Basic Flutter layout concepts

Contents

Welcome to the Flutter layout codelab, where you learn how to build a Flutter UI without downloading and installing Flutter or Dart!

> **Important:** This codelab covers basic Flutter layout concepts using an experimental code editor called DartPad. DartPad hasn't been fully tested on all browsers. If you experience any difficulties while using DartPad on a specific browser, please create a DartPad issue and specify which browser you're using in the issue title.

Flutter is different from other frameworks because its UI is built in code, not (for example) in an XML file or similar. Widgets are the basic building blocks of a Flutter UI. As you progress through this codelab, you'll learn that almost everything in Flutter is a widget. A widget is an immutable object that describes a specific part of a UI. You'll also learn that Flutter widgets are composable,

meaning, that you can combine existing widgets to make more sophisticated widgets. At the end of this codelab, you'll get to apply what you've learned into building a Flutter UI that displays a business card.

Estimated time to complete this codelab: 45-60 minutes.

Row and Column classes

`Row` and `Column` are classes that contain and lay out widgets. Widgets inside of a `Row` or `Column` are called *children*, and `Row` and `Column` are referred to as *parents*. `Row` lays out its widgets horizontally, and `Column`lays out its widgets vertically.

Example: Creating a Column

The following example displays the differences between a `Row` and `Column`.

1. Click the **Run** button.

2. In the code, change the `Row` to a `Column`, and run again.

Axis size and alignment

So far, the `BlueBox` widgets have been squished together (either to the left or at the top of the UI Output). You can change how the `BlueBox` widgets are spaced out using the axis size and alignment properties.

mainAxisSize property

`Row` and `Column` occupy different main axes. A `Row`'s main axis is horizontal, and a `Column`'s main axis is vertical. The `mainAxisSize` property determines how much space a `Row` and `Column` can occupy on their main axes.
The `mainAxisSize` property has two possible values:

`MainAxisSize.max`
`Row` and `Column` occupy all of the space on their main axes. If the combined width of their children is less than the total space on their main axes, their children are laid out with extra space.

`MainAxisSize.min`
`Row` and `Column` only occupy enough space on their main axes for their children. Their children are laid out without extra space and at the middle of their main axes.

> **Tip:** `MainAxisSize.max` is the `mainAxisSize` property's default value. If you don't specify another value, the default value is used, as shown in the previous example.

Example: Modifying axis size

The following example explicitly sets `mainAxisSize` to its default value, `MainAxisSize.max`.

1. Click the **Run** button.

2. Change `MainAxisSize.max` to `MainAxisSize.min`, and run again.

mainAxisAlignment property

When `mainAxisSize` is set to `MainAxisSize.max`, `Row` and `Column` might lay out their children with extra space. The `mainAxisAlignment` property determines how `Row` and `Column` can position their children in that extra space. `mainAxisAlignment` has six possible values:

`MainAxisAlignment.start`

Positions children near the beginning of the main axis. (Left for `Row`, top for `Column`)

`MainAxisAlignment.end`
Positions children near the end of the main axis. (Right for `Row`, bottom for `Column`)

`MainAxisAlignment.center`
Positions children at the middle of the main axis.

`MainAxisAlignment.spaceBetween`
Divides the extra space evenly between children.

`MainAxisAlignment.spaceEvenly`
Divides the extra space evenly between children and before and after the children.

`MainAxisAlignment.spaceAround`
Similar to `MainAxisAlignment.spaceEvenly`, but reduces half of the space before the first child and after the last child to half of the width between the children.

Example: Modifying main axis alignment

The following example explicitly sets `mainAxisAlignment` to its default value,`MainAxisAlignment.start`.

1. Click the **Run** button.

2. Change `MainAxisAlignment.start` to `MainAxisAlignment.end`, and run again.

Tip: Before moving to the next section, change `MainAxisAlignment.end` to another value.

crossAxisAlignment property

The `crossAxisAlignment` property determines how `Row` and `Column` can position their children on their cross axes. A `Row`'s cross axis is vertical, and a `Column`'s cross axis is horizontal. The `crossAxisAlignment`property has five possible values:

`CrossAxisAlignment.start`
Positions children near the start of the cross axis. (Top for `Row`, Left for `Column`)

`CrossAxisAlignment.end`
Positions children near the end of the cross axis. (Bottom for `Row`, Right for `Column`)

`CrossAxisAlignment.center`
Positions children at the middle of the cross axis. (Middle for `Row`, Center for `Column`)

`CrossAxisAlignment.stretch`
Stretches children across the cross axis. (Top-to-bottom for `Row`, left-to-right for `Column`)

`CrossAxisAlignment.baseline`
Aligns children by their character baselines. (`Text` class only, and requires that the `textBaseline` property is set to `TextBaseline.alphabetic`. See the Text widget section for an example.)

Example: Modifying cross axis alignment

The following example explicitly sets `crossAxisAlignment` to its default value, `CrossAxisAlignment.center`.

To demonstrate cross axis alignment, `mainAxisAlignment` is set to`MainAxisAlignment.spaceAround`, and `Row` now contains a `BiggerBlueBox` widget that is taller than the `BlueBox` widgets.

1. Click the **Run** button.

2. Change `CrossAxisAlignment.center` to `CrossAxisAlignment.start`, and run again.

Tip: Before moving to the next section, change `CrossAxisAlignment.start` to another value.

Flexible widget

As you've seen, the `mainAxisAlignment` and `crossAxisAlignment` properties determine how `Row` and `Column` position widgets along both axes. `Row` and `Column` first lay out widgets of a fixed size. Fixed size widgets are considered *inflexible* because they can't resize themselves after they've been laid out.

The `Flexible` widget wraps a widget, so the widget becomes resizable. When the `Flexible` widget wraps a widget, the widget becomes the `Flexible` widget's child and is considered *flexible*. After inflexible widgets are laid out, the widgets are resized according to their `flex` and `fit` properties.:

`flex`
Compares itself against other `flex` properties before determining what fraction of the total remaining space each `Flexible` widget receives.

`fit`
Determines whether a `Flexible` widget fills all of its extra space.

Example: Changing fit properties

The following example demonstrates the `fit` property, which can have one of two values:

`FlexFit.loose`
The widget's preferred size is used. (Default)

`FlexFit.tight`
Forces the widget to fill all of its extra space.

In this example, change the `fit` properties to make the `Flexible` widgets fill the extra space.

1. Click the **Run** button.

2. Change both `fit` values to `FlexFit.tight`, and run again.

Example: Testing flex values

In the following example, `Row` contains one `BlueBox` widget and two `Flexible` widgets that wrap two `BlueBox` widgets. The `Flexible` widgets contain `flex` properties with `flex` values set to 1 (the default value).

When `flex` properties are compared against one another, the ratio between their `flex` values determines what fraction of the total remaining space each `Flexible` widget receives.

```
remainingSpace * (flex / totalOfAllFlexValues)
```
content_copy

In this example, the sum of the `flex` values (2), determines that both `Flexible` widgets receive half of the total remaining space. The `BlueBox` widget (or fixed-size widget) remains the same size.

Tip: Before moving to the next example, try changing the `flex` properties to other values, such as 2 and 1.

Expanded widget

Similar to `Flexible`, the `Expanded` widget can wrap a widget and force the widget to fill extra space.

Tip: What's the difference between Flexible and Expanded? Use `Flexible` to resize widgets in a `Row` or `Column`. That way, you can adjust a child widget's spacing while keeping its size in relation to

its parent widget. `Expanded` changes the constraints of a child widget, so it fills any empty space.

Example: Filling extra space

The following example demonstrates how the `Expanded` widget forces its child widget to fill extra space.

1. Click the **Run** button.

2. Wrap the second `BlueBox` widget in an `Expanded` widget.

For example:

```
Expanded(child: BlueBox(),),
```

content_copy

3. Select the **Format** button to properly format the code, and run again.

SizedBox widget

The `SizedBox` widget can be used in one of two ways when creating exact dimensions. When `SizedBox`wraps a widget, it resizes the widget using the `height` and `width` properties. When it doesn't wrap a widget, it uses the `height` and `width` properties to create empty space.

Example: Resizing a widget

The following example wraps the middle `BlueBox` widget inside of a `SizedBox` widget and sets the `BlueBox`'s width to 100 logical pixels.

1. Click the **Run** button.

2. Add a `height` property equal to 100 logical pixels inside the `SizedBox` widget, and run again.

Example: Creating space

The following example contains three `BlueBox` widgets and one `SizedBox` widget that separates the first and second `BlueBox` widgets. The `SizedBox` widget contains a `width` property equal to 50 logical pixels.

1. Click the **Run** button.

2. Create more space by adding another `SizedBox` widget (25 logical pixels wide) between the second and third `BlueBox` widgets, and run again.

Spacer widget

Similar to `SizedBox`, the `Spacer` widget also can create space between widgets.

Tip: What's the difference between SizedBox and Spacer? Use `Spacer` when you want to create space using a `flex` property. Use `SizedBox` when you want to create space using a specific number of logical pixels.

Example: Creating more space

The following example separates the first two `BlueBox` widgets using a `Spacer` widget with a `flex`value of 1.

1. Click the **Run** button.

2. Add another `Spacer` widget (also with a `flex` value of 1) between the second and third `BlueBox`widgets.

Text widget

The `Text` widget displays text and can be configured for different fonts, sizes, and colors.

Example: Aligning text

The following example displays “Hey!” three times, but at different font sizes and in different colors.`Row` specifies the `crossAxisAlignment` and `textBaseline` properties.

1. Click the **Run** button.

2. Change `CrossAxisAlignment.center` to `CrossAxisAlignment.baseline`, and run again.

Icon widget

The `Icon` widget displays a graphical symbol that represents an aspect of the UI. Flutter is preloaded with icon packages for Material and Cupertino applications.

Example: Creating an Icon

The following example displays the widget `Icons.widget` from the Material Icon library in red and blue.

1. Click the **Run** button.

2. Add another `Icon` from the Material Icon library with a size of 50.

3. Give the `Icon` a color of `Colors.amber` from the Material Color palette, and run again.

Image widget

The `Image` widget displays an image. You either can reference images using a URL, or you can include images inside your app package. Since DartPad can't package an image, the following example uses an image from the network.

Example: Displaying an image

The following example displays an image that's stored remotely on GitHub. The `Image.network`method takes a string parameter that contains an image's URL.

In this example, `Image.network` contains a short URL.

1. Click the **Run** button.

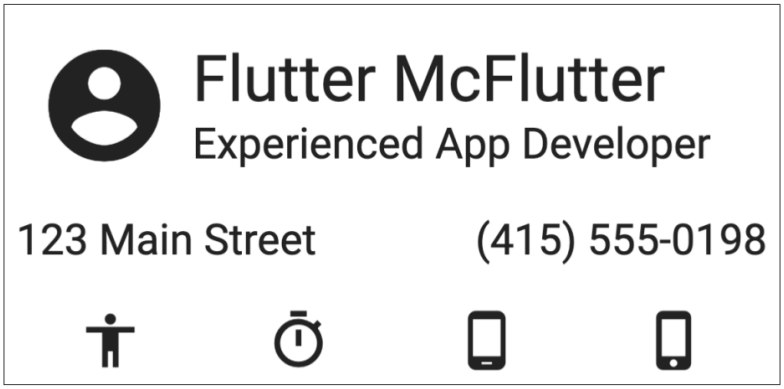
2. Change the short URL to the actual URL:

```
https://github.com/flutter/website/blob/master/examples/layout/sizing/images/pic3.jpg?raw=true
```

3. Then change `pic3.jpg` to `pic1.jpg` or `pic2.jpg`, and run again.

Putting it all together

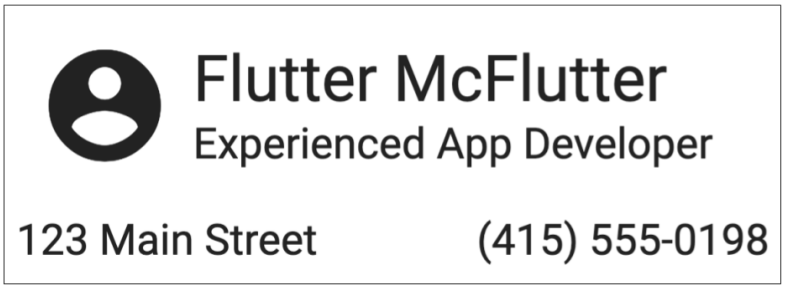
You're almost at the end of this codelab. If you'd like to test your knowledge of the techniques that you've learned, why not apply those skills into building a Flutter UI that displays a business card!



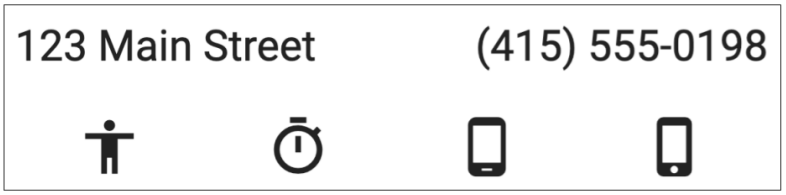
You'll break down Flutter's layout into parts, which is how you'd create a Flutter UI in the real world.

In Part 1, you'll implement a `Column` that contains the name and title. Then you'll wrap the `Column` in a `Row`that contains the icon, which is positioned to the left of the name and title.

In Part 2, you'll wrap the `Row` in a `Column`, so the code contains a `Column` within a `Row` within a `Column`. Then you'll tweak the outermost `Column`'s layout, so it looks nice. Finally, you'll add the contact information to the outermost `Column`'s list of children, so it's displayed below the name, title, and icon.



In Part 3, you'll finish building the business card display by adding four more icons, which are positioned below the contact information.



Part 1

Exercise: Create the name and title

Implement a `Column` that contains two text widgets:

- The first `Text` widget has the name `Flutter McFlutter` and the `style` property set to `Theme.of(context).textTheme.headline`.
- The second `Text` widget contains the title `Experienced Developer`.

For the `Column`,
set `mainAxisSize` to `MainAxisSize.min` and `crossAxisAlignment` to `CrossAxisAlignment.start`.

Exercise: Wrap the Column in a Row

Wrap the `Column` you implemented in a `Row` that contains the following widgets:

- An `Icon` widget set to `Icons.account_circle` and with a size of 50 pixels.
- A `Padding` widget that creates a space of 8 pixels around the `Icon` widget.

  To do this, you can specify `const EdgeInsets.all(8.0)` for the `padding` property.

  The `Row` should look like this:

```
      Row(
        children: [
         Padding(
           padding: const EdgeInsets.all(8.0),
           child: Icon(Icons.account_circle, size: 50),
         ),
         Column( ... ), // <--- The Column you first
implemented
       ],
      );
```

content_copy

Part 2

Exercise: Tweak the layout

Wrap the `Row` in a `Column` that has a `mainAxisSize` property set to `MainAxisSize.min` and a`crossAxisAlignment` property set to `CrossAxisAlignment.stretch`. The `Column` contains the following widgets:

- A `SizedBox` widget with a height of 8.
- An empty `Row` where you'll add the contact information in a later step.
- A second `SizedBox` widget with a height of 16.
- A second empty `Row` where you'll add four icons (Part 3).

The `Column`'s list of widgets should be formatted as follows, so the contact information and icons are displayed below the name and title:

```
      ],
    ), // <--- Closing parenthesis for the Row
    SizedBox(),
    Row(), // First empty Row
    SizedBox(),
    Row(), // Second empty Row
   ],
  ); // <--- Closing parenthesis for the Column that wraps
the Row
```

content_copy

Exercise: Enter contact information

Enter two `Text` widgets inside the first empty `Row` :

- The first `Text` widget contains the address `123 Main Street`.
- The second `Text` widget contains the phone number `(415) 555-0198`.

For the first empty `Row`, set the `mainAxisAlignment` property to `MainAxisAlignment.spaceBetween`.

Part 3

Exercise: Add four icons

Enter the following `Icon` widgets inside the second empty `Row`:

- `Icons.accessibility`
- `Icons.timer`
- `Icons.phone_android`
- `Icons.phone_iphone`

For the second empty `Row`, set the `mainAxisAlignment` property to`MainAxisAlignment.spaceAround`.

What’s next?

Congratulations, you’ve finished this codelab! If you’d like to know more about Flutter, here are a few suggestions for resources worth exploring:

- Learn more about layouts in Flutter by visiting the Building layouts page.
- Check out the sample apps.
- Visit Flutter’s YouTube channel, where you can watch a variety videos from videos that focus on individual widgets to videos of developers building apps.

You can download Flutter from the install page.