

# Chapitre 4 : Widgets de layout

## ▼ Introduction au système de layout

### Avec flutter tout est un Widget

Pour créer des applications `flutter`, nous utilisons des `Widgets` y compris pour les `layouts`, c'est-à-dire la disposition et le style des éléments.

En `Web`, nous utiliserions le `HTML` et le `CSS` pour gérer l'affichage des éléments (position, styles etc).

Avec `Flutter`, les éléments de `Layout` comme par exemple les **rangées**, **colonnes** et les **grilles**, sont des `widgets` écrits en `Dart`.

Il n'existe donc pas d'autre langage de `layout` à maîtriser : il suffit de **connaître les bases de Dart** que nous avons vues pour pouvoir gérer également l'affichage.

### Les widgets de layout

Nous verrons au fur et à mesure des chapitres les `widgets de layout`.

Il existe en effet plus de **30** `widgets` de layout et tous les voir d'un coup serait totalement contreproductif.

Nous allons voir les plus importants dans ce chapitre, à savoir les `widgets container`, `padding`, `center`, `column`, `row`, `stack` et `expanded`.

### Paramètre nommé commun

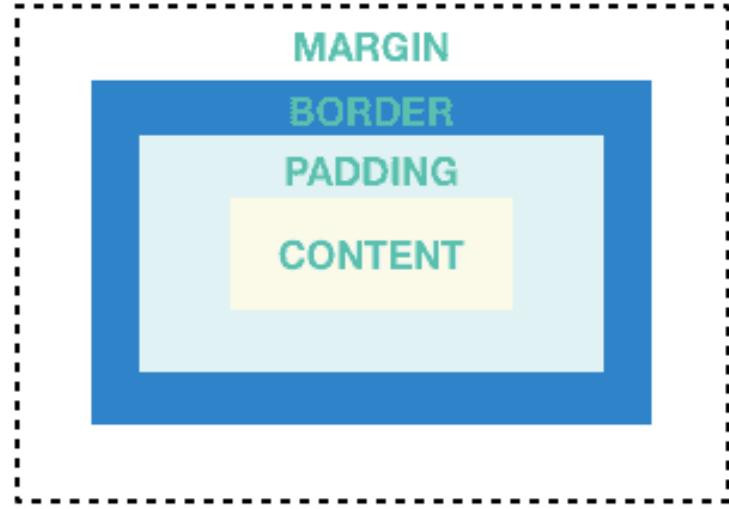
Tous les `widgets de layout` prennent en propriété nommé soit un `child` soit des `children` suivant qu'ils s'appliquent à un ou plusieurs autres `widgets`.

## ▼ Le widget container

Le `widget container` permet de gérer le `widget enfant` qu'il contient.

Il permet d'ajouter du `padding`, de la `margin` et des `borders` à son `widget enfant` (`child`).

Le fonctionnement est le suivant (avec `border` comme `decoration`, le `content` comme `child`):



Il permet également de changer le `background` pour le colorer ou y placer une image.

Bien qu'un `widget container` ne contienne qu'un seul `widget` enfant, ce dernier peut en contenir plusieurs.

## Un premier container

Nous allons prendre un premier exemple minimaliste :

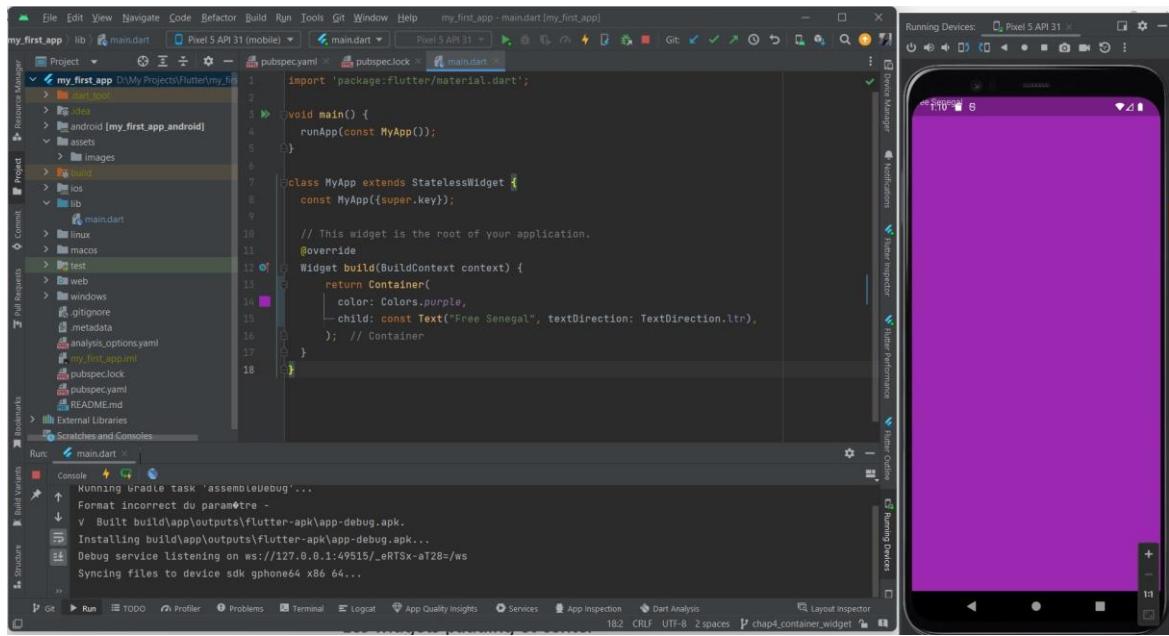
```
import 'package:flutter/material.dart';

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

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

  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return Container(
      color: Colors.purple,
      child: const Text("Free Senegal", textDirection: TextDirection.ltr),
    );
}
}
```

Voici le résultat :



**Problème :** Le texte s'affiche tout en haut à gauche et est illisible ! En outre, le container a pris toute la place disponible.

## Le constructeur de Container

Le constructeur est composé des propriétés suivantes:

```

Container({
  Key key,
  this.alignment,
  this.padding,
  Color color,
  Decoration decoration,
  this.foregroundDecoration,
  double width,
  double height,
  BoxConstraints constraints,
  this.margin,
  this.transform,
  this.child,
})

```

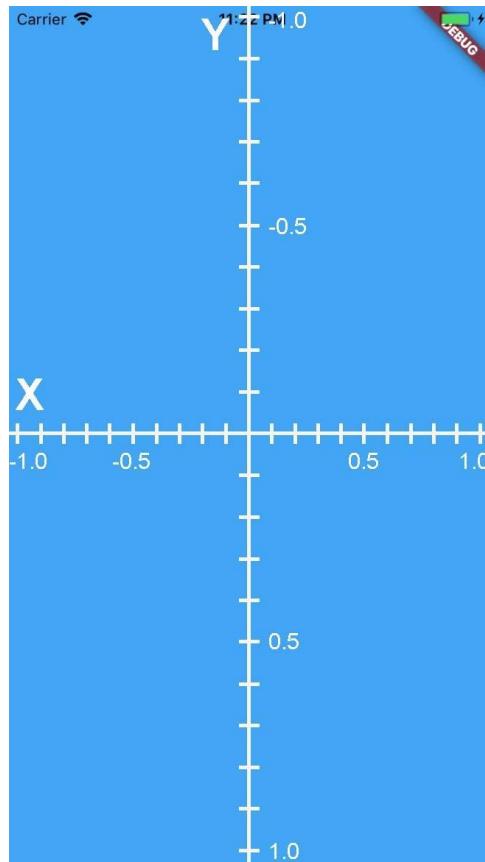
### Le paramètre nommé alignment

Nous allons commencer par régler notre problème en utilisant la propriété `alignment`

Cette propriété permet de gérer l'alignement de l'enfant dans le `container` qui est un rectangle.

Elle prend `deux doubles` en argument représentant la **distance** par rapport au **centre du rectangle ( 0,0 )**. Les valeurs au dessus de `1.0` sont en dehors du `rectangle`.

Pour être dans le `rectangle` les valeurs doivent varier entre `-1` et `1`:



Le **premier paramètre** est l'`axe horizontal`, et le **second paramètre** est l'`axe vertical`.

Il existe également des `enums` pour les alignements principaux (`bottomCenter`, `bottomLeft`, `bottomRight`, `center`, `centerLeft`, `centerRight`, `topCenter`, `topLeft` et `topRight`).

Ainsi,

- Le **centre du rectangle** est `Alignment(0.0, 0.0)` OU `Alignment.center` .
- Le **coin inférieur droit du rectangle** est `Alignment(1.0, 1.0)` OU `Alignment.bottomRight` .
- Le **coin supérieur gauche du rectangle** est `Alignment(-1.0, -1.0)` OU `Alignment.topLeft` . Le **coin inférieur gauche du rectangle** est `Alignment(-1.0, 1.0)` OU `Alignment.bottomLeft` .

**A Retenir** : L'alignement d'un élément se définit sur une grille allant de  $-1$  à  $1$  sur l'abscisse et sur l'ordonnée

Modifions notre exemple pour placer notre texte au centre du rectangle:

```

import 'package:flutter/material.dart';

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

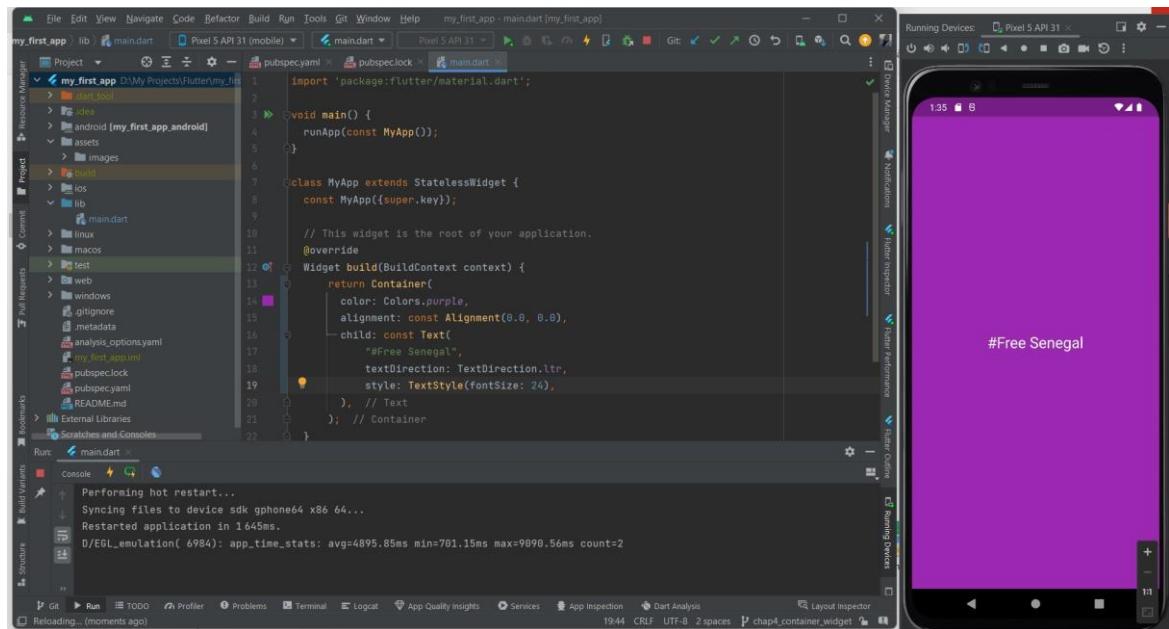
class MyApp extends StatelessWidget {
  const MyApp({super.key});

  // This widget is the root of your application.

  @override
  Widget build(BuildContext context) {
    return Container(
      color: Colors.purple,
      alignment: const Alignment(0.0, 0.0),
      child: const Text(
        "#Free Senegal",
        textDirection: TextDirection.ltr,
        style: TextStyle(fontSize: 24),
      ),
    );
  }
}

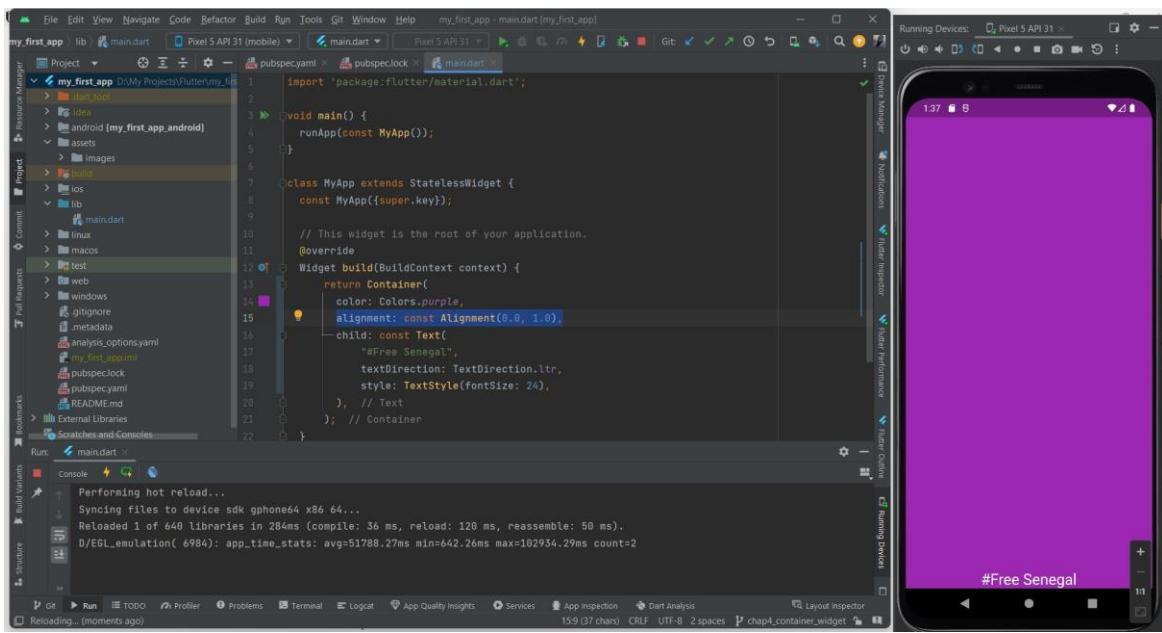
```

Notre problème est maintenant résolu :



`Alignment(0.0, 1.0)` permet donc de centrer horizontalement et de mettre l'enfant au niveau du bord inférieur du rectangle.

Voici le résultat :



## Le paramètre nommé margin

Ce paramètre permet de mettre de la `margin` autour du `Container`, c'est-à-dire de mettre de l'espace vide. `margin` prend la classe `EdgeInsets` en argument qui permet de définir des décalages dans toutes les directions.

**NB:** `EdgeInsets` s'utilise également pour le padding comme nous le verrons.

- Le constructeur nommé `EdgeInsets.all()`

Pour appliquer un décalage sur tous les côtés, il faut utiliser `EdgeInsets.all()` et lui passer en argument un `double` qui est le **nombre de pixels du décalage**

En reprenant notre exemple :

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

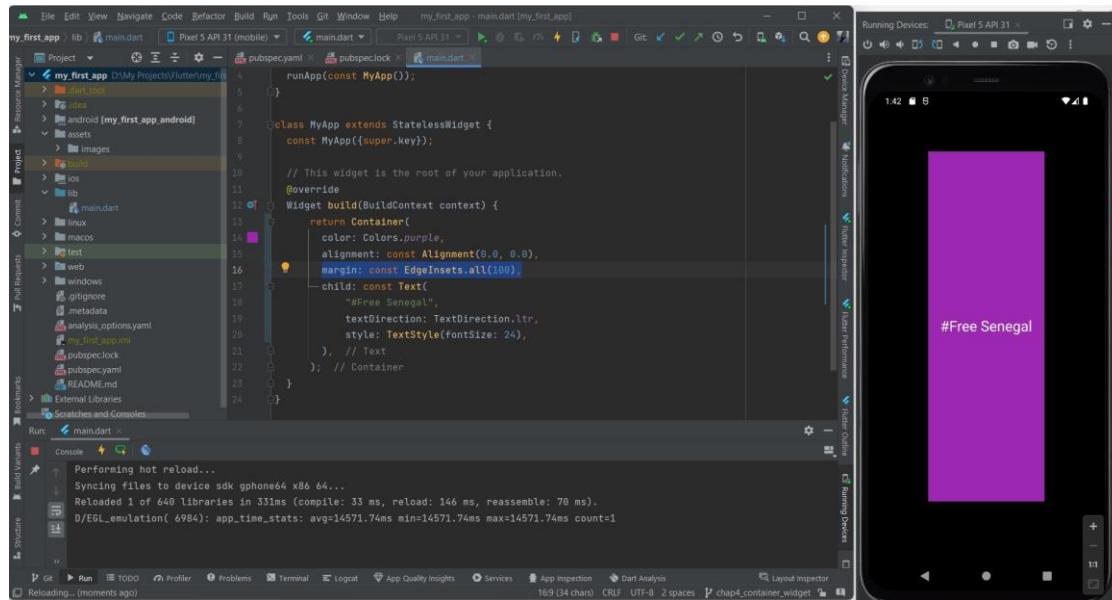
void main() {
    runApp(const MyApp());
}

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

    // This widget is the root of your application.
    @override
    Widget build(BuildContext context) {
        return Container(
            color: Colors.purple,
            alignment: const Alignment(0.0, 0.0),
            margin: const EdgeInsets.all(100),
            child: const Text(
                "#Free Senegal",
                textDirection: TextDirection.ltr,
                style: TextStyle(fontSize: 24),
            ),
        );
}
```

```
}
```

Ce qui donne une `margin` dans toutes les directions autour du `Container` de 100 pixels :

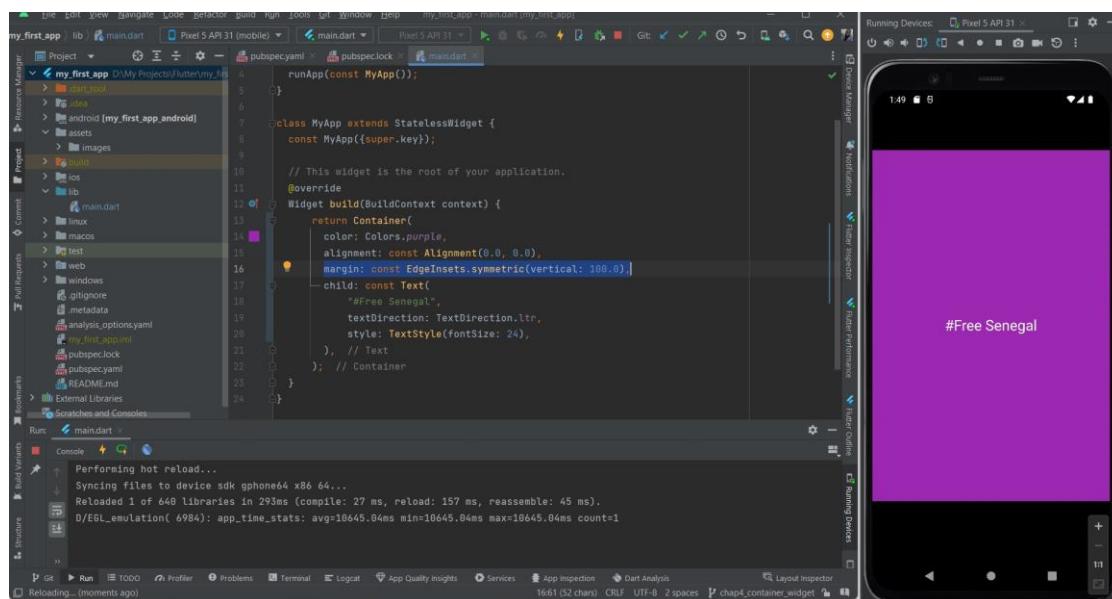


- Le constructeur nommé `EdgeInsets.symmetric()`

Ce constructeur permet de définir le décalage à appliquer **verticalement** (en haut et en bas) et / ou **horizontalement** (à gauche et à droite).

```
margin: const EdgeInsets.symmetric(horizontal: 0.0, vertical: 0.0),
```

Par exemple pour définir que de la margin verticale :

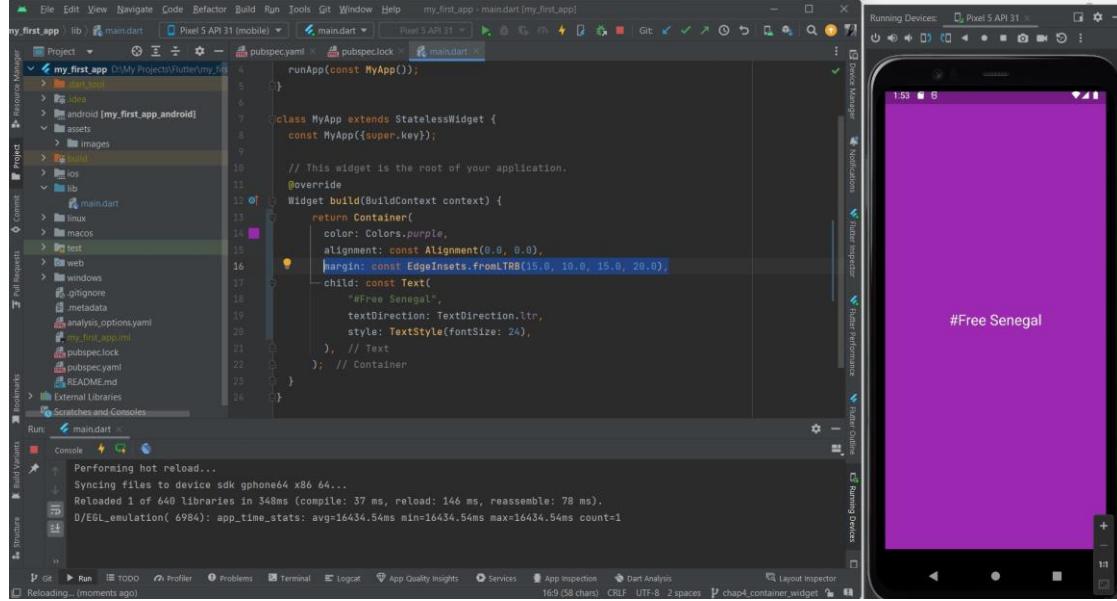


- Le constructeur nommé `EdgeInsets.fromLTRB()`

Ce constructeur permet de définir les décalages dans l'ordre `left`, `top`, `right` et `bottom`.

Par exemple :

```
margin: const EdgeInsets.fromLTRB(15.0, 10.0, 15.0, 20.0),
```

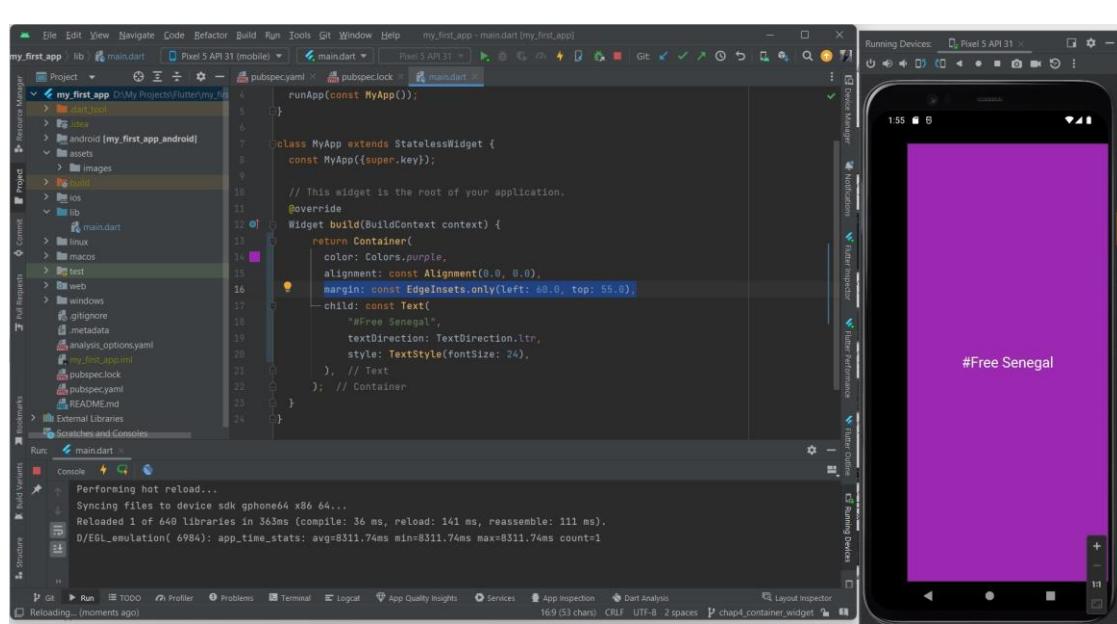


- **Le constructeur nommé `EdgeInsets.only()`**

Ce constructeur permet de définir seulement que le ou les directions spécifiées.

Par exemple :

```
margin: const EdgeInsets.only(left: 60.0, top: 55.0),
```



## Le paramètre nommé padding

Ce paramètre permet de mettre du `padding` autour du `child`, c'est-à-dire de l'**espace vide entre la decoration et le child**.

Le `padding` utilise exactement les mêmes constructeurs de la classe `EdgeInsets` que nous venons de voir.

Voici un exemple :

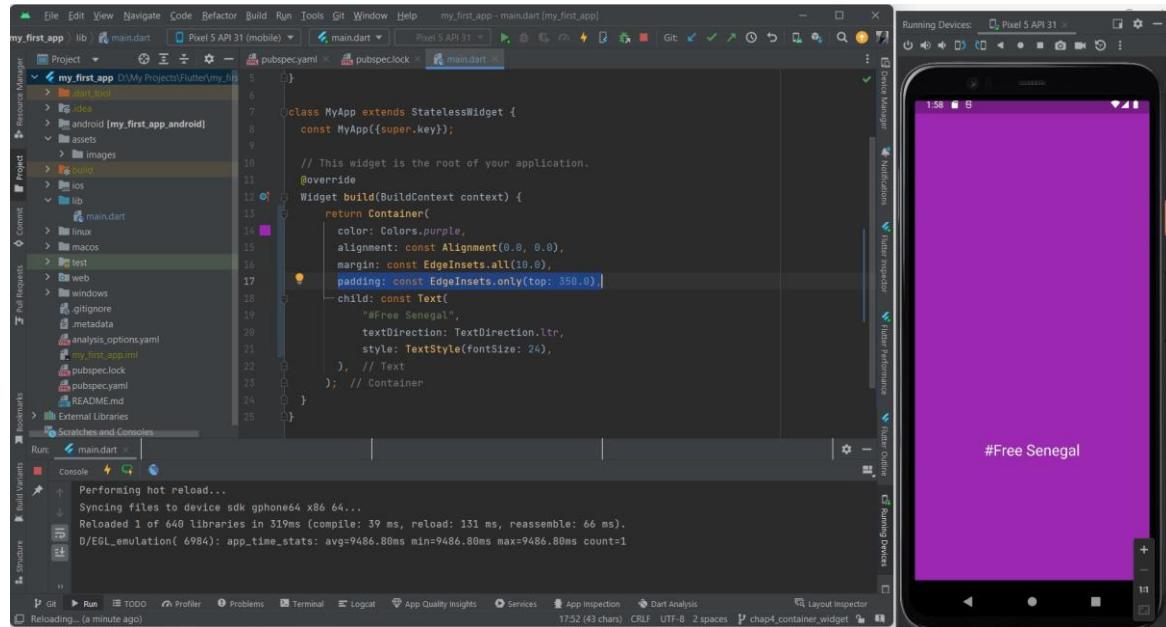
```
import 'package:flutter/material.dart';

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

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

    // This widget is the root of your application.
    @override
    Widget build(BuildContext context) {
        return Container(
            color: Colors.purple,
            alignment: const Alignment(0.0, 0.0),
            margin: const EdgeInsets.all(10.0),
            padding: const EdgeInsets.only(top: 350.0),
            child: const Text(
                "#Free Senegal",
                textDirection: TextDirection.ltr,
                style: TextStyle(fontSize: 24),
            ),
        );
    }
}
```

Voici le résultat :



## Le paramètre nommé constraint

Ce paramètre permet d'imposer des contraintes à un widget. Par exemple, une **hauteur minimale** et ou **maximale**.

Prenons un exemple basique que nous allons expliquer :

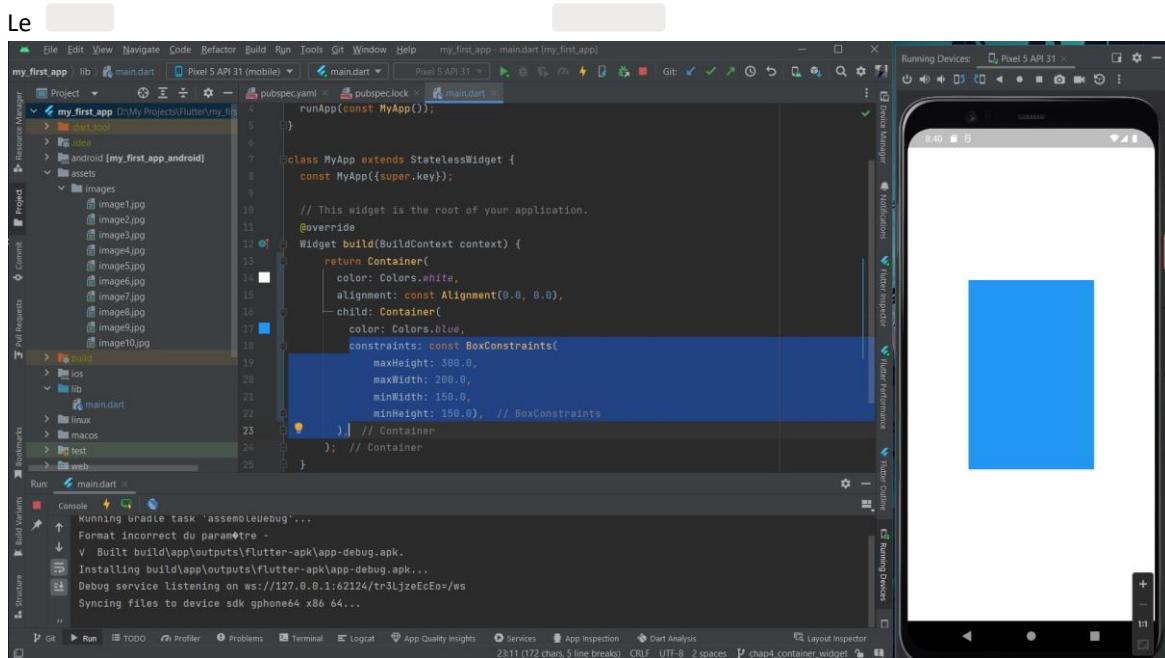
```
import 'package:flutter/material.dart';

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

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

    // This widget is the root of your application.
    @override
    Widget build(BuildContext context) {
        return Container(
            color: Colors.white,
            alignment: const Alignment(0.0, 0.0),
            child: Container(
                color: Colors.blue,
                constraints: const BoxConstraints(
                    maxHeight: 300.0,
                    maxWidth: 200.0,
                    minWidth: 150.0,
                    minHeight: 150.0),
            ),
        );
    }
}
```

On a comme résultat:



Nous avons un **widget container** qui a un **fond blanc** et qui aligne son **child** de manière centrée comme nous l'avions vu précédemment.

**child** du premier **container** est un second **container** qui a un **fond bleu** et nous lui imposons des contraintes de taille en lui fixant des hauteurs et des largeurs minimales et maximales.

Le constructeur `BoxConstraints` permet de créer une boîte minimale et une boîte maximale. Le child doit forcément respecter la taille de la boîte minimale ou de la boîte maximale.

Il respectera la boîte maximale si il n'a pas de child, sinon il tentera de respecter d'abord la boîte minimale. Comme nous pouvons le voir ici :

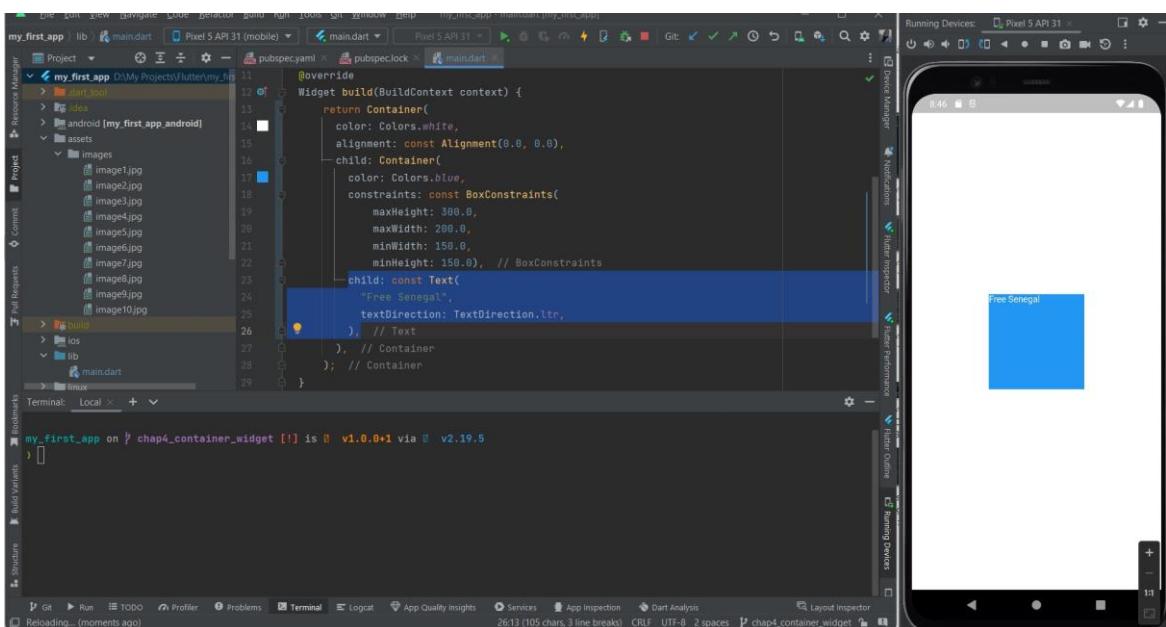
```
import 'package:flutter/material.dart';

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

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

    // This widget is the root of your application.
    @override
    Widget build(BuildContext context) {
        return Container(
            color: Colors.white,
            alignment: const Alignment(0.0, 0.0),
            child: Container(
                color: Colors.blue,
                constraints: const BoxConstraints(
                    maxHeight: 300.0,
                    maxWidth: 200.0,
                    minWidth: 150.0,
                    minHeight: 150.0),
                child: const Text(
                    "Free Senegal",
                    textDirection: TextDirection.ltr,
                ),
            ),
        );
    }
}
```

On a comme résultat :

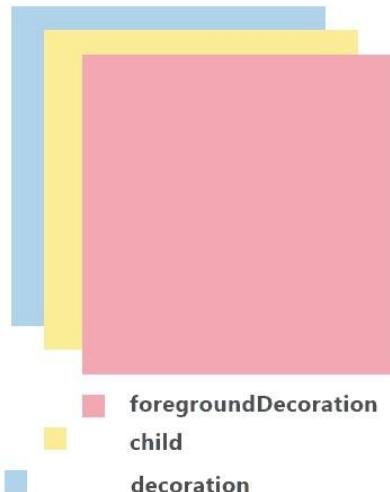


Il existe d'autres constructeurs pour les contraintes, que nous verrons au fur et à mesure de nos besoins.

## Le paramètre nommé decoration

Nous avons déjà abordé ce paramètre avec le `widget Text`.

Ici, la `decoration` s'applique **derrière** le `child`, alors que la `foregroundDecoration` que nous allons voir, s'applique **devant** comme illustré sur l'image ci-dessous:



Les `decorations` peuvent être de plusieurs types, chaque type étant une classe qui implémente la classe `Decoration`.

Nous allons voir seulement les deux principales classes qui nous serons utiles.

- La classe `BoxDecoration`

La classe `BoxDecoration` permet de dessiner des boîtes de forme rectangulaire ou circulaire.

Son constructeur est le suivant :

```
const BoxDecoration({  
  Color color,  
  DecorationImage image,  
  BoxBorder border,  
  BorderRadiusGeometry borderRadius,  
  List<BoxShadow> boxShadow,  
  Gradient gradient,  
  BlendMode backgroundBlendMode,  
  BoxShape shape: BoxShape.rectangle  
})
```

Voici un exemple de `BoxDecoration` de forme circulaire :

```
import 'package:flutter/material.dart';  
  
void main() {  
  runApp(const MyApp());  
}
```

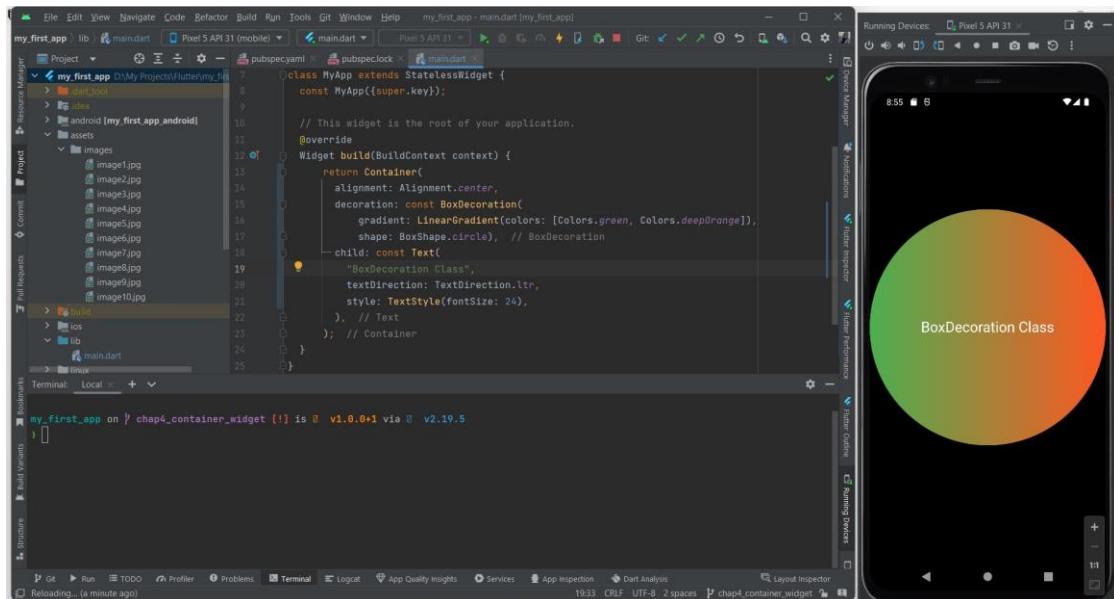
```

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

  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return Container(
      alignment: Alignment.center,
      decoration: const BoxDecoration(
        gradient: LinearGradient(colors: [Colors.green, Colors.deepOrange]),
        shape: BoxShape.circle),
      child: const Text(
        "BoxDecoration Class",
        textDirection: TextDirection.ltr,
        style: TextStyle(fontSize: 24),
      ),
    );
  }
}

```

Voici le résultat :



Nous avons utilisé la propriété `gradient`, permettant de créer un gradient entre plusieurs couleurs.

Il est possible également d'utiliser d'autres propriétés comme `border` qui permet de paramétrer une bordure, par exemple :

```

import 'package:flutter/material.dart';

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

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

  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {

```

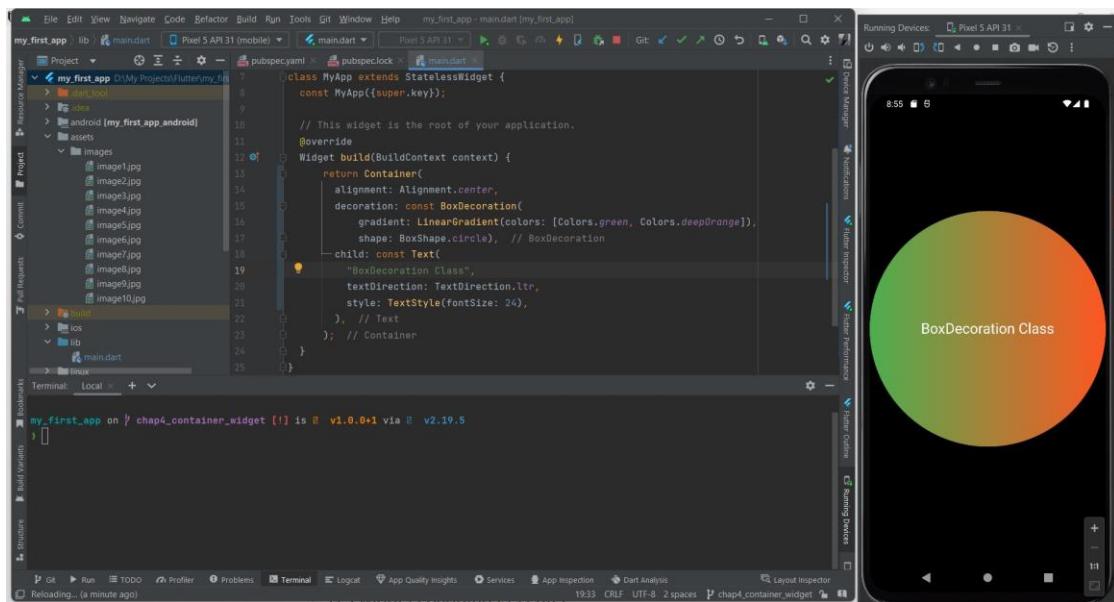
```

        return Container(
            alignment: Alignment.center,
            decoration: BoxDecoration(
                gradient: const LinearGradient(colors: [Colors.green, Colors.deepOrange]),
                shape: BoxShape.circle,
                border: Border.all(
                    color: Colors.white,
                    width: 10.0,
                ),
            ),
            child: const Text(
                "BoxDecoration Class",
                textDirection: TextDirection.ltr,
                style: TextStyle(fontSize: 24),
            ),
        );
    );
}

```

Ce code ajoute une bordure blanche de 10 pixels autour du cercle.

Voici le résultat :



Il est possible d'utiliser également d'autres propriétés que nous verrons au fur à mesure.

- [La classe ShapeDecoration](#)

La classe `ShapeDecoration` dans Flutter permet de décorer une forme géométrique donnée en appliquant différents effets visuels.

Son constructeur est le suivant :

```

const ShapeDecoration({
    Color color,
    DecorationImage image,
    Gradient gradient,
    List<BoxShadow> shadows,
    required ShapeBorder shape
})

```

Voici les propriétés disponibles dans la classe `ShapeDecoration` :

- `color` : La couleur de remplissage de la forme.
- `gradient` : Le dégradé à appliquer comme remplissage de la forme. `shape`
- : La forme de la décoration (qui est obligatoire). `shadows` : La liste des
- `ombres` à appliquer à la forme. `image` : L'image à afficher comme fond de
- la forme.
- `border` : La bordure de la forme.

Nous pouvons prendre une forme parmi les nombreuses disponibles :

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

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

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

  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return Container(
      color: Colors.white,
      alignment: Alignment.center,
      child: Container(
        constraints: const BoxConstraints(
          maxHeight: 350.0,
          maxWidth: 300.0,
        ),
        padding: const EdgeInsets.all(10),
        decoration: const ShapeDecoration(
          shape: RoundedRectangleBorder(
            borderRadius: BorderRadius.all(
              Radius.circular(20.0),
            ),
          ),
          color: Colors.blue,
        ),
        child: const Text("ShapeDecoration Class",
          textDirection: TextDirection.ltr,
          style: TextStyle(
            fontSize: 30.0,
          )),
      ),
    );
  }
}
```

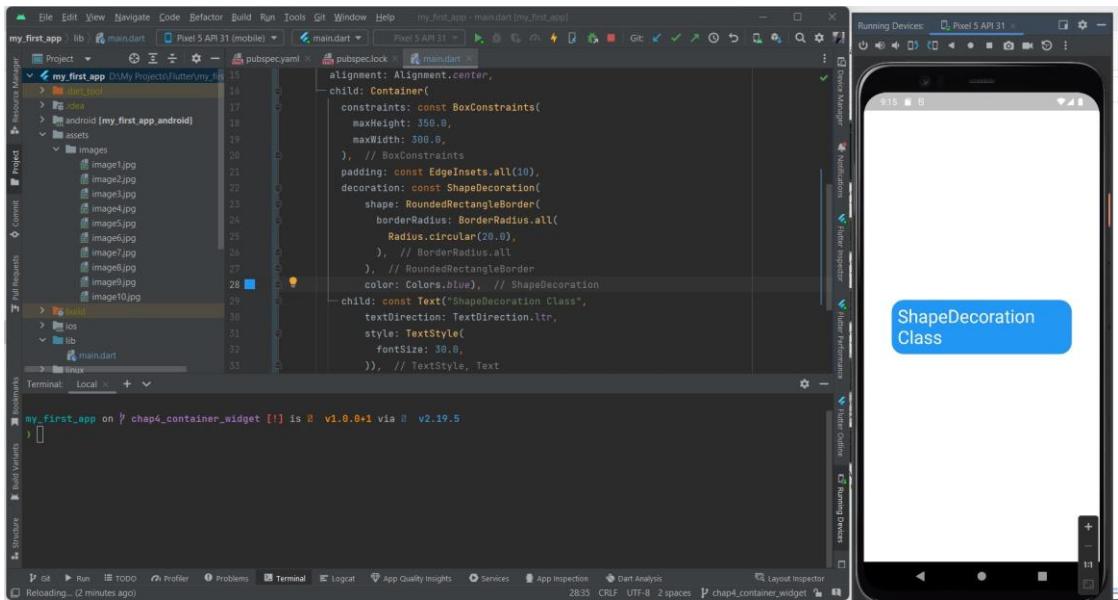
Ici, nous utilisons `RoundedRectangleBorder` qui nous permet de dessiner un rectangle avec des bords arrondis.

Nous utilisons un `widget container` parent afin de pouvoir imposer des**contraintes** de **hauteur** et de **largeur**.

Nous ajoutons du `padding` afin que le rectangle prenne plus que la taille du texte.

Nous n'oublions pas de centrer le `container` enfant, en effet, sans alignement, nous ne pouvons imposer de contraintes.

Le résultat est le suivant :



## Le paramètre nommé foregroundDecoration

Ce paramètre permet de dessiner une décoration **devant** le `child`, contrairement à `decoration` qui permet de la dessiner **derrière**.

Par exemple, nous pouvons créer une boîte avec une bordure arrondie avec un rayon de 10 et une bordure noire de largeur 2 et la remplir d'une couleur tout en diminuant son opacité pour voir transparaître le texte :

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

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

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

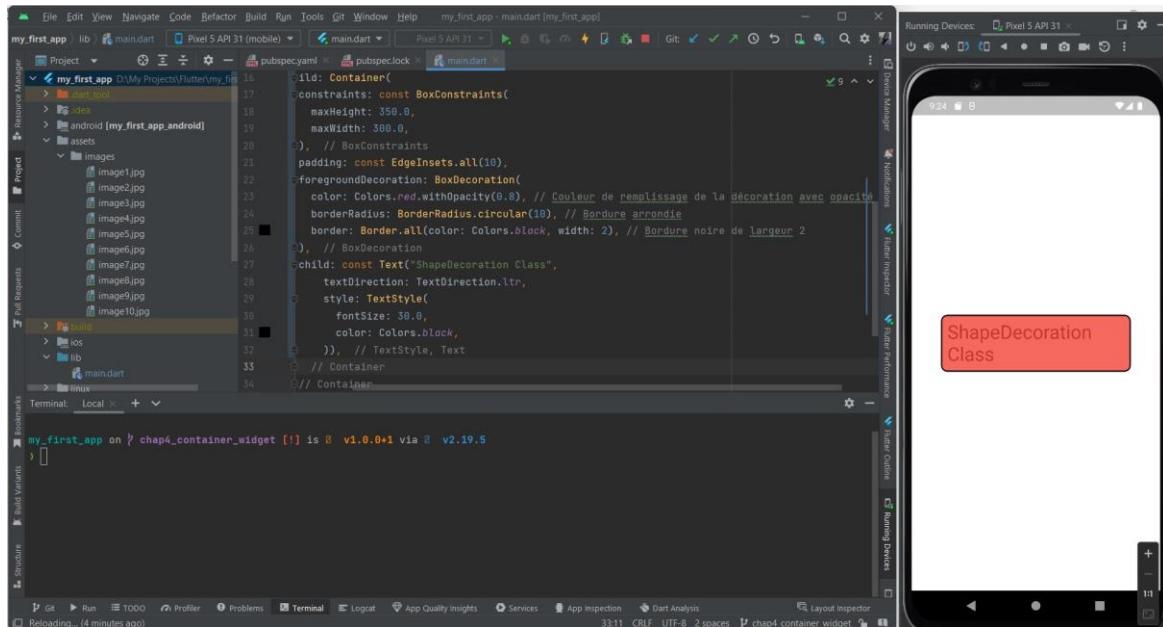
    // This widget is the root of your application.
    @override
    Widget build(BuildContext context) {
        return Container(
            color: Colors.white,
            alignment: Alignment.center,
            child: Container(
                constraints: const BoxConstraints(
                    maxHeight: 350.0,
                    maxWidth: 300.0,
                ),
                padding: const EdgeInsets.all(10),
                foregroundDecoration: BoxDecoration(
                    color: Colors.red.withOpacity(0.8), // Couleur de remplissage de la décoration avec opacité
                    borderRadius: BorderRadius.circular(10), // Bordure arrondie
                    border: Border.all(color: Colors.black, width: 2), // Bordure noire de largeur 2
                ),
                child: const Text("ShapeDecoration Class",
                    textDirection: TextDirection.ltr,
                    style: TextStyle(
                        fontSize: 30.0,
```

```

        color: Colors.black,
    )),
),
);
}
}

```

Voici le résultat :



## Le paramètre nommé transform

Le paramètre nommé `transform` permet d'appliquer une transformation sur un élément, tel qu'un widget, pour le déplacer, le faire pivoter, l'agrandir ou le réduire.

Il prend en argument un objet de la classe `Matrix4`, qui représente une transformation 4x4 dans l'espace en 3D.

C'est un peu complexe car cela implique des maths pour les gérer, mais nous verrons les plus communes au fur et à mesure.

Voici un exemple où nous effectuons une translation de 100 pixels horizontalement. Cela déplacera le texte vers la droite de 100 pixels. :

```

import 'dart:math';

import 'package:flutter/material.dart';

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

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

    // This widget is the root of your application.
    @override
    Widget build(BuildContext context) {
        return Container(
            color: Colors.white,

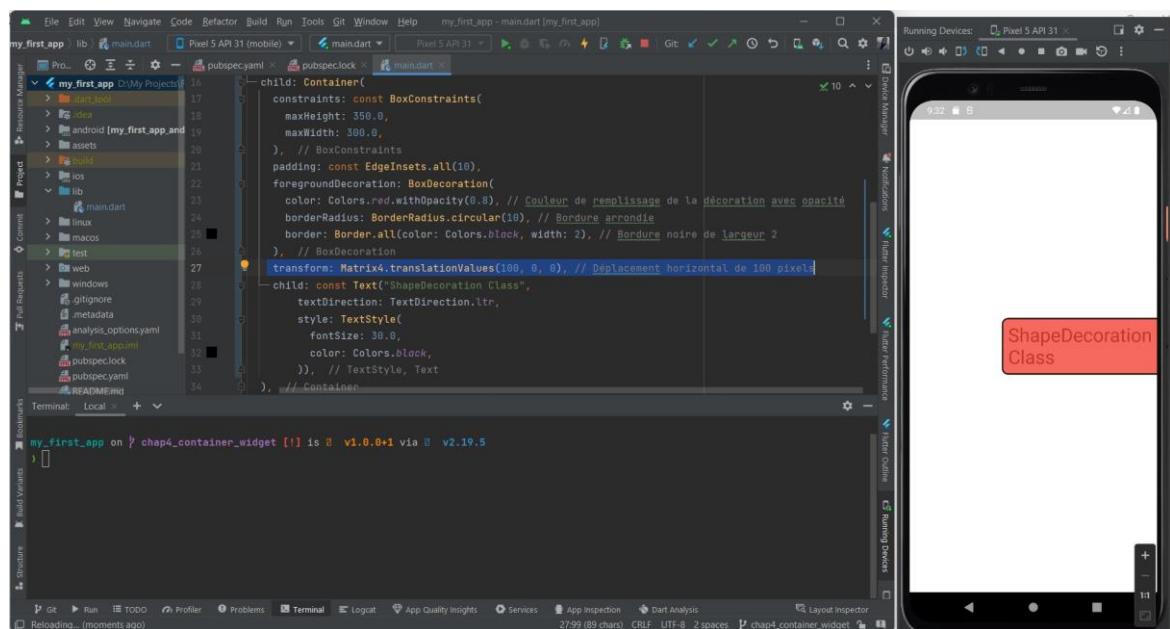
```

```

        alignment: Alignment.center,
        child: Container(
            constraints: const BoxConstraints(
                maxHeight: 350.0,
                maxWidth: 300.0,
            ),
            padding: const EdgeInsets.all(10),
            foregroundDecoration: BoxDecoration(
                color: Colors.red.withOpacity(0.8), // Couleur de remplissage de la décoration avec opacité
                borderRadius: BorderRadius.circular(10), // Bordure arrondie
                border: Border.all(color: Colors.black, width: 2), // Bordure noire de largeur 2
            ),
            transform: Matrix4.rotationZ(pi / 2), // Déplacement horizontal de 100 pixels
            child: const Text("ShapeDecoration Class",
                textDirection: TextDirection.ltr,
                style: TextStyle(
                    fontSize: 30.0,
                    color: Colors.black,
                )),
        ),
    );
}
}

```

Voici le résultat :



D'autres méthodes de la classe `Matrix4` peuvent être utilisées pour appliquer d'autres types de transformations, comme `rotationX`, `rotationY`, `scale`, etc. Vous pouvez combiner plusieurs transformations en les multipliant avec l'opérateur `*`.

Voici un exemple où nous effectuons une rotation de 90 degrés :

```

import 'dart:math';

import 'package:flutter/material.dart';

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

```

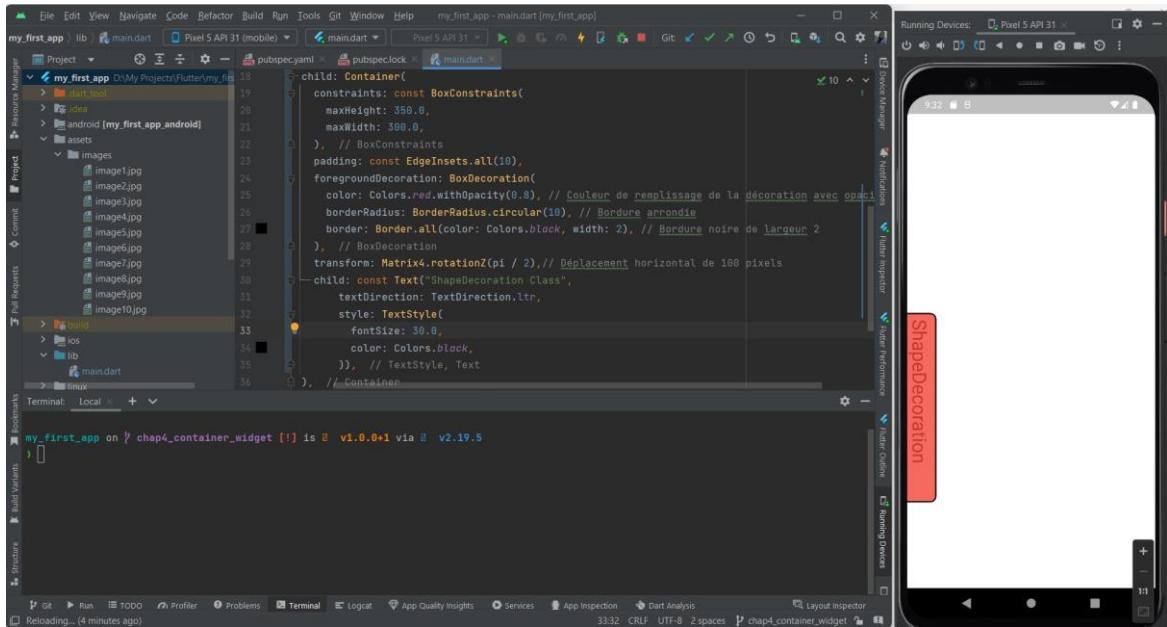
```

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

    // This widget is the root of your application.
    @override
    Widget build(BuildContext context) {
        return Container(
            color: Colors.white,
            alignment: Alignment.center,
            child: Container(
                constraints: const BoxConstraints(
                    maxHeight: 350.0,
                    maxWidth: 300.0,
                ),
                padding: const EdgeInsets.all(10),
                foregroundDecoration: BoxDecoration(
                    color: Colors.red.withOpacity(0.8), // Couleur de remplissage de la décoration avec opacité
                    borderRadius: BorderRadius.circular(10), // Bordure arrondie
                    border: Border.all(color: Colors.black, width: 2), // Bordure noire de largeur 2
                ),
                transform: Matrix4.rotationZ(pi / 2), // Rotation de 90 degrés
                child: const Text("ShapeDecoration Class",
                    textDirection: TextDirection.ltr,
                    style: TextStyle(
                        fontSize: 30.0,
                        color: Colors.black,
                    )),
            ),
        );
    }
}

```

**Voici le résultat :**



L'utilisation du paramètre `transform` est utile lorsque vous souhaitez animer ou modifier visuellement la position ou l'apparence d'un élément.

Notez que la classe `Matrix4` permet également de spécifier des transformations en 2D en utilisant les composants de la matrice 4x4 appropriés.

## ▼ Les widgets padding et center

### Le widget Padding

Ce `widget` est extrêmement simple, il permet de créer un **décalage d'espace vide** autour de son `child`. Il permet de définir un décalage (padding) à **gauche**, à **droite**, en **haut** et en **bas** de l'enfant.

Voici un exemple d'utilisation du widget `Padding` :

```
Padding(  
  padding: EdgeInsets.all(16.0), // Définit un décalage de 16 pixels sur tous les côtés  
  child: Text('Contenu du widget'),  
)
```

Dans cet exemple, le widget `Padding` entoure le widget `Text` avec un décalage de 16 pixels sur tous les côtés. Cela crée un espace vide de 16 pixels autour du texte.

Le paramètre `padding` du widget `Padding` est de type `EdgeInsets`, qui permet de spécifier les décalages individuels pour chaque côté (gauche, haut, droite, bas) à l'aide des propriétés `EdgeInsets.only()`, `EdgeInsets.symmetric()` ou `EdgeInsets.all()`. Par exemple :

```
Padding(  
  padding: EdgeInsets.only(left: 20.0, top: 10.0),  
  child: Text('Contenu du widget'),  
)
```

Dans cet exemple, le widget `Text` est entouré d'un décalage de 20 pixels à gauche et de 10 pixels en haut, tandis que les autres côtés n'ont pas de décalage.

Le widget `Padding` est utile lorsque vous souhaitez ajouter de l'espace vide autour d'un widget spécifique sans utiliser un `Container` pour encapsuler le widget.

En fait, le `widget Container` utilise ce widget lorsque vous utilisez la propriété `padding`. Il construit ce widget pour vous.

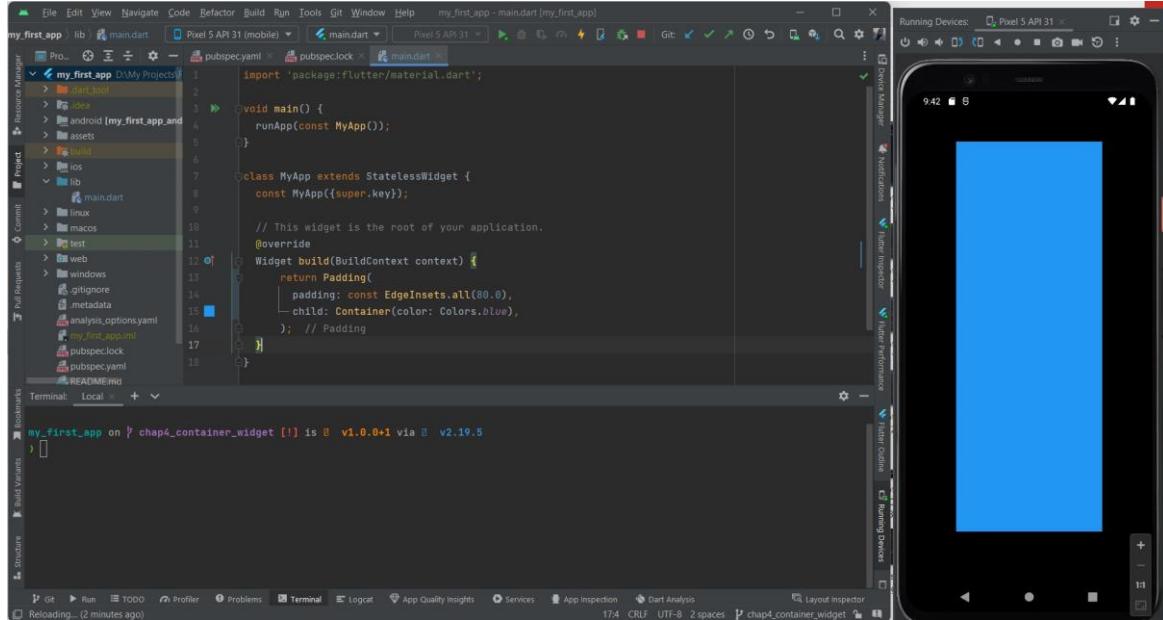
Si vous ne voulez pas utiliser un container, vous pouvez simplement utiliser le widget `Padding`.

Voici un exemple simple avec un `widget Container` :

```
import 'package:flutter/material.dart';  
  
void main() {  
  runApp(const MyApp());  
}  
  
class MyApp extends StatelessWidget {  
  const MyApp({super.key});  
  
  // This widget is the root of your application.  
  @override  
  Widget build(BuildContext context) {  
    return Padding(  
      padding: const EdgeInsets.all(80.0),  
      child: Container(color: Colors.blue),  
    );  
}
```

```
}
```

Voici le résultat :



## Le widget Center

Ce widget est également très simple, il permet de centrer le child passé en argument.

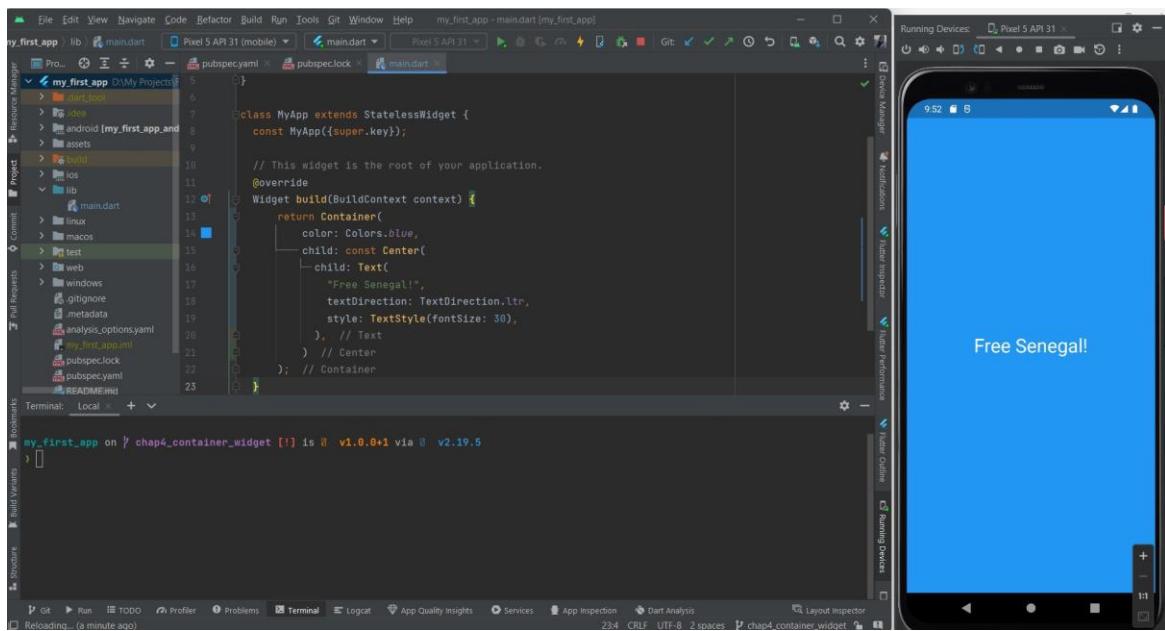
```
import 'package:flutter/material.dart';

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

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

  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return Container(
      color: Colors.blue,
      child: const Center(
        child: Text(
          "Free Senegal!",
          textDirection: TextDirection.ltr,
          style: TextStyle(fontSize: 30),
        ),
      ),
    );
}
}
```

Voici le résultat :



## ▼ Les widgets column et expanded

### Le widget Column

Le `widget Column` permet d'afficher des `widgets` enfants en colonne.

Ce `widget` utilise toujours `children` car l'objectif est d'aligner verticalement des éléments.

Prenons un exemple de base avec des carrés de couleur :

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

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

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

    // This widget is the root of your application.
    @override
    Widget build(BuildContext context) {
        return Column(
            children: [
                Container(
                    color: Colors.green,
                    height: 100,
                    width: 100,
                ),
                Container(
                    color: Colors.yellow,
                    height: 100,
                    width: 100,
                ),
                Container(
                    color: Colors.red,
                    height: 100,
                    width: 100,
                ),
                Container(
                    color: Colors.blue,
                    height: 100,
                    width: 100,
                )
            ],
        );
    }
}
```

Voici le résultat :

The screenshot shows the Android Studio interface with the following details:

- File Bar:** File, Edit, View, Navigate, Code, Refactor, Build, Run, Tools, Git, Window, Help.
- Project Structure:** my\_first\_app (D:\My Projects\), lib, assets, build, ios, test, web, windows, metadata, analysis\_options.yaml, my\_first\_app.iml, pubspec.lock, pubspec.yaml, README.md, External Libraries, Scratches and Consoles.
- Code Editor:** The main.dart file is open, displaying a Column widget with five Container children, each with a different color (green, yellow, red, blue) and dimensions (height: 100, width: 100).
- Terminal:** Local, showing the command: my\_first\_app on my\_pc chap4\_container\_widget [!] is v1.0.0+1 via v2.19.5.
- Emulator:** Running on Pixel 5 API 31, showing a vertical stack of colored boxes (green, yellow, red, blue) corresponding to the widget structure.
- Bottom Navigation:** P, Git, Run, TODO, Profiler, Problems, Terminal, Logcat, App Quality Insights, Services, App Inspection, Dart Analysis, Layout Inspector.

Nous allons ensuite gérer finement l'alignement et l'espacement entre les éléments.

Pour gérer l'alignement il faut comprendre les notions d'**axe principal** et d'**axe croisé**.

Lorsque nous utilisons le `widget Column`, le l'axe principal (le `main axis`) est l'axe vertical.

L'axe croisé (le ~~cross axis~~) est l'axe horizontal.



## 1. L'alignement sur l'axe croisé avec la propriété crossAxisAlignment

La propriété `crossAxisAlignment` permet de gérer l'alignement sur l'axe horizontal lorsque nous utilisons le `widget Column`.

Il est obligatoire de préciser la propriété `textDirection` afin que `Flutter` puisse savoir comment positionner les éléments.

Il peut avoir plusieurs valeurs, commençons par `start`, `center` et `end`.

- `start` : permet de placer les children au début de l'axe croisé, ici horizontal, donc le plus à gauche.

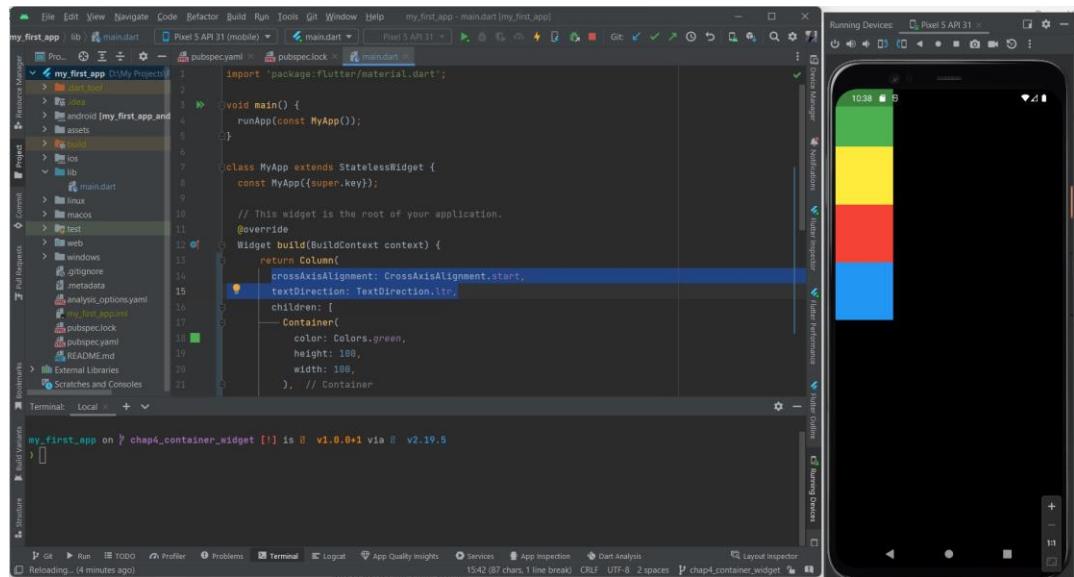
```
import 'package:flutter/material.dart';

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

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

    // This widget is the root of your application.
    @override
    Widget build(BuildContext context) {
        return Column(
            crossAxisAlignment: CrossAxisAlignment.start,
            textDirection: TextDirection.ltr,
            children: [
                Container(
                    color: Colors.green,
                    height: 100,
                    width: 100,
                ),
                Container(
                    color: Colors.yellow,
                    height: 100,
                    width: 100,
                ),
                Container(
                    color: Colors.red,
                    height: 100,
                    width: 100,
                ),
                Container(
                    color: Colors.blue,
                    height: 100,
                    width: 100,
                )
            ],
        );
    }
}
```

Voici le résultat :



- `center` : permet de les placer au milieu de l'axe croisé, ici horizontal.

```

import 'package:flutter/material.dart';

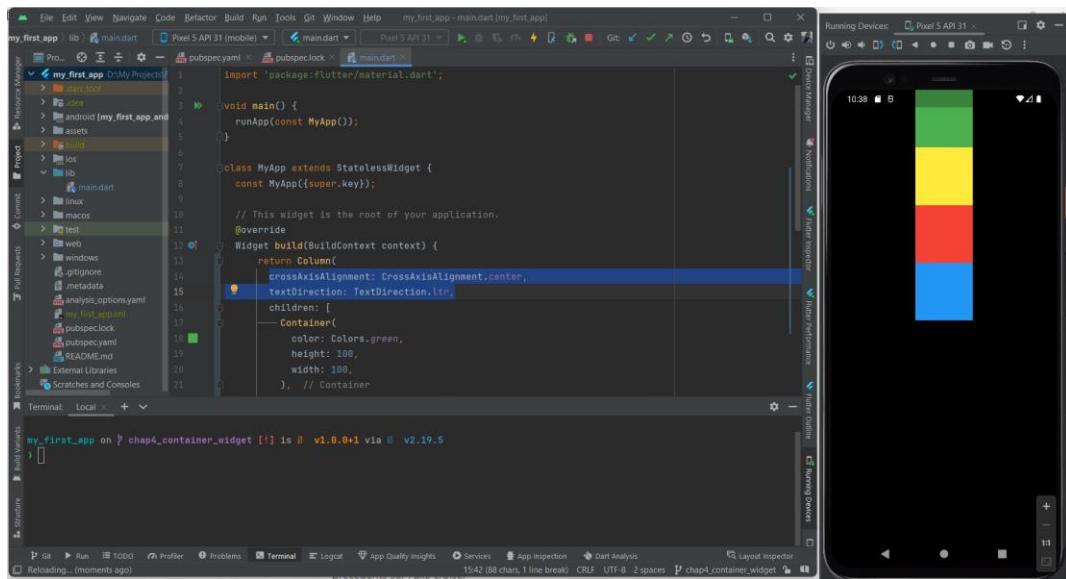
void main() {
    runApp(const MyApp());
}

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

    // This widget is the root of your application.
    @override
    Widget build(BuildContext context) {
        return Column(
            mainAxisAlignment: MainAxisAlignment.center,
            textDirection: TextDirection.ltr,
            children: [
                Container(
                    color: Colors.green,
                    height: 100,
                    width: 100),
                Container(
                    color: Colors.yellow,
                    height: 100,
                    width: 100),
                Container(
                    color: Colors.red,
                    height: 100,
                    width: 100),
                Container(
                    color: Colors.blue,
                    height: 100,
                    width: 100),
            ],
        );
    }
}

```

Voici le résultat :



- `end` : permet de les placer à la fin de l'axe croisé, ici horizontal.

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

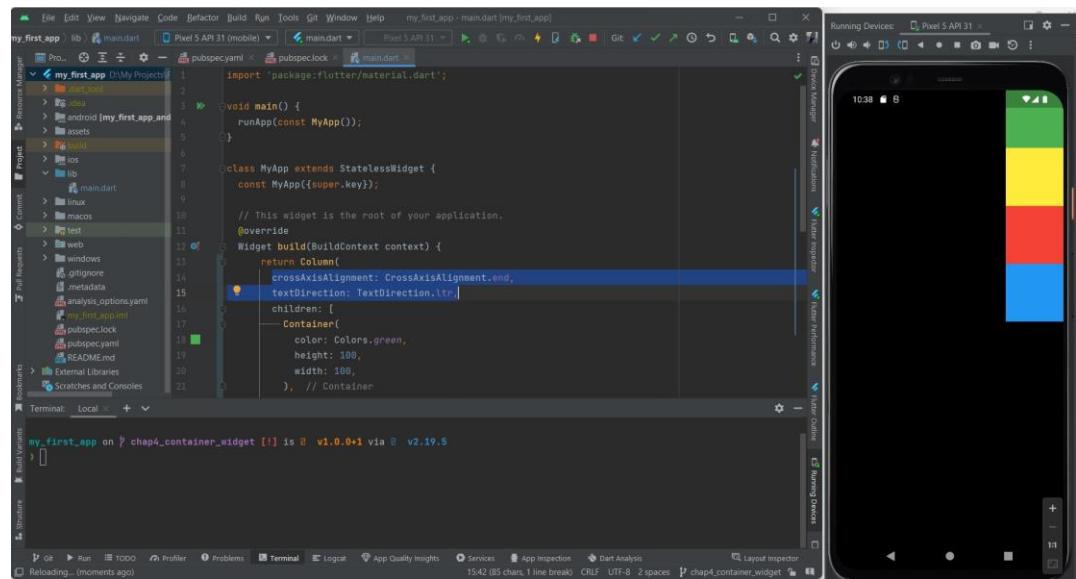
void main() {
    runApp(const MyApp());
}

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

    // This widget is the root of your application.
    @override
    Widget build(BuildContext context) {
        return Column(
            mainAxisAlignment: MainAxisAlignment.end,
            textDirection: TextDirection.ltr,
            children: [
                Container(
                    color: Colors.green,
                    height: 100,
                    width: 100,
                ),
                Container(
                    color: Colors.yellow,
                    height: 100,
                    width: 100,
                ),
                Container(
                    color: Colors.red,
                    height: 100,
                    width: 100,
                ),
                Container(
                    color: Colors.blue,
                    height: 100,
                    width: 100,
                )
            ],
        );
    }
}
```



Voici le résultat :



La dernière valeur possible est `stretch` qui permet d'étendre les éléments au maximum de l'axe croisé, ici horizontal :

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

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

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

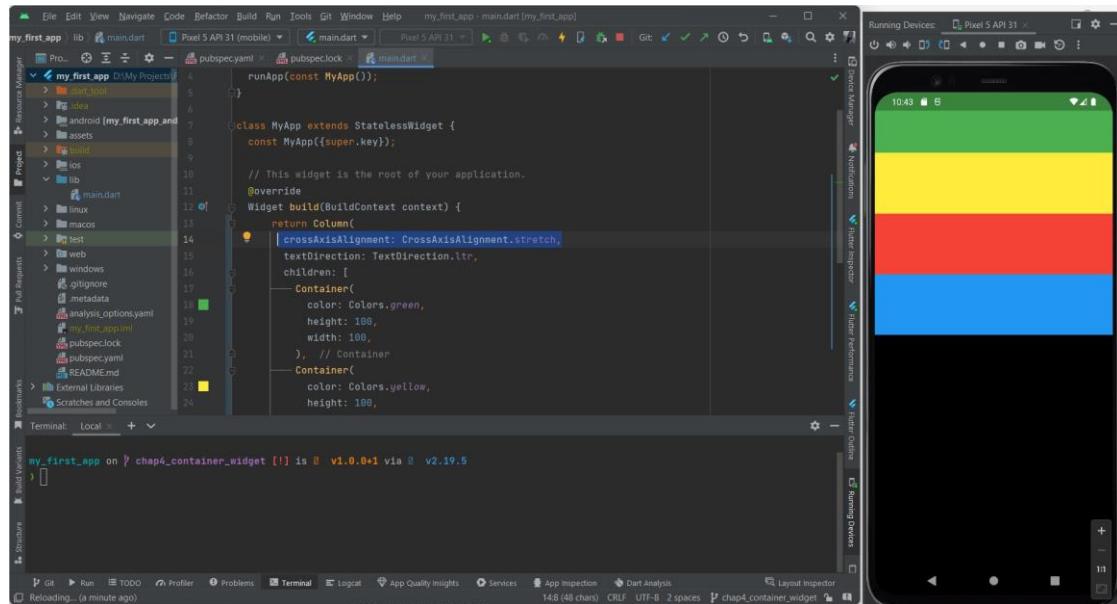
  @override
  Widget build(BuildContext context) {
    return Column(
      mainAxisAlignment: MainAxisAlignment.end,
      textDirection: TextDirection.ltr,
      children: [
        Container(
          color: Colors.green,
          height: 100,
          width: 100,
        ),
        Container(
          color: Colors.yellow,
          height: 100,
          width: 100,
        ),
        Container(
          color: Colors.red,
          height: 100,
          width: 100,
        ),
        Container(
          color: Colors.blue,
          height: 100,
          width: 100,
        ),
      ],
    );
}
```

```

        ],
    );
}
}

```

Voici le résultat :



2. **L'alignement sur l'axe principal avec la propriété mainAxisAlignment** L'alignement sur l'axe principal est ici l'alignement sur l'axe vertical.

Il permet de déterminer comment sont placés les éléments sur l'axe vertical.

La propriété `mainAxisAlignment` peut prendre les valeurs `start`, `center`, `end`, `spaceAround`, `spaceBetween` et `spaceEvenly`.

Il est également obligatoire de préciser la propriété `textDirection` afin de `Flutter` puisse savoir comment positionner les éléments.

Nous allons commencer par voir `start`, `center` et `end`, en gardant les alignements précédents sur l'axe croisé.

Voici trois `widgets Column` placés en rangé (ne vous focalisez pas sur `Row` pour le moment, nous l'étudierons dans la leçon suivante) :

```

import 'package:flutter/material.dart';

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

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

    // This widget is the root of your application.
    @override
    Widget build(BuildContext context) {
        return Row(
            textDirection: TextDirection.ltr,

```



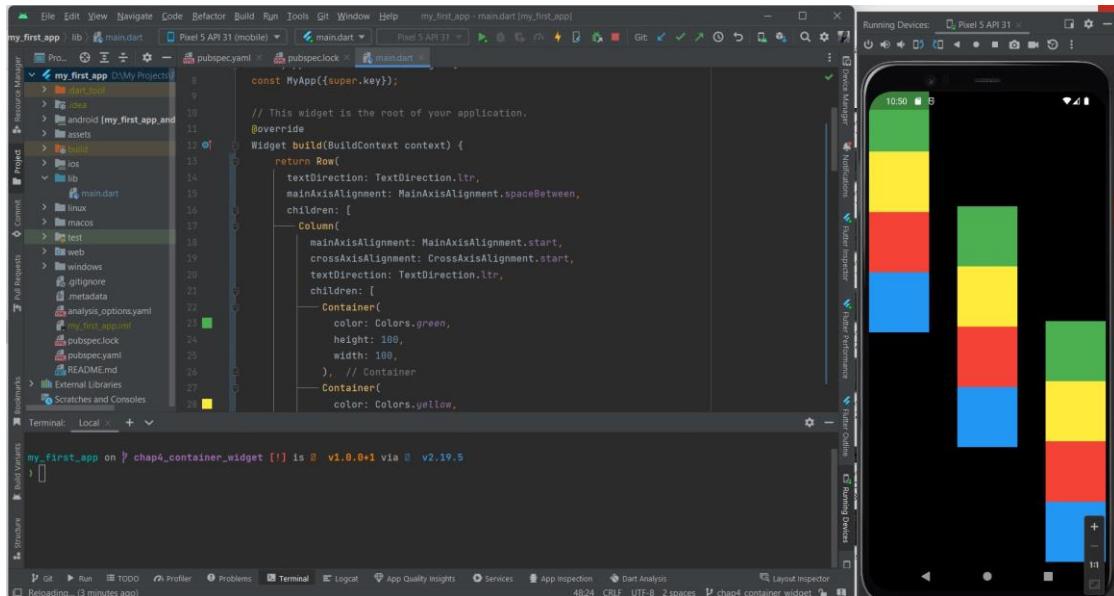


```

        width: 100,
    ),
    Container(
        color: Colors.red,
        height: 100,
        width: 100,
    ),
    Container(
        color: Colors.blue,
        height: 100,
        width: 100,
    )
),
],
);
}
}

```

Voici le résultat :



La première `Column` affiche ses `children` au **début** de l'axe principal (vertical) et de l'axe croisé (horizontal), la deuxième les aligne au **milieu** de chaque axe, et la dernière les aligne à la **fin** de chaque axe.

Nous allons maintenant passer à `spaceAround`, `spaceBetween` et `spaceEvenly` :

```

import 'package:flutter/material.dart';

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

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

    // This widget is the root of your application.
    @override
    Widget build(BuildContext context) {
        return Row(
            textDirection: TextDirection.ltr,

```

```
    mainAxisAlignment: MainAxisAlignment.spaceBetween,
    children: [
      Column(
        mainAxisAlignment: MainAxisAlignment.spaceBetween,
        crossAxisAlignment: CrossAxisAlignment.start,
        textDirection: TextDirection.ltr,
        children: [
          Container(
            color: Colors.green,
            height: 100,
            width: 100,
            ),
          Container(
            color: Colors.yellow,
            height: 100,
            width: 100,
            ),
          Container(
            color: Colors.red,
            height: 100,
            width: 100,
            ),
          Container(
            color: Colors.blue,
            height: 100,
            width: 100,
            )
        ],
      ),
      Column(
        mainAxisAlignment: MainAxisAlignment.spaceAround,
        crossAxisAlignment: CrossAxisAlignment.center,
        textDirection: TextDirection.ltr,
        children: [
          Container(
            color: Colors.green,
            height: 100,
            width: 100,
            ),
          Container(
            color: Colors.yellow,
            height: 100,
            width: 100,
            ),
          Container(
            color: Colors.red,
            height: 100,
            width: 100,
            ),
          Container(
            color: Colors.blue,
            height: 100,
            width: 100,
            )
        ],
      ),
      Column(
        mainAxisAlignment: MainAxisAlignment.spaceEvenly,
        crossAxisAlignment: CrossAxisAlignment.end,
        textDirection: TextDirection.ltr,
        children: [
          Container(
            color: Colors.green,
            height: 100,
            width: 100,
            ),
          Container(
            color: Colors.yellow,
            height: 100,
            width: 100,
            )
        ],
      ),
    ],
  ),

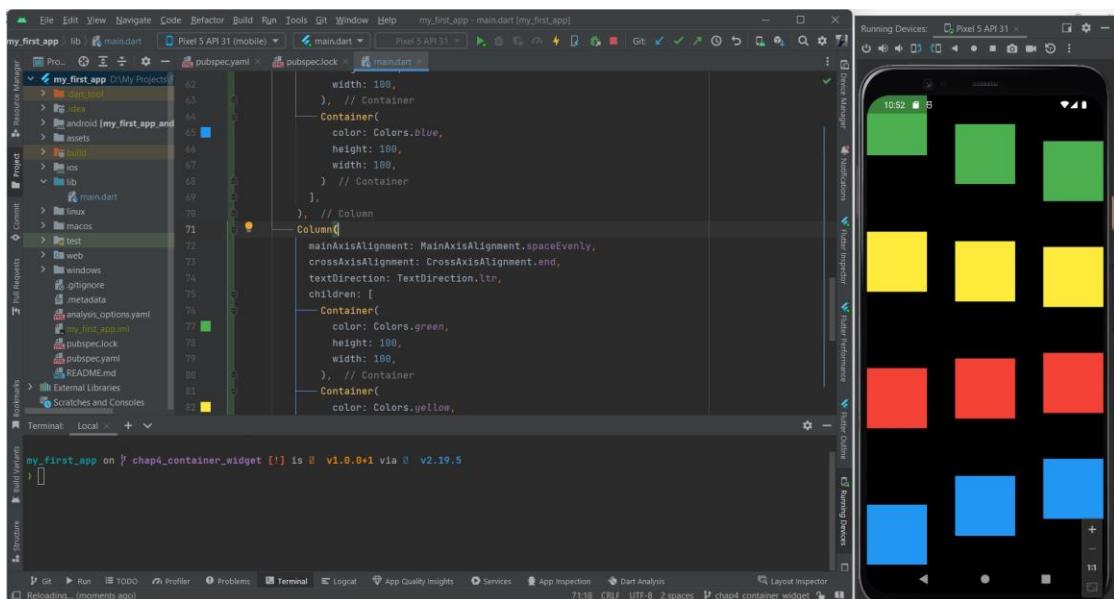
```

```

        width: 100,
    ),
    Container(
        color: Colors.red,
        height: 100,
        width: 100,
    ),
    Container(
        color: Colors.blue,
        height: 100,
        width: 100,
    )
),
],
);
}
}

```

Voici le résultat :



Nous pouvons observer que `spaceBetween` permet de placer tout l'espace disponible de manière égale entre les `children`.

Nous pouvons également voir que `spaceAround` permet de placer l'espace disponible de manière égale entre les `children` tout en gardant de l'espace pour placer une moitié de l'espace entre deux éléments **avant** et une moitié **après** les `children`.

Enfin `spaceEvenly` permet de placer tout l'espace disponible de manière égale avant, entre et après les `children`.

## Le widget Expanded

Ce `widget` permet d'étendre un `child` d'une `Column` ou d'une `Row` pour qu'il remplisse tout l'espace disponible.

Le `child` s'étendra sur l'axe principal donc vertical pour `Column`.

Voici un premier exemple

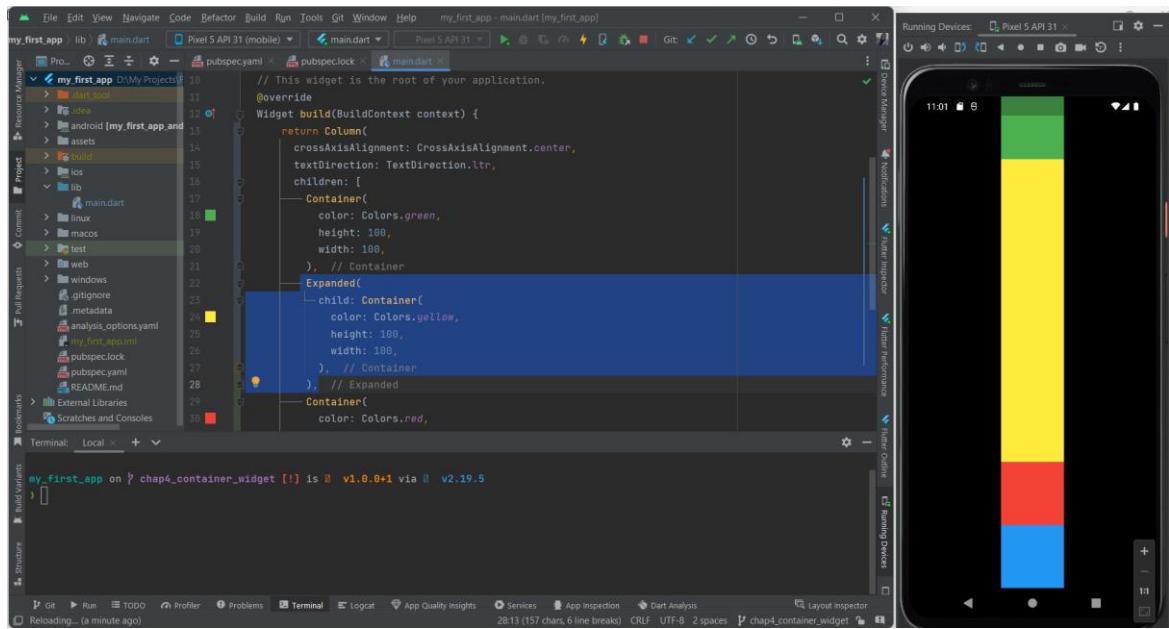
```
import 'package:flutter/material.dart';

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

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

  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return Column(
      mainAxisAlignment: MainAxisAlignment.center,
      textDirection: TextDirection.ltr,
      children: [
        Container(
          color: Colors.green,
          height: 100,
          width: 100,
        ),
        Expanded(
          child: Container(
            color: Colors.yellow,
            height: 100,
            width: 100,
          ),
        ),
        Container(
          color: Colors.red,
          height: 100,
          width: 100,
        ),
        Container(
          color: Colors.blue,
          height: 100,
          width: 100,
        )
      ],
    );
  }
}
```

Nous avons étendu le Container jaune avec Expanded, cela donne :



Il est également possible d'utiliser `Expanded` sur plusieurs `child`.

Dans ce cas, ils se répartiront de manière équitable l'espace disponible.

Mais vous pouvez la répartir à votre convenance avec la propriété `flex`.

En donnant par exemple à un `child` d'un `widget Expanded` un `flex` de `3`, et en donnant un `flex` de `1` à un autre, il prendra 3 fois plus d'espace disponible.

Par exemple :

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

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

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

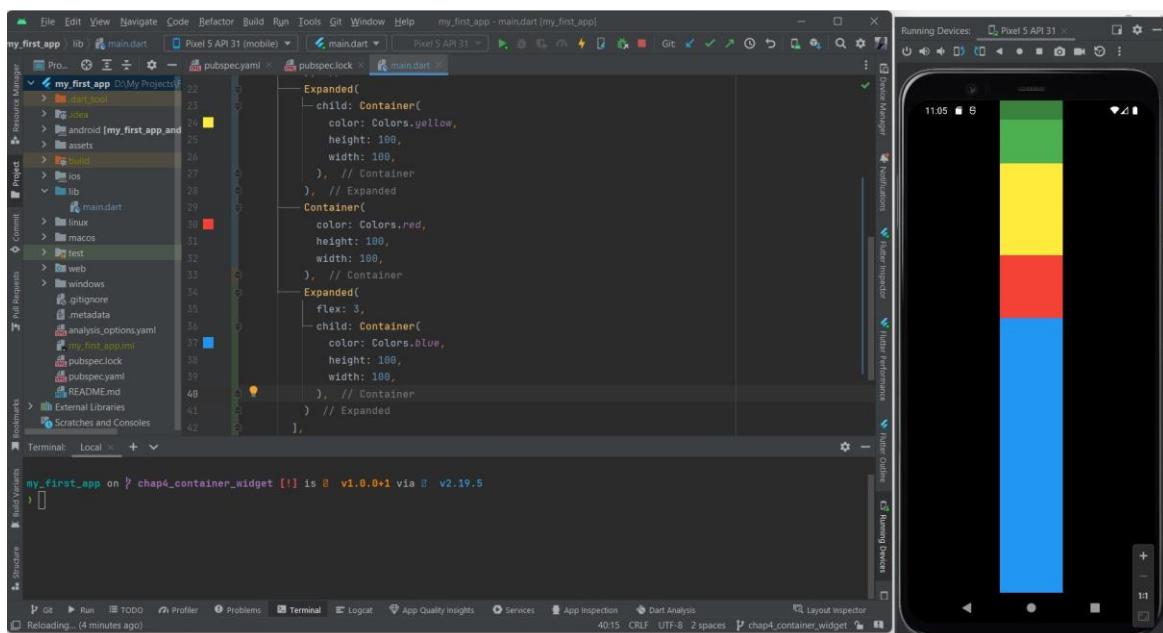
    // This widget is the root of your application.
    @override
    Widget build(BuildContext context) {
        return Column(
            crossAxisAlignment: CrossAxisAlignment.center,
            textDirection: TextDirection.ltr,
            children: [
                Container(
                    color: Colors.green,
                    height: 100,
                    width: 100,
                ),
                Expanded(
                    flex: 3,
                    child: Container(
                        color: Colors.yellow,
                        height: 100,
                        width: 100,
                    ),
                ),
                Container(
                    color: Colors.red,
                    height: 100,
                    width: 100,
                ),
            ],
        );
    }
}
```

```

        width: 100,
    ),
    Expanded(
        flex: 3,
        child: Container(
            color: Colors.blue,
            height: 100,
            width: 100,
        ),
    ),
),
],
);
}
}

```

Voici le résultat :



Le child bleu et le child jaune sont Expanded et se répartissent donc l'espace disponible.

Cependant, le child bleu a un flex de 3, et il récupère donc le triple d'espace disponible par rapport au jaune.

## Définition de la différence entre Flexible et Expanded

Les widgets `Flexible` et `Expanded` sont utilisés pour contrôler la façon dont les widgets se redimensionnent et se répartissent l'espace disponible dans une `Row` ou une `Column`. Ils sont souvent utilisés pour créer des mises en page flexibles et réactives.

Voici la différence entre les deux :

1. `Flexible` : Le widget `Flexible` est utilisé pour permettre à un enfant de s'agrandir ou de se rétrécir de manière flexible pour remplir l'espace disponible. Il est défini avec un facteur de flexibilité (`flex`) qui détermine comment l'enfant doit se comporter par rapport aux autres enfants flexibles dans le même conteneur. Le facteur de flexibilité est utilisé pour répartir l'espace restant après que tous les enfants non flexibles ont été placés. Par exemple, si un

`Flexible` a un `flex` de 2 et un autre à un `flex` de 1, le premier occupera deux fois plus

d'espace que le second. Si tous les enfants ont le même facteur de flexibilité, ils se partageront l'espace disponible de manière égale.

2. `Expanded` : Le widget `Expanded` est une version spécifique du widget `flexible` avec un facteur de flexibilité de 1. Il est utilisé pour remplir tout l'espace disponible dans le conteneur parent. Si vous avez plusieurs enfants `Expanded` dans le même conteneur, ils se partageront l'espace disponible de manière égale. Si vous voulez qu'un enfant occupe plus d'espace que les autres, vous pouvez utiliser un `flex` différent en utilisant le widget `flexible` au lieu de `Expanded`.

En résumé, le widget `flexible` permet un contrôle plus granulaire sur la flexibilité des enfants et leur répartition d'espace, tandis que le widget `Expanded` est une manière concise de spécifier qu'un enfant doit occuper tout l'espace disponible.

## ▼ Les widgets row

Le widget `Row` est très similaire au widget `Column`, la différence étant que l'**axe principal** est l'**axe horizontal** et l'**axe croisé** est l'**axe vertical**.

Dans un widget `Row`, l'axe principal est horizontal, ce qui signifie que les enfants sont disposés horizontalement de gauche à droite. L'axe croisé est vertical, ce qui signifie que les enfants sont alignés verticalement les uns par rapport aux autres.

### La propriété `mainAxisAlignment`

Cette propriété va cette fois-ci contrôler l'alignement des éléments sur l'axe horizontal.

Nous n'allons pas reprendre tous les exemples, car cela fonctionne exactement pareil que pour le widget `Column`.

Voici le résumé des valeurs possibles et de leur effet :

## Row



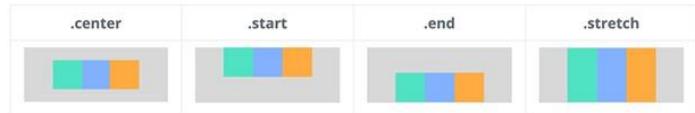
### La propriété crossAxisAlignment

Cette fois-ci la propriété `crossAxisAlignment` va gérer l'alignement sur l'axe vertical.

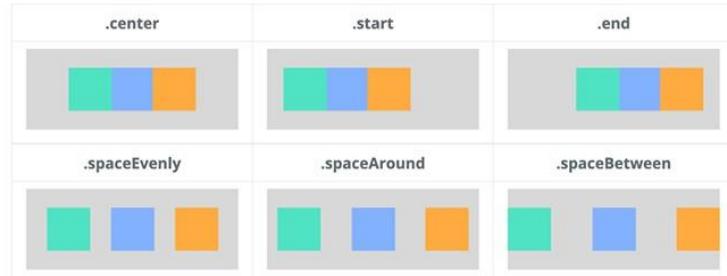
Voici le résumé des valeurs possibles et de leur effet :

# Row

## CrossAxis Alignment



## MainAxis Alignment



## Utilisation du widget Expanded avec le widget Row

La propriété mainAxisAlignment

Vous pouvez également utiliser le widget `Expanded` sur les `child` des widgets `Row`.

Prenons par exemple :

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

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

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

    // This widget is the root of your application.
    @override
    Widget build(BuildContext context) {
        return Row(
            mainAxisAlignment: MainAxisAlignment.center,
            textDirection: TextDirection.ltr,
            children: [
                Container(
                    color: Colors.green,
                    height: 80,
                    width: 80,
                ),
                Expanded(
                    child: Container(
                        color: Colors.yellow,
                        height: 80,
                        width: 80,
                    ),
                ),
                Container(
                    color: Colors.red,
                    height: 80,
                    width: 80,
                ),
                Container(

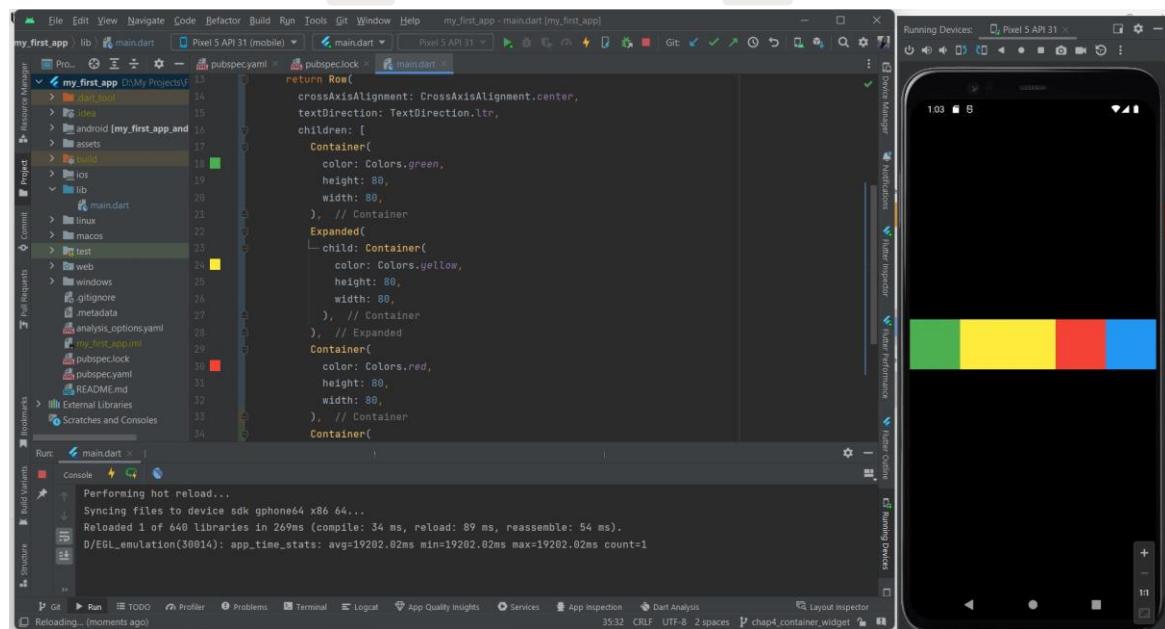
```

```

        color: Colors.blue,
        height: 80,
        width: 80,
    ),
),
);
}
}

```

Ce qui donne :



## ▼ Les widgets stack

Le widget `Stack` qui se traduit "pile" en français, permet de **superposer plusieurs éléments** `child`.

Dans un `Stack`, les éléments `child` sont empilés les uns sur les autres selon l'ordre dans lequel ils sont définis. Le premier `child` défini sera à l'arrière et les `child` suivants seront superposés par-dessus.

Par exemple, il permet de placer un texte ou un bouton sur une image.

**NB: La notion de stack ressemble au z-index en CSS**

Commençons par un premier exemple basique :

```

import 'package:flutter/material.dart';

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

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

    // This widget is the root of your application.
    @override

```

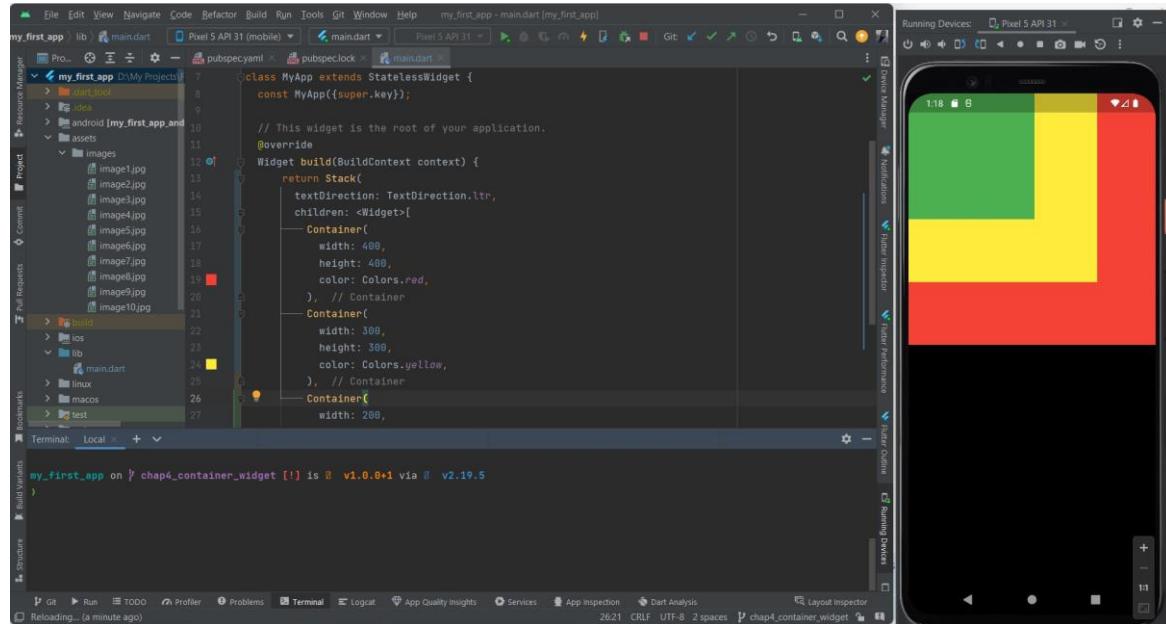
```

Widget build(BuildContext context) {
    return Stack(
        textDirection: TextDirection.ltr,
        children: <Widget>[
            Container(
                width: 400,
                height: 400,
                color: Colors.red,
            ),
            Container(
                width: 300,
                height: 300,
                color: Colors.yellow,
            ),
            Container(
                width: 200,
                height: 200,
                color: Colors.green,
            ),
        ],
    );
}
}

```

Nous sommes obligé de préciser la `textDirection` pour que Flutter sache si il faut dessiner les éléments en haut à gauche ou en haut à droite par défaut.

Le code précédent donne :



Le **carré rouge** est le **premier** dans la `List` et il est donc dessiné en **premier** (c'est donc celui le plus en dessous).

Le **carré jaune** est placé en **second** dans la `List` et il est donc au **milieu** des deux carrés.

Le **carré vert** est en **dernier** dans la `List` et c'est donc le carré le plus au dessus.

## La propriété `alignment`

Nous avons déjà vu cette propriété pour les `Container` et elle fonctionne exactement pareil, avec les mêmes enums ou alors en passant deux arguments double.

Voici un exemple d'alignement :

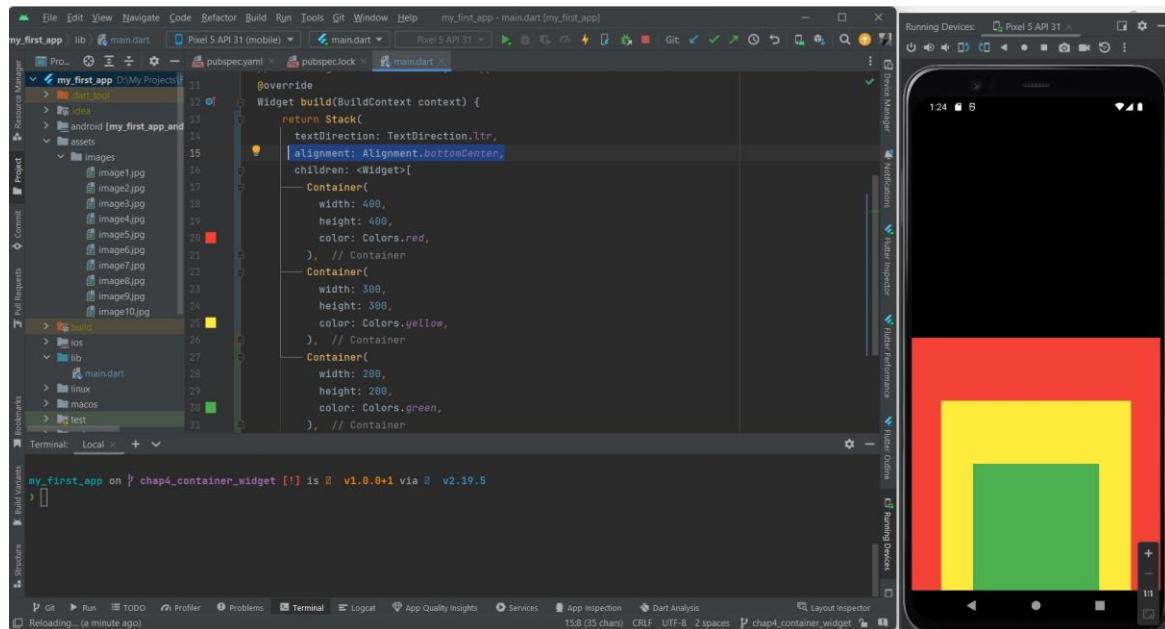
```
import 'package:flutter/material.dart';

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

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

  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return Stack(
      textDirection: TextDirection.ltr,
      alignment: Alignment.bottomCenter,
      children: <Widget>[
        Container(
          width: 400,
          height: 400,
          color: Colors.red,
        ),
        Container(
          width: 300,
          height: 300,
          color: Colors.yellow,
        ),
        Container(
          width: 200,
          height: 200,
          color: Colors.green,
        ),
      ],
    );
  }
}
```

Voici le résultat :



## La propriété Positioned

Il est important de noter que les éléments `child` dans un `stack` peuvent se chevaucher. Vous pouvez utiliser des propriétés telles que `positioned` pour contrôler la position de chaque élément

Voici un exemple :

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

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

class MyApp extends StatelessWidget {
    const MyApp({Key? key});

    @override
    Widget build(BuildContext context) {
        return Stack(
            textDirection: TextDirection.ltr,
            alignment: Alignment.bottomCenter,
            children: <Widget>[
                Positioned(
                    left: 50,
                    top: 50,
                    child: Container(
                        width: 200,
                        height: 200,
                        color: Colors.red,
                    ),
                ),
                Positioned(
                    right: 50,
                    top: 100,
                    child: Container(
                        width: 300,
                        height: 300,
                        color: Colors.yellow,
                    ),
                ),
                Positioned(

```

`child` de manière plus précise.

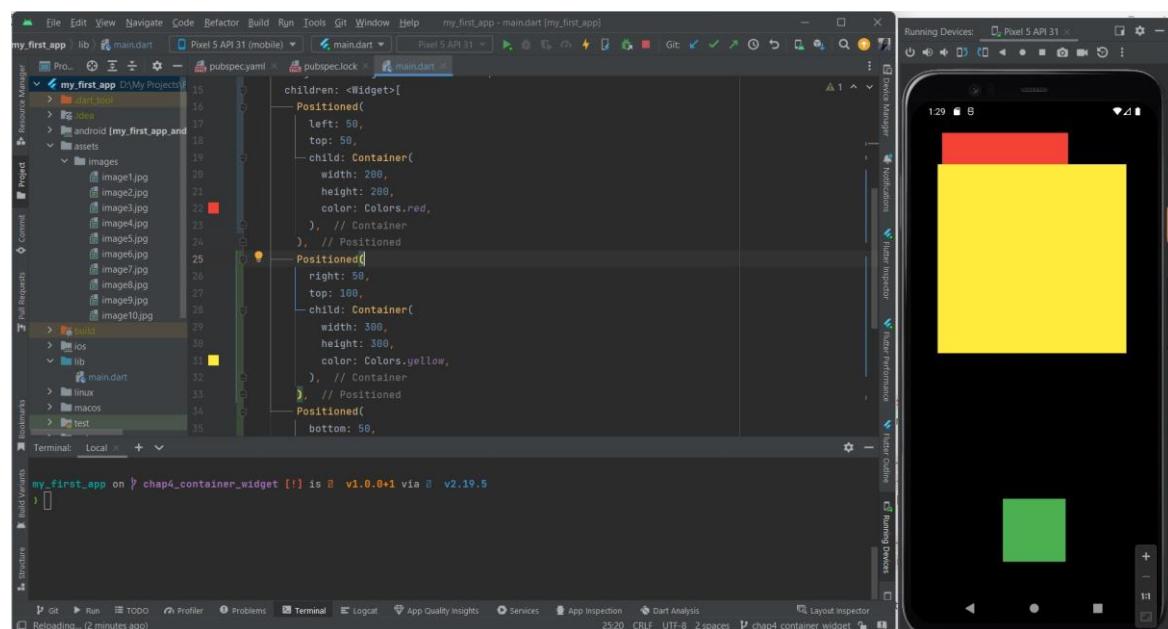
```

        bottom: 50,
        child: Container(
            width: 100,
            height: 100,
            color: Colors.green,
        ),
    ),
),
],
);
}
}

```

## Le widget

Voici le résultat:



Dans cet exemple, chaque `Container` est enveloppé dans un widget `Positioned`. Le widget `Positioned` vous permet de spécifier la position de l'élément enfant à l'intérieur du `Stack` en utilisant des propriétés telles que `left`, `top`, `right` et `bottom`. Vous pouvez ajuster ces valeurs pour positionner les éléments comme souhaité.

Le premier `Container` est positionné à `(left: 50, top: 50)`, le deuxième `Container` est positionné à `(right: 50, top: 100)`, et le troisième `Container` est positionné à `(bottom: 50)`. Cela crée une disposition où les conteneurs sont placés à des positions spécifiques à l'intérieur du `Stack`.

Vous pouvez expérimenter avec différentes valeurs pour les propriétés `left`, `top`, `right` et `bottom` afin d'obtenir le positionnement souhaité des éléments enfants dans le `Stack`.

`Stack` est très utile pour créer des mises en page complexes où vous avez besoin de superposer des éléments les uns sur les autres, comme des calques dans une application graphique.