<MyApp> {
13   double _nombreSaisi;
14
15   final List<String> _unitesMesures = [
16     'mètres',
17     'kilomètres',
18     'grammes',
19     'kilogrammes',
20     'pieds',
21     'miles',
22     'livres',
23     'onces'
24   ];
25
26   @override
27   void initState() {
28     _nombreSaisi = 0.0;
29     super.initState();
30   }
31
32   @override
33   Widget build(BuildContext context) {
34     return MaterialApp(
35       title: 'Convertisseur de Mesures',
36       home: Scaffold(
37         appBar: AppBar(
38           title: Text('Convertisseur de Mesures'),
39         ), // AppBar
40         body: Center(
41           child: Column(
42             children: [
43               DropdownButton(
44                 items: _unitesMesures.map((String unite) {
45                   return DropdownMenuItem<String>(
46                     value: unite,
47                     child: Text(unite),
48                   ); // DropdownMenuItem
49                 }).toList(),
50                 onChanged: (_) {},
51               ), // DropdownButton
52               TextField(
53                 onChanged: (texte) {
54                   var vr = double.tryParse(texte);
55                   if (vr != null) {
56                     _nombreSaisi = vr;
57                   }
58                 },
59               ), // TextField
60               Text((_nombreSaisi == null) ? '' : _nombreSaisi.toString()),
61             ],
62           ),
63         ),
64       ),
65     );
66   }
67 }
```

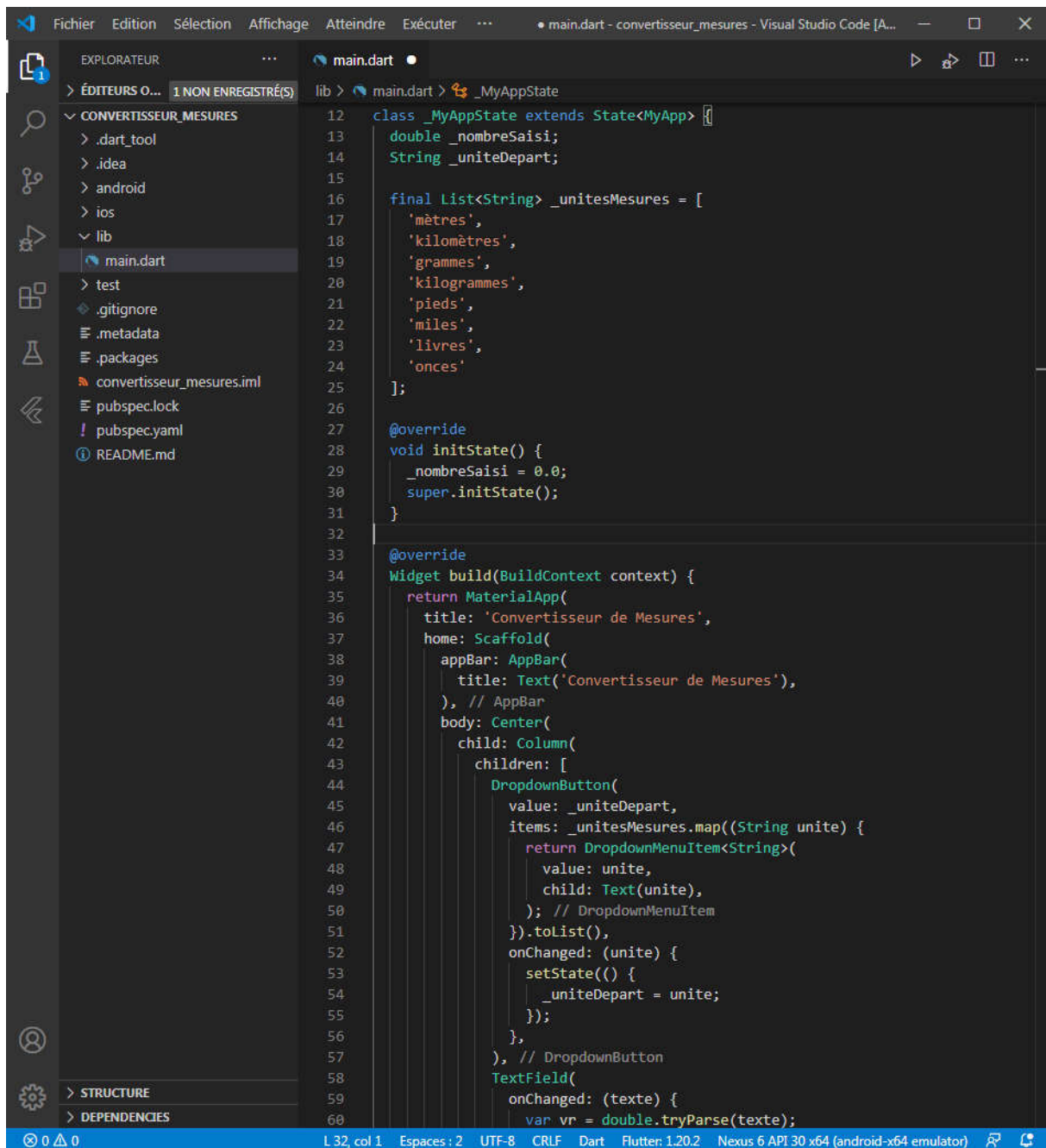
3.2.4.2 Mise à jour

Modifions la propriété *onChanged* en procédant comme suit :

- Créez une nouvelle chaîne appelée *_uniteDepart* en haut de la classe *MyAppState*. Il contiendra la valeur sélectionnée dans *DropdownButton*:
 - Au lieu du trait de soulignement dans l'événement *onChanged*, appelez le paramètre qui est passé à la fonction et appelez la méthode *setState()* pour mettre à jour *_uniteDepart* avec la nouvelle valeur.
 - Lisez la valeur sélectionnée afin de mettre à jour *DropdownButton* à chaque fois qu'elle change en ajoutant définissant la propriété *value* avec la nouvelle valeur.
- ⇒ Solution sur la page suivante.



TP dirigé 1 « Miles ou kilomètres »



```
12 class _MyAppState extends State<MyApp> {
13   double _nombreSaisi;
14   String _uniteDepart;
15
16   final List<String> _unitesMesures = [
17     'mètres',
18     'kilomètres',
19     'grammes',
20     'kilogrammes',
21     'pieds',
22     'miles',
23     'livres',
24     'onces'
25   ];
26
27   @override
28   void initState() {
29     _nombreSaisi = 0.0;
30     super.initState();
31   }
32
33   @override
34   Widget build(BuildContext context) {
35     return MaterialApp(
36       title: 'Convertisseur de Mesures',
37       home: Scaffold(
38         appBar: AppBar(
39           title: Text('Convertisseur de Mesures'),
40         ), // AppBar
41         body: Center(
42           child: Column(
43             children: [
44               DropdownButton(
45                 value: _uniteDepart,
46                 items: _unitesMesures.map((String unite) {
47                   return DropdownMenuItem<String>(
48                     value: unite,
49                     child: Text(unite),
50                   ); // DropdownMenuItem
51                 }).toList(),
52                 onChanged: (unite) {
53                   setState(() {
54                     _uniteDepart = unite;
55                   });
56                 },
57               ), // DropdownButton
58               TextField(
59                 onChanged: (texte) {
60                   var vr = double.tryParse(texte);
```

Maintenant, si vous essayez l'application, lorsque vous sélectionnez une valeur dans la liste, la valeur apparaît dans *DropdownButton*, qui est exactement le comportement que vous attendez d'elle.

3.2.5 Compléter l'UI pour la page d'accueil

Complétez maintenant l'interface utilisateur de votre application. Le résultat final est montré dans la capture d'écran ci-contre :



Vous devez en fait afficher huit widgets à l'écran :

1. *Text* contenant '**Valeur à convertir**'
2. *TextField* pour la valeur de départ
3. *Text* contenant '**Depuis**'
4. *DropDownButton* pour choisir l'unité de départ
5. *Text* contenant '**Vers**'
6. *DropDownButton* pour choisir l'unité d'arrivée après la conversion
7. *RaisedButton* pour appeler la méthode qui convertira la valeur.
8. *Text* pour le résultat de la conversion

Chaque élément de la colonne doit également être espacé et stylisé.

- a) Pour commencer, je vous propose de définir deux widgets *TextStyle* (nommé *styleEntree* et *styleLabel*) en haut de la méthode *build()* à l'aide de l'exemple ci-dessous. L'avantage de cette approche est que vous pouvez les utiliser plusieurs fois sans avoir besoin de spécifier les détails de style pour chaque widget.

```
final TextStyle styleEntree = TextStyle(
  fontSize: 20,
  color: Colors.blue[900],
);
final TextStyle styleLabel = TextStyle(
  fontSize: 20,
  color: Colors.grey[700],
);
```

- b) Il serait bien que le widget *Column* prenne une certaine distance par rapport aux bords horizontaux de l'appareil. Ainsi, au lieu de renvoyer un widget *Center*, nous pouvons renvoyer un widget *Container*, qui prend un remplissage de 20 pixels. *EdgeInsets.symmetric* vous permet de spécifier une valeur pour le remplissage horizontal ou vertical.

```
body: Container(
  padding: EdgeInsets.symmetric(horizontal: 20),
  child: Column(
    children: [
```

- c) En parlant d'espacement, vous voulez aussi espacer verticalement les widgets entre eux dans la colonne. Un moyen simple d'y parvenir consiste à utiliser le widget *Spacer*. *Spacer* crée un espace vide qui peut être utilisé pour définir l'espacement entre les widgets dans un conteneur flexible, tel que la colonne ou ligne. Un widget *Spacer* a une propriété *flex*, dont la valeur par défaut est 1, qui détermine le nombre d'espace que nous voulons utiliser (flex : 1 (par défaut) utilise 1 espace, flex:2 utilise 2 espaces, ...). Commencez par ajouter un espace en haut de la colonne :

```
child: Column(
  children: [
    Spacer(),
```

- d) Sous ce dernier widget, ajoutez le premier texte de la colonne contenant la chaîne «**Valeur à convertir**». Vous n'oubliez pas appliquer *styleEntree* à ce widget, et de remettre un widget *Spacer* en dessous :

```
Text(
  'Valeur à convertir',
  style: styleEntree,
), // Text
Spacer(),
```

- e) Sous ce dernier widget, vous devez placer le *TextField* que nous avons créé précédemment, pour permettre à l'utilisateur de saisir le nombre qu'il souhaite convertir. Modifiez *TextField* pour qu'il utilise le *styleEntree*. Vous allez également définir la propriété de décoration du *TextField*.

```
TextField(
  style: styleEntree,
  textAlign: TextAlign.center,
  decoration: InputDecoration(
    hintText: 'Saisissez la mesure à convertir',
  ), // InputDecoration
  onChanged: (texte) {
    var vr = double.tryParse(texte);
    if (vr != null) {
      _nombreSaisi = vr;
    }
  },
), // TextField
```

hintText est un texte qui est affiché lorsque *TextField* est vide, pour suggérer le type d'entrée attendu de l'utilisateur. Dans ce cas, ajoutez "Saisissez la mesure à convertir" comme une invite *hintText* pour notre *TextField* en appliquant alignement du texte en centré.

- f) Sous *TextField*, placez un nouveau *Spacer*, puis un widget *Text* contenant '**Depuis**' et le style *styleLabel*.
- g) Sous le texte '**Depuis**', placez le widget *DropDownButton*, dont la valeur est *_uniteDepart*, que vous avez défini précédemment.
- h) Ajoutez un autre widget *Text* avec le texte '**Vers**', et le style sera *styleLabel*, comme auparavant.
- i) Sous le texte 'Vers', vous devez placer le deuxième widget *DropDownButton*, et cela nécessite une autre variable de la classe *MyAppState* de type *String _uniteArrivee*. Le premier widget *DropDownButton* a utilisé *_uniteDepart* pour sa valeur ; ce nouveau utilisera *_uniteArrivee*. Comme d'habitude, n'oubliez pas d'ajouter un *Spacer* avant le widget.
- j) Ensuite, ajoutez le bouton qui appellera la méthode de conversion. Ce bouton sera un widget *RaisedButton* avec un texte '**Convertir**', et le style de *styleEntree*. Pour le moment, l'événement *onPressed* ne fera rien, car nous n'avons pas encore la logique métier de l'application. Avant et après le bouton, n'oubliez pas de placer un *Spacer*, mais cette fois, nous définirons également sa propriété *flex* sur 2 (pour mettre double espaces).

```
Spacer(  
  flex: 2,  
), // Spacer  
RaisedButton(  
  child: Text(  
    'Convertir',  
    style: styleEntree,  
  ), // Text  
  onPressed: () => true,  
), // RaisedButton  
Spacer(  
  flex: 2,  
), // Spacer
```

- k) Enfin, vous ajouterez le texte du résultat de la conversion. Pour l'instant, laissons simplement la valeur `_nombreSaisi` comme `Text`.
- l) À la fin du résultat, nous ajouterons le plus grand `Spacer` de cet écran, avec une valeur de `flex` de 8, afin de laisser un peu d'espace en bas de l'écran.

```
Text(  
  (_nombreSaisi == null) ? '' : _nombreSaisi.toString(),  
  style: styleLabel,  
), // Text  
Spacer(  
  flex: 8,  
), // Spacer
```

⇒ La solution à partir de la page suivante.


```

1  import 'package:flutter/material.dart';
2
3  Run | Debug
4  void main() {
5    runApp(MyApp());
6  }
7
8  class MyApp extends StatefulWidget {
9    @override
10   _MyAppState createState() => _MyAppState();
11 }
12
13 class _MyAppState extends State<MyApp> {
14   double _nombreSaisi;
15   String _uniteDepart;
16   String _uniteArrive;
17
18   final List<String> _unitesMesures = [
19     'mètres',
20     'kilomètres',
21     'grammes',
22     'kilogrammes',
23     'pieds',
24     'miles',
25     'livres',
26     'onces'
27   ];
28
29   @override
30   void initState() {
31     _nombreSaisi = 0.0;
32     super.initState();
33   }
34
35   @override
36   Widget build(BuildContext context) {
37     final TextStyle styleEntree = TextStyle(
38       fontSize: 20,
39       color: Colors.blue[900],
40     );
41     final TextStyle styleLabel = TextStyle(
42       fontSize: 20,
43       color: Colors.grey[700],
44     );
45     return MaterialApp(
46       title: 'Convertisseur de Mesures',
47       home: Scaffold(
48         appBar: AppBar(
49           title: Text('Convertisseur de Mesures'),

```

```

49     ), // AppBar
50     body: Container(
51       padding: EdgeInsets.symmetric(horizontal: 20),
52       child: Column(
53         children: [
54           Spacer(),
55           Text(
56             'Valeur à convertir',
57             style: styleEntree,
58           ), // Text
59           Spacer(),
60           TextField(
61             style: styleEntree,
62             textAlign: TextAlign.center,
63             decoration: InputDecoration(
64               hintText: 'Saisissez la mesure à convertir',
65             ), // InputDecoration
66             onChanged: (texte) {
67               var vr = double.tryParse(texte);
68               if (vr != null) {
69                 _nombreSaisi = vr;
70               }
71             },
72           ), // TextField
73           Spacer(),
74           Text(
75             'Depuis',
76             style: styleLabel,
77           ), // Text
78           Spacer(),
79           DropdownButton(
80             value: _uniteDepart,
81             items: _unitesMesures.map((String unite) {
82               return DropdownMenuItem<String>(
83                 value: unite,
84                 child: Text(unite),
85               ); // DropdownMenuItem
86             }).toList(),
87             onChanged: (unite) {
88               setState(() {
89                 _uniteDepart = unite;
90               });
91             },
92           ), // DropdownButton
93           Spacer(),
94           Text(
95             'Vers',
96             style: styleLabel,
97           ), // Text

```

```

98     Spacer(),
99     DropdownButton(
100       value: _uniteArrive,
101       items: _unitesMesures.map((String unite) {
102         return DropdownMenuItem<String>(
103           value: unite,
104           child: Text(unite),
105         ); // DropdownMenuItem
106       }).toList(),
107       onChanged: (unite) {
108         setState(() {
109           _uniteArrive = unite;
110         });
111       },
112     ), // DropdownButton
113     Spacer(
114       flex: 2,
115     ), // Spacer
116     RaisedButton(
117       child: Text(
118         'Convertir',
119         style: styleEntree,
120       ), // Text
121       onPressed: () => true,
122     ), // RaisedButton
123     Spacer(
124       flex: 2,
125     ), // Spacer
126     Text(
127       (_nombreSaisi == null) ? '' : _nombreSaisi.toString(),
128       style: styleLabel,
129     ), // Text
130     Spacer(
131       flex: 8,
132     ), // Spacer
133   ],
134 ), // Column
135 ), // Container
136 ), // Scaffold
137 ); // MaterialApp
138 }
139 }
140

```

3.3 Ajout de la logique métier

Vous avez terminé la mise en page de l'application, mais pour le moment, il manque à l'application la partie qui convertit les valeurs basées sur l'entrée de l'utilisateur. De manière générale, c'est toujours une bonne idée de séparer la logique de vos applications de l'interface utilisateur.

Vous en utiliserez certains, tels que `ScopedModel` et `Business Logic Components (BLoC)`, dans les TP suivants, mais pour l'instant, nous pouvons simplement ajouter les fonctions de conversion dans notre classe.

Il existe certainement plusieurs façons d'écrire le code pour effectuer la conversion entre les mesures pour cette application. L'approche la plus simple consiste à voir les formules que nous devons appliquer sous forme de tableau bidimensionnel, également appelé matrice. Cette matrice contient toutes les combinaisons de choix possibles que l'utilisateur peut effectuer. Un diagramme de cette approche est présenté ci-dessous :

| MESURES | 0-mètres | 1-kilomètres | 2-grammes | 3-kilogrammes | 4-pied | 5-miles | 6-livres | 7-onces |
|---------------|----------|--------------|-----------|---------------|---------|---------|----------|---------|
| 0-mètres | 1 | 0.0001 | 0 | 0 | 3.28084 | 0.00062 | 0 | 0 |
| 1-kilomètres | 1000 | 1 | 0 | 0 | 3280.84 | 0.62137 | 0 | 0 |
| 2-grammes | 0 | 0 | 1 | 0.0001 | 0 | 0 | 0.0022 | 0.03527 |
| 3-kilogrammes | 0 | 0 | 1000 | 1 | 0 | 0 | 2.20462 | 35.274 |
| 4-pied | 0.3048 | 0.0003 | 0 | 0 | 1 | 0.00019 | 0 | 0 |
| 5-miles | 1609.34 | 1.60934 | 0 | 0 | 5280 | 1 | 0 | 0 |
| 6-livres | 0 | 0 | 453.592 | 0.45359 | 0 | 0 | 1 | 16 |
| 7-onces | 0 | 0 | 28.3495 | 0.022835 | 0 | 0 | 0.0625 | 1 |

Ainsi, par exemple, lorsque vous souhaitez convertir 100 kilomètres en miles, vous multipliez 100 par le nombre que vous trouvez dans le tableau (dans ce cas, 0,62137). Lorsque la conversion n'est pas possible, le multiplicateur est 0, donc toute conversion impossible retourne 0. Vous utiliserez `List` pour créer le tableau. Dans ce cas, c'est un tableau ou une matrice à deux dimensions, et par conséquent, vous allez créer un objet qui contient `List`.

- a) Vous devriez convertir les chaînes des unités de mesure en nombres. En haut de la classe `MyAppState`, ajoutez le code suivant :

```
final Map<String, int> _mesuresMap = {
  'mètres': 0,
  'kilomètres': 1,
  'grammes': 2,
  'kilogrammes': 3,
  'pied': 4,
  'miles': 5,
  'livres': 6,
  'onces': 7,
};
```

Les `Map` vous permettent d'insérer des paires clé-valeur, où le premier élément est la clé et le second est la valeur. Lorsque vous avez besoin de récupérer une valeur de `Map`, vous pouvez utiliser la syntaxe suivante :

```
maValeur = _mesuresMap['miles']; // maValeur recupere la valeur 5
```

- b) Ensuite, vous devez créer une liste qui contient tous les multiplicateurs indiqués dans le diagramme précédent :

```
final dynamic _formules = {
  '0': [1, 0.001, 0, 0, 3.28084, 0.000621371, 0, 0],
  '1': [1000, 1, 0, 0, 3280.84, 0.621371, 0, 0],
  '2': [0, 0, 1, 0.0001, 0, 0, 0.00220462, 0.035274],
  '3': [0, 0, 1000, 1, 0, 0, 2.20462, 35.274],
  '4': [0.3048, 0.0003048, 0, 0, 1, 0.000189394, 0, 0],
  '5': [1609.34, 1.60934, 0, 0, 5280, 1, 0, 0],
  '6': [0, 0, 453.592, 0.453592, 0, 0, 1, 16],
  '7': [0, 0, 28.3495, 0.0283495, 3.28084, 0, 0.0625, 1],
};
```

- c) Maintenant que vous avez créé une matrice qui contient toutes les combinaisons possibles de formules de conversion, il suffit d'écrire la méthode qui convertira les valeurs à l'aide des formules et de la Map des mesures en ajoutant le code suivant au bas de la classe *MyAppState* :

```
void convertir(double valeur, String depuis, String vers) {
  int numDepuis = _mesuresMap[depuis];
  int numVers = _mesuresMap[vers];
  var multiplicateur = _formules[numDepuis.toString()][numVers];
  var resultat = valeur * multiplicateur;
}
```

La méthode *convertir()* prend trois paramètres:

- ✚ La valeur à convertir (*double valeur*),
 - ✚ L'unité de mesure dans laquelle cette valeur est actuellement exprimée (*String depuis*),
 - ✚ L'unité de mesure dans laquelle la valeur sera convertie (*String vers*).
- d) Ensuite, vous devez afficher le résultat de la conversion à l'utilisateur. Pour cela, déclarez une variable *String _message*; en haut de la classe *MyAppState* :

```
String _message;
```

- e) Dans la méthode *convertir()*, après avoir calculé le résultat, remplissez la chaîne *_message* et appelez la méthode *setState()* pour notifier le framework qu'une mise à jour de l'interface utilisateur est nécessaire:

```
if (resultat == 0) {
  _message = 'Cette conversion ne peut être réalisée';
} else {
  _message = '${_nombreSaisi.toString()} $_uniteDepart est égal à ${resultat.toString()} $_uniteArrive';
}
setState(() {
  _message = _message;
});
```

- f) Enfin, vous devez appeler la méthode *convertir()* lorsque l'utilisateur appuie sur le bouton **Convertir**. Avant d'appeler la méthode, nous vérifierons que chaque valeur a été définie pour éviter les erreurs potentielles.

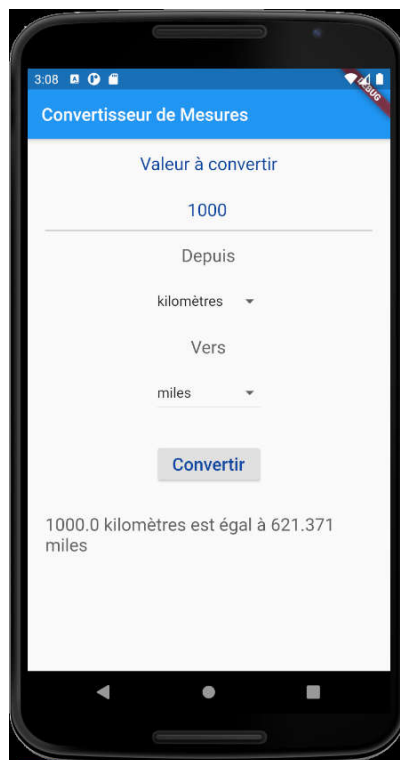
g) Modifiez *RaisedButton*, comme indiqué ci-dessous :

```
onPressed: () {
  if (_uniteDepart.isEmpty ||
      _uniteArrive.isEmpty ||
      _nombreSaisi == 0) {
    return;
  } else {
    convertir(_nombreSaisi, _uniteDepart, _uniteArrive);
  }
}), // RaisedButton
```

h) Enfin, affichez le résultat en mettant à jour le widget *Text*, afin qu'il affiche la chaîne qui contient le message à l'utilisateur :

```
Text(
  (_nombreSaisi == null) ? '' : _nombreSaisi.toString(),
  style: styleLabel,
), // Text
```

Félicitations, l'application est maintenant terminée ! Si vous l'essayez maintenant, vous devriez voir un écran comme celui-ci :



3.4 Challenge

Découper le fichier *main.dart* en deux. L'une avec la partie principale (interface utilisateur et la fonction *main*), l'autre avec la fonction métier '*convertir*' transformée en classe appellable.