# Widgets and layouts: UI perspectiva

# Cookbook

This cookbook contains recipes that demonstrate how to solve common problems while writing Flutter apps. Each recipe is self-contained and can be used as a reference to help you build up an application.

## Animation

- Animate a page route transition
- Animate a widget using a physics simulation
- Animate the properties of a container
- Fade a widget in and out

## Design

- Add a drawer to a screen
- Display a snackbar
- Export fonts from a package
- Update the UI based on orientation
- Use a custom font
- Use themes to share colors and font styles
- Work with tabs

## Effects

- Create a download button
- Create a nested navigation flow
- Create a photo filter carousel
- Create a scrolling parallax effect
- Create a shimmer loading effect
- Create a staggered menu animation
- Create a typing indicator
- Create an expandable FAB
- Create gradient chat bubbles

Contents



https://docs.flutter.dev/cookbook

# Codelabs & workshops

The Flutter codelabs provide a guided, hands-on coding experience. Some codelabs run in DartPad—no downloads required!

Flutter workshops are similar to the codelabs, but are instructor led and always use DartPad. The provided workshop link takes you to the relevant YouTube video, which tells you where to find the associated DartPad link.

> You might want to check out the workshops created by our Google Developer Experts (GDEs). You can find them on the Flutter community blog.

Contents



## Good for beginners

If you're new to Flutter, we recommend starting with one of the following codelabs:

- Your first Flutter app
  Create a simple app that automatically generates cool-sounding names, such as "newstay", "lightstream", "mainbrake", or "graypine". This app is responsive and runs on mobile, desktop, and web. (This also replaces the previous "write your first Flutter app" for mobile, part 1 and part 2 codelabs.)
- Write your first Flutter app on the web
  Implement a simple web app in DartPad (no downloads required!) that displays a sign-in screen containing three text fields. As the user fills out the fields, a progress bar animates along the top of the sign-in area. This codelab is written specifically for the web, but if you have downloaded and configured Android and iOS tooling, the completed app works on Android and iOS devices, as well.

## Next steps

- Building scrolling experiences in Flutter (workshop)
  Start with an app that performs simple, straightforward scrolling and enhance it to create fancy and custom scrolling effects by using slivers.
- Dart null safety in Action (workshop)
  An instructor-led workshop based on the Null safety codelab on the dart.dev site.
- How to manage application states using inherited widgets (workshop)
  L
  s

https://docs.flutter.dev/codelabs

## Designing a Flutter UI

# widgets

# widgets

- The central idea is that you build your UI out of widgets. Widgets describe what their view should look like given their current configuration and state. When a widget's state changes, the widget rebuilds its description, which the framework diffs against the previous description in order to determine the minimal changes needed in the underlying render tree to transition from one state to the next.

https://docs.flutter.dev/development/ui/widgets-intro


## Flutter Widget of the Week

84 videos • 2,005,215 views • Last updated on 2 Jul 2020

Fighting the good fight for Widget Awareness! Widget of the Week is a series of quick, animated videos, each of which covers a particular widget from the Flutter SDK. Need to know why you'd choose to use a SafeArea widget, or how to position a child within a Stack, but don't have much time? Then, this series is for you!

Subscribe to the Flutter YouTube channel → https://bit.ly/flutterdev

Follow us on Twitter → https://twitter.com/flutterio
Check us out on the web → https://flutter.dev/

Flutter SUBSCRIBED

1 Introducing Widget of the Week! — Flutter

2 SafeArea (Flutter Widget of the — Google Developers

3 Expanded (Flutter Widget of the — Google Developers

4 Wrap (Flutter Widget of the — Google Developers

5 AnimatedContainer (Flutter Widget of — Google Developers

6 Opacity (Flutter Widget of the — Google Developers

7 FutureBuilder (Flutter Widget of — Google Developers

[Flutter Widget of the Week
https://www.youtube.com/playlist?list=PLjxrf2q8roU23XGwz3Km7sQZFTdB996iG](https://www.youtube.com/playlist?list=PLjxrf2q8roU23XGwz3Km7sQZFTdB996iG)


## Flutter in Focus

25 videos • 269,226 views • Last updated on 22 Jun 2020

Let's learn Flutter features in 10 minutes or less with our new series, Flutter in Focus! We'll not only show you the right code for a situation, but also explain the under-the-hood details that make it the right code. If you've ever been interested in learning how Dart's streams really work, or how a CustomPaint widget wires itself into Flutter's paint cycle, this is the series for you!

Subscribe to the Flutter YouTube channel → http://bit.ly/flutterdev

Follow us on Twitter → https://twitter.com/flutterio
Check us out on the web → https://flutter.dev/

1 Introducing Flutter in Focus!
Flutter

2 Building your first Flutter Widget
Google Developers

3 Using Material Design with Flutter
Google Developers

4 Introducing Flutter Widgets 101
Google Developers

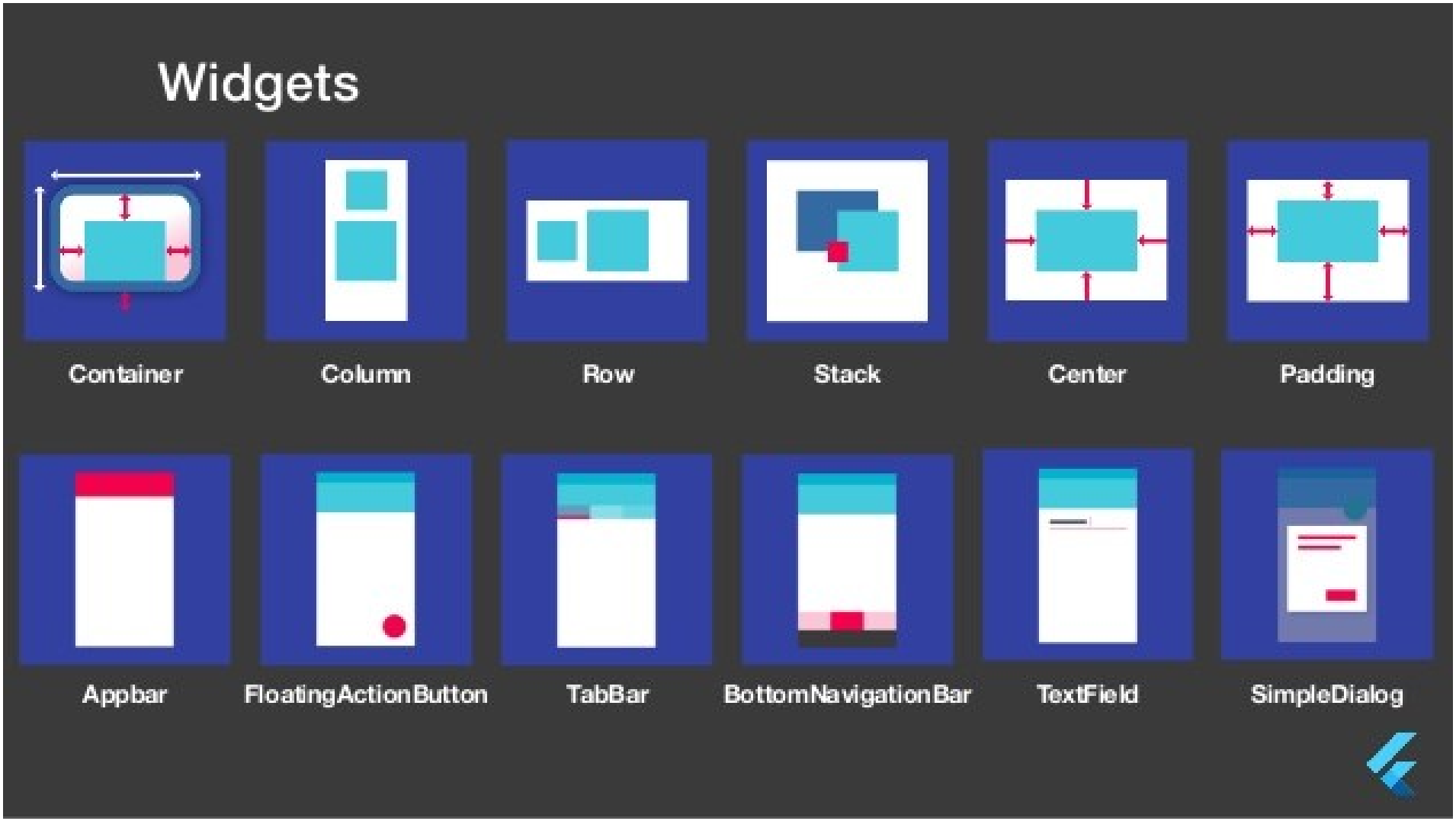
5 How to Create Stateless Widgets -
Google Developers

6 How Stateful Widgets Are Used
Google Developers

7 Inherited Widgets Explained - Flutter

8 Flutter Widgets

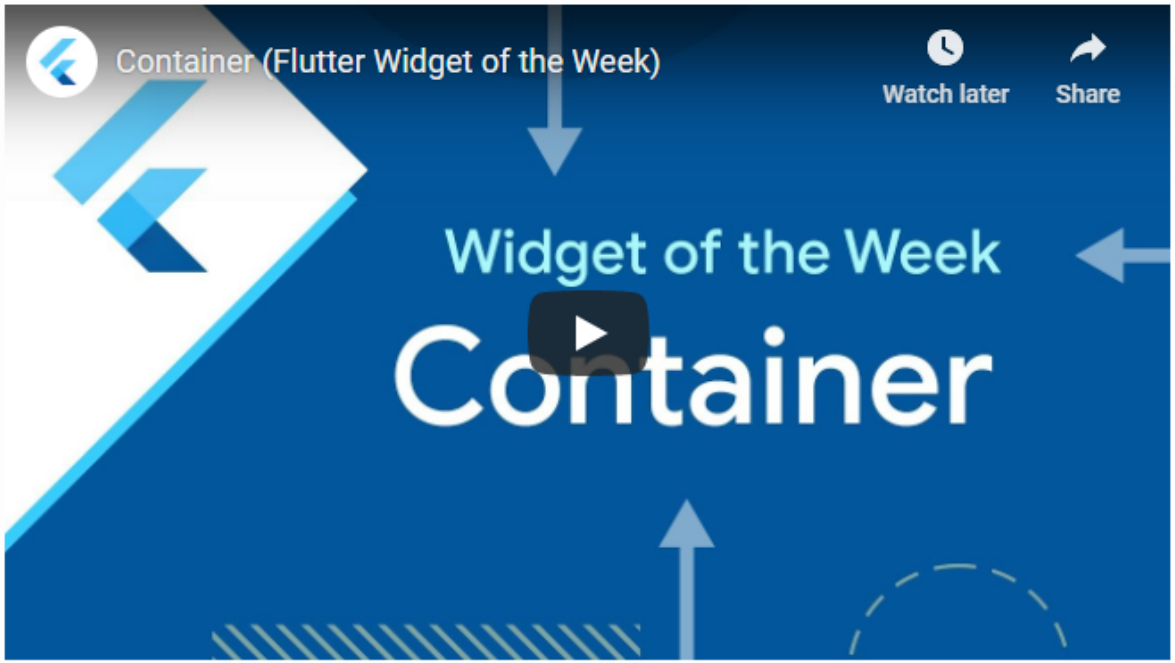
Flutter in Focus
https://www.youtube.com/playlist?list=PLjxrf2q8roU2HdJQDjJzOeO6J3FoFLWr2




https://www.slideshare.net/tdc-globalcode/tdc2018sp-trilha-mobile-flutter-do-zero-a-publicacao

# Container class

A convenience widget that combines common painting, positioning, and sizing widgets.




A container first surrounds the child with padding (inflated by any borders present in the decoration) and then applies additional constraints to the padded extent (incorporating the `width` and `height` as constraints, if either is non-null). The container is then surrounded by additional empty space described from the margin.

During painting, the container first applies the given transform, then paints the decoration to fill the padded extent, then it paints the child, and finally paints the foregroundDecoration, also filling the padded extent.

Containers with no children try to be as big as possible unless the incoming constraints are unbounded, in which case they try to be as small as possible. Containers with children size themselves to their children. The width, height, and constraints arguments to the constructor override this.

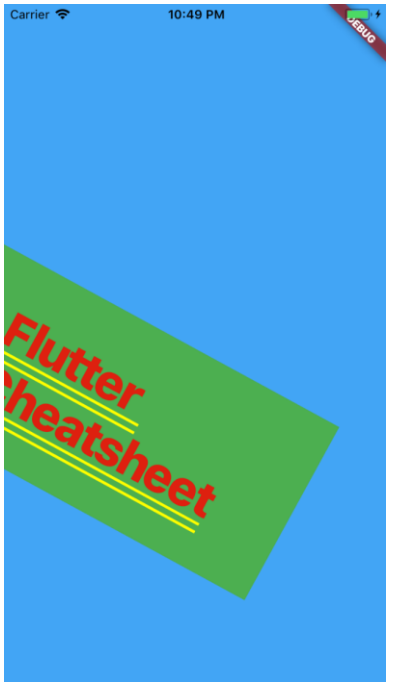
https://api.flutter.dev/flutter/widgets/Container-class.html
https://www.youtube.com/watch?v=c1xLMaTUWCY&list=RDCMUCwXdFgeE9KYzlDdR7TG9cMw&index=9

# Flutter — Container Cheat Sheet

Julien Louage Follow
May 20, 2018 · 9 min read

- Container
  - used to contain a child widget with the ability to apply some styling properties.
  - no child it will automatically fill the

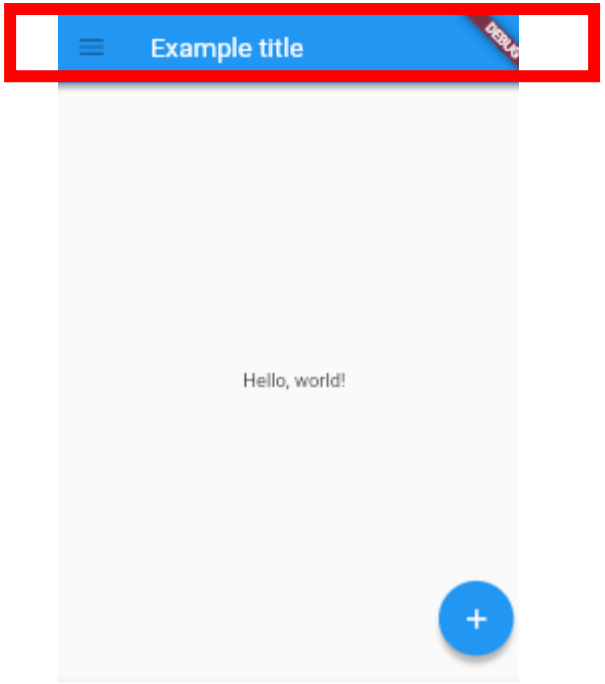
```
Center(
  child: Container(
    color: Color.fromARGB(255, 66, 165, 245),
    alignment: AlignmentDirectional(0.0, 0.0),
    child: Container(
      padding: new EdgeInsets.all(40.0),
      color: Colors.green,
      child: Text("Flutter Cheatsheet"),
      transform: new Matrix4.rotationZ(0.5)
    ),
  ),
);
```



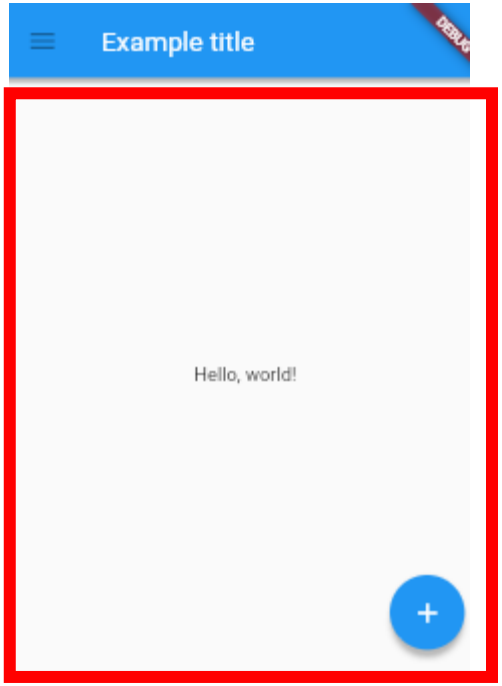

https://medium.com/jlouage/container-de5b0d3ad184

```dart
void main() {
  runApp(
    const Center(
      child: Text(
        'Hello, world!',
        textDirection: TextDirection.ltr,
      ),
    ),
  );
}
```

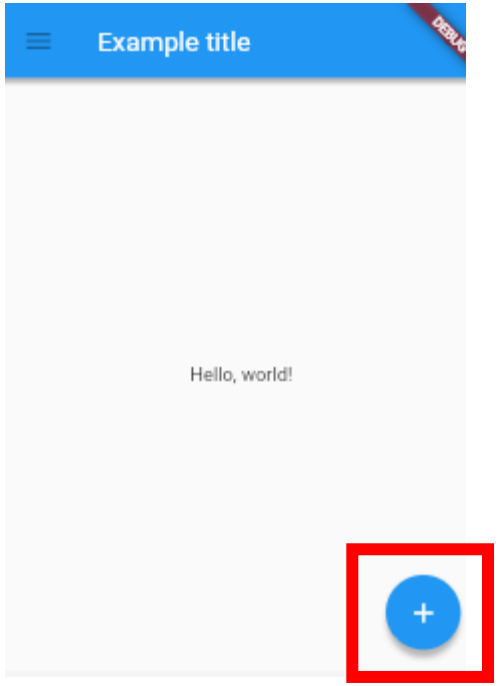
```
Widget build(BuildContext context) {
  // Scaffold is a layout for
  // the major Material Components.
  return Scaffold(
    appBar: AppBar(
      leading: const IconButton(
        icon: Icon(Icons.menu),
        tooltip: 'Navigation menu',
        onPressed: null,
      ),
      title: const Text('Example title'),
      actions: const [
        IconButton(
          icon: Icon(Icons.search),
          tooltip: 'Search',
          onPressed: null,
        ),
      ],
    ),
    // body is the majority of the screen.
    body: const Center(
      child: Text('Hello, world!'),
    ),
    floatingActionButton: const FloatingActionButton(
      tooltip: 'Add', // used by assistive technologies
      onPressed: null,
      child: Icon(Icons.add),
    ),
  );
```



```dart
Widget build(BuildContext context) {
  // Scaffold is a layout for
  // the major Material Components.
  return Scaffold(
    appBar: AppBar(
      leading: const IconButton(
        icon: Icon(Icons.menu),
        tooltip: 'Navigation menu',
        onPressed: null,
      ),
      title: const Text('Example title'),
      actions: const [
        IconButton(
          icon: Icon(Icons.search),
          tooltip: 'Search',
          onPressed: null,
        ),
      ],
    ),
    // body is the majority of the screen.
    body: const Center(
      child: Text('Hello, world!'),
    ),
    floatingActionButton: const FloatingActionButton(
      tooltip: 'Add', // used by assistive technologies
      onPressed: null,
      child: Icon(Icons.add),
    ),
  );
```



```
Widget build(BuildContext context) {
  // Scaffold is a layout for
  // the major Material Components.
  return Scaffold(
    appBar: AppBar(
      leading: const IconButton(
        icon: Icon(Icons.menu),
        tooltip: 'Navigation menu',
        onPressed: null,
      ),
      title: const Text('Example title'),
      actions: const [
        IconButton(
          icon: Icon(Icons.search),
          tooltip: 'Search',
          onPressed: null,
        ),
      ],
    ),
    // body is the majority of the screen.
    body: const Center(
      child: Text('Hello, world!'),
    ),
    floatingActionButton: const FloatingActionButton(
      tooltip: 'Add', // used by assistive technologie
      onPressed: null,
      child: Icon(Icons.add),
    ),
  );
```



# Scaffold class

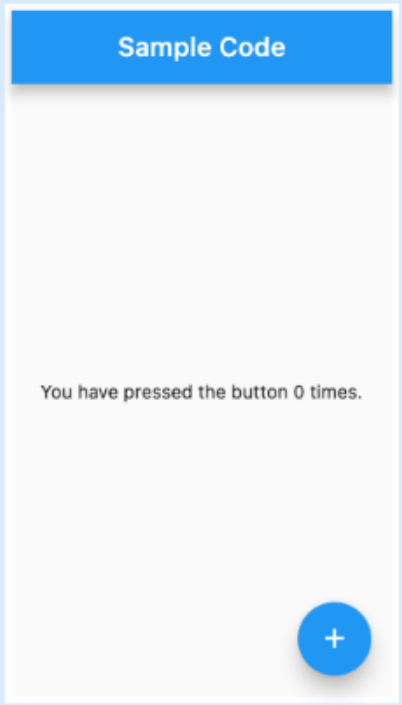
Null safety

Implements the basic Material Design visual layout structure.

This class provides APIs for showing drawers and bottom sheets.

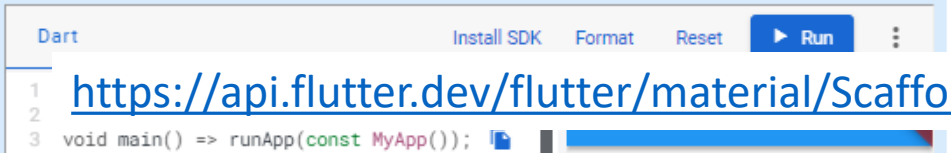
To display a persistent bottom sheet, obtain the ScaffoldState for the current BuildContext via Scaffold.of and use the ScaffoldState.showBottomSheet function.

This example shows a Scaffold with a body and FloatingActionButton. The body is a Text placed in a Center in order to center the text within the Scaffold. The FloatingActionButton is connected to a callback that increments a counter.




To create a local project with this code sample, run:

```
flutter create --sample=material.Scaffold.1 mysample
```




https://api.flutter.dev/flutter/material/Scaffold-class.html

# Scaffold class

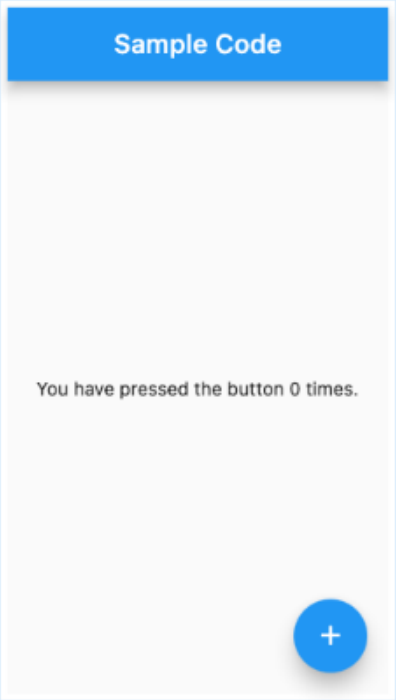
Null safety

Implements the basic Material Design visual layout structure.

This class provides APIs for showing drawers and bottom sheets.

To display a persistent bottom sheet, obtain the ScaffoldState for the current BuildContext via Scaffold.of and use the ScaffoldState.showBottomSheet function.

This example shows a Scaffold with a body and FloatingActionButton. The body is a Text placed in a Center in order to center the text within the Scaffold. The FloatingActionButton is connected to a callback that increments a counter.

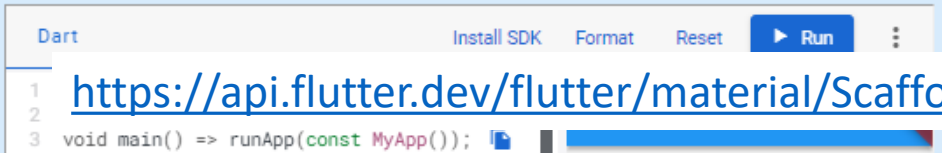



https
-clas

- AppBar, which is a horizontal bar typically shown at the top of an app using the appBar property.
- BottomAppBar, which is a horizontal bar typically shown at the bottom of an app using the bottomNavigationBar property.
- FloatingActionButton, which is a circular button typically shown in the bottom right corner of the app using the floatingActionButton property.
- Drawer, which is a vertical panel that is typically displayed to the left of the body (and often hidden on phones) using the drawer property.
- BottomNavigationBar, which is a horizontal array of buttons typically shown along the bottom of the app using the bottomNavigationBar property.
- BottomSheet, which is an overlay typically shown near the bottom of the app. A bottom sheet can either be persistent, in which case it is shown using the ScaffoldState.showBottomSheet method, or modal, in which case it is shown using the showModalBottomSheet function.
- SnackBar, which is a lightweight message with an optional action which briefly displays at the bottom of the screen. Use the ScaffoldMessengerState.showSnackBar method to show snack bars.
- MaterialBanner, which displays an important, succinct message, at the top of the screen, below the app bar. Use the ScaffoldMessengerState.showMaterialBanner method to show material banners.
- ScaffoldState, which is the state associated with this widget.
- material.io/design/layout/responsive-layout-grid.html
- Cookbook: Add a Drawer to a screen

To create a local project with this code sample, run

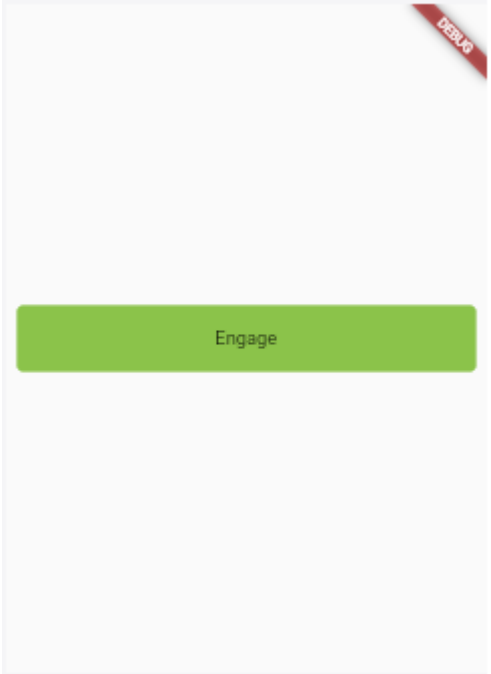
```
flutter create --sample=material.Scaffold.1 mys
```




https://api.flutter.dev/flutter/material/Scaffold-class.html

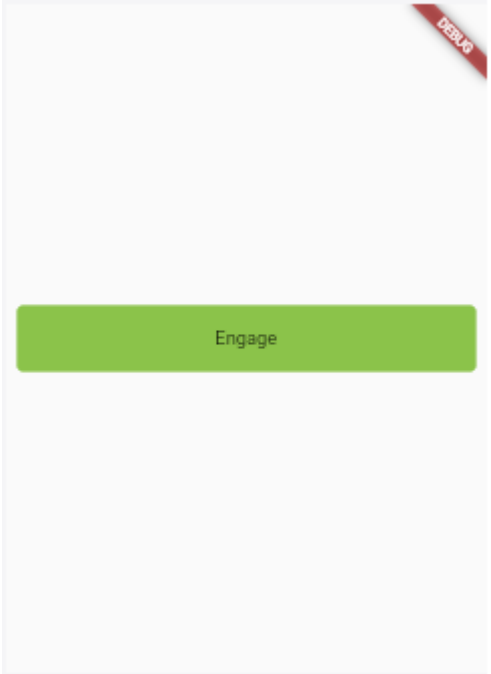
# Handling gestures

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

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

  @override
  Widget build(BuildContext context) {
    return GestureDetector(
      onTap: () {
        print('MyButton was tapped!');
      },
      child: Container(
        height: 50.0,
        padding: const EdgeInsets.all(8.0),
        margin: const EdgeInsets.symmetric(horizontal: 8.0
        decoration: BoxDecoration(
          borderRadius: BorderRadius.circular(5.0),
          color: Colors.lightGreen[500],
        ),
        child: const Center(
          child: Text('Engage'),
        ),
      ),
    );
  }
```



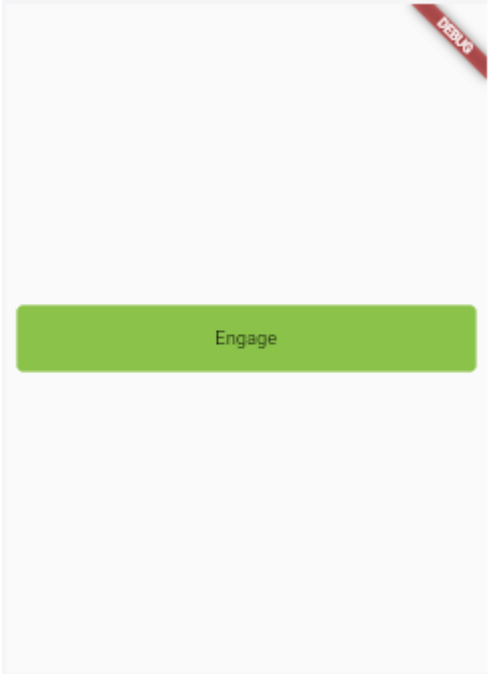
# Handling gestures

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

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

  @override
  Widget build(BuildContext context) {
    return GestureDetector(
      onTap: () {
        print('MyButton was tapped!');
      },
      child: Container(
        height: 50.0,
        padding: const EdgeInsets.all(8.0),
        margin: const EdgeInsets.symmetric(horizontal: 8.0
        decoration: BoxDecoration(
          borderRadius: BorderRadius.circular(5.0),
          color: Colors.lightGreen[500],
        ),
        child: const Center(
          child: Text('Engage'),
        ),
      ),
    );
  }
```



# Handling gestures

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

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

  @override
  Widget build(BuildContext context) {
    return GestureDetector(
      onTap: () {
        print('MyButton was tapped!');
      },
      child: Container(
        height: 50.0,
        padding: const EdgeInsets.all(8.0),
        margin: const EdgeInsets.symmetric(horizontal: 8.0
        decoration: BoxDecoration(
          borderRadius: BorderRadius.circular(5.0),
          color: Colors.lightGreen[500],
        ),
        child: const Center(
          child: Text('Engage'),
        ),
      ),
    );
  }
```



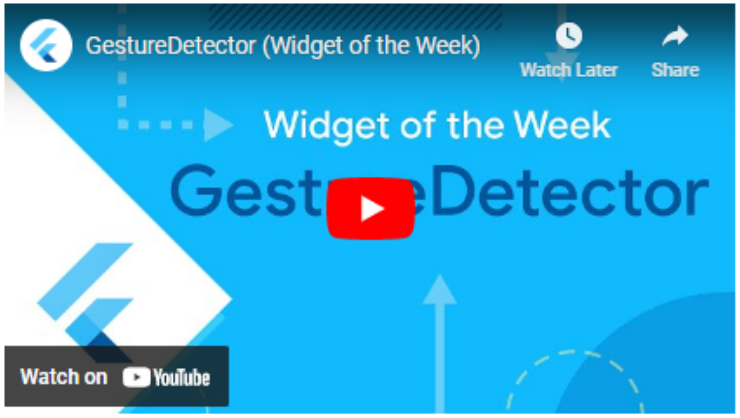
# GestureDetector class

Null safety

A widget that detects gestures.

Attempts to recognize gestures that correspond to its non-null callbacks.

If this widget has a child, it defers to that child for its sizing behavior. If it does not have a child, it grows to fit the parent instead.




By default a GestureDetector with an invisible child ignores touches; this behavior can be controlled with behavior.

GestureDetector also listens for accessibility events and maps them to the callbacks. To ignore accessibility events, set excludeFromSemantics to true.

See flutter.dev/gestures/ for additional information.

Material design applications typically react to touches with ink splash effects. The InkWell class implements this effect and can be used in place of a GestureDetector for handling taps.

This example contains a black light bulb wrapped in a GestureDetector. It turns the light bulb yellow when the "TURN LIGHT ON" button is tapped by setting the `_lights` field, and off again when "TURN LIGHT OFF" is tapped.

To create a local project with this code sample, run:

```
flutter create --sample=widgets.GestureDetector.1 mysample
```

Dart | Install SDK | Format | Reset | Run

```
1  import 'package:flutter/material.dart';
5  class MyApp extends StatelessWidget {
```

https://api.flutter.dev/flutter/widgets/GestureDetector-class.html

# GestureDetector class Null safety

A widget that detects gestures.

Attempts to recognize gestures that corresp

If this widget has a child, it defers to that ch
grows to fit the parent instead.


By default a GestureDetector with an invisib
controlled with behavior.

GestureDetector also listens for accessibilit
accessibility events, set excludeFromSemar

See flutter.dev/gestures/ for additional info

Material design applications typically react
implements this effect and can be used in p

This example contains a black light bulb
bulb yellow when the "TURN LIGHT ON"
and off again when "TURN LIGHT OFF" is

To create a local project with this code sa

```
flutter create --sample=widgets.Gestur
```

Dart

```dart
import 'package:flutter/material
void main() => runApp(const MyAp
class MyApp extends StatelessWid
```

onDoubleTap → GestureTapCallback?
The user has tapped the screen with a primary button at the same location twice in quick succession.
final

onDoubleTapCancel → GestureTapCancelCallback?
The pointer that previously triggered onDoubleTapDown will not end up causing a double tap.
final

onDoubleTapDown → GestureTapDownCallback?
A pointer that might cause a double tap has contacted the screen at a particular location.
final

onForcePressEnd → GestureForcePressEndCallback?
The pointer is no longer in contact with the screen.
final

onForcePressPeak → GestureForcePressPeakCallback?
The pointer is in contact with the screen and has pressed with the maximum force. The amount of force is at least ForcePressGestureRecognizer.peakPressure.
final

onForcePressStart → GestureForcePressStartCallback?
The pointer is in contact with the screen and has pressed with sufficient force to initiate a force press. The amount of force is at least ForcePressGestureRecognizer.startPressure.
final

onForcePressUpdate → GestureForcePressUpdateCallback?
A pointer is in contact with the screen, has previously passed the ForcePressGestureRecognizer.startPressure and is either moving on the plane of the screen, pressing the screen with varying forces or both simultaneously.
final

onHorizontalDragCancel → GestureDragCancelCallback?
The pointer that previously triggered onHorizontalDragDown did not complete.
final

onHorizontalDragDown → GestureDragDownCallback?
A pointer has contacted the screen with a primary button and might begin to move horizontally.
final

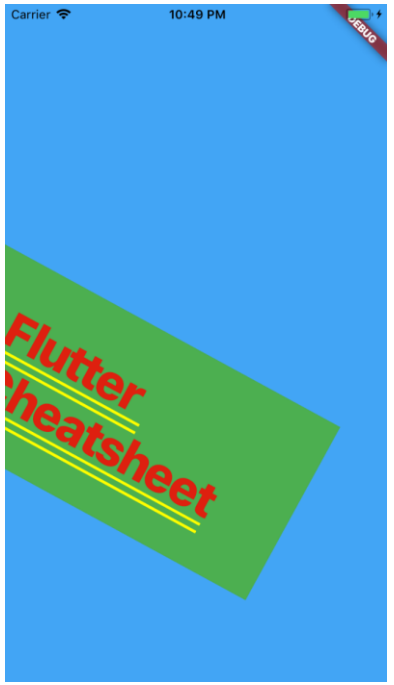
onHorizontalDragEnd → GestureDragEndCallback?
A pointer that was previously in contact with the screen with a primary button and moving horizontally is no longer in contact with the screen and was moving at a specific velocity when it stopped contacting the screen.

# Flutter — Container Cheat Sheet

Julien Louage Follow
May 20, 2018 · 9 min read

- Container
  - used to contain a child widget with the ability to apply some styling properties.
  - no child it will automatically fill the

```
Center(
  child: Container(
    color: Color.fromARGB(255, 66, 165, 245),
    alignment: AlignmentDirectional(0.0, 0.0),
    child: Container(
      padding: new EdgeInsets.all(40.0),
      color: Colors.green,
      child: Text("Flutter Cheatsheet"),
      transform: new Matrix4.rotationZ(0.5)
    ),
  ),
);
```



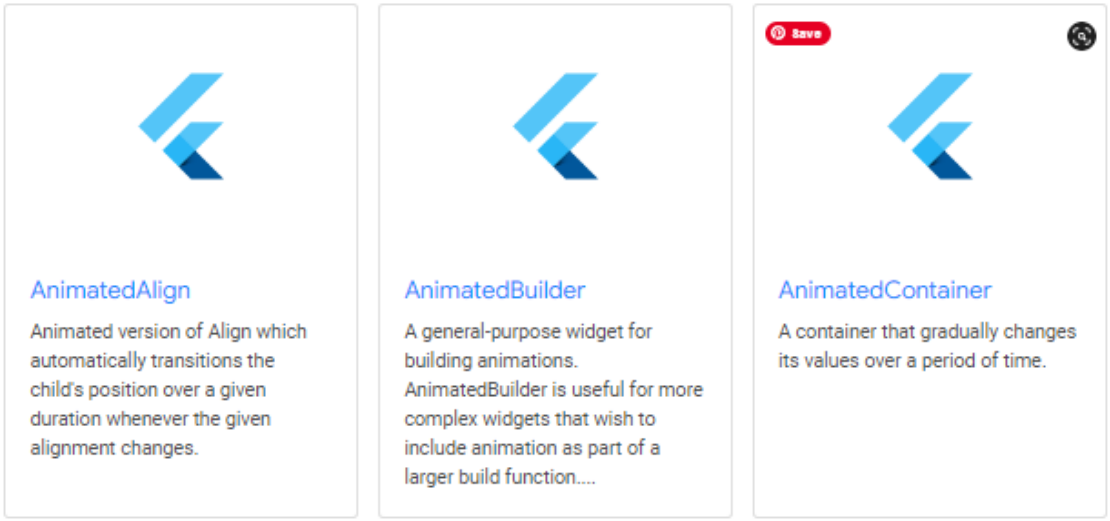

https://medium.com/jlouage/container-de5b0d3ad184

# Animation and motion widgets

UI > Widgets > Animation

Bring animations to your app.

See more widgets in the widget catalog.



### AnimatedAlign

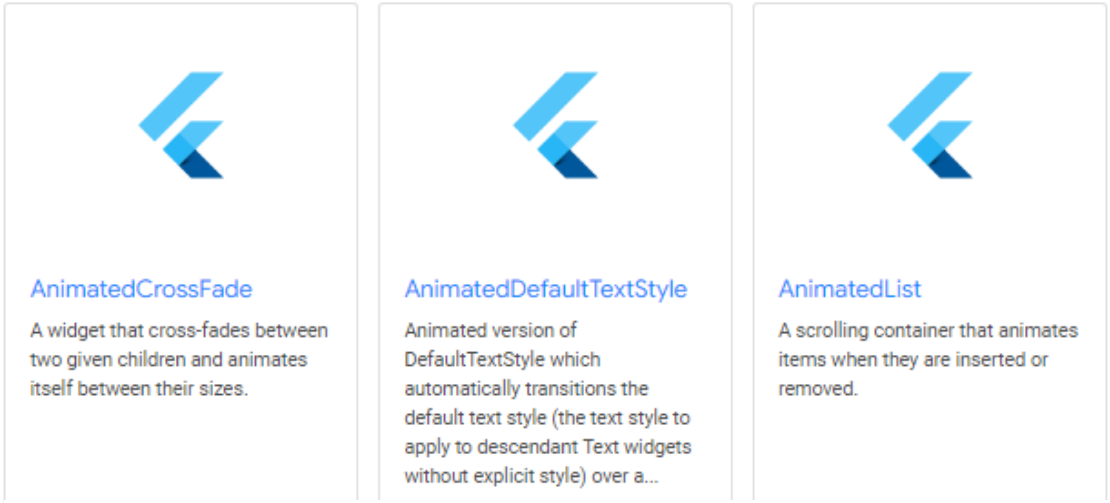
Animated version of Align which automatically transitions the child's position over a given duration whenever the given alignment changes.

### AnimatedBuilder

A general-purpose widget for building animations. AnimatedBuilder is useful for more complex widgets that wish to include animation as part of a larger build function....

### AnimatedContainer

A container that gradually changes its values over a period of time.



### AnimatedCrossFade

A widget that cross-fades between two given children and animates itself between their sizes.

### AnimatedDefaultTextStyle

Animated version of DefaultTextStyle which automatically transitions the default text style (the text style to apply to descendant Text widgets without explicit style) over a...

### AnimatedList

A scrolling container that animates items when they are inserted or removed.


https://docs.flutter.dev/development/ui/widgets/animation

# Animations tutorial

UI > Animations > Tutorial

### What you'll learn

- How to use the fundamental classes from the animation library to add animation to a widget.
- When to use `AnimatedWidget` vs. `AnimatedBuilder`.

This tutorial shows you how to build explicit animations in Flutter. After introducing some of the essential concepts, classes, and methods in the animation library, it walks you through 5 animation examples. The examples build on each other, introducing you to different aspects of the animation library.

The Flutter SDK also provides built-in explicit animations, such as `FadeTransition`, `SizeTransition`, and `SlideTransition`. These simple animations are triggered by setting a beginning and ending point. They are simpler to implement than custom explicit animations, which are described here.

Contents




# Essential animation concepts and classes

### What's the point?

- `Animation`, a core class in Flutter's animation library, interpolates the values used to guide an animation.
- An `Animation` object knows the current state of an animation (for example, whether it's started, stopped, or moving forward or in reverse), but doesn't know anything about what appears onscreen.
- An `AnimationController` manages the `Animation`.
- A `CurvedAnimation` defines progression as a non-linear curve.
- A `Tween` interpolates between the range of data as used by the object being animated. For example, a `Tween` might define an interpolation from red to blue, or from 0 to 255.
- Use `Listener`s and `StatusListener`s to monitor animation state changes.

The animation system in Flutter is based on typed `Animation` objects. Widgets can either incorporate these animations in their build functions directly by reading their current value and listening to their state changes or they can use the animations as the basis of more elaborate animations that they pass along to other widgets.

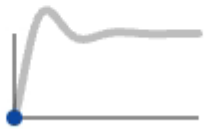
## Animation<double>

In F
und
types is `Animation<double>`.

https://docs.flutter.dev/development/ui/animations/tutorial

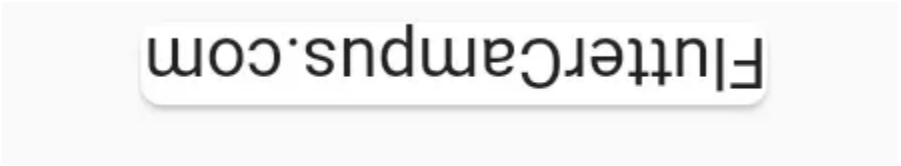

```dart
@override
Widget build(BuildContext context) {
  return Scaffold(
    body: Center(
      child: RotationTransition(
        turns: _animation,
        child: const Padding(
          padding: EdgeInsets.all(8.0),
          child: FlutterLogo(size: 150.0),
        ),
      ),
    ),
  );
}
```

RotationTransition




```
Transform.rotate(
  angle: -math.pi / 4,
  child: Card(
      child: Text("FlutterCampus.com")
  ),
)
```


```
RotatedBox(
  quarterTurns: -2,
  child: Card(
      child: Text("FlutterCampus.com")
  )
)
```




https://www.fluttercampus.com/guide/204/how-to-rotate-widget-in-flutter/



# How to rotate a widget in Flutter with examples

Various / 5 February 2023

In Flutter, there are several options available to rotate an image, including:

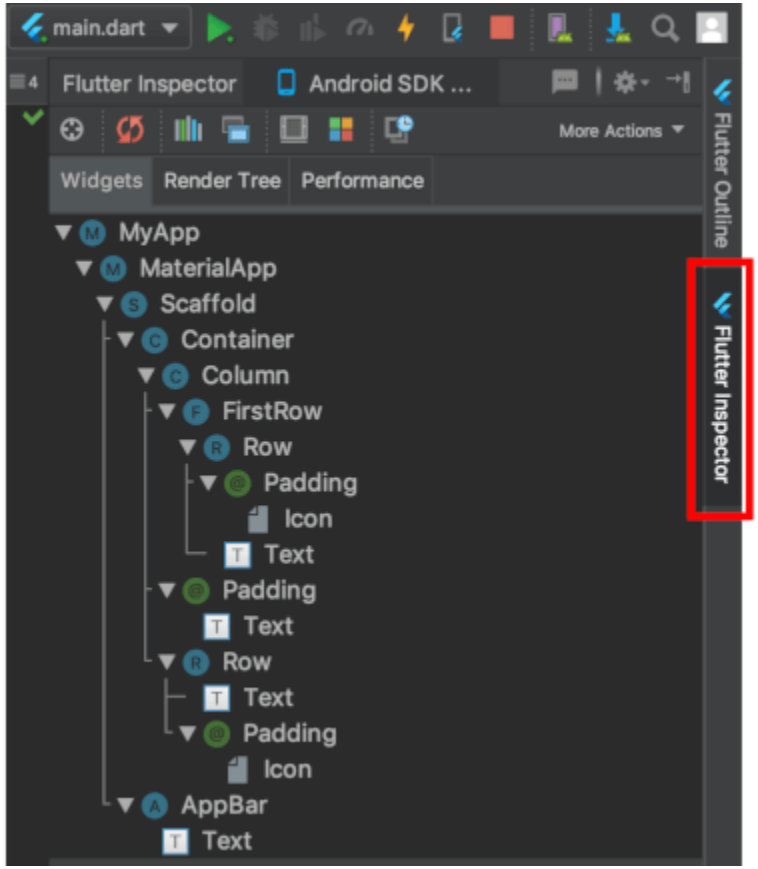
1. `Transform` Widget: The Transform widget is used to apply a transformation to its child widget. You can use this widget to rotate an image by setting the rotation property in a Matrix4 object.
2. `RotatedBox` Widget: The RotatedBox widget rotates its child by a given number of quarter turns. This widget is useful if you need to rotate an image by a specific number of turns.
3. `TweenAnimationBuilder` Widget: The TweenAnimationBuilder widget that allows you to perform animations by specifying the start and end values.
4. `AnimatedBuilder` Widget: The AnimatedBuilder widget can be used to animate the rotation of an image. You can use this widget to rotate an

https://flutterassets.com/rotate-a-widget-in-flutter-examples/

# layouts

# Flutter Inspector

In Android Studio you can find the Flutter Inspector tab on the far right. Here we see our layout as a widget tree.



# Layouts in Flutter

UI > Layout

## What's the point?

- Widgets are classes used to build UIs.
- Widgets are used for both layout and UI elements.
- Compose simple widgets to build complex widgets.

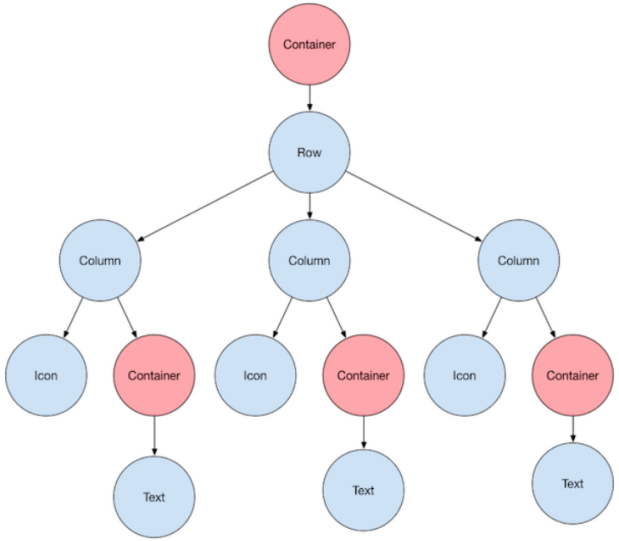
The core of Flutter's layout mechanism is widgets. In Flutter, almost everything is a widget—even layout models are widgets. The images, icons, and text that you see in a Flutter app are all widgets. But things you don't see are also widgets, such as the rows, columns, and grids that arrange, constrain, and align the visible widgets.

You create a layout by composing widgets to build more complex widgets. For example, the first screenshot below shows 3 icons with a label under each one:


The second screenshot displays the visual layout, showing a row of 3 columns where each column contains an icon and a label.

**Note:** Most of the screenshots in this tutorial are displayed with `debugPaintSizeEnabled` set to true so you can see the visual layout. For more information, see Debugging layout issues visually, a section in Using the Flutter inspector.

Here's a diagram of the widget tree for this UI:

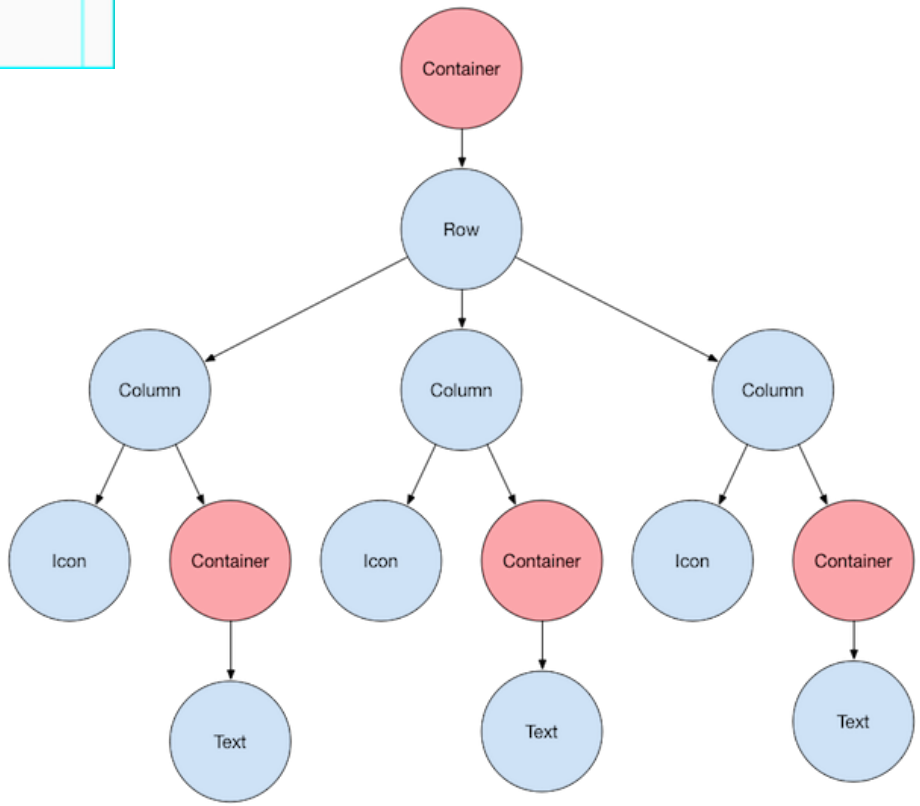


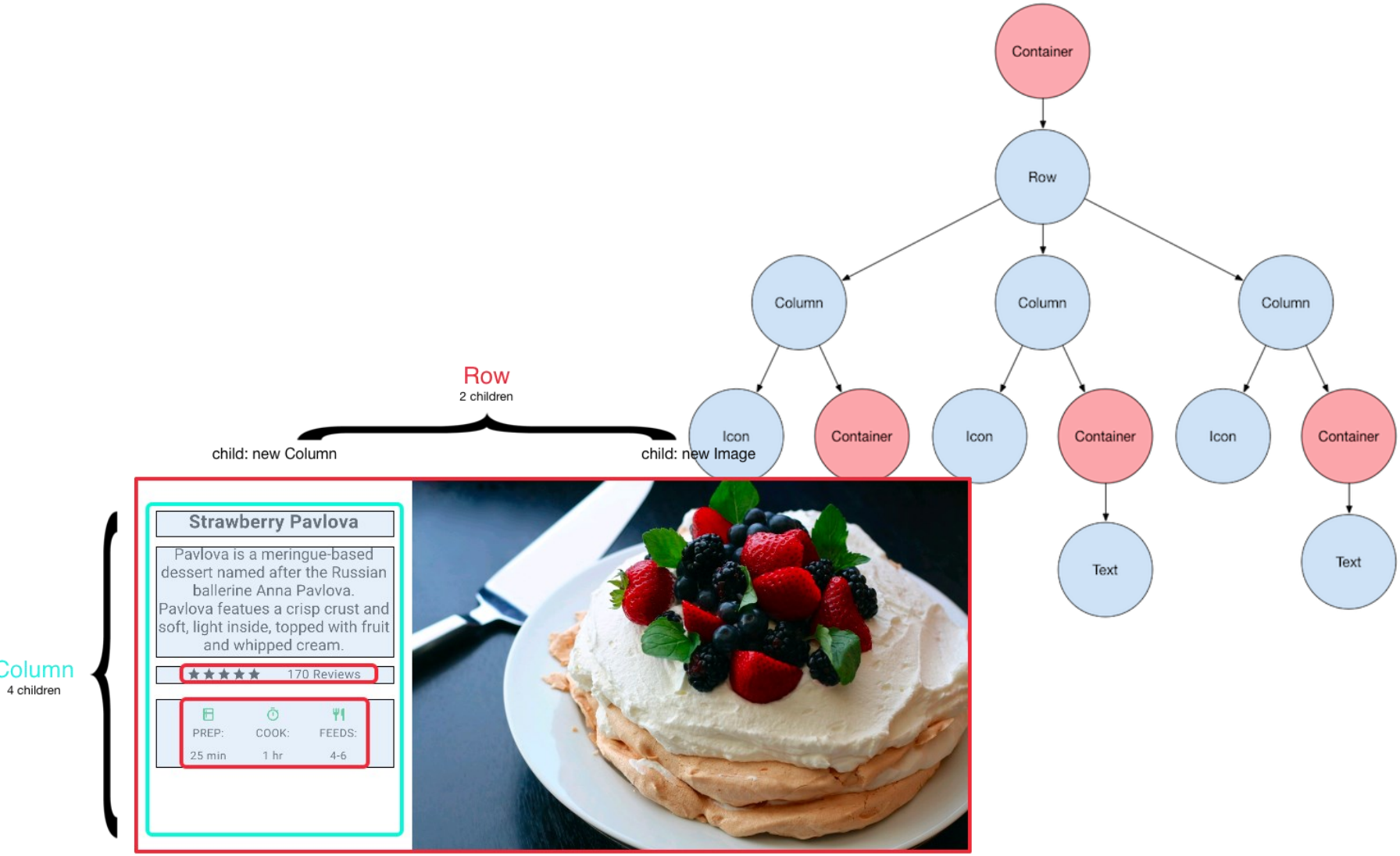

Most of this shoul
widget class that a
background color,

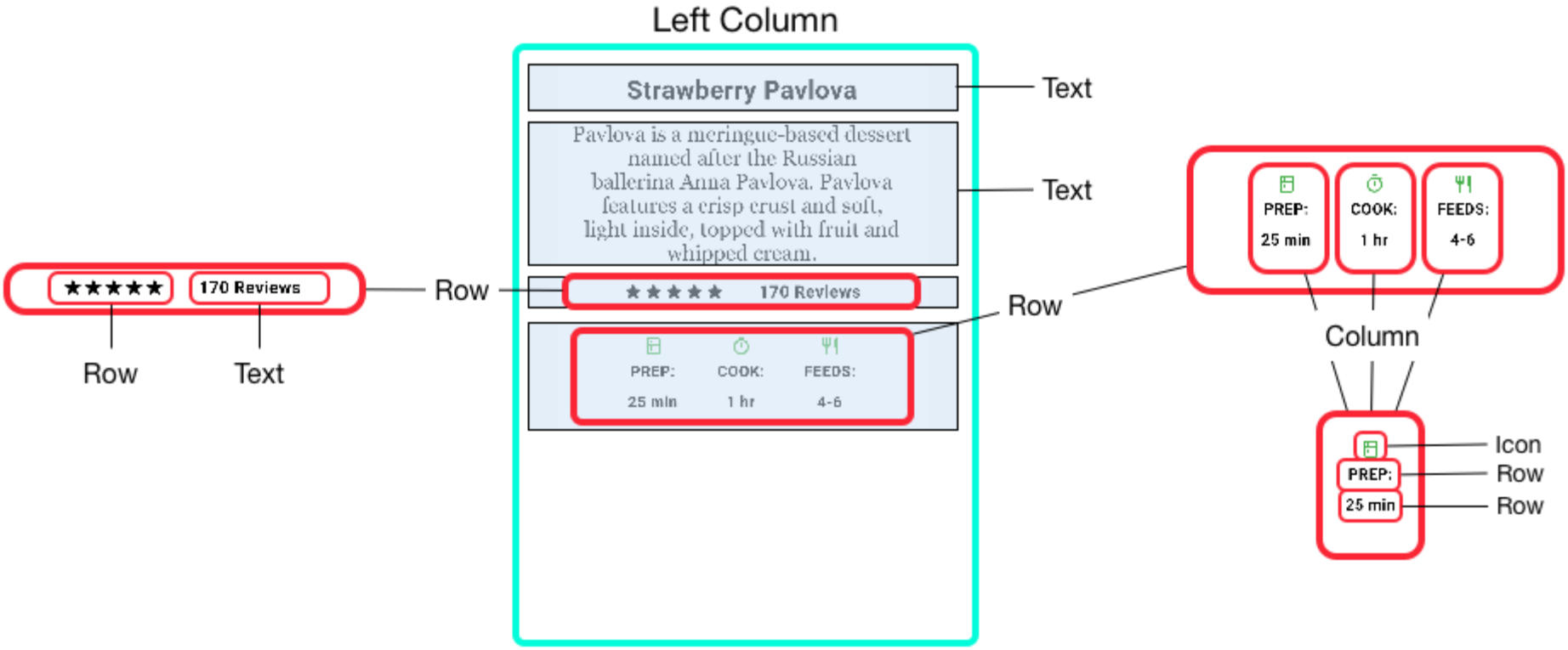
https://docs.flutter.dev/development/ui/layout


CALL
ROUTE
SHARE


CALL
ROUTE
SHARE


Container
Row
Column
Column
Column
Icon
Container
Icon
Container
Icon
Container
Text
Text
Text


Container
Row
Column
Column
Column
Icon
Container
Icon
Container
Icon
Container
Text
Text
Row
2 children
child: new Column
child: new Image
Column
4 children
Strawberry Pavlova
Pavlova is a meringue-based dessert named after the Russian ballerine Anna Pavlova. Pavlova featues a crisp crust and soft, light inside, topped with fruit and whipped cream.
170 Reviews
PREP:
25 min
COOK:
1 hr
FEEDS:
4-6

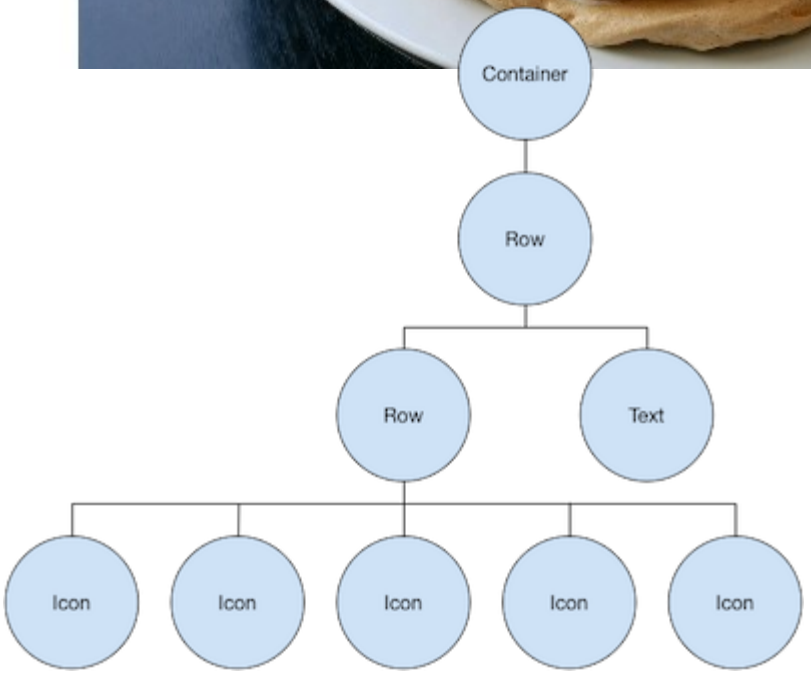

Left Column
Strawberry Pavlova
Text
Pavlova is a meringue-based dessert named after the Russian ballerina Anna Pavlova. Pavlova features a crisp crust and soft, light inside, topped with fruit and whipped cream.
Text
★★★★★ 170 Reviews
Row
PREP: 25 min
COOK: 1 hr
FEEDS: 4-6
Row
PREP: 25 min
COOK: 1 hr
FEEDS: 4-6
Column
Icon
Row
Row
★★★★★ 170 Reviews
Row
Text

## Strawberry Pavlova

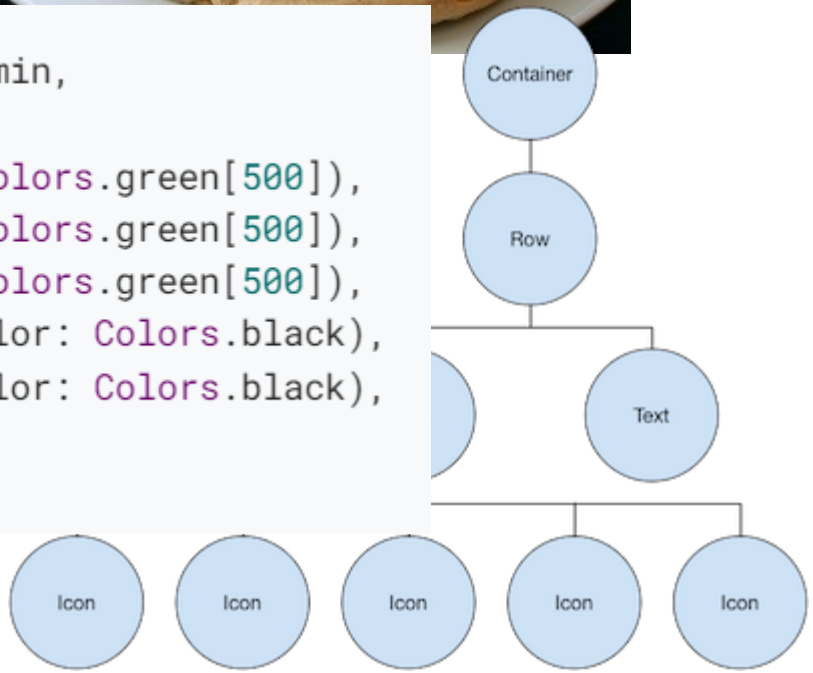
Pavlova is a meringue-based dessert named after the Russian ballerina Anna Pavlova. Pavlova features a crisp crust and soft, light inside, topped with fruit and whipped cream.

★★★★★ 170 Reviews

| PREP: | COOK: | FEEDS: |
|---|---|---|
| 25 min | 1 hr | 4-6 |




```
var stars = Row(
  mainAxisSize: MainAxisSize.min,
  children: [
    Icon(Icons.star, color: Colors.green[500]),
    Icon(Icons.star, color: Colors.green[500]),
    Icon(Icons.star, color: Colors.green[500]),
    const Icon(Icons.star, color: Colors.black),
    const Icon(Icons.star, color: Colors.black),
  ],
);
```



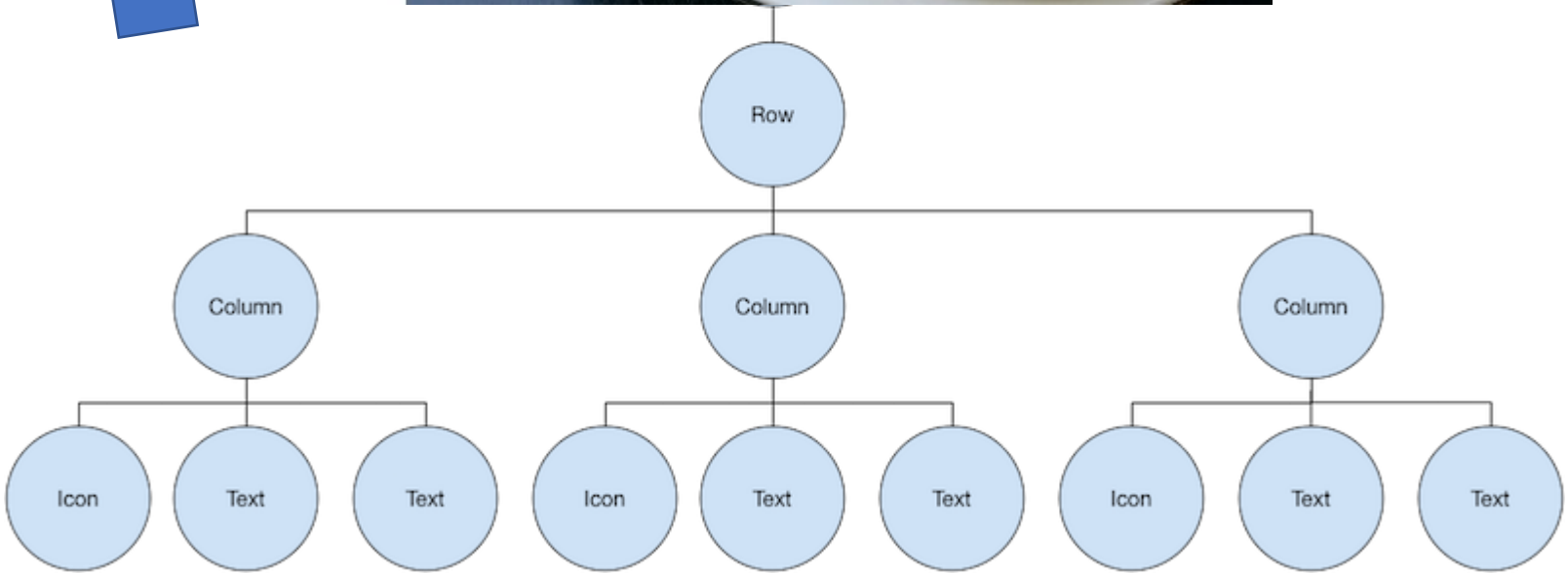

```dart
final ratings = Container(
  padding: const EdgeInsets.all(20),
  child: Row(
    mainAxisAlignment: MainAxisAlignment.spaceEvenly,
    children: [
      stars,
      const Text(
        '170 Reviews',
        style: TextStyle(
          color: Colors.black,
          fontWeight: FontWeight.w800,
          fontFamily: 'Roboto',
          letterSpacing: 0.5,
          fontSize: 20,
        ),
      ),
    ],
  ),
);
```

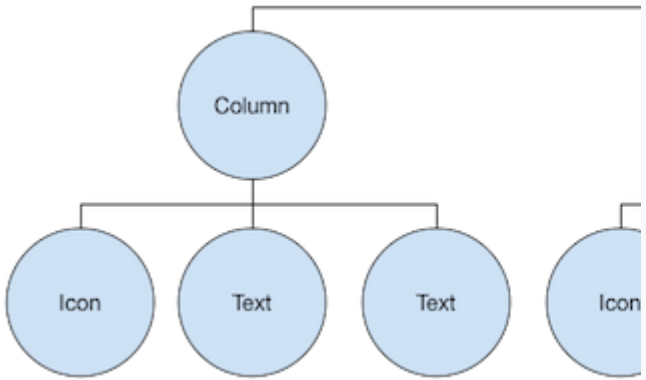
## Strawberry Pavlova

Pavlova is a meringue-based dessert named after the Russian ballerina Anna Pavlova. Pavlova features a crisp crust and soft, light inside, topped with fruit and whipped cream.

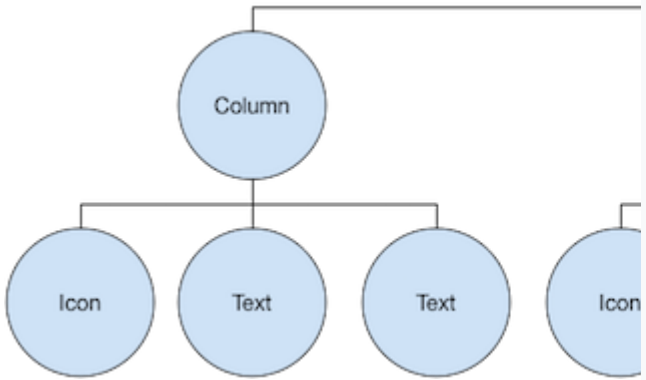
★★★★★ 170 Reviews

| PREP: | COOK: | FEEDS: |
|---|---|---|
| 25 min | 1 hr | 4-6 |

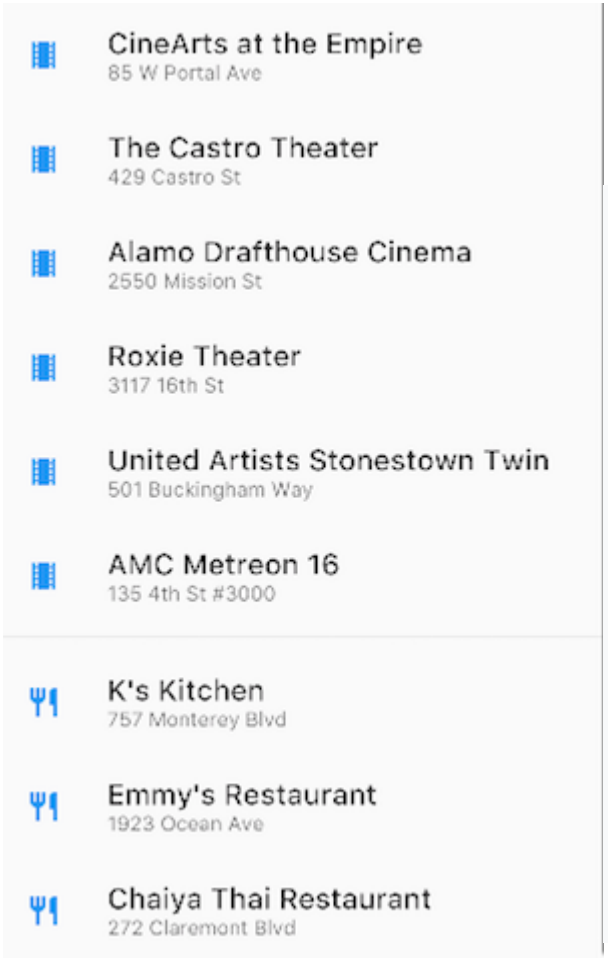



```dart
final iconList = DefaultTextStyle.merge(
  style: descTextStyle,
  child: Container(
    padding: const EdgeInsets.all(20),
    child: Row(
      mainAxisAlignment: MainAxisAlignment.spaceEvenly,
      children: [
        Column(
          children: [
            Icon(Icons.kitchen, color: Colors.green[500]),
            const Text('PREP:'),
            const Text('25 min'),
          ],
        ),
        Column(
          children: [
            Icon(Icons.timer, color: Colors.green[500]),
            const Text('COOK:'),
            const Text('1 hr'),
          ],
        ),
        Column(
          children: [
            Icon(Icons.restaurant, color: Colors.green[500]),
            const Text('FEEDS:'),
            const Text('4-6'),
          ],
        ),
      ],
    ),
  ),
);
```


```dart
final iconList = DefaultTextStyle.merge(
  style: descTextStyle,
  child: Container(
    padding: const EdgeInsets.all(20),
    child: Row(
      mainAxisAlignment: MainAxisAlignment.spaceEvenly,
      children: [
        Column(
          children: [
            Icon(Icons.kitchen, color: Colors.green[500]),
            const Text('PREP:'),
            const Text('25 min'),
          ],
        ),
        Column(
          children: [
            Icon(Icons.timer, color: Colors.green[500]),
            const Text('COOK:'),
            const Text('1 hr'),
          ],
        ),
        Column(
          children: [
            Icon(Icons.restaurant, color: Colors.green[500]),
            const Text('FEEDS:'),
            const Text('4-6'),
          ],
        ),
      ],
    ),
  ),
);
```




```
Widget _buildList() {
  return ListView(
    children: [
      _tile('CineArts at the Empire', '85 W Portal Ave', Icons.theaters),
      _tile('The Castro Theater', '429 Castro St', Icons.theaters),
      _tile('Alamo Drafthouse Cinema', '2550 Mission St', Icons.theaters),
      _tile('Roxie Theater', '3117 16th St', Icons.theaters),
      _tile('United Artists Stonestown Twin', '501 Buckingham Way',
          Icons.theaters),
      _tile('AMC Metreon 16', '135 4th St #3000', Icons.theaters),
      const Divider(),
      _tile('K\'s Kitchen', '757 Monterey Blvd', Icons.restaurant),
      _tile('Emmy\'s Restaurant', '1923 Ocean Ave', Icons.restaurant),
      _tile(
          'Chaiya Thai Restaurant', '272 Claremont Blvd', Icons.restaurant),
      _tile('La Ciccia', '291 30th St', Icons.restaurant),
    ],
  );
}
```

https://docs.flutter.dev/development/ui/layout

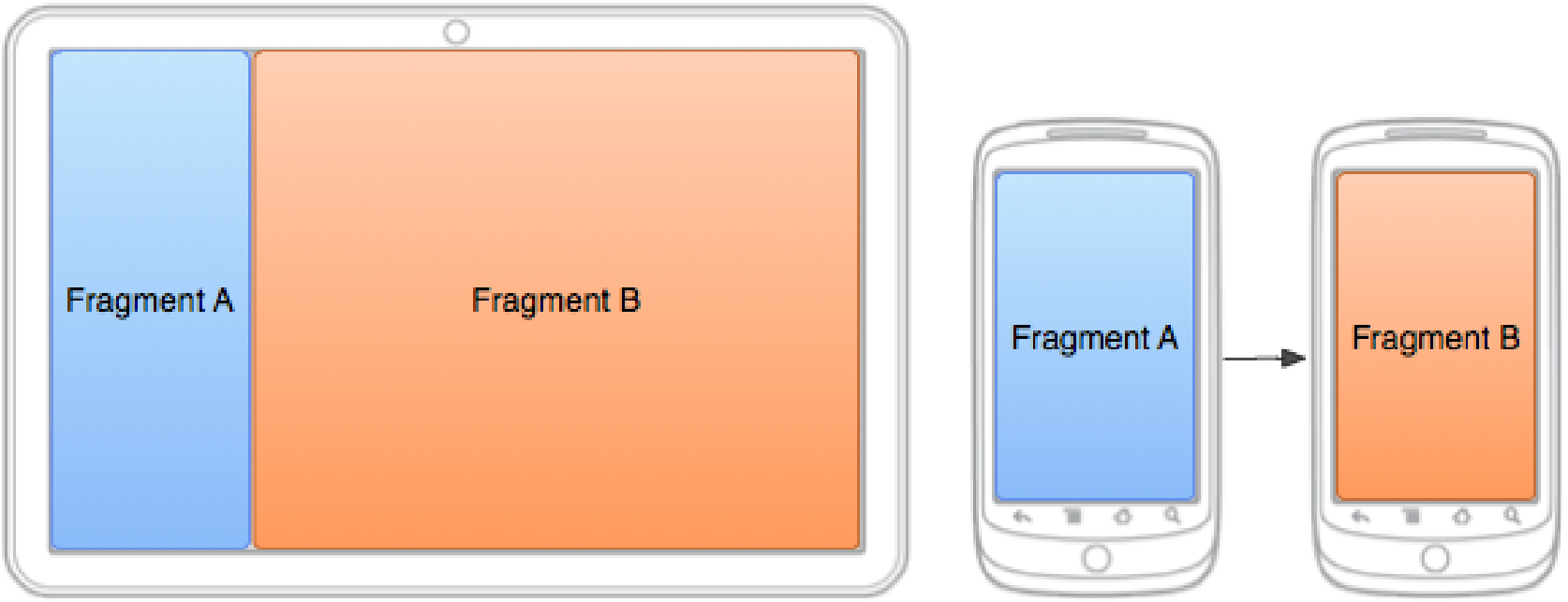
# No more fragments

# Developing for Multiple Screen Sizes and Orientations in Flutter (Fragments in Flutter)

Making Adaptive Screens in Google's Flutter Mobile SDK

Deven Joshi Follow

Oct 24, 2018 · 10 min read



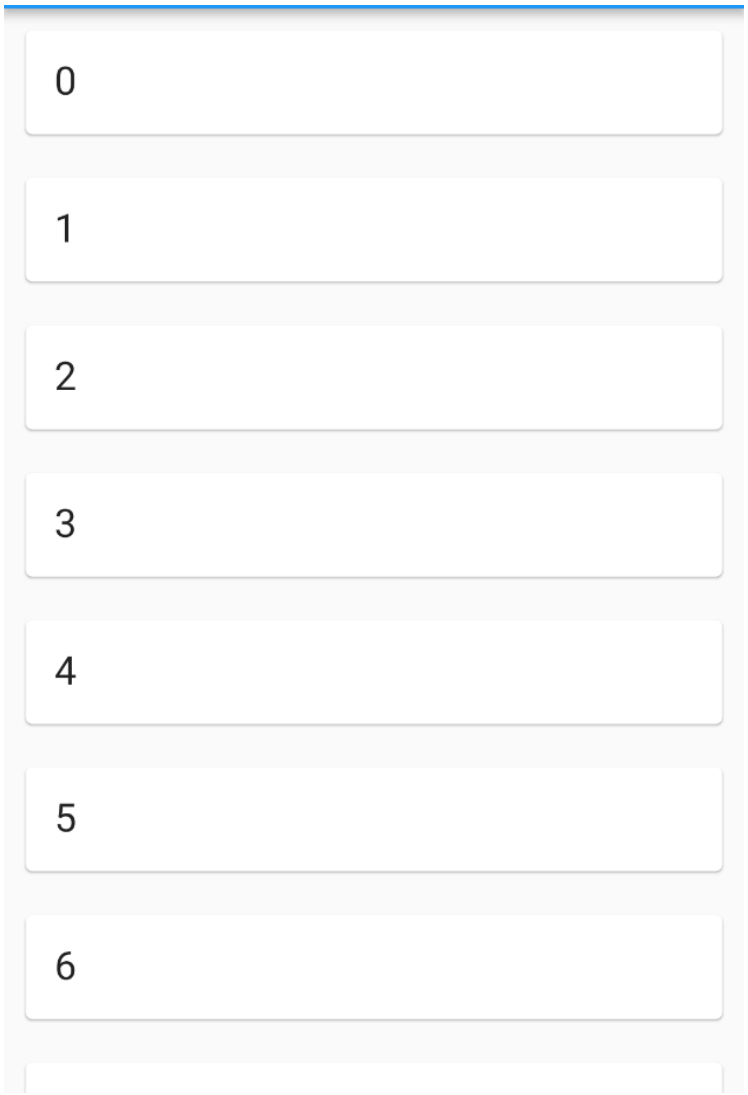

https://medium.com/flutter-community/developing-for-multiple-screen-sizes-and-orientations-in-flutter-fragments-in-flutter-a4c51b849434

# Fragments


So **Fragment A is the master list** fragment and **B is the detail fragment**.

# In flutter

- No need of Fragments
- widgets can easily fill the part
  - **Every widget in Flutter is by nature, reusable.**
  - **Every widget in Flutter is like a Fragment.**
  - Natural integration in programming model
  - no split between layout and lifecycle resources  as in Android
- Master – detail
  - Only need to do is define **two widgets**.
    - One for the master list,
    - one for the detail view.
  - **We simply check if the device has enough width to handle both the list and detail part.**
    - If it does, we use both widgets.
    - If the device does not have enough width to support both, we only show the list and navigate to a separate screen to show the detail content.

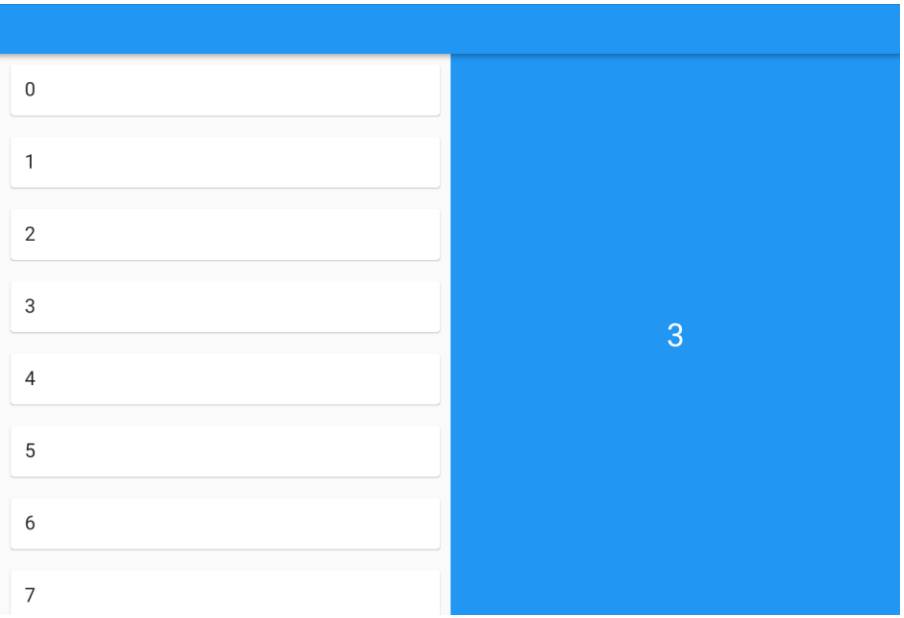



```
typedef Null ItemSelectedCallback(int value);

class ListWidget extends StatefulWidget {
  final int count;
  final ItemSelectedCallback onItemSelected;

  ListWidget(
    this.count,
    this.onItemSelected,
  );

  @override
  _ListWidgetState createState() => _ListWidgetState();
}

class _ListWidgetState extends State<ListWidget> {
  @override
  Widget build(BuildContext context) {
    return ListView.builder(
      itemCount: widget.count,
      itemBuilder: (context, position) {
        return Padding(
          padding: const EdgeInsets.all(8.0),
          child: Card(
            child: InkWell(
              onTap: () {
                widget.onItemSelected(position);
              },
              child: Row(
                children: <Widget>[
                  Padding(
                    padding: const EdgeInsets.all(16.0),
                    child: Text(position.toString(), style:
TextStyle(fontSize: 22.0),),
                  ),
                ],
              ),
            ),
          ),
        );
      },
    );
  }
}
```

1

```
class DetailWidget extends StatefulWidget {

  final int data;

  DetailWidget(this.data);

  @override
  _DetailWidgetState createState() => _DetailWidgetState();
}

class _DetailWidgetState extends State<DetailWidget> {
  @override
  Widget build(BuildContext context) {
    return Container(
      color: Colors.blue,
      child: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text(widget.data.toString(), style: TextStyle(fontSize:
36.0, color: Colors.white),),
          ],
        ),
      ),
    );
  }
}
```

# The Main Screen

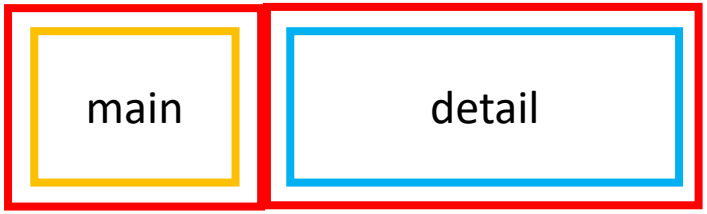



```
class _MasterDetailPageState extends State<MasterDetailPage> {
  var selectedValue = 0;
  var isLargeScreen = false;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(),
      body: OrientationBuilder(builder: (context, orientation) {

        if (MediaQuery.of(context).size.width > 600) {
          isLargeScreen = true;
        } else {
          isLargeScreen = false;
        }

        return Row(children: <Widget>[
          Expanded(
            child: ListWidget(10, (value) {
              if (isLargeScreen) {
                selectedValue = value;
                setState(() {});
              } else {
                Navigator.push(context, MaterialPageRoute(
                  builder: (context) {
                    return DetailPage(value);
                  },
                ));
              }
            }),
          ),
          isLargeScreen ? Expanded(child:
DetailWidget(selectedValue)) : Container(),
        ]);
      }),
    );
  }
}
```

If the screen is large, we add a detail widget, and if it is not, we return an empty container. We use the Expanded widgets around it to fill the screen or divide the screen into proportions in case of a larger screen. So Expanded allows each widget to fill half of the screen or even a certain percentage by setting the Flex property.

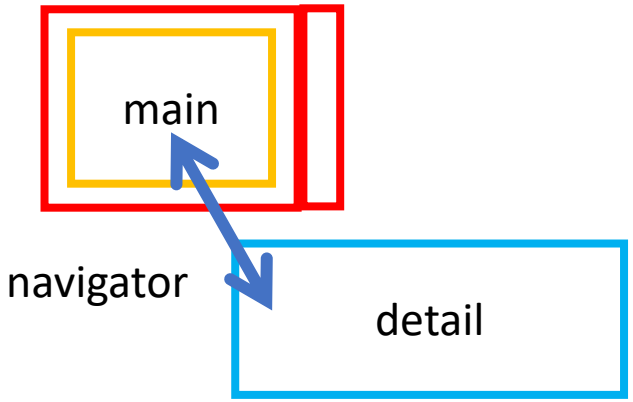
Has Space ? yes

Second container with detail




First container with main

Has Space ? no




Second container empty

# Has space

```
class _MasterDetailPageState extends State<MasterDetailPage> {
  var selectedValue = 0;
  var isLargeScreen = false;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
```

```
return Row(children: <Widget>[
  Expanded(
    child: ListWidget(10, (value) {
      if (isLargeScreen) {
        selectedValue = value;
        setState(() {});
      } else {
        Navigator.push(context, MaterialPageRoute(
          builder: (context) {
            return DetailPage(value);
          },
        ));
      }
    }),
  ),
  isLargeScreen ? Expanded(child:
DetailWidget(selectedValue)) : Container(),
]);
```

Main

Detail

# Has no space

```
class _MasterDetailPageState extends State<MasterDetailPage> {
  var selectedValue = 0;
  var isLargeScreen = false;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
```

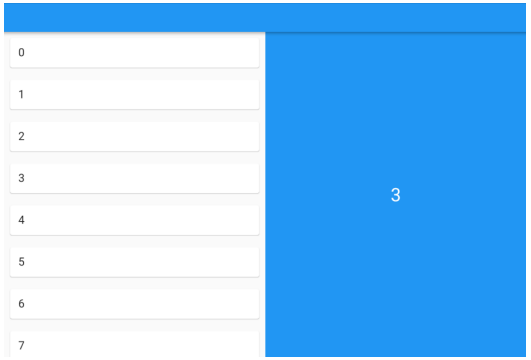
```
0
1
2
3
4
5
6
7
```

```
return Row(children: <Widget>[
  Expanded(
    child: ListWidget(10, (value) {
      if (isLargeScreen) {
        selectedValue = value;
        setState(() {});
      } else {
        Navigator.push(context, MaterialPageRoute(
          builder: (context) {
            return DetailPage(value);
          },
        ));
      }
    }),
  ),
  isLargeScreen ? Expanded(child:
DetailWidget(selectedValue)) : Container(),
]);
}),
```

Main

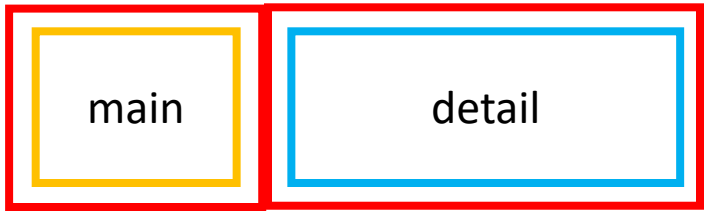
Detail

# Responsive Main-detail UI




Has Space ? yes

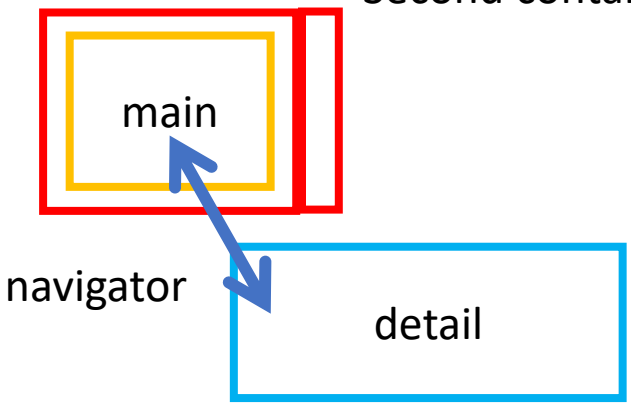
Second container with detail




First container with main


Has Space ? no

Second container empty



# Responsive

# What is Responsive Design?

- the concept of **Responsive Design** is all about using one set of code that *respond* to various changes to layout. Platforms such as the iOS and Android native SDKs tackled this issue with “universal layouts.” The universal layouts respond to layout changes by using constraints and automatically resizing elements.
  - Handling Different Device Types and Screen Sizes
  - Handling Keyboard State Changes
  - Handling Orientation Changes

Responsive Design for Flutter: Getting Started
https://www.raywenderlich.com/4324124-responsive-design-for-flutter-getting-started

# Creating responsive and adaptive apps

UI > Layout > Responsive and adaptive

One of Flutter's primary goals is to create a framework that allows you to develop apps from a single codebase that look and feel great on any platform.

This means that your app may appear on screens of many different sizes, from a watch, to a foldable phone with two screens, to a high def monitor.

Two terms that describe concepts for this scenario are *adaptive* and *responsive*. Ideally, you'd want your app to be *both* but what, exactly, does this mean? These terms are similar, but they are not the same.

## The difference between an adaptive and a responsive app

*Adaptive* and *responsive* can be viewed as separate dimensions of an app: you can have an adaptive app that is not responsive, or vice versa. And, of course, an app can be both, or neither.

**Responsive**
Typically, a *responsive* app has had its layout tuned for the available screen size. Often this means (for example), re-laying out the UI if the user resizes the window, or changes the device's orientation. This is especially necessary when the same app can run on a variety of devices, from a watch, phone, tablet, to a laptop or desktop computer.

**Adaptive**
*Adapting* an app to run on different device types, such as mobile and desktop, requires dealing with mouse and keyboard input, as well as touch input. It also means there are different expectations about the app's visual density, how component selection works (cascading menus vs bottom sheets, for example), using platform-specific features (such as top-level windows), and more.

## Creating a responsive Flutter app

Flutter allows you to create apps that self-adapt to the device's screen size and orientation.

There are two basic approaches to creating Flutter apps with responsive design:

**Use the `LayoutBuilder` class**
From its `builder` property, you get a `BoxConstraints` object. Examine the constraint's properties to decide what to display. For example, if your `maxWidth` is greater than your width breakpoint, return a `Scaffold` object with a row that has a list on the left. If it's narrower, return a `Scaffold` object with a drawer containing that list. You can also adjust your display based on the device's height, the aspect ratio, or some other property. When the constraints change (for example, the user rotates the phone, or puts your app into a tile UI in Nougat), the build function runs.

**Use the `MediaQuery.of()` method in your build functions**
This gives you the size, orientation, etc, of your current app. This is more useful if you want to make decisions based on the complete context rather than on just the size of your particular widget. Again, if you use this, then your build function automatically runs if the user somehow changes the app's size.

Other useful widgets and classes for creating a responsive UI:

- `AspectRatio`
- `CustomSingleChildLayout`
- `CustomMultiChildLayout`
- `FittedBox`
- `FractionallySizedBox`

https://docs.flutter.dev/development/ui/layout/adaptive-responsive

C

# Creating responsive and adaptive apps

UI > Layout > Responsive and adaptive

One of Flutter's primary goals is to create a framework that allows you to develop apps from a single codebase that look and feel great on any platform.

This means that your app may appear on screens of many different sizes, from a watch, to a foldable phone with two screens, to a high def monitor.

Two terms that describe concepts for this scenario are *adaptive* and *responsive*. Ideally, you'd want your app to be *both* but what, exactly, does this mean? These terms are similar, but they are not the same.

## The difference between an adaptive and a responsive app

*Adaptive* and *responsive* can be viewed as separate dimensions of an app: you can have an adaptive app that is not responsive, or vice versa. And, of course, an app can be both, or neither.

**Responsive**
Typically, a *responsive* app has ha
the UI if the user resizes the wind
run on a variety of devices, from a

**Adaptive**
*Adapting* an app to run on differer
as well as touch input. It also mea
works (cascading menus vs botto
more.

## Creating a re

Flutter allows you to create apps t

There are two basic approaches t

**Use the `LayoutBuilder` class**
From its `builder` property, you ge
example, if your `maxWidth` is great
it's narrower, return a `Scaffold` ob
height, the aspect ratio, or some c
your app into a tile UI in Nougat), t

**Use the `MediaQuery.of()` metho**
This gives you the size, orientatio
complete context rather than on ju
automatically runs if the user sor

Other useful widgets and classes for creating a responsive UI:

- `AspectRatio`
- `CustomSingleChildLayout`
- `CustomMultiChildLayout`
- `FittedBox`
- `FractionallySizedBox`

Other useful widgets and classes for creating a responsive UI:

- `AspectRatio`
- `CustomSingleChildLayout`
- `CustomMultiChildLayout`
- `FittedBox`
- `FractionallySizedBox`
- `LayoutBuilder`
- `MediaQuery`
- `MediaQueryData`
- `OrientationBuilder`

https://docs.flutter.dev/development/ui/layout/adaptive-responsive

C

# Creating responsive and adaptive apps

UI > Layout > Responsive and adaptive

One of Flutter's primary goals is to create a framework that allows you to develop apps from a single codebase that look and feel great on any platform.

This means that your app may appear on screens of many different sizes, from a watch, to a foldable phone with two screens, to a high def monitor.

Two terms that describe concepts for this scenario are *adaptive* and *responsive*. Ideally, you'd want your app to be *both* but what, exactly, does this mean? These terms are similar, but they are not the same.

## The difference between an adaptive and a responsive app

*Adaptive* and *responsive* can be viewed as separate dimensions of an app: you can have an adaptive app that is not responsive, or vice versa. And, of course, an app can be both, or neither.

**Responsive**
Typically, a *responsive* app has ha
the UI if the user resizes the wind
run on a variety of devices, from a

**Adaptive**
*Adapting* an app to run on differer
as well as touch input. It also mea
works (cascading menus vs botto
more.

## Creating a re

Flutter allows you to create apps t

There are two basic approaches t

**Use the `LayoutBuilder` class**
From its `builder` property, you ge
example, if your `maxWidth` is great
it's narrower, return a `Scaffold` ob
height, the aspect ratio, or some c
your app into a tile UI in Nougat), t

**Use the `MediaQuery.of()` metho**
This gives you the size, orientatio
complete context rather than on ju
automatically runs if the user sor

Other useful widgets and classes for creating a responsive UI:

- `AspectRatio`
- `CustomSingleChildLayout`
- `CustomMultiChildLayout`
- `FittedBox`
- `FractionallySizedBox`

Other useful widgets and classes for creating a responsive UI:

- `AspectRatio`
- `CustomSingleChildLayout`
- `CustomMultiChildLayout`
- `FittedBox`
- `FractionallySizedBox`
- `LayoutBuilder`
- `MediaQuery`
- `MediaQueryData`
- `OrientationBuilder`

https://docs.flutter.dev/development/ui/layout/adaptive-responsive

C

# How to detect what platform a Flutter app is running on

2020-06-12 by marc

Android? iOS? Web? MacOS? – With the growing number of supported build platforms, Flutter increases its value for developers who like to have only a single code base.
Also, this creates the necessity to make design decisions based on the platform it's running on because the underlying APIs to access the hardware can vary.

Just to name a few things that are (not necessarily) available on the web platform:

- Camera
- Vibration
- Push-Notifications
- Acceleration sensor
- Microphone
- ...

Flutter provides two different APIs that enables the caller to get to know more about the current platform: the kIsWeb constant that is part of the foundation library and the Platform class being part of the platform library.

The aim I have is to implement a class that provides an API that abstracts from these two quite low-level

https://www.flutterclutter.dev/flutter/tutorials/how-to-detect-what-platform-a-flutter-app-is-running-on/2020/127/

*getCurrentPlatformType*, *isWeb*, *isAppOS* and *isDesktop*. The first one should return an enum that is defined

# How to detect what platform a Flutter app is running on

2020-06-12 by marc

Android? iOS? Web? MacOS? – With the growing number of supported build platforms, Flutter increases its value for developers who like to have only a single code base.
Also, this creates the necessity to make design decisions based on the platform it's running on because the underlying APIs to access the hardware can vary.

Just to name a few things that are (not necessarily) available on the web platform:

- Camera
- Vibration
- Push-Notifications
- Acceleration sensor
- Microphone
- ...

Flutter provides two different APIs that enables the caller to get to know more about the current platform: the kIsWeb constant that is part of the foundation library and the Platform class being part of the platform library.

The aim I have is to implement a class that provides an API that abstracts from these two quite low-level

https://www.flutterclutter.dev/flutter/tutorials/how-to-detect-what-platform-a-flutter-app-is-running-on/2020/127/

*getCurrentPlatformType*, *isWeb*, *isAppOS* and *isDesktop*. The first one should return an enum that is defined

# foundation library Null safety

Core Flutter framework primitives.

The features defined in this library are the lowest-level utility classes and functions used by all the other layers of the Flutter framework.

# Platform class Null safety

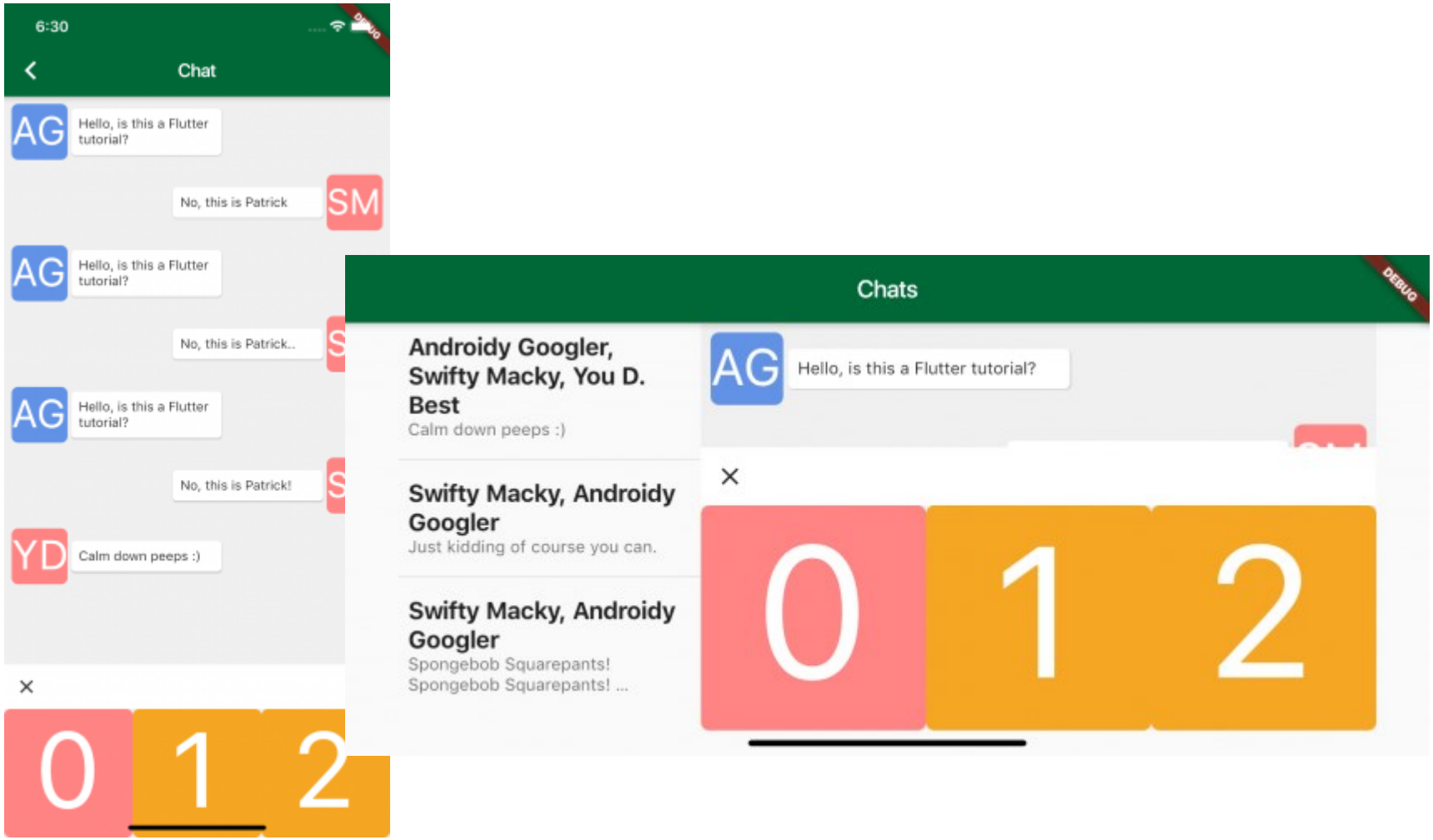
Provides API parity with the `Platform` class in `dart:io`, but using instance properties rather than static properties. This difference enables the use of these APIs in tests, where you can provide mock implementations.

https://api.flutter.dev/flutter/foundation/foundation-library.html
https://api.flutter.dev/flutter/package-platform_platform/Platform-class.html

```dart
import 'dart:io';
import 'package:flutter/foundation.dart' show kIsWeb;

class PlatformInfo {
  bool isDesktopOS() {
    return Platform.isMacOS || Platform.isLinux || Platform.isWindows;
  }

  bool isAppOS() {
    return Platform.isMacOS || Platform.isAndroid;
  }

  bool isWeb() {
    return kIsWeb;
  }

  PlatformType getCurrentPlatformType() {
    if (kIsWeb) {
      return PlatformType.Web;
    }

    if (Platform.isMacOS) {
      return PlatformType.MacOS;
    }

    if (Platform.isFuchsia) {
      return PlatformType.Fuchsia;
    }
```

```dart
enum PlatformType
  Web,
  iOS,
  Android,
  MacOS,
  Fuchsia,
  Linux,
  Windows,
  Unknown
}
```

# orientation



# LayoutBuilder class

Builds a widget tree that can depend on the parent widget's size.

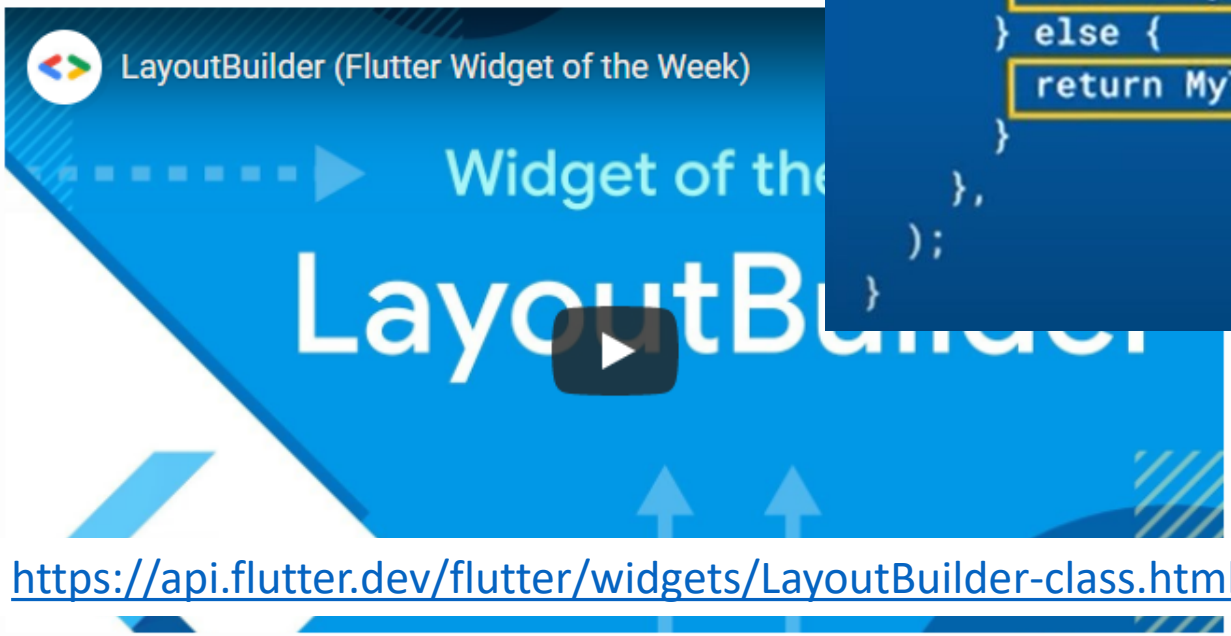
Similar to the Builder widget except that the framework calls the builder function at layout time and provides the parent widget's constraints. This is useful when the parent constrains the child's size and doesn't depend on the child's intrinsic size. The LayoutBuilder's final size will match its child's size.

The builder function is called in the following situations:

- The first time the widget is laid out.
- When the parent widget passes different layout constraints.
- When the parent widget updates this widget.
- When the dependencies that the builder function subscribes t

The builder function is *not* called during layout if the parent passes t

```
Widget build(BuildContext context) {
  return LayoutBuilder(
    builder: (context, constraints) {
      if (constraints.maxWidth < 600) {
        return MyOneColumnLayout();
      } else {
        return MyTwoColumnLayout();
      }
    },
  );
}
```




https://api.flutter.dev/flutter/widgets/LayoutBuilder-class.html

# LayoutBuilder

```dart
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: const Text('LayoutBuilder Example')),
    body: LayoutBuilder(
      builder: (BuildContext context, BoxConstraints constraints) {
        if (constraints.maxWidth > 600) {
          return _buildWideContainers();
        } else {
          return _buildNormalContainer();
        }
      },
    ),
  );
}
```

https://api.flutter.dev/flutter/widgets/LayoutBuilder-class.html

# LayoutBuilder

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: const Text('LayoutBuilder Example')),
    body: LayoutBuilder(
      builder: (BuildContext context, BoxConstraints constraints) {
      ) {
      );

      ();
```

## BoxConstraints class Null safety

Immutable layout constraints for RenderBox layout.

A Size respects a BoxConstraints if, and only if, all of the following relations hold:

- minWidth <= Size.width <= maxWidth
- minHeight <= Size.height <= maxHeight

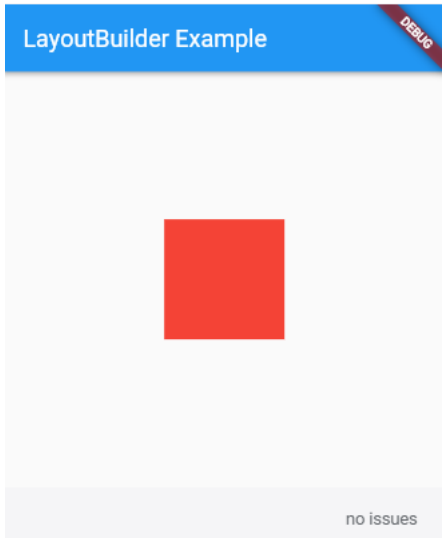
The constraints themselves must satisfy these relations:

- 0.0 <= minWidth <= maxWidth <= double.infinity
- 0.0 <= minHeight <= maxHeight <= double.infinity

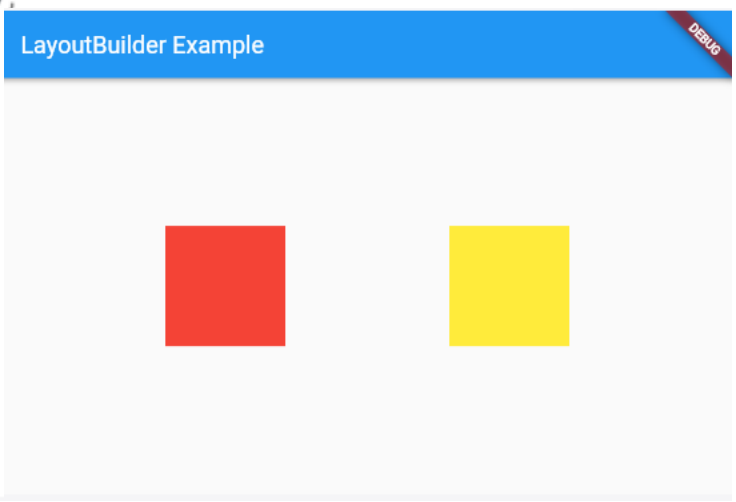
double.infinity is a legal value for each constraint.

https://api.flutter.dev/flutter/widgets/LayoutBuilder-class.html

```dart
Widget _buildNormalContainer() {
  return Center(
    child: Container(
      height: 100.0,
      width: 100.0,
      color: Colors.red,
    ),
  );
}
```

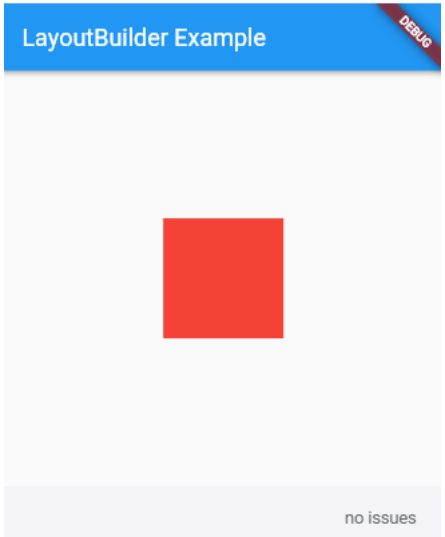



```dart
Widget _buildWideContainers() {
  return Center(
    child: Row(
      mainAxisAlignment: MainAxisAlignment.spaceEvenly,
      children: <Widget>[
        Container(
          height: 100.0,
          width: 100.0,
          color: Colors.red,
        ),
        Container(
          height: 100.0,
          width: 100.0,
          color: Colors.yellow,
        ),
      ],
    ),
  );
}
```

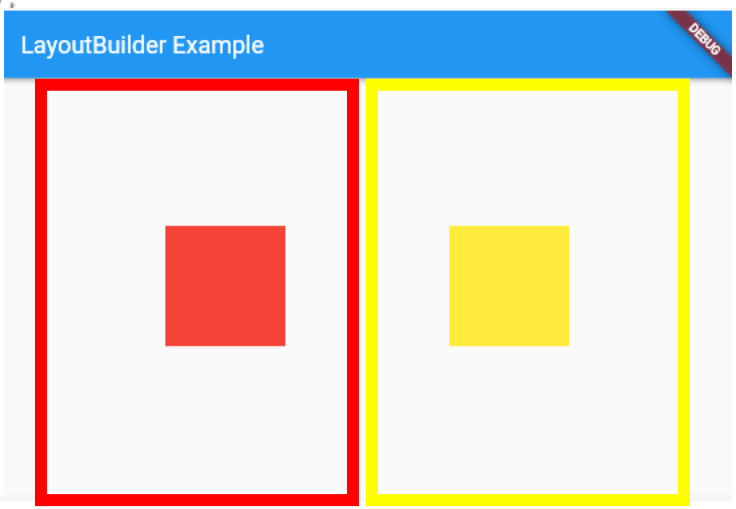



https://api.flutter.dev/flutter/widgets/LayoutBuilder-class.html

```dart
Widget _buildNormalContainer() {
  return Center(
    child: Container(
      height: 100.0,
      width: 100.0,
      color: Colors.red,
    ),
  );
}
```




```dart
Widget _buildWideContainers() {
  return Center(
    child: Row(
      mainAxisAlignment: MainAxisAlignment.spaceEvenly,
      children: <Widget>[
        Container(
          height: 100.0,
          width: 100.0,
          color: Colors.red,
        ),
        Container(
          height: 100.0,
          width: 100.0,
          color: Colors.yellow,
        ),
      ],
    ),
  );
}
```



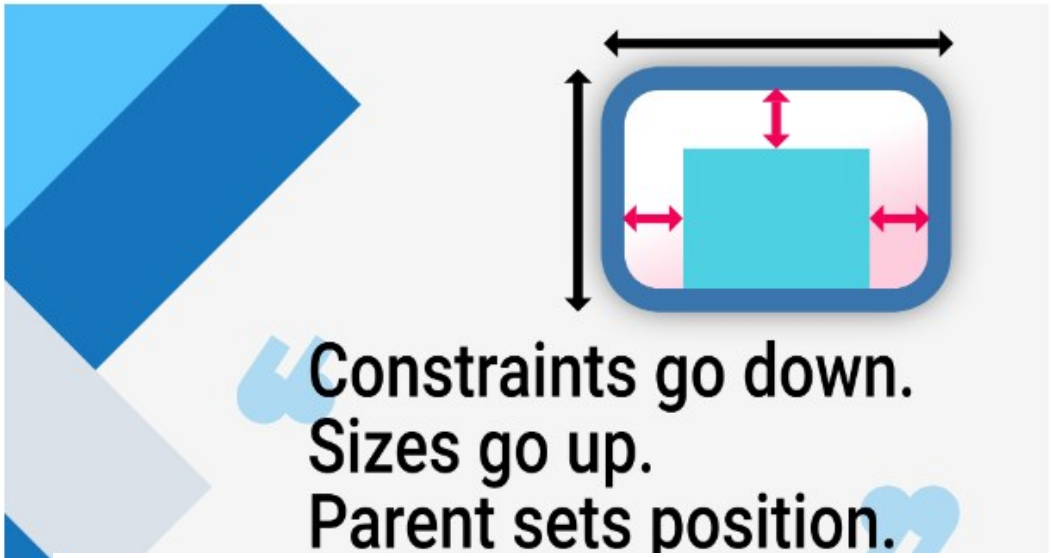

https://api.flutter.dev/flutter/widgets/LayoutBuilder-class.html

# Understanding constraints

UI > Layout > Understanding constraints


**Note:** To better understand how Flutter implements layout constraints, check out the following 5-minute video:


Decoding Flutter: Unbounded height and width




Understanding constraints
https://docs.flutter.dev/development/ui/layout/constraints

When someone learning Flutter asks you why some widget with `width:100` isn't 100 pixels wide, the default answer is to tell

# MediaQuery

- https://api.flutter.dev/flutter/widgets/MediaQuery-class.html
- https://www.geeksforgeeks.org/flutter-managing-the-mediaquery-object/

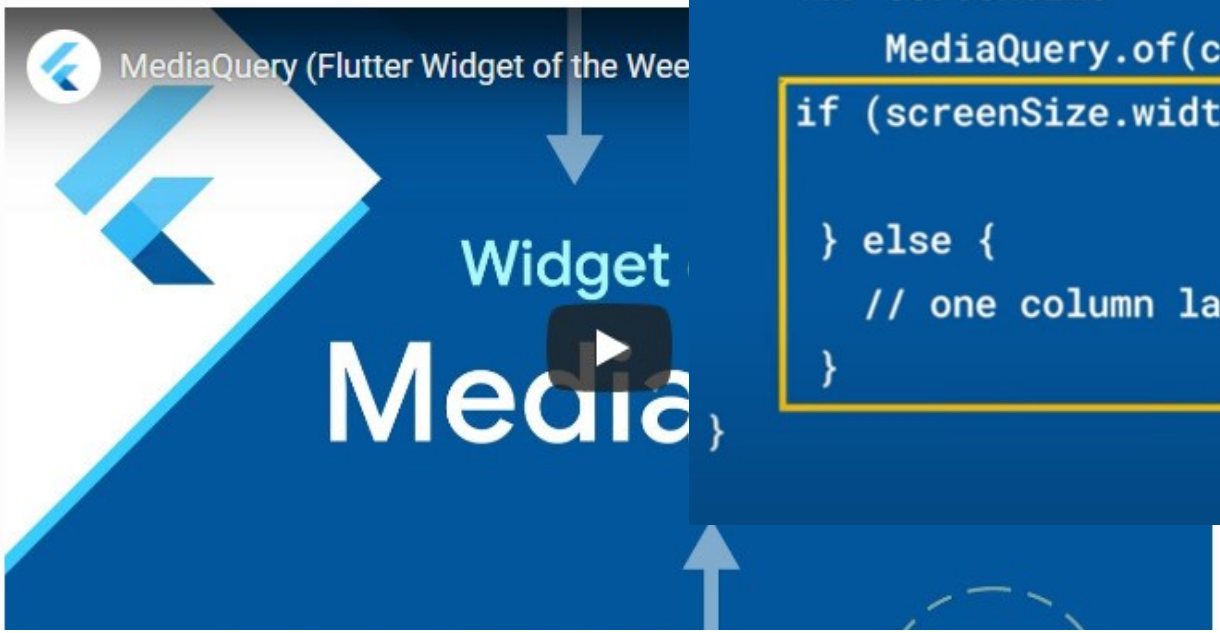
# MediaQuery class

Establishes a subtree in which media queries resolve to the given data.

For example, to learn the size of the current media (e.g., the window containing your app), you can read the MediaQueryData.size property from the MediaQueryData returned by MediaQuery.of: `MediaQuery.of(context).size`.

Querying the current media using MediaQuery.of will cause your widget to rebuild automatically whenever the MediaQueryData changes (e.g., if the user rotates their device).

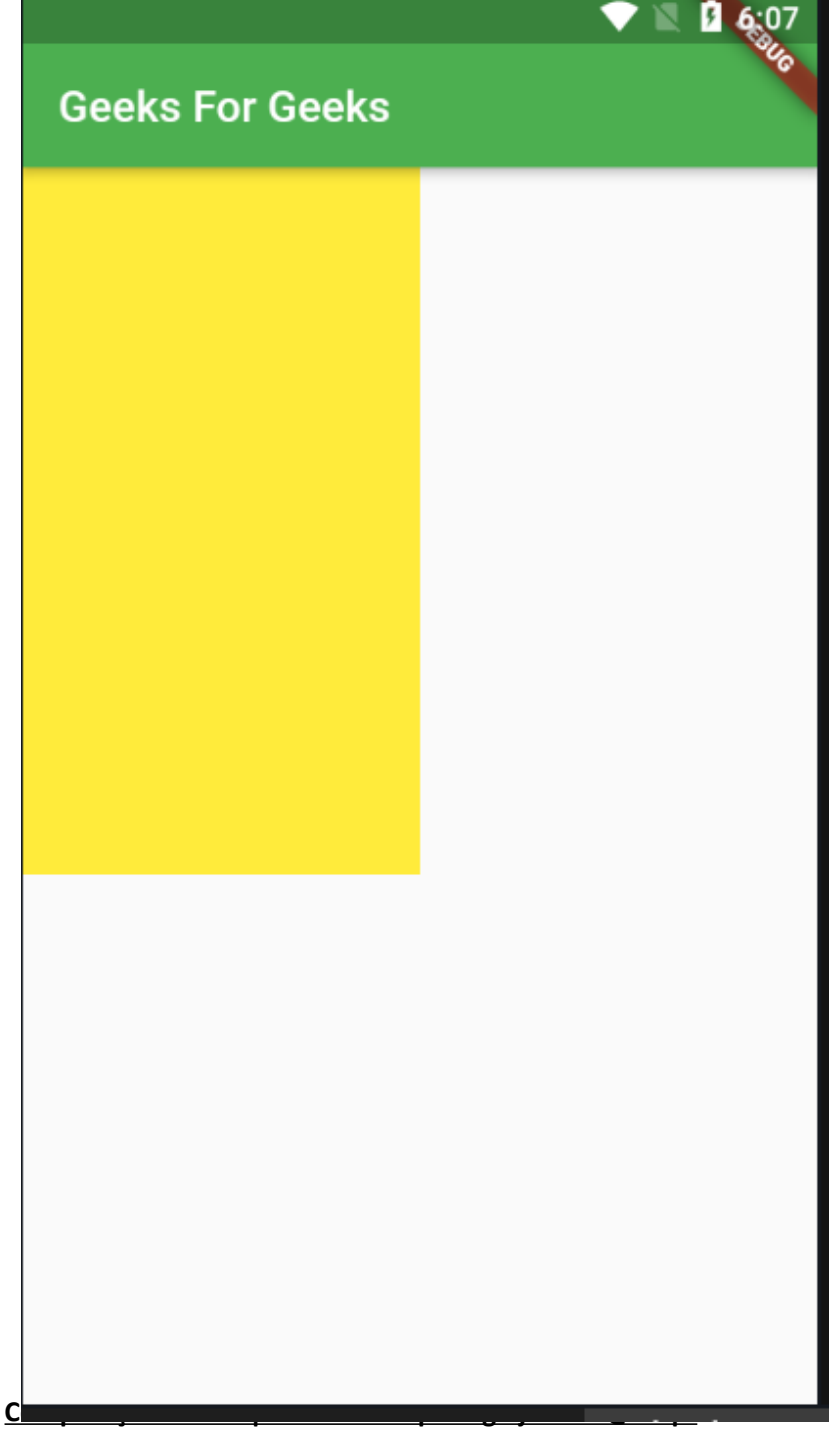
If no MediaQuery is in scope then the MediaQuery.of met
`nullOk` argument is set to true, in which case it returns



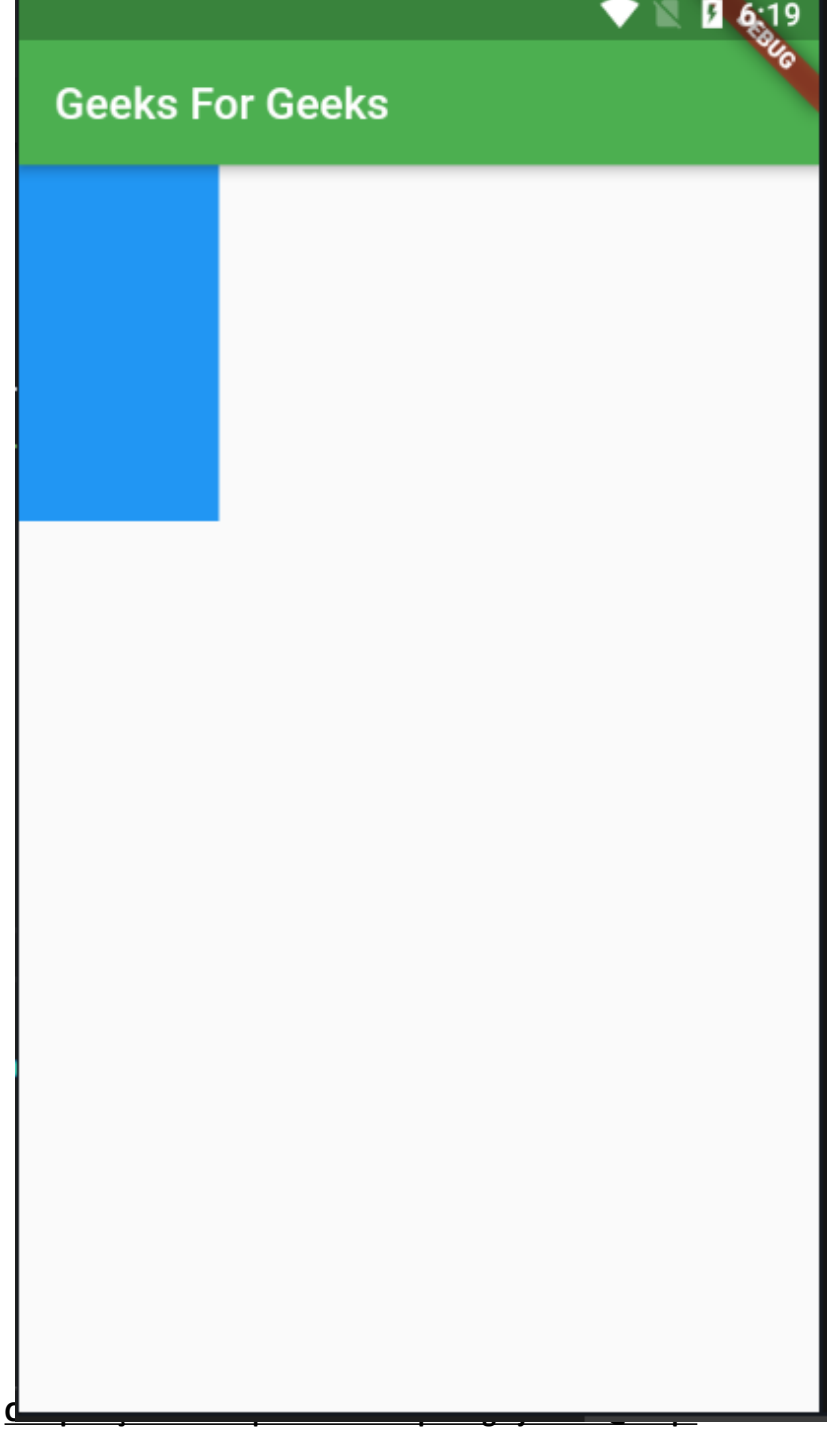

```
build(BuildContext context) {
    var screenSize =
        MediaQuery.of(context).size;
    if (screenSize.width > oneColumnLayout) {

    } else {
      // one column layout
    }
}
```

https://api.flutter.dev/flutter/widgets/MediaQuery-class.html

See also:




```dart
class Home extends StatelessWidget {
var size,height,width;

  @override
  Widget build(BuildContext context) {

    // getting the size of the window
    size = MediaQuery.of(context).size;
    height = size.height;
    width = size.width;

    return Scaffold(
      appBar: AppBar(
        title: Text("Geeks For Geeks"),
        backgroundColor: Colors.green,
      ),
      body: Container(
        color: Colors.yellow,
        height: height/2,//half of the height size
        width: width/2,//half of the width size
      ),
    );
  }
}
```




```
class Home extends StatelessWidget {
var orientation, size,height,width;


  @override
  Widget build(BuildContext context) {

    // getting the orientation of the app
    orientation = MediaQuery.of(context).orientation;

    //size of the window
    size = MediaQuery.of(context).size;
    height = size.height;
    width = size.width;

    return Scaffold(
      appBar: AppBar(
        title: Text("Geeks For Geeks"),
        backgroundColor: Colors.green,
      ),

      // checking the orientation
      body: orientation == Orientation.portrait?Container(
        color: Colors.blue,
        height: height/4,
        width: width/4,
      ):Container(
        height: height/3,
        width: width/3,
        color: Colors.red,
      ),
    );
  }
}
```

# Some useful “widgets”

Other useful widgets and classes for creating a responsive UI:

- `AspectRatio`
- `CustomSingleChildLayout`
- `CustomMultiChildLayout`
- `FittedBox`
- `FractionallySizedBox`
- `LayoutBuilder`
- `MediaQuery`
- `MediaQueryData`
- `OrientationBuilder`

https://flutter.dev/docs/development/ui/layout/responsive

- Building adaptive apps
- https://docs.flutter.dev/development/ui/layout/building-adaptive-apps

- Layout widgets
- https://docs.flutter.dev/development/ui/widgets/layout

# Layout widgets

UI > Widgets > Layout

Arrange other widgets columns, rows, grids, and many other layouts.

- Single-child layout widgets
- Multi-child layout widgets
- Sliver widgets

See more widgets in the widget catalog.

## Single-child layout widgets

Align

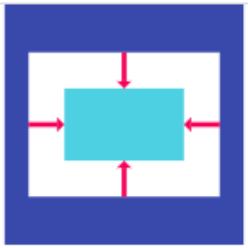
A widget that aligns its child within itself and optionally sizes itself based on the child's size.

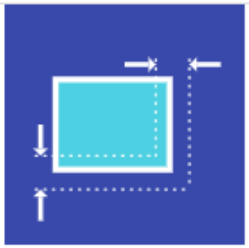
AspectRatio

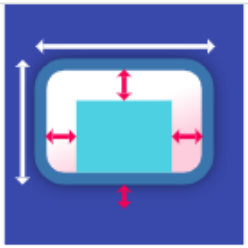
A widget that attempts to size the child to a specific aspect ratio.


Baseline

A widget that positions its child according to the child's baseline.







Ce
A w
witl

Layout widgets
https://docs.flutter.dev/development/ui/widgets/layout

```
Widget foo = LayoutBuilder(
    builder: (context, constraints) {
  bool useVerticalLayout = constraints.maxWidth < 400.0;
  return Flex(
    children: [
      Text('Hello'),
      Text('World'),
    ],
    direction: useVerticalLayout ? Axis.vertical : Axis.horizontal,
  );
});
```

https://docs.flutter.dev/development/ui/layout/building-adaptive-apps#building-adaptive-layouts

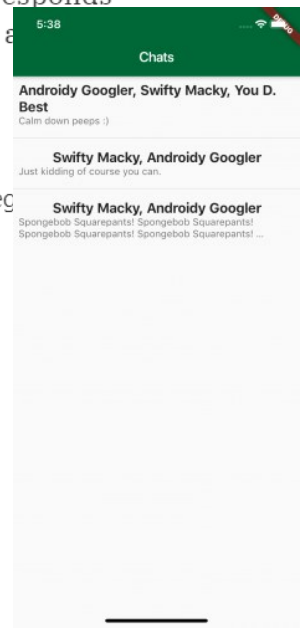
Home · Flutter Tutorials

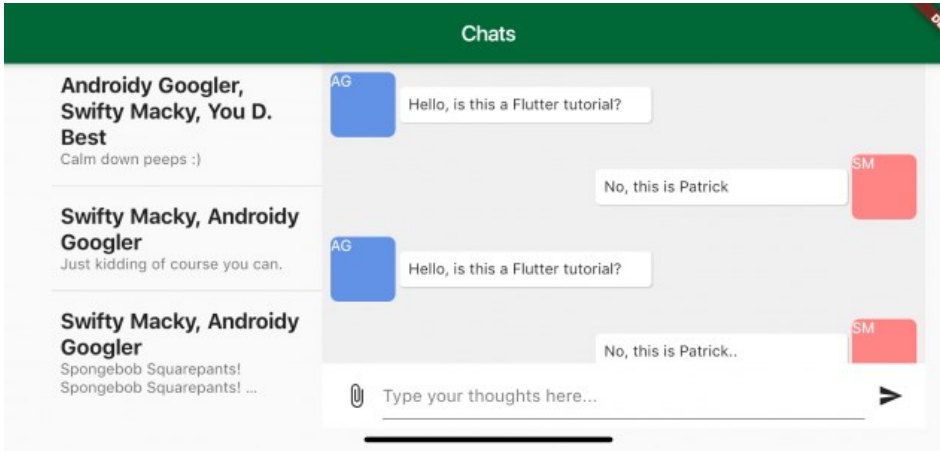
# Responsive Design for Flutter: Getting Started

In this Flutter Responsive Design tutorial you'll learn how to build a Flutter app that responds to layout changes such as screen size a orientation.

By JB Lorenzo

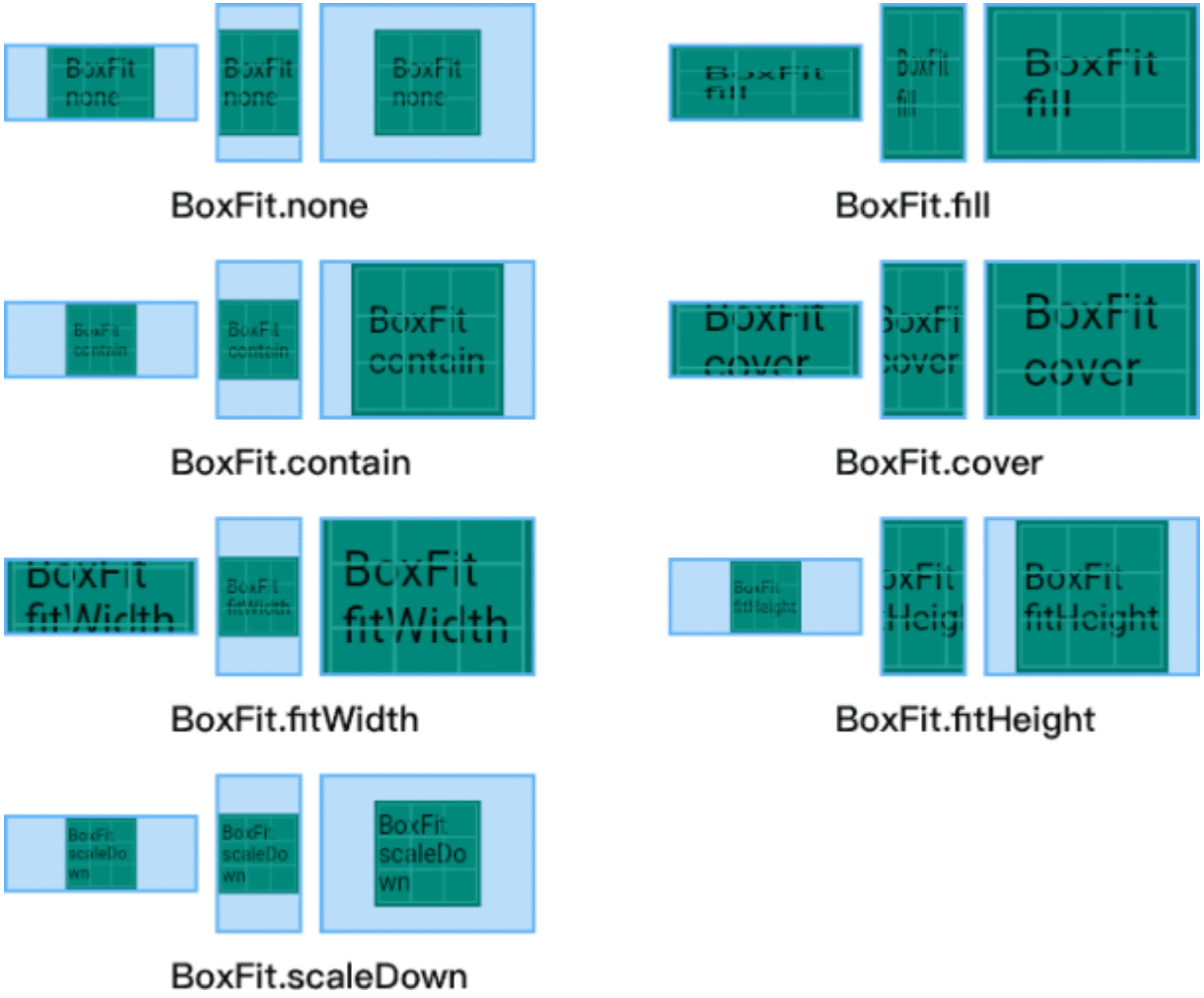
Aug 26 2019 · Article (15 mins) · Beg




Responsive Design for Flutter: Getting Started

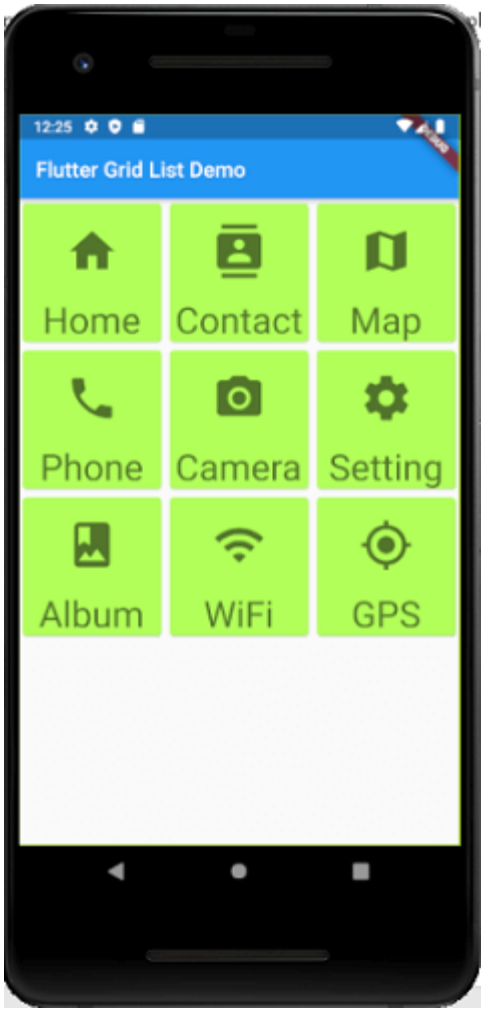
https://www.raywenderlich.com/4324124-responsive-design-for-flutter-getting-started



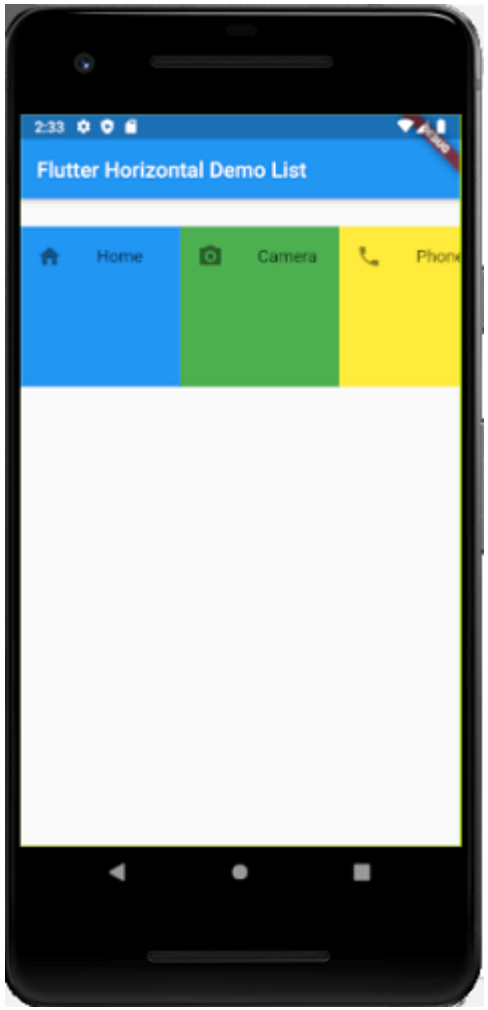

Responsive Design for Flutter: Getting Started
https://www.raywenderlich.com/4324124-responsive-design-for-flutter-getting-started

# Not addressed

Changing orientation




Custom layouts




Scrollable contents

GridView

# Scrolling widgets

Docs > Development > UI > Widgets > Scrolling


Scroll multiple widgets as children of the parent.

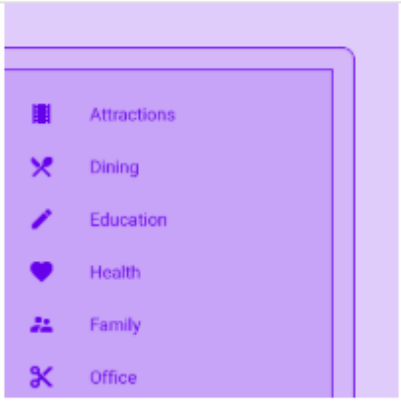
See more widgets in the widget catalog.

## CustomScrollView

A ScrollView that creates custom scroll effects using slivers.

## GridView

A grid list consists of a repeated pattern of cells arrayed in a vertical and horizontal layout. The GridView widget implements this




## ListView

A scrollable, linear list of widgets. ListView is the most commonly used scrolling widget. It displays its children one after another in the

axis,

ill the

ListView.

https://flutter.dev/docs/development/ui/widgets/scrolling
https://api.flutter.dev/flutter/widgets/CustomScrollView-class.html

# The END