3.3 Exploring Basic Widgets

Perhaps it is time that we move on to explaining what widgets do and how to configure them properly.

3.3.1 Center

This widget creates a widget that centers its child.

Center Constructors

```
Center({
    Key key,
    double widthFactor,
    double heightFactor,
    Widget child
    })
```

As you can see the Center widget has four parameter, don't worry about the `Key` parameter for now, however the other three are useful for us in this particular way, imagine that you provide the Center widget with a child and there is no constraints of size is enforced; so the widget will take up as much as space that it can, but if you provide it with a `widthFactor` or `heightFactor`, the width or height of the widget will be effected by these parameters. For example if `widthFactor` is 2 then the width of Center widget will be twice the width of its child.

3.3.2 Text

We are already somewhat familiar with this widget, the main function of the Text is to output (display) a string of text with a single style, although that style argument is optional.

Text Constructors

```
Text(
    String data,                    // Takes a stream of data with string type
    {Key key,                       // Identifier
    TextStyle style,                // Styling the data (See the next section)
    StrutStyle strutStyle,          // Defines the min height relative to
     ↪ baseline
```

```
    TextAlign textAlign,                // Alignment of text (e.g. center)
    TextDirection textDirection,        // Direction of text (e.g. left-to-right)
    Locale locale,                      // Identifier of language and formatting
     ↪ (e.g. en)
    bool softWrap,                      // Text break at soft line breaks
    TextOverflow overflow,              // Handling of overflowing text
    double textScaleFactor,             // Text scale factor (e.g. 1.5 => 50%
     ↪ larger)
    int maxLines,                       // Max number of lines for text to span
    String semanticsLabel               // Indication of purpose for accessibility
     ↪ reasons
    })
```

-TextStyle

The Text widget uses the TextStyle class for providing configurations of styling with these arguments:

TextStyle

```
TextStyle({
bool inherit: true,                     // Inherit from parent
Color color,                            // Color of text
Color backgroundColor,                  // Background color of text
double fontSize,                        // Size of font
FontWeight fontWeight,                  // Thickness of typeface (e.g. bold)
FontStyle fontStyle,                    // Variant of typeface (e.g. italics)
double letterSpacing,                   // Amount of space between letters
double wordSpacing,                     // Amount of space between words
TextBaseline textBaseline,              // Common baseline between this text and
 ↪ its parent
double height,                          // Height of text span as a multiple of
 ↪ font size
```

```
Locale locale,                          // Used locale for region-specific
 ↪ formatting
Paint foreground,                       // Used for painting text as foreground in
 ↪ canvas
Paint background,                       //Used for painting text as background in
 ↪ canvas
List<Shadow> shadows,                   // A list of shadows painted beneath the
 ↪ text
TextDecoration decoration,              // Decoration of text (e.g. underline)
Color decorationColor,                  // Color of decoration
TextDecorationStyle decorationStyle,    // Style of decoration (e.g. dashed)
double decorationThickness,             // Thickness of decoration strokes
String debugLabel,                      // A human friendly description about this
 ↪ style
String fontFamily,                      // Family of the used font
List<String> fontFamilyFallback,        // A list of fonts to fall back on when a
 ↪ higher priority font family cannot be found.
String package                          // Name of the package from which this
 ↪ style is included
})
```

■ **Example 3.1** Here is a combination of Text widget and TextStyle:

Text&TextStyle

```
Text(
    'Hello World',
    style: TextStyle(
        color: Colors.red,
        fontSize: 20,
        fontWeight: FontWeight.bold,
        decoration: TextDecoration.underline,
```

```
        letterSpacing: 5.5),
),
```

■

3.3.3 Icon

Flutter comes with bundle of material icons which we can use easily with the help of `Icon` widget. Have in mind that icons are not interactive themselves, however we can use a widget called `IconButton` for this purpose, we will discuss this widget later on. Let's look at `Icon` properties (Remember that it is necessary to provide an IconData, all other properties are optional):

Icon Constructors

```
Icon(
IconData icon,                  // Provides a description of an icon drawed using
 ↪  a font glyph
{
Key key,                        // Key identifier of widget
double size,                    // Size of icon
Color color,                    // Color configuration of icon
String semanticLabel,           // Indication of purpose for accessibility reasons
TextDirection textDirection     // Direction which icon appears with text (e.g.
 ↪  ltr)
})
```

■ **Example 3.2** An example of an alarm icon with blue color and size of 25:

Icon

```
Icon(
    Icons.alarm,
    color: Colors.blue,
    size: 25)
```

■

3.3.4 Container

Flutter provides us with a number of wonderful widgets to help out easily layout the applications with an extended amount of freedom and abilities. One of the most primarily layout widget is called `Container`. This widget has a suite of common configurations for structuring the layout like painting, sizing, padding and positioning.

Container like so many other things in Flutter is based on its own etymology, hence the Container widget contains a child of type widget, surrounds it with a padding then applies additional constraints like width or height.

We can pass certain arguments to Container like alignment, padding, color, transformation and more. Let's take a look at all of the properties of Container widget:

Container Constructors

```
Container({
Key key,                          // Identifier of the widget
AlignmentGeometry alignment,      // Alignment of the widget (e.g.
 ↪ AlignmentDirectional.center)
EdgeInsetsGeometry padding,       // Ass padding (e.g. EdgeInsets.all(10) )
Color color,                      // Color of the container
Decoration decoration,            // Provides decoration using other methods
 ↪ like BoxDecoration()
Decoration foregroundDecoration,  // Provides foreground decoration
double width,                     // Width of the container
double height,                    // Height of the container
BoxConstraints constraints,       // Provides min and max constraints of width
 ↪ and height
EdgeInsetsGeometry margin,        // Adds margin (e.g. EdgeInsets.all(10) )
Matrix4 transform,                // Transforms the container (e.g. rotation)
Widget child                      // Child of the container
})
```

Please be aware that the `color` argument is a shorthand for `decoration:BoxDecoration(color: color)` so you cannot use `color` and `decoration` together. Instead you must use it one at a time or only within the `decoration` property.

■ **Example 3.3** Let us now construct a container with these properties:

1. A red container in center of the screen with width of 50.
2. A centered red container with width and height of 200, padding of 10 which shows a string of "Hello" with a font size of 50.
3. In the above condition, center the "Hello" string.

Here is the boilerplate code of this app without the Container specifications:

Boilerplate #MyApp

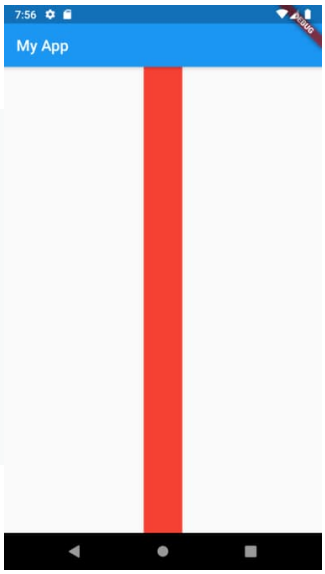
```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('My App'),
        ),
        body: Container(

        ),
      ),
    );
  }
}
```

You should put given containers inside the body part of code.

Container #1:

Container #1

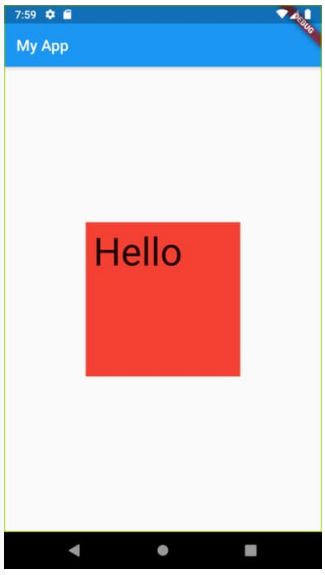
```
Center(
  child: Container(
    child: null,
    color: Colors.red,
    width: 50,
  ),
),
```



Container #2:

Container #2

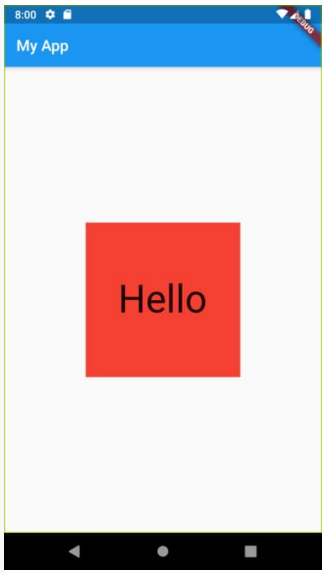
```
Center(
  child: Container(
    child: Text(
      'Hello',
      style: TextStyle(fontSize: 50),
    ),
    color: Colors.red,
    width: 200,
    height: 200,
    padding: EdgeInsets.all(10),
  ),
),
```



Container #3:

Container #3

```
Center(
  child: Container(
    child: Text(
      'Hello',
      style: TextStyle(fontSize: 50),
    ),
    color: Colors.red,
    width: 200,
    height: 200,
    padding: EdgeInsets.all(10),
    alignment: AlignmentDirectional.center,
  ),
),
```



Notice that by adding an alignment we can control the alignment property of the child (In this case a text). ∎

3.3.5 Column

Now that we are acquainted with the Container widget and its much useful properties, we must move on to another layout component of Flutter, which is called Column widget, to simply put this widget will represent its children in a vertical array. It is important to know that the Column widget does not support scrolling, subsequently if you put more children in a Column that it can fit, it is considered as an error.

There are other widgets that provides us with scrolling options which we will discuss later. Here is the basic syntax of Column widget at its simplest form: `Column(children: <Widget>[])`

As you can see on the second line the Column widget will accept a list of type widgets as its children and then puts them in a vertical array with a first come first served attitude. Let us explain the properties of Column widget and then we will wrap it up with some examples.

Column Constructors

```
Column({
Key key,
// Widget Identifier
MainAxisAlignment mainAxisAlignment: MainAxisAlignment.start,
// The vertical axis alignment (e.g. center, end)
MainAxisSize mainAxisSize: MainAxisSize.max,
// The vertical axis size (max and min)
CrossAxisAlignment crossAxisAlignment: CrossAxisAlignment.center,
// The horizontal axis alignment (e.g. baseline, stretch)
TextDirection textDirection,
// Direction of Text
VerticalDirection verticalDirection: VerticalDirection.down,
// Vertical direction of the array (up and down)
TextBaseline textBaseline,
// A horizontal line used for aligning text
List<Widget> children: const []
// List of widgets as Column children
})
```

■ **Example 3.4** In this example we will construct a basic column with three widgets inside,we will use the same boilerplate code for wrapping column inside of it. (Note that the Column widget will be inside the Center widget).

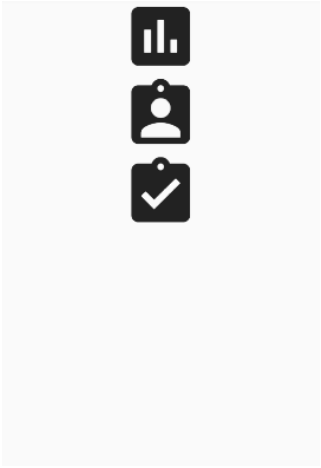
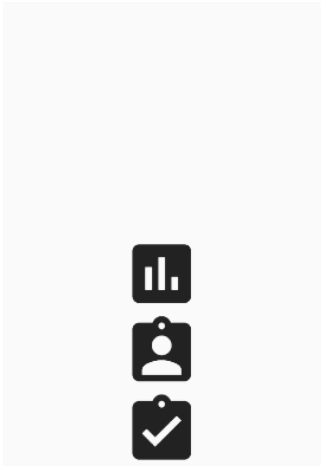
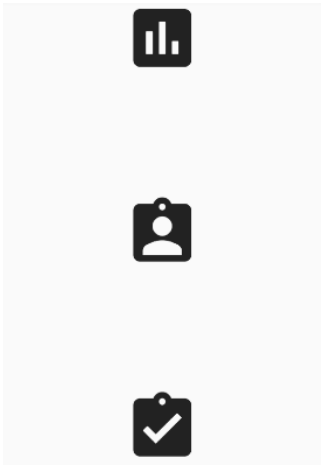
ColumnExample

```
Column(
    children: <Widget>[
      Icon(Icons.assessment, size: 100),
      Icon(Icons.assignment_ind, size: 100),
      Icon(Icons.assignment_turned_in, size: 100)],)
```

■

For better understanding of Column widget properties let us construct a table of some of the most important configurations:

```
mainAxisAlignment:
```

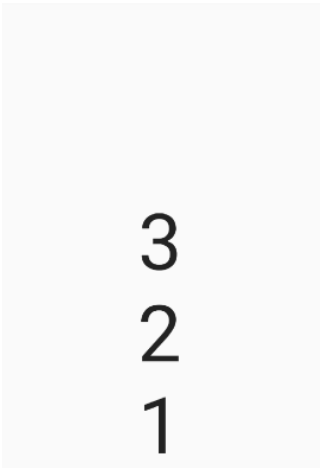
MainAxisAlignment.start	MainAxisAlignment.center	MainAxisAlignment.end
		
MainAxisAlignment.spaceArou	MainAxisAlignment.spaceEven	MainAxisAlignment.spaceBetween
		

Syntax

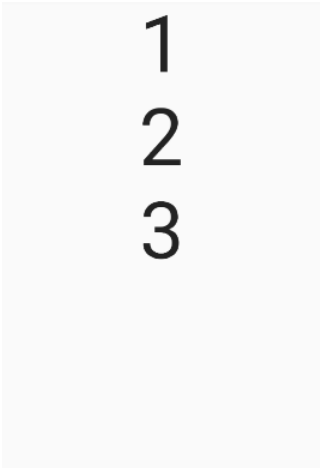
```
Column(
   mainAxisAlignment: MainAxisAlignment.spaceAround, //.end    .center    .start
    ↪  .spaceEvenly    .spaceBetween
   children: <Widget>[
     Icon(Icons.assessment, size: 100),
     Icon(Icons.assignment_ind, size: 100),
     Icon(Icons.assignment_turned_in, size: 100)
   ],
 )
```

```
verticalDirection:
```

`VerticalDirection.up`



`VerticalDirection.down`



3.3.6 Row

The Row widget is in a way just a rotated form of Column widget which takes a list of children and then displays them on a horizontal array. The syntax is very similar to Column widget. Hence, if you

are interested go and check Column widget properties. Let us show an example of Row widget:

■ **Example 3.5** A row of three digits (1, 2, 3)

RowExample

```
Row(
    mainAxisAlignment:
     ↪ MainAxisAlignment.spaceAround,
    children: <Widget>[
      Text("1", style: TextStyle(fontSize:
       ↪ 100),),
      Text("2", style: TextStyle(fontSize:
       ↪ 100),),
      Text("3", style: TextStyle(fontSize:
       ↪ 100),)
    ],
  )
```

3.3.7 Image

Working with images in Flutter can be a little inflexible at times, however Flutter provides many ways to include and display images inside the application:

- `Image`
- `Image.asset`
- `Image.network`
- `Image.file`
- `Image.memory`

In this section we will look at two of the most convenient and useful ways of adding images in Flutter; the `Image.network` and `Image.asset`.

–Image.network

By using the `Image.network` widget we can directly summon images from the internet and then display them on our applications. Here is the properties of this widget:

Image.network Constructors

```
Image.network(
String src,
// The source (Commonly a URL) of image in string form
{
Key key,
// Widget Identifier
double scale: 1.0,
// Scale of the image (By default 1.0 for original scale)
String semanticLabel,
// Semantic label for the widget
bool excludeFromSemantics: false,
// Whether to exclude gestures of this widget from semantic tree
double width,
// The width of image
double height,
// The height of image
Color color,
// Color of the image
BlendMode colorBlendMode,
// Blending color options for image
BoxFit fit,
// Adjusting fitting of image
AlignmentGeometry alignment: Alignment.center,
// Alignment of image
ImageRepeat repeat: ImageRepeat.noRepeat,
// Whether to repeat image or not
Rect centerSlice,
```

```
// Slicing the image
bool matchTextDirection: false,
// Whether to paint image in direction of text
bool gaplessPlayback: false,
// Whether to continue showing the old image (true), or briefly show nothing
 ↪  (false), when the image provider changes.
FilterQuality filterQuality: FilterQuality.low,
// Quality levels of image filters
Map<String, String> headers
// Custom headers for retrieving the image if needed
})
```

■ **Example 3.6** Getting image from the internet directly

Image.network

```
Image.network(
  "https://www.google.com/logo.png",
)
```

■

■ **Example 3.7** Changing the color of that image

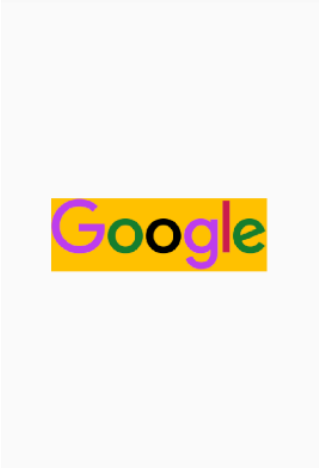
Image.network

```
Image.network(
  "https://www.google.com/logo.png",
  color: Colors.amber
)
```

■ **Example 3.8** Blending with the color of image

Image.network

```
Image.network(
  "https://www.google.com/logo.png",
  color: Colors.amber,
  colorBlendMode: BlendMode.difference,
)
```



■ **Example 3.9** Image.network custom headers usage:

Image.network

```
Image.network(
  'https://someremotserver.com/picture1',
  headers: {
    'Authentication': 'My Auth 001',
  },
)
```