

« Mobile & Image »  
« Développement d'une application mobile »  
TP dirigé 2 « Temps qui passe . . . »

---

Par Alexis GUILLOTEAU & Didier JUGE-HUBERT

# Sommaire

---

1	Introduction.....	2
2	Présentation de l'application .....	3
3	Etapes de création de l'application « gestion_temps » .....	4
3.1	Création du nouveau projet .....	4
3.2	Création de la page d'accueil .....	4
3.2.1	Structure type d'une application de base .....	4
3.2.2	Création de votre widget générique .....	6
3.2.3	Mettre les boutons sur la page d'accueil .....	7
3.2.4	Utilisation d'un widget tierce : <i>percent_indicator</i> .....	11
3.3	Ajout de la logique métier .....	13
3.3.1	Utilisation d'un flux et de la programmation asynchrone dans Flutter .....	14
3.3.2	Programmation de <i>Stream</i> .....	14
3.3.3	Affichage dans la page principale : <i>StreamBuilder</i> .....	16
3.3.4	Activation des boutons.....	17
3.3.4.1	Code des boutons du pied de page .....	17
3.3.4.2	Code des boutons de l'entête .....	18
3.4	Challenge .....	21
3.4.1	Créer la base d'une page de paramètres .....	21
3.4.2	Créer la navigation entre ces deux pages.....	23
3.4.3	Création de la disposition de la page « Paramètres » .....	24
3.4.3.1	Utilisation du constructeur <i>GridView.Count ()</i> .....	24
3.4.3.2	Ajout des boutons dans le <i>GridView</i> .....	25
3.4.4	Utilisation de <i>SharedPreferences</i> pour lire et écrire des données d'application .....	28

## 1 Introduction






Le travail en profondeur est dans un état de concentration qui vous permet de maximiser vos capacités cognitives. Vous pouvez utiliser le travail en profondeur lorsque vous étudiez un nouveau langage comme celui-ci, lorsque vous écrivez une application, en bref, chaque fois que vous devez effectuer un travail qui crée de la valeur ou améliore vos compétences.

Il existe une solution simple, et nous aborderons cela dans l'application que vous allez créer dans ce TP dirigé. Vous devez planifier le temps de travail et le temps de pause.

Dans ce TP dirigé, vous allez créer une application qui vous aidera à définir les intervalles de temps qui vous conviennent et à mesurer votre temps de travail et votre temps de pause. En fait, vous allez créer une application de productivité contenant un compte à rebours qui vous indique votre temps de travail ou de pause restant, avec une animation à l'écran.

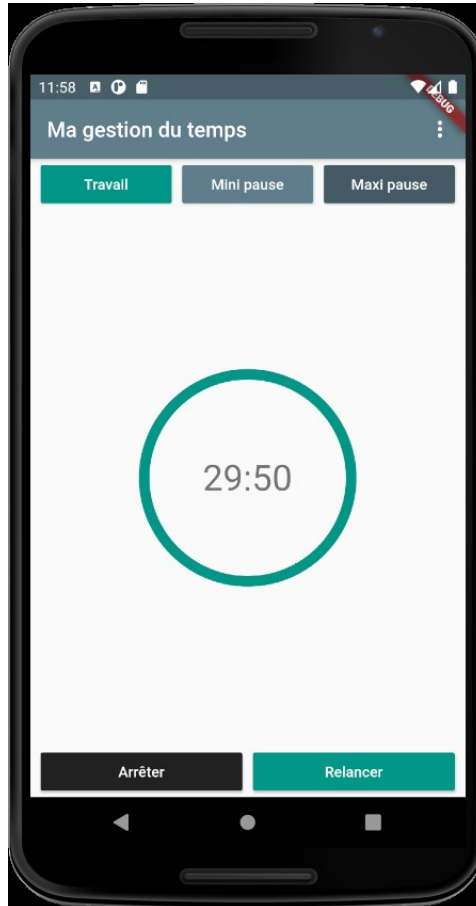
Pendant le challenge, sur une autre page, vous pourrez également définir les durées de références pour votre temps de travail, vos courtes pauses et vos longues pauses, et les enregistrer sur votre appareil.

A la fin du TP, vous saurez comment utiliser *Stream* et *StreamBuilder*, ajouter une navigation simple à vos applications, intégrer des bibliothèques externes dans vos projets Flutter et utiliser *SharedPreferences* pour conserver les données. Ce sera un bon exercice pour apprendre plusieurs fonctionnalités importantes de Flutter que nous n'aurions pas déjà abordées pendant le cours, telles que :

-  Créer une mise en page en tirant parti d'une bibliothèque externe
-  Écouter des flux de données et utiliser la programmation asynchrone
-  Naviguer d'un écran à un autre dans votre application
-  Utilisation des préférences partagées pour conserver les données sur votre appareil
-  Utilisation d'un *GridView* et choix des bonnes couleurs pour votre application

## 2 Présentation de l'application

Dans la capture d'écran suivante, vous pouvez voir la mise en page que nous allons construire dans cette première partie. Afin de faciliter la compréhension de ce que nous devons faire pour cette mise en page, j'ai ajouté des bordures qui montrent comment les widgets seront placés à l'écran :



Je pense que le moyen le plus simple de créer cette mise en page consiste à utiliser une combinaison de widgets Colonne (*Column*) et Ligne (*Row*). Le widget conteneur principal de cet écran sera une colonne qui divisera l'espace en trois parties, comme suit :

1. Les trois boutons en haut : « Travail », « Mini Pause » et « Maxi Pause »
2. Le minuteur au milieu
3. Les deux boutons en bas : « Arrêter » et « Relancer »

### 3 Etapes de création de l'application « gestion\_temps »

#### 3.1 Création du nouveau projet

Vous allez maintenant créer une nouvelle application que nous utiliserons tout au long de ce TP pour créer le minuteur de gestion du temps. À partir de votre éditeur préféré, créez une nouvelle application et nommez la nouvelle application « gestion\_temps », vous pouvez vous aider des étapes ci-dessous :

- a) Je vous propose de créer un répertoire « AppsFlutter » sous le répertoire « tools » qui a été défini lors de l'installation.
- b) Démarrer votre émulateur avant la création du projet afin qu'il soit reconnu pour l'exécution.
- c) Pour créer le projet via Visual Studio Code,
  1. Dans le menu « **Affichage => Palette de commandes...** ».
  2. Saisissez dans le champ de recherche « > » le texte « *flutter* ».
  3. Choisissez « *Flutter : New Project* » puis saisissez le nom de votre nouveau projet « gestion\_temps » puis validez avec « **Entrée** ».

#### 3.2 Création de la page d'accueil

C'est un début générique pour une application Flutter de base, que vous pouvez réutiliser chaque fois que vous démarrez un nouveau projet.

##### 3.2.1 Structure type d'une application de base

- a) Dans le fichier *main.dart*, supprimez l'exemple de code, puis créez la fonction *main* ainsi que le widget *Stateless* de l'application.
- b) Dans le *build* du widget application, saisissez le code ci-dessous :

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: "Gestion du temps",
      theme: ThemeData(
        primaryColor: Colors.blueGrey,
      ),
      home: PageAccueilMinuterie(),
    );
  }
}
```

- c) Vous remarquerez que vous importez le fichier '*page\_accueil\_minuterie.dart*' qui contiendra le code de la page d'accueil de votre application.



Ce code crée un *Scaffold*, qui est la disposition de base pour la plupart de vos écrans, et place un titre dans *AppBar* et un texte au centre du corps. Le résultat doit être similaire à la capture d'écran suivante :



Si vous utilisez VS Code, Android Studio ou IntelliJ IDEA, vous pouvez également utiliser le raccourci *stless* pour que le framework écrive une partie de votre code. En début de ligne dans votre fichier '*page\_accueil\_minuterie.dart*', tapez simplement *stless* et validez, puis saisissez le nom de la classe *PageAccueilMinuterie*. Le résultat final doit être le suivant :

```
class PageAccueilMinuterie extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return Container(  
  
    );  
  }  
}
```

### 3.2.2 Création de votre widget générique

Maintenant, vous êtes prêts à commencer à placer les widgets sur l'écran. Comme vous devez créer cinq widgets boutons qui auront des fonctionnalités très similaires, une bonne pratique est de créer une nouvelle classe pour ceux-ci, afin de garder le reste du code plus propre et d'économiser un peu de frappe.

- Créez un nouveau fichier dans le dossier *lib* de l'application, appelé « *widgets.dart* ».
- Créez un nouveau widget *Stateless* appelé *BoutonGenerique*. Ce widget exposera quatre champs, avec un constructeur qui définit ces valeurs : une couleur, un texte, une taille et une méthode d'action.

```

class BoutonGenerique extends StatelessWidget {
  final Color couleur;
  final String texte;
  final double taille;
  final VoidCallback action;
  BoutonGenerique(
    {@required this.couleur,
     @required this.texte,
     @required this.taille,
     @required this.action});
  @override
  Widget build(BuildContext context) {
    return MaterialButton(
      child: Text(
        this.texte,
        style: TextStyle(
          color: Colors.white,
        ),
      ),
      onPressed: this.action,
      color: this.couleur,
      minWidth: this.taille,
    );
  }
}

```

Vous avez peut-être remarqué que les paramètres sont inclus entre accolades ({} ) et ont une annotation `@required`. Vous utilisez ici des paramètres nommés.

Pour rappel, le but de l'utilisation de paramètres nommés est que lorsque vous appelez la fonction et que vous transmettez des valeurs, vous spécifiez également le nom du paramètre que vous définissez. Par exemple, lors de la création d'une instance de *BoutonGenerique*, vous pouvez utiliser la syntaxe *BoutonGenerique(couleur: Colors.blueAccent, texte: 'Bonjour', taille: 20.0, action: quelque\_chose)*. Les paramètres nommés étant référencés par leur nom, ils peuvent être utilisés dans n'importe quel ordre. Les paramètres nommés sont facultatifs, mais vous pouvez les annoter avec l'annotation `@required` pour indiquer que le paramètre est obligatoire.

### 3.2.3 Mettre les boutons sur la page d'accueil

Maintenant que vous avez créé un widget bouton générique, vous devez placer quelques instances du bouton sur l'écran. Les boutons du haut doivent être placés sur une seule ligne en haut de l'écran. Ils doivent prendre tout l'espace horizontal disponible, économiser de l'espace pour les marges et faire varier leur largeur en fonction de la taille et de l'orientation de l'écran.

Vous pouvez créer une méthode vide temporaire pour avoir une méthode à transmettre aux boutons. Vous la supprimerez plus tard.

```
void methodevide () {}
```



- a) Je vous propose de déclarer une constante pour le remplissage par défaut que nous voulons utiliser dans notre écran, de la manière suivante avant la classe `MyApp` dans `main.dart`. Cette constante sera de haut niveau.

```
const double REMPLISSAGE_DEFAULT = 5.0;
```

- b) Maintenant, placez les boutons du haut sur l'écran. Pour cela vous devez utiliser un widget `Row`, et l'inclure comme premier élément du widget `Column`. Dans Flutter, il est en fait possible d'inclure des widgets `Row` dans les widgets `Column`, et l'inverse est également vrai. Nous voulons que les boutons prennent tout l'espace horizontal disponible. Pour ce faire, nous utiliserons un widget `Expanded` qui prend tout l'espace disponible d'une colonne (ou d'une ligne) après avoir placé les éléments fixes. Chaque bouton aura un `padding` avant et après, pour créer un espace entre les éléments.

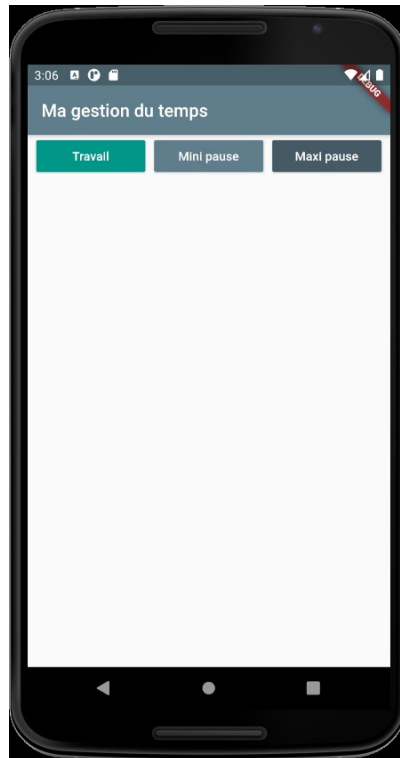
```
body: LayoutBuilder(
  builder: (BuildContext context, BoxConstraints constraints) {
    return Column(
      children: <Widget>[
        Row(
          children: <Widget>[
            Padding(
              padding: EdgeInsets.all(REPLISSAGE_DEFAULT),
            ),
            Expanded(
              child: BoutonGenerique(
                couleur: Color(0xff009688),
                texte: 'Travail',
                taille: 20.0,
                action: () => methodeVide(),
              ),
            ),
            Padding(
              padding: EdgeInsets.all(REPLISSAGE_DEFAULT),
            ),
            Expanded(
              child: BoutonGenerique(
                couleur: Color(0xff607D8B),
                texte: 'Mini pause',
                taille: 20.0,
                action: () => methodeVide(),
              ),
            ),
            Padding(
              padding: EdgeInsets.all(REPLISSAGE_DEFAULT),
            ),
            Expanded(
              child: BoutonGenerique(
                couleur: Color(0xff455A64),
```

```

        texte: 'Maxi pause',
        taille: 20.0,
        action: () => methodeVide(),
    ),
),
Padding(
  padding: EdgeInsets.all(REMPLISSAGE_DEFAULT),
),
],
),
],
);
},
)

```

- c) Essayez l'appli. Le résultat du code précédent doit être similaire à la capture d'écran suivante :



- d) La minuterie devra être placée au milieu de l'écran et prendre tout l'espace restant après avoir placé les lignes de boutons en haut et en bas. Pour l'instant, nous allons simplement utiliser un texte "Bonjour" comme espace réservé, sous le widget Colonne. Notez que, dans ce cas, *Expanded* est utilisé dans une colonne au lieu de la ligne, donc il prend tout l'espace vertical disponible, comme illustré dans l'extrait de code suivant :

```

Expanded(
  child: Text('Bonjour')
),
Row(
  children: <Widget>[

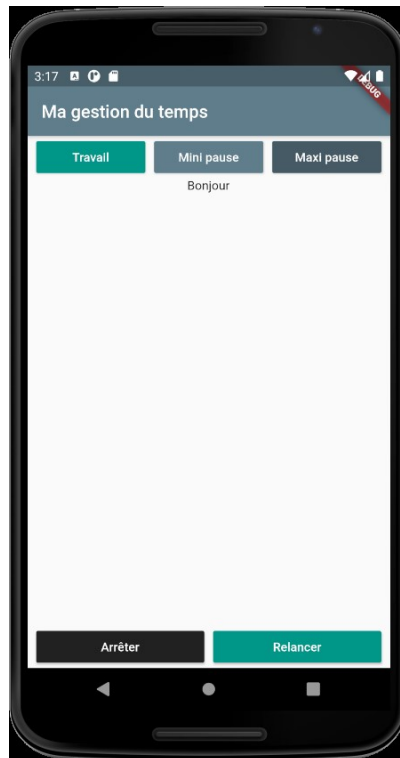
```

```

        Padding(
          padding: EdgeInsets.all(REPLISSAGE_DEFAULT),
        ),
        Expanded(
          child: BoutonGenerique(
            couleur: Color(0xff212121),
            texte: 'Arrêter',
            taille: 20.0,
            action: () => methodeVide(),
          ),
        ),
        Padding(
          padding: EdgeInsets.all(REPLISSAGE_DEFAULT),
        ),
        Expanded(
          child: BoutonGenerique(
            couleur: Color(0xff009688),
            texte: 'Relancer',
            taille: 20.0,
            action: () => methodeVide(),
          ),
        ),
        Padding(
          padding: EdgeInsets.all(REPLISSAGE_DEFAULT),
        ),
      ],
    ),
  ),

```

e) Le résultat final devrait ressembler à la capture d'écran suivante :



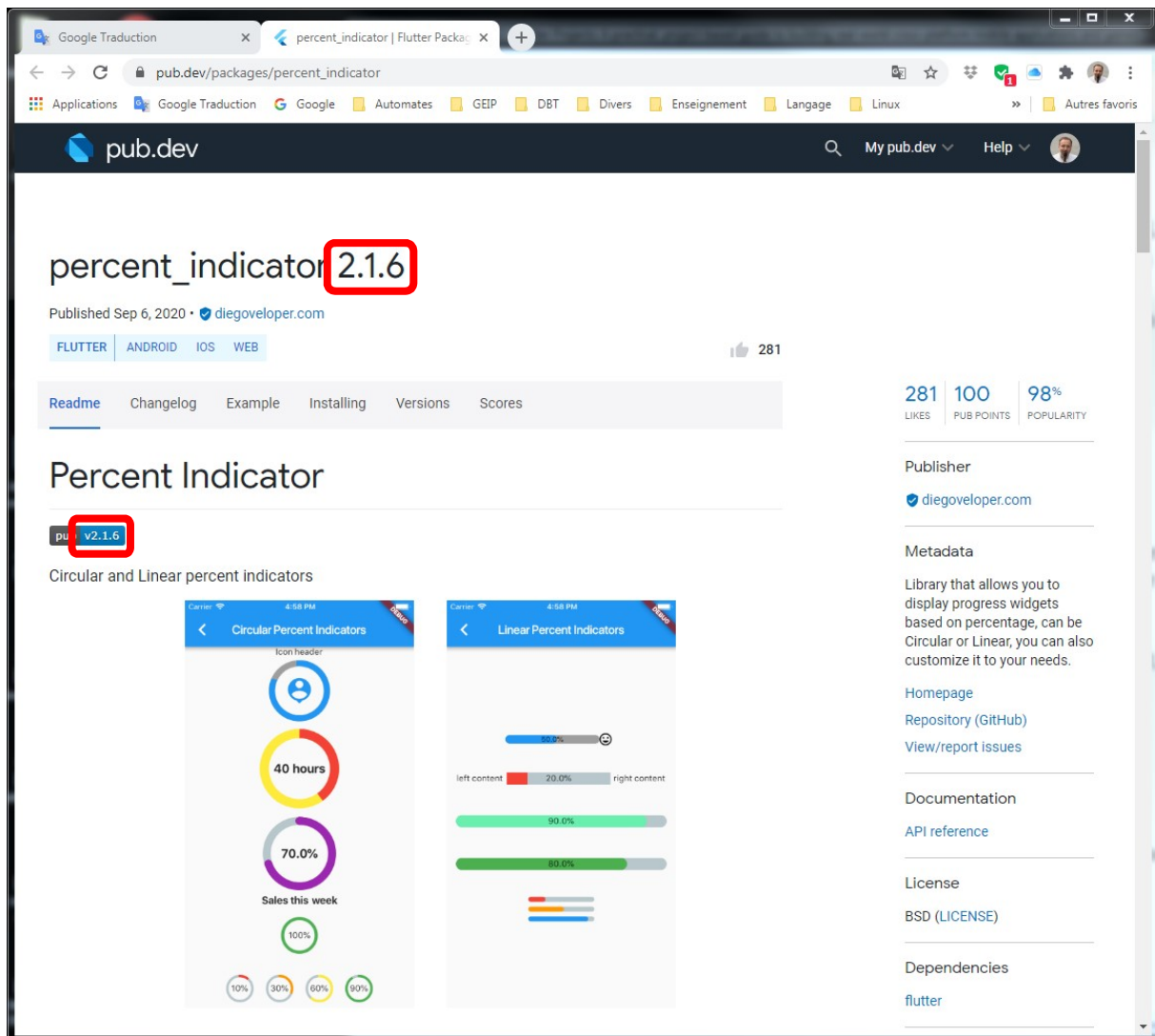
Maintenant, vous avez terminé la mise en page des boutons de votre application, mais vous devez toujours placer le contenu principal au centre de l'écran, qui est le minuteur lui-même.

### 3.2.4 Utilisation d'un widget tierce : *percent\_indicator*

Pour utiliser un widget tiers, vous devez obtenir les informations sur le package le contenant : nom du package, version, ... sur le principal site de package : <https://pub.dev/flutter>. Vous allez utiliser une procédure permettant d'installer n'importe quel package dans vos applications Flutter.

Dans votre exemple, vous devez placer un indicateur de pourcentage au centre de l'écran à la place du texte "Bonjour". Pour celui-ci, vous utiliserez le widget *CircularPercentIndicator*, qui est inclus dans le package *percent\_indicator*.

- a) Obtenez via le web les informations d'utilisation de ce package sur le site en recherchant *percent\_indicator* sur le site Web <https://pub.dev/flutter>. Le premier résultat devrait être le package dont vous avez besoin, qui est la bibliothèque *percent\_indicator*, comme indiqué dans la capture d'écran suivante :



C'est un widget qui facilite la création d'indicateurs de pourcentage circulaires et linéaires dans vos applications. Dans Flutter, les packages sont des morceaux de code réutilisables généralement développés par la communauté que vous pouvez inclure dans vos projets. À l'aide de packages, vous pouvez créer rapidement une application sans avoir à tout développer à partir de zéro.

La page du package affiche des informations et des exemples sur la façon d'installer et d'utiliser le package. En particulier, pour tout package, nous devons ajouter la dépendance dans le fichier de configuration *pubspec.yaml*.

- b) Ouvrez le fichier *pubspec.yaml* dans le dossier racine de votre application. Chaque projet Flutter a un fichier nommé *pubspec.yaml*. Comme vous le savez déjà, c'est ici que vous spécifiez les dépendances requises dans votre projet Flutter. Recherchez la section des dépendances et ajoutez la dépendance *percent\_indicator* sous le SDK Flutter avec le numéro de version récupéré sur le site web précédent (encadré en rouge sur la copie d'écran du site) :

```
dependencies:
  flutter:
    sdk: flutter
  percent_indicator: ^2.1.6
```

La dépendance *percent\_indicator* DOIT être indentée comme la dépendance flutter, comme indiqué dans le code précédent, car les fichiers YAML utilisent l'indentation pour représenter les relations entre les couches.

- c) Ensuite, de retour dans le fichier *page\_accueil\_minuterie.dart*, ajoutez l'importation *percent\_indicator* :

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

- d) Ensuite, dans la colonne, supprimez le texte "Bonjour", et, à sa place, utilisez un *CircularPercentIndicator*. Nous l'inclurons dans un widget *Expanded* afin qu'il prenne tout l'espace vertical disponible dans la colonne. Le code est affiché sous le paragraphe e). Un *CircularPercentIndicator* requiert une propriété *radius* qui représente la taille du cercle, en pixels logiques. Vous pourriez certainement choisir une taille arbitraire, telle que 200, mais une meilleure approche pourrait être de choisir une taille relative qui dépend de l'espace disponible sur l'écran. Dans ce cas, vous utilisez un *LayoutBuilder*. Un *LayoutBuilder* fournit les contraintes du widget parent, de sorte que vous puissiez découvrir l'espace dont vous disposez pour vos widgets. Dans le corps du *Scaffold*, au lieu de renvoyer une colonne, retournons un *LayoutBuilder* dans sa méthode *builder*. Vous trouverez la largeur disponible en appelant la propriété *maxWidth* de l'instance de *BoxConstraints* qui a été passée à la méthode et en la plaçant dans une constante *largeurDisponible* :

```
body: LayoutBuilder(  
  builder: (BuildContext context, BoxConstraints constraints) {  
    final double largeurDisponible = constraints.maxWidth;  
    return Column(  
      children: [
```

- e) À l'intérieur du widget *Column*, sous la première ligne contenant les boutons 'Travail', 'Mini pause' et 'Maxi pause', ajoutons un *CircularPercentIndicator*. Le rayon du cercle correspond à la moitié de la largeur disponible et la largeur de la ligne à 10. Si vous aimez une bordure plus épaisse, vous pouvez également essayer une autre valeur, telle que 15 ou même 20.

```
Expanded(  
  child: CircularPercentIndicator(  
    radius: largeurDisponible / 2,  
    lineWidth: 10.0,  
    percent: 1,  
    center: Text(  
      "30:00",  
      style: Theme.of(context).textTheme.headline4,  
    ), // Text  
    progressColor: Color(0xff009688),  
  ), // CircularPercentIndicator
```

La mise en page de l'écran principal de notre application est maintenant prête. Maintenant, vous devez ajouter la logique métier pour que la minuterie compte réellement le temps.

### 3.3 Ajout de la logique métier

Vous allez découvrir dans ce chapitre une approche très utile : le flux qui est souvent lié à la programmation asynchrone vu dans le cours.

### 3.3.1 Utilisation d'un flux et de la programmation asynchrone dans Flutter

Pour le moment, vous avez vu deux types de widgets *Stateless* et *Stateful*. Dans le cas du *Stateful*, l'état est ce qui vous permet d'utiliser des données qui peuvent changer au cours de la durée de vie du widget. Et, bien que cela fonctionne parfaitement dans plusieurs cas, il existe d'autres moyens de modifier les données dans votre application, et l'un d'entre eux utilise les flux 'Streams'.

Les flux fournissent une séquence asynchrone de données. Le concept clé ici est que les flux sont asynchrones. C'est un concept très puissant en programmation. La programmation asynchrone permet à un morceau de code de s'exécuter séparément de la ligne principale d'exécution. Cela signifie que l'exécution de plusieurs tâches peut s'exécuter en même temps, au lieu de s'exécuter séquentiellement. Comme déjà dit, Dart est un langage de programmation monothreading et utilise des isolats pour traiter plusieurs tâches en même temps. Un isolat est un espace dans le thread de votre application, avec sa propre mémoire privée et sa propre ligne d'exécution.

### 3.3.2 Programmation de Stream

Pour mettre en œuvre un *Stream* dans votre application, vous devez exécuter les étapes suivantes :

- a) Créez une classe modèle pour le *CircularPercentIndicator* qui prend un texte et un pourcentage. Dans le dossier lib de notre application, ajoutez un fichier appelé *minuteur.dart*. Dans ce fichier, ajoutez une classe appelée *ModeleMinuteur*, avec deux champs et un constructeur qui les définit tous les deux :

```
class ModeleMinuteur {  
  String temps;  
  double pourcentage;  
  
  ModeleMinuteur(this.temps, this.pourcentage);  
}
```

- b) Dans le même fichier, importer le package « dart:async » pour pouvoir travailler en programmation asynchrone.

```
import 'dart:async';
```

- c) Créez une nouvelle classe *Minuteur* avec les 4 paramètres ci-dessous :

```
class Minuteur {  
  double _rayon = 1;  
  bool _estActif = true;  
  Duration _temps;  
  Duration _tempsTotal;  
}
```

Dans le code précédent, les 4 champs correspondent :

- *\_rayon* : représente ce que vous utiliserez pour exprimer le pourcentage de temps terminé
  - *\_estActif* : indique si le compteur est actif ou non. Lorsque l'utilisateur appuie sur le bouton d'arrêt, il devient inactif.
  - *\_temps* : représente le temps restant,
  - *\_tempsTotal* : représente le temps initial (une courte pause, par exemple, 5 minutes).
- d) Avant de renvoyer l'heure qui sera affichée dans *CircularProgressIndicator*, vous devez effectuer une mise en forme. Dans la classe *Minuteur*, créez une fonction *retournerTemps* qui

reçoit une durée (*Duration*) et retourne une chaîne de caractère (*String*). La durée (*Duration*) est une classe Dart utilisée pour contenir un intervalle de temps. Le code doit transformer cette durée en une chaîne, avec deux chiffres pour les minutes et deux chiffres pour les secondes.

Les propriétés *inMinutes* retourne les minutes et *inSeconds* les secondes dans un objet *Duration*. Vous devez vous assurer que, si les minutes ou les secondes n'ont qu'un seul chiffre, vous ajoutez un "0" avant le nombre, puis concaténez les deux valeurs avec un signe ":". La fonction retourne la chaîne formatée du type « MM:SS ».

```
String retournerTemps(Duration t) {
  String minutes = (t.inMinutes < 10)
    ? '0' + t.inMinutes.toString()
    : t.inMinutes.toString();
  int numSeconds = t.inSeconds - (t.inMinutes * 60);
  String secondes = (numSeconds < 10)
    ? '0' + numSeconds.toString()
    : numSeconds.toString();
  String tempsFormate = minutes + ':' + secondes;
  return tempsFormate;
}
```

e) Ensuite, vous créez la méthode *stream()* en vous inspirant du code ci-dessous :

```
Stream<ModeleMinuteur> stream() async* {
  yield* Stream.periodic(Duration(seconds: 1), (int a) {
    String temps;
    if (this._estActif) {
      _temps = _temps - Duration(seconds: 1);
      _rayon = _temps.inSeconds / _tempsTotal.inSeconds;
      if (_temps.inSeconds <= 0) {
        _estActif = false;
      }
    }
    temps = retournerTemps(_temps);
    return ModeleMinuteur(temps, _rayon);
  }); // Stream.periodic
}
```

L'astérisque (\*) après *async* est utilisé pour indiquer qu'un *Stream* est renvoyé via son pointeur. La méthode *stream()* renvoie un *Stream*. Un *Stream* est générique, ce qui signifie que vous pouvez renvoyer un *Stream* de n'importe quel type. Dans ce cas, nous retournons un *Stream* de *ModeleMinuteur*. La méthode est asynchrone (*async\**).

Dans Flutter, vous utilisez *async* (sans le signe \*) pour *Futures* et *async\** (avec le signe \*) pour *Streams*.

Quelle est la différence entre *Stream* et *Future*? Un *Stream* retourne n'importe quel nombre d'événements, alors qu'un *Future* n'en retourne qu'un seul. Lorsque vous marquez une fonction comme *async\**, vous créez une méthode *build*. Vous utilisez l'instruction *yield\** pour fournir un résultat. C'est comme une instruction *return*, mais cela ne termine pas la fonction. Comme indiqué précédemment, vous utilisez le signe "\*" après *yield* car nous renvoyons un *Stream*; s'il s'agissait d'une valeur unique, vous utiliseriez simplement *yield*.

Le code précédent *yield\** peut être vu comme une méthode créant un *Stream* qui émet des événements aux intervalles spécifiés dans le premier paramètre.

Ainsi, cette fonction renvoie un *Stream* de *TimerModel*, décrémentant la durée toutes les secondes.



### 3.3.3 Affichage dans la page principale : *StreamBuilder*

À l'heure actuelle, notre page principale ne change jamais, nous devons donc montrer le compte à rebours à l'utilisateur, et également nous assurer que l'utilisateur peut démarrer et arrêter le minuteur quand il en a besoin.

- a) Vous devez commencer par créer la fonction qui comptera le temps de travail. Pour l'instant, le temps de travail sera de 30 minutes (vous rendrez cette valeur modifiable par la suite). Donc, tout d'abord, dans la classe *Minuteur* dans le fichier *minuteur.dart*, créez un champ appelé *tempsTravail* et initialisez-le à 30. Il s'agit du nombre de minutes par défaut pour le temps de travail.
- b) Ensuite, toujours dans la classe *Minuteur*, créez une méthode de type *void* qui définira la durée *\_temps* sur le nombre de minutes contenu dans la variable de *tempsTravail*, et de même pour le champ *\_tempsTotal*. Cette méthode s'appellera *void demarrerTravail()*.
- c) La méthode *demarrerTravail()* doit être appelée depuis l'écran principal lors de son chargement. Revenez donc au fichier *page\_accueil\_minuterie.dart* et importons le fichier *minuteur.dart*.
- d) Ensuite, en haut de la classe *PageAccueilMinuterie*, créez une variable *Minuteur* appelée *minuteur*.
- e) En haut de la méthode *build()*, appelez la méthode *demarrerTravail()*.
- f) Maintenant, vous pouvez accéder aux propriétés du minuteur – temps et rayon – et les afficher à l'écran dans *CircularPercentIndicator* en ajoutant le code suivant :

```
Expanded(  
  child: CircularPercentIndicator(  
    radius: largeurDisponible / 2,  
    lineWidth: 10.0,  
    percent: 1,  
    center: Text('30:00',  
      style: Theme.of(context).textTheme.headline4,  
    ),  
    progressColor: Color(0xff009688),  
  ),  
)
```

- g) Si vous essayez l'application maintenant, vous devriez voir le minuteur, mais le compte à rebours n'est pas actif. C'est parce qu'il vous manque encore une partie importante du *Stream*, qui est le *StreamBuilder*. C'est ce que vous devez utiliser lorsque vous souhaitez écouter des événements provenant de *Streams*. Un *StreamBuilder* reconstruit ses enfants à tout changement dans le *Stream*. Utilisons-le dans notre application, y compris le widget *Expanded* dans un *StreamBuilder*.
  - a. ***initialData*** : définit la valeur initiale que le constructeur affiche pendant qu'il attend les premières données provenant du flux.
  - b. ***stream*** : définit le flux lui-même.
  - c. ***build*** : définit les widgets enfants en prenant comme paramètres un contexte et un instantané de type *AsyncSnapshot*. À partir de là, l'enfant est reconstruit à chaque fois que des données proviennent du flux. *AsyncSnapshot* contient les données de l'interaction la plus récente avec *StreamBuilder* (ou *FutureBuilder*).

```
Expanded(  
  child: StreamBuilder(  
    initialData: ModeleMinuteur('00:00', 1),  
  ),  
)
```

```

        stream: minuteur.stream(),
        builder: (BuildContext context, AsyncSnapshot snapshot) {
          ModeleMinuteur minuteur = snapshot.data;
          return Container(
            child: CircularPercentIndicator(
              radius: largeurDisponible / 2,
              lineWidth: 10.0,
              percent: (minuteur.pourcentage == null)
                ? 1
                : minuteur.pourcentage,
              center: Text(
                (minuteur.temps == null) ? '00:00' : minuteur.temps,
                style: Theme.of(context).textTheme.headline4,
              ),
              progressColor: Color(0xff009688),
            ),
          );
        },
      ),
    ),
  ),
),

```

Dans le code précédent, notez que le *snapshot* contient une propriété *data* : c'est ce qui a été reçu du *yield\** dans la méthode *stream()* de la classe *Minuteur*, qui a renvoyé un objet de type *ModeleMinuteur*. Si vous essayez l'application maintenant, la minuterie devrait fonctionner correctement.

Cependant, pendant que la minuterie fonctionne, l'utilisateur ne peut pas interagir avec notre application pour le moment.

### 3.3.4 Activation des boutons

#### 3.3.4.1 Code des boutons du pied de page

En premier, vous allez faire fonctionner les boutons de démarrage et d'arrêt. Pour cela, revenez sur le fichier *minuteur.dart*, puis implémenter les étapes suivantes:

- Ajoutez une nouvelle méthode *void arreterMinuteur()* qui positionnera la propriété *\_estActif* à la valeur *false* pour arrêter le minuteur.
- Ensuite, écrivez une autre méthode appelée *relancerMinuteur* qui vérifiera si le temps restant est supérieur à 0 seconde et définira le booléen *\_estActif* sur *true*.
- Enfin, dans le fichier *page\_accueil\_minuterie.dart*, appelons ces deux nouvelles méthodes à partir des boutons « Arrêter » et « Relancer ».

```

Row(
  children: <Widget>[
    Padding(
      padding: EdgeInsets.all(REPLISSAGE_DEFAULT),
    ),
    Expanded(
      child: BoutonGenerique(
        couleur: Color(0xff212121),

```

```

        texte: 'Arrêter',
        taille: 20.0,
        action: () => minuteur.arreterMinuteur(),
      ),
    ),
    Padding(
      padding: EdgeInsets.all(REMPLISSAGE_DEFAULT),
    ),
    Expanded(
      child: BoutonGenerique(
        couleur: Color(0xff009688),
        texte: 'Relancer',
        taille: 20.0,
        action: () => minuteur.relancerMinuteur(),
      ),
    ),
    Padding(
      padding: EdgeInsets.all(REMPLISSAGE_DEFAULT),
    ),
  ],
),

```

Si vous essayez l'application maintenant, vous pourrez arrêter et démarrer le minuteur à volonté.

#### 3.3.4.2 Code des boutons de l'entête

Vous devez mettre à la disposition de l'utilisateur les boutons « Travail », « Mini Pause » et « Maxi Pause ». Temporairement, vous coderez en dur la durée des trois boutons, mais plus loin, vous donnerez à l'utilisateur le pouvoir de définir ces valeurs.

- Dans le fichier *minuteur.dart*, dans la classe *Minuteur*, déclarez deux autres variables pour les temps des pauses courtes et longues, respectivement *tempsPauseCourte* et *tempsPauseLongue* avec les valeurs initiales de 5 et 20.
- Ensuite, ajoutez une méthode pour démarrer une pause courte ou une pause longue. Cette méthode *demarrerPause* prendra un paramètre de type booléen qui précisera si la pause est courte ou pas.

```

int tempsTravail = 30;
int tempsPauseCourte = 5;
int tempsPauseLongue = 20;

void arreterMinuteur() {
  this._estActif = false;
}

void relancerMinuteur() {
  if (_temps.inSeconds > 0) {
    this._estActif = true;
  }
}

```

```

void demarrerPause(bool estCourte) {
  _rayon = 1;
  _temps = Duration(
    minutes: (estCourte) ? tempsPauseCourte : tempsPauseLongue,
    seconds: 0,
  );
  _tempsTotal = _temps;
}

void demarrerTravail() async {
  await lireParametres();
  _rayon = 1;
  _temps = Duration(minutes: this.tempsTravail, seconds: 0);
  _tempsTotal = _temps;
}

```

- d) Enfin, dans le fichier *page\_accueil\_minuterie.dart*, appelons ces deux nouvelles méthodes à partir des boutons « Travail », « Mini Pause » et « Maxi Pause ».

```

Row(
  children: <Widget>[
    Padding(
      padding: EdgeInsets.all(REPLISSAGE_DEFAULT),
    ),
    Expanded(
      child: BoutonGenerique(
        couleur: Color(0xff009688),
        texte: 'Travail',
        taille: 20.0,
        action: () => minuteur.demarrerTravail(),
      ),
    ),
    Padding(
      padding: EdgeInsets.all(REPLISSAGE_DEFAULT),
    ),
    Expanded(
      child: BoutonGenerique(
        couleur: Color(0xff607D8B),
        texte: 'Mini pause',
        taille: 20.0,
        action: () => minuteur.demarrerPause(true),
      ),
    ),
    Padding(
      padding: EdgeInsets.all(REPLISSAGE_DEFAULT),
    ),
  ],
)

```

```

Expanded(
  child: BoutonGenerique(
    couleur: Color(0xff455A64),
    texte: 'Maxi pause',
    taille: 20.0,
    action: () => minuteur.demarrerPause(false),
  ),
),
Padding(
  padding: EdgeInsets.all(REPLISSAGE_DEFAULT),
),
],
),

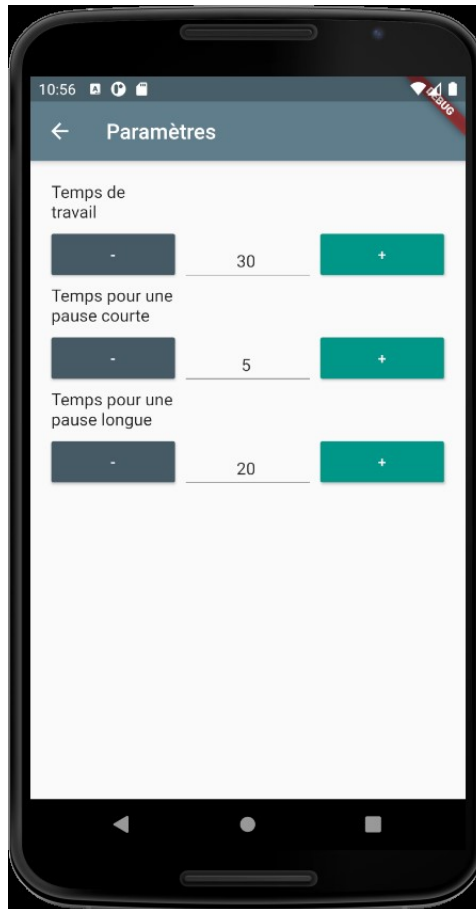
```

Notez que le paramètre *onPressed* prend une fonction comme valeur. En effet, dans Dart et Flutter, vous pouvez passer une fonction en tant que paramètre, dans un constructeur ou dans toute autre méthode.

Si vous essayez l'application maintenant, vous remarquerez que toutes les fonctions principales fonctionnent correctement !

### 3.4 Challenge

Tout le monde ne travaille pas par cycle de 30 minutes, et ne prend pas des pauses de 5 ou 20 minutes. Pour corriger cela, je vous propose de créer une page « paramètres » permettant de changer ces valeurs et de les stocker dans la mémoire de l'appareil au travers de *SharedPreferences*.



#### 3.4.1 Créer la base d'une page de paramètres

- Ajoutez un nouveau fichier « `page_parametres.dart` » dans le dossier `lib` de notre application.
- Vous allez créer une nouvelle page sur la base d'un widget *Stateless* qui, dans la méthode `build()`, renverra un *Scaffold*, avec un *AppBar* dont le titre sera « Paramètres » et un conteneur *Container* vide.
- Pour le moment, il n'y a aucun moyen d'accéder à cette page « Paramètres », vous devez donc ajouter une fonction pour l'ouvrir à partir de l'écran principal. Dans Flutter, les écrans ou les pages sont appelés itinéraires. Pour ce faire, revenez au fichier `page_accueil_minuterie.dart` et, dans la méthode `build()` de la classe `PageAccueilMinuterie()`, ajoutez le code suivant:

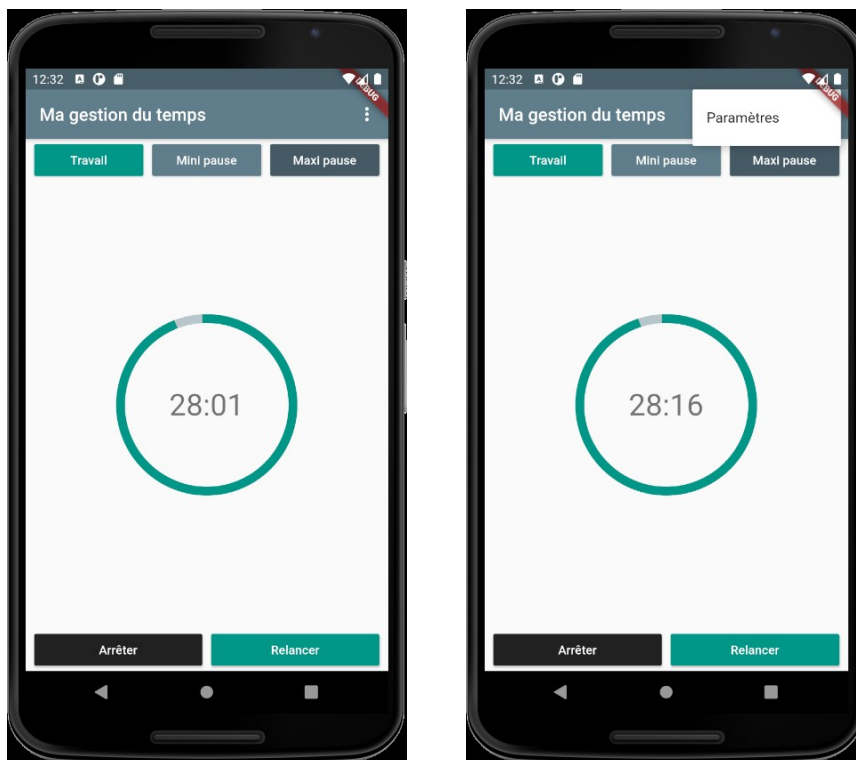
```
final List<PopupMenuItem<String>> elementsMenu =  
    List<PopupMenuItem<String>>();  
  
elementsMenu.add(PopupMenuItem(  
    value: 'Paramètres',  
    child: Text('Paramètres'),  
));
```

Dans Flutter, un *PopupMenuButton* affiche un menu lorsqu'il est enfoncé. Dans sa propriété *itemBuilder*, il peut afficher une liste de *PopupMenuItems*. C'est pourquoi, dans cette partie du code, nous avons créé une liste de *PopupMenuItems*, même si le menu n'en a qu'un à ce moment.

- d) Afin de faire apparaître le *PopupMenuButton* à l'écran, ajoutez-le à *AppBar* dans *Scaffold* :

```
appBar: AppBar(  
  title: Text('Ma gestion du temps'),  
  actions: <Widget>[  
    PopupMenuButton(  
      itemBuilder: (BuildContext context) {  
        return elementsMenu.toList();  
      },  
    ),  
  ],  
)
```



Dans la capture d'écran suivante, vous pouvez voir un *PopupMenuButton* avant et après avoir cliqué dessus :



- e) Ensuite, créons une méthode *allerParametres* qui navigue réellement vers la route de la page « Paramètres ». La navigation dans Flutter est basée sur une pile. Une pile contient les écrans créés par une application depuis le début de son exécution. Chaque fois que vous avez besoin de changer d'écran dans une application Flutter, vous pouvez utiliser l'objet *Navigator*.

### 3.4.2 Créer la navigation entre ces deux pages

*Navigator* a plusieurs méthodes qui interagissent avec la pile, mais nous n'avons à nous soucier que de deux pour l'instant : la méthode *push()* et la méthode *pop()*.

-  La méthode *push()* place une nouvelle page en haut de la pile.
-  La méthode *pop()* supprime la page en haut de la pile afin que l'écran précédent de votre pile redevienne visible.

Lorsque vous utilisez la méthode *push()*, vous devez spécifier une route, qui est l'écran que vous souhaitez charger. Pour cela, vous utilisez la classe *MaterialPageRoute*, dans laquelle vous spécifiez le nom de la page que vous souhaitez utiliser. *Push()* et *pop()* nécessitent le contexte actuel.

- a) Revenez à votre code, écrivez la méthode *allerParametres()*, comme indiqué dans l'extrait de code suivant:

```
void allerParametres(BuildContext context) {  
  Navigator.push(  
    context,  
    MaterialPageRoute(  
      builder: (context) => PageParametres(),  
    ),  
  );  
}
```

N'oubliez pas d'importer le fichier *page\_parametres.dart* pour importer pour importer la classe *PageParametres*.

- b) Ensuite, dans *AppBar* du *Scaffold*, ajoutez les actions qui spécifieront un *PopupMenuButton* avec l'élément *itemBuilder* contenant votre *elementsMenu*, comme suit:

```
appBar: AppBar(  
  title: Text('Ma gestion du temps'),  
  actions: <Widget>[  
    PopupMenuButton(  
      itemBuilder: (BuildContext context) {  
        return elementsMenu.toList();  
      },  
      onSelect: (s) {  
        if (s == 'Paramètres') {  
          allerParametres(context);  
        }  
      },  
    ),  
  ],  
)
```

Maintenant, si vous essayez l'application, vous pouvez en fait naviguer d'un écran à l'autre !



### 3.4.3 Création de la disposition de la page « Paramètres »

Les paramètres de cette application devront conserver leurs états, vous allez donc créer un widget *Stateful*. Si vous utilisez l'un des éditeurs pris en charge (VS Code, IntelliJ IDEA ou Android Studio), vous pouvez simplement taper le raccourci *stful*. Cela créera le code standard pour un nouveau widget *Stateful*.

- a) Dans le fichier *page\_parametres.dart*, à la fin du fichier, saisissez *stful* et saisissez *Parametres* comme nom du widget.

Vous pourriez utiliser une combinaison de widgets *Row* et *Column* pour construire cet écran, mais vous utiliserez un nouveau widget le *GridView*.

#### 3.4.3.1 Utilisation du constructeur *GridView.Count()*

Un *GridView* est un tableau 2D permettant de dérouler des widgets, et vous pouvez l'utiliser pour afficher certaines données à vos utilisateurs sous forme de tableau. Les cas d'utilisation possibles du *GridView* incluent une galerie d'images, une table de chansons, une liste de films et bien d'autres.

Le *GridView* est scrollable et a deux dimensions : en d'autres termes, c'est un tableau déroulant. Il peut défiler horizontalement ou verticalement. Il existe plusieurs constructeurs pour *GridView* qui couvrent plusieurs cas différents d'utilisation, mais pour cette application, vous utiliserez le constructeur *GridView.Count()*.

Ce constructeur est utilisable lorsque vous connaissez le nombre d'éléments que la grille affichera à l'écran.

```
@override
Widget build(BuildContext context) {
  return Container(
    child: GridView.count(
      scrollDirection: Axis.vertical,
      crossAxisCount: 3,
      childAspectRatio: 3,
      crossAxisSpacing: 10,
      mainAxisSpacing: 10,
      children: <Widget>[],
      padding: const EdgeInsets.all(20.0),
    );
}
```

La première propriété que vous définissez est la direction du défilement, qui est *Axis.Vertical*. Cela signifie que, si le contenu de *GridView* est plus grand que l'espace disponible, le contenu défilera verticalement.

Ensuite, vous définissez la propriété *crossAxisCount*: lorsque l'utilisateur fera défiler verticalement, c'est le nombre d'éléments qui apparaîtront sur chaque ligne.

La propriété *childAspectRatio* détermine la taille des enfants dans le *GridView*. La valeur représente le rapport *itemWidth/itemHeight*. Dans ce cas, en mettant 3, nous disons que la largeur doit être trois fois la hauteur. Comme il n'y a pas d'espace entre les enfants d'un *GridView* par défaut, vous pouvez ajouter un espacement pour l'axe principal, en utilisant le paramètre *mainAxisSpacing*, et en lui donnant une valeur de 10. Vous pouvez également faire de même pour l'axe transversal, toujours

avec une valeur de 10. Et pour compléter cet exemple, vous rajoutez ajouté un bourrage en prenant un *EdgeInsets.all* de 20.

### 3.4.3.2 Ajout des boutons dans le GridView

Comme vous l'avez fait pour le *BoutonGenerique*, afin d'éviter une duplication de code inutile, vous pouvez créer un bouton que vous réutiliseriez plusieurs fois dans l'écran Paramètres. Ce bouton a des propriétés qui sont différentes de *BoutonGenerique*. De ce fait, vous allez créer un nouveau widget en suivant les étapes suivantes :

- a) Dans le fichier *widgets.dart*, créez un nouveau widget *Stateless* appelé *BoutonParametre*.
- b) Définissez les propriétés suivantes dans cette classe :
  - a. **couleur** : couleur du bouton de type *Color*.
  - b. **texte** : texte sur le bouton de type *String*.
  - c. **valeur** : valeur de l'incrément de type *int*.
  - d. **parametre** : nom du paramètre à modifier de type *String*.
  - e. **action** : fonction à exécuter en cas d'appui sur le bouton de type *CallbackSetting*.
- c) Ce widget retourne un widget de type *MaterialButton* qui utilisera les propriétés qui ont été définies dans le constructeur. La méthode *onPressed* utilisera la propriété *action* avec deux paramètres comme indiqué ci-dessous :

```
class BoutonParametre extends StatelessWidget {
  final Color couleur;
  final String texte;
  final int valeur;
  final String parametre;
  final CallbackSetting action;

  BoutonParametre(
    this.couleur, this.texte, this.valeur, this.parametre, this.action);
  @override
  Widget build(BuildContext context) {
    return MaterialButton(
      child: Text(
        this.texte,
        style: TextStyle(color: Colors.white),
      ),
      onPressed: () => this.action(this.parametre, this.valeur),
      color: this.couleur,
    );
  }
}
```

- d) Revenez dans le fichier *page\_parametres.dart*, afin que vous intégriez le nouveau *BoutonParametre* dans le *GridView*. En haut de la méthode *build()* de la classe *\_ParametreState*, créez un *styleTexte* qui permettra de définir la taille de la police, comme suit:

```
TextStyle styleTexte = TextStyle(fontSize: 16);
```

- e) Ensuite, dans le paramètre *children* du constructeur *GridView.count()*, insérez tous les widgets que vous devez placer à l'écran, comme suit:

```
children: <Widget>[
  Text(
    'Temps de travail',
    style: styleTexte,
  ),
  Text(''),
  Text(''),
  BoutonParametre(
    Color(0xff455A64), '-', -1, TEMPS_TRAVAIL, majParametres),
  TextField(
    controller: txtTempsTravail,
    style: styleTexte,
    textAlign: TextAlign.center,
    keyboardType: TextInputType.number,
  ),
  BoutonParametre(
    Color(0xff009688), '+', 1, TEMPS_TRAVAIL, majParametres),
  Text(
    'Temps pour une pause courte',
    style: styleTexte,
  ),
  Text(''),
  Text(''),
  BoutonParametre(
    Color(0xff455A64), '-', -1, PAUSE_COURTE, majParametres),
  TextField(
    controller: txtTempsPauseCourte,
    style: styleTexte,
    textAlign: TextAlign.center,
    keyboardType: TextInputType.number,
  ),
  BoutonParametre(
    Color(0xff009688), '+', 1, PAUSE_COURTE, majParametres),
  Text(
    'Temps pour une pause longue',
    style: styleTexte,
  ),
  Text(''),
  Text(''),
  BoutonParametre(
    Color(0xff455A64), '-', -1, PAUSE_LONGUE, majParametres),
  TextField(
    controller: txtTempsPauseLongue,
    style: styleTexte,
    textAlign: TextAlign.center,
    keyboardType: TextInputType.number,
```

```

    ),
    BoutonParametre(
      Color(0xff009688), '+', 1, PAUSE_LONGUE, majParametres),
  ],

```

Lorsque vous créez un *GridView*, chaque cellule a la même taille. Comme vous avez défini la propriété *crossAxisCount* sur 3, pour chaque ligne de la grille, il y a trois éléments par ligne.

Dans la première ligne, vous placez simplement trois textes, un contenant "Temps de travail" et deux vides. Les deux textes vides sont juste pour s'assurer que le widget suivant se retrouvera dans la deuxième ligne.

Dans la deuxième ligne, vous avez deux boutons et un *TextField* qui permettra de saisir directement et d'afficher la valeur du paramètre.

- f) Vous n'avez plus qu'à définir la méthode *majParametres* comme suit :

```

void majParametres(String key, int value) {
  switch (key) {
    case TEMPS_TRAVAIL:
      break;
    case PAUSE_COURTE:
      break;
    case PAUSE_LONGUE:
      break;
  }
}

```

- g) Ce modèle est répété pour les lignes suivantes permettant ainsi de lire et écrire les trois paramètres : « temps de travail », « temps pour une courte pause » et « temps pour une longue pause », comme illustré ci-contre.

Si vous essayez l'application maintenant, l'écran « Paramètres » devrait ressembler à la capture d'écran ci-contre.

Maintenant que vous avez la disposition du deuxième écran, vous devez ajouter la logique, car vous voulez lire et écrire les paramètres de votre application.



### 3.4.4 Utilisation de *SharedPreferences* pour lire et écrire des données d'application

Il existe plusieurs façons d'enregistrer des données sur un appareil mobile. Vous pouvez conserver les données dans un fichier, ou vous pouvez utiliser une base de données locale, telle que SQLite, ou vous pouvez utiliser *SharedPreferences* (sur Android) ou *NSUserDefaults* (sur iOS).

*SharedPreferences* ne doit pas être utilisé pour les données critiques car les données qui y sont stockées ne sont pas chiffrées et les écritures ne sont pas toujours garanties. Lorsque vous utilisez Flutter, vous pouvez tirer parti de la bibliothèque *SharedPreferences*. Elle encapsule à la fois *NSUserDefaults* et *SharedPreferences* afin que vous puissiez stocker des données simples de manière transparente à la fois sur iOS et Android sans traiter les spécificités des deux systèmes d'exploitation.

Les données sont toujours conservées sur le disque de manière asynchrone lorsque vous utilisez *SharedPreferences*. *SharedPreferences* est un moyen simple de conserver les données clé-valeur sur le disque. Vous ne pouvez stocker que des données primitives, c'est-à-dire des objets variable de base (*int*, *double*, *bool*, *String*) et un objet variable composé (*stringList*). Les données *SharedPreferences* sont enregistrées dans l'application, donc, lorsque l'utilisateur désinstalle votre application, les données seront également supprimées.

L'objet *SharedPreferences* n'est pas conçu pour stocker beaucoup de données, mais, pour votre application, cet outil est parfait.

- a) Pour utiliser *SharedPreferences*, vous devez inclure *SharedPreferences* dans votre projet. Recherchez sur internet les informations sur ce package puis modifiez en conséquence le fichier *pubspec.yaml*.
- b) Ensuite, pour utiliser le package, vous devez faire *import* dans le fichier *page\_parametres.dart*.

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

Avant d'entrer dans les détails de l'utilisation de *SharedPreferences*, vous avez besoin d'un moyen de lire les données à partir des *TextFields* lorsque l'utilisateur change de valeur, et d'écrire dans le *TextField* lorsque vous chargez l'écran à partir des valeurs stockées dans *SharedPreferences*. Lors de l'utilisation de *TextFields*, un moyen efficace de lire et d'écrire des données consiste à utiliser un *TextEditingController*.

- c) Ajoutez le code suivant en haut de la classe *\_ParametresState* :

```
TextEditingController txtTempsTravail;  
TextEditingController txtTempsPauseCourte;  
TextEditingController txtTempsPauseLongue;
```

- d) Ensuite, remplacez la méthode *initState()* pour définir les nouveaux *TextEditingController* :

```
@override  
void initState() {  
  txtTempsTravail = TextEditingController();  
  txtTempsPauseCourte = TextEditingController();  
  txtTempsPauseLongue = TextEditingController();  
  super.initState();  
}
```

Ici, vous créez les objets qui vous permettront de lire et d'écrire dans les widgets *TextField*.

- e) Ensuite, ajoutez le *TextEditingController* aux *TextFields* pertinents, et vous le ferez en utilisant la propriété *controller* dans chacun des trois *TextFields* que vous avez créés auparavant dans la méthode *build()*.
- f) En haut de la classe *\_ParametresState*, créez les constantes et les variables que nous utiliserons pour interagir avec *SharedPreferences*, comme suit :

```
static const String TEMPS_TRAVAIL = 'Temps de travail';  
static const String PAUSE_COURTE = 'Pause courte';  
static const String PAUSE_LONGUE = 'Pause longue';  
int tempsTravail;  
int tempsPauseCourte;  
int tempsPauseLongue;
```

- g) Créez également une variable pour les *SharedPreferences*, toujours au début de la classe *\_ParametresState*, comme ceci :

```
SharedPreferences preferences;
```

- h) Ensuite, vous devez créer deux méthodes : la première lira à partir de *SharedPreferences* et la seconde écrira tout changement effectué par l'utilisateur.
  - a. Commencez par lire les paramètres. Après la méthode *build()* de la classe *\_ParametresState*, ajoutons une méthode appelée *lireParametres()*. Cette méthode sera asynchrone (*async*) et pourra renvoyer un objet *Future*. Pour suspendre l'exécution jusqu'à la fin d'un *Future*, vous utiliserez *await* dans une fonction *async*. *SharedPreferences.getInstance()* est asynchrone, vous devez donc utiliser l'instruction *await* pour nous assurer que *preferences* est instancié avant l'exécution des prochaines lignes de code.

```
lireParametres() async {  
  preferences = await SharedPreferences.getInstance();  
  int tempsTravail = preferences.getInt(TEMPS_TRAVAIL);  
  if (tempsTravail == null) {  
    await preferences.setInt(TEMPS_TRAVAIL, int.parse('30'));  
  }  
}
```

Lorsque vous appelez *preferences.getInt(KEY)*, vous appelez une méthode qui renvoie un entier à partir de *SharedPreferences* - en particulier, l'entier qui, en tant que clé, a la valeur que nous passons en argument. Donc, si nous avons une clé appelée "Temps de travail" et une valeur de 25, cette fonction retournera 25. S'il n'y a pas de valeur à la clé que vous avez passée, cette fonction retournera *null*.

Dans votre exemple, si le retour est null, vous initialisez la valeur à '30'.

Vous répétez cela pour toutes les valeurs que vous voulez stocker pour les paramètres.

- b. Ensuite, vous mettez à jour l'état de la classe en modifiant la propriété *text* des *textControllers*. Pour cela vous utilisez la méthode *setState()*. Par exemple pour le premier paramètre :

```
setState(() {  
  txtTempsTravail.text = tempsTravail.toString();  
});
```

```
}
```

En bref, cette fonction lit les valeurs des paramètres à partir de *SharedPreferences*, puis elle écrit les valeurs dans *textFields*.

- c. Maintenant, modifiez la méthode *majParametres()*. Cette méthode prend deux paramètres : une clé et une valeur.

Vous voulez que l'utilisateur mette à jour la valeur en cliquant sur les boutons + et - à l'écran, de sorte que la valeur sera 1 pour le bouton + ou -1 pour le bouton -. La clé sera l'une des constantes que nous avons déclarées en haut de la classe. Cette méthode lit la valeur de la clé qui a été transmise et ajoute la valeur (+1 ou -1).

Par exemple, le code suivant lit le temps de travail enregistré et y ajoute de la valeur :

```
int tempsTravail = preferences.getInt(TEMPS_TRAVAIL);
tempsTravail += value;
```

Ensuite, vous vérifiez si *tempsTravail* est dans la plage acceptée (entre 1 et 180 minutes). Si oui, alors vous sauvegardez la nouvelle valeur dans *SharedPreferences* puis vous mettez l'affichage du *textField* à jour à l'aide de *setState()*.

```
setState(() {
  txtTempsTravail.text = tempsTravail.toString();
});
```

Vous répétez ensuite les mêmes étapes pour les deux autres paramètres : *tempsPauseCourte* et *tempsPauseLongue*. Maintenant, la question est : quand appelez-vous ces deux méthodes ? Lorsque l'écran est affiché, vous devez lire les valeurs immédiatement, car vous voulez les afficher dans les *TextFields*.

- a) Pour cela dans la méthode *initState()*, avant d'appeler *super.initState()*, ajoutez un appel à la méthode *lireParametres()* comme suit:

```
@override
void initState() {
  txtTempsTravail = TextEditingController();
  txtTempsPauseCourte = TextEditingController();
  txtTempsPauseLongue = TextEditingController();
  lireParametres();
  super.initState();
}
```

Si tout fonctionne comme il se doit, les *TextFields* contiennent maintenant les valeurs (nulle la première fois que vous essayez l'application).

- b) Vous voulez mettre à jour les paramètres à chaque fois que l'utilisateur change la valeur en appuyant sur l'un des boutons + ou -. Cela devrait changer les valeurs dans le *TextField* approprié et également mettre à jour le paramètre dans *SharedPreferences*. Lorsque nous appuyons sur l'un des boutons, une méthode doit alors être appelée, mettant à jour le *TextField* et le paramètre approprié. Cela est réalisé en partie par l'appel de *BoutonGenerique* dans le fichier *widgets.dart*.

La dernière étape pour terminer votre application consiste également à lire le paramètre à partir du fichier *minuteur.dart*. Pour cela, vous devez réaliser les opérations suivantes :

- a) En haut du fichier *minuteur.dart*, importez le package *shared\_preferences*, comme précédemment.
- b) Créez une méthode *lireParametres()* à la fin de la classe *Minuteur* qui récupère les paramètres enregistrés dans l'instance *SharedPreferences* ou définit les valeurs par défaut. Vous pouvez vous inspirer de la méthode *lireParametres* écrite pour la page paramètres.
- c) Ajoutez un appel à la méthode *lireParametres()* en haut de la méthode *demarrerTravail()* en transformant cette méthode en méthode asynchrone. Vous pouvez vous inspirer du code ci-dessous

```
void demarrerTravail() async {  
  await lireParametres();  
  _rayon = 1;  
  _temps = Duration(minutes: this.tempsTravail, seconds: 0);  
  _tempsTotal = _temps;  
}
```

- d) Faites de même, pour la méthode *demarrerPause()*.

Si vous essayez l'application maintenant, vous verrez qu'elle fonctionne enfin ! Bravo, vous avez terminé une application Flutter plutôt riche ! Vous pouvez télécharger le code solution sur le site MLS.