Layouts in Flutter

Docs Development UI Layout

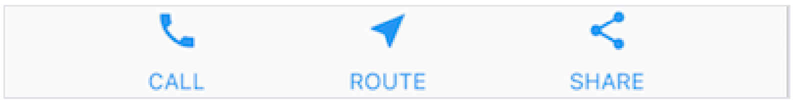
Contents

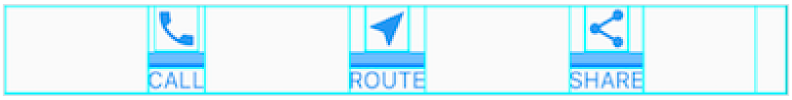
What's the point?

- 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:

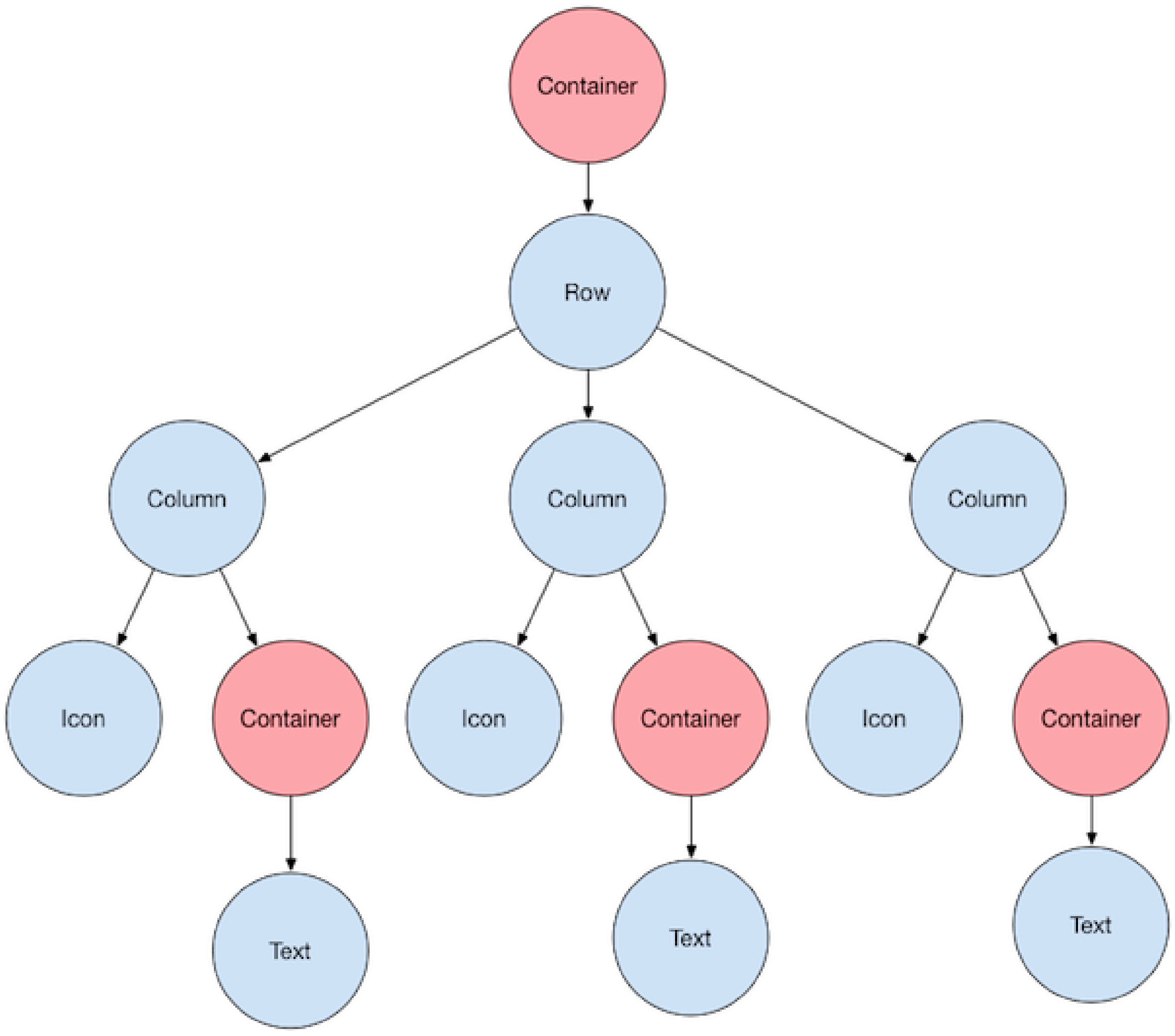




The second screenshot displays the visual layout, showing a row of 3 columns where each column contains an icon and a label.

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 should look as you might expect, but you might be wondering about the containers (shown in pink). `Container` is a widget class that allows you to customize its child widget. Use a `Container` when you want to add padding, margins, borders, or background color, to name some of its capabilities.

In this example, each `Text` widget is placed in a `Container` to add margins. The entire `Row` is also placed in a `Container` to add padding around the row.

The rest of the UI in this example is controlled by properties. Set an `Icon`’s color using its `color` property. Use the `Text.style`property to set the font, its color, weight, and so on. Columns and rows have properties that allow you to specify how their children are aligned vertically or horizontally, and how much space the children should occupy.

Lay out a widget

How do you lay out a single widget in Flutter? This section shows you how to create and display a simple widget. It also shows the entire code for a simple Hello World app.

In Flutter, it takes only a few steps to put text, an icon, or an image on the screen.

1. Select a layout widget

Choose from a variety of layout widgets based on how you want to align or constrain the visible widget, as these characteristics are typically passed on to the contained widget.

This example uses `Center` which centers its content horizontally and vertically.

2. Create a visible widget

For example, create a `Text` widget:

```
Text('Hello World'),
```

content_copy

Create an `Image` widget:

```
Image.asset(
  'images/lake.jpg',
  fit: BoxFit.cover,
),
```

content_copy

Create an `Icon` widget:

```
Icon(
  Icons.star,
  color: Colors.red[500],
),
```

content_copy

3. Add the visible widget to the layout widget

All layout widgets have either of the following:

- A `child` property if they take a single child—for example, `Center` or `Container`
- A `children` property if they take a list of widgets—for example, `Row`, `Column`, `ListView`, or `Stack`.

Add the `Text` widget to the `Center` widget:

```
const Center(
  child: Text('Hello World'),
),
```

content_copy

4. Add the layout widget to the page

A Flutter app is itself a widget, and most widgets have a `build()` method. Instantiating and returning a widget in the app's `build()`method displays the widget.

Material apps

For a `Material` app, you can use a [`Scaffold`]() widget; it provides a default banner, background color, and has API for adding drawers, snack bars, and bottom sheets. Then you can add the `Center` widget directly to the `body` property for the home page.

lib/main.dart (MyApp)

content_copy

```dart
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter layout demo',
      home: Scaffold(
        appBar: AppBar(
          title: const Text('Flutter layout demo'),
        ),
        body: const Center(
          child: Text('Hello World'),
        ),
      ),
    );
  }
}
```

Note: The Material library implements widgets that follow Material Design principles. When designing your UI, you can exclusively use widgets from the standard widgets library, or you can use widgets from the Material library. You can mix widgets from both libraries, you can customize existing widgets, or you can build your own set of custom widgets.

Non-Material apps

For a non-Material app, you can add the `Center` widget to the app's `build()` method:

lib/main.dart (MyApp)

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

  @override
  Widget build(BuildContext context) {
    return Container(
      decoration: const BoxDecoration(color: Colors.white),
      child: const Center(
        child: Text(
          'Hello World',
          textDirection: TextDirection.ltr,
          style: TextStyle(
            fontSize: 32,
            color: Colors.black87,
          ),
        ),
      ),
    );
  }
}
```

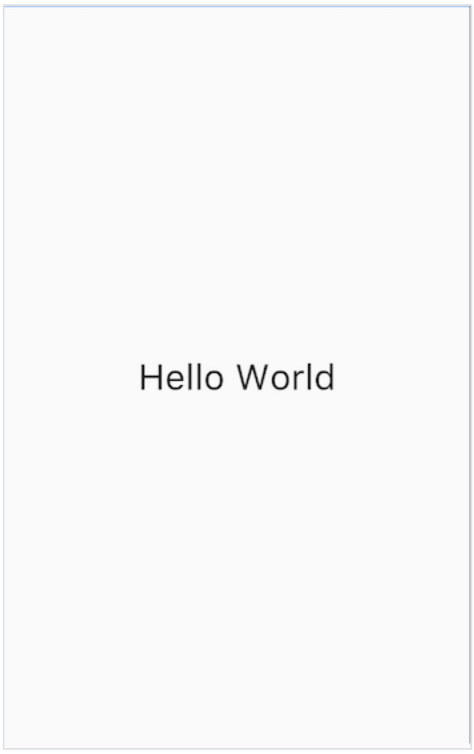
content_copy

By default a non-Material app doesn't include an `AppBar`, title, or background color. If you want these features in a non-Material app, you have to build them yourself. This app changes the background color to white and the text to dark grey to mimic a Material app.

That's it! When you run the app, you should see *Hello World*.

App source code:

- Material app
- Non-Material app



Lay out multiple widgets vertically and horizontally

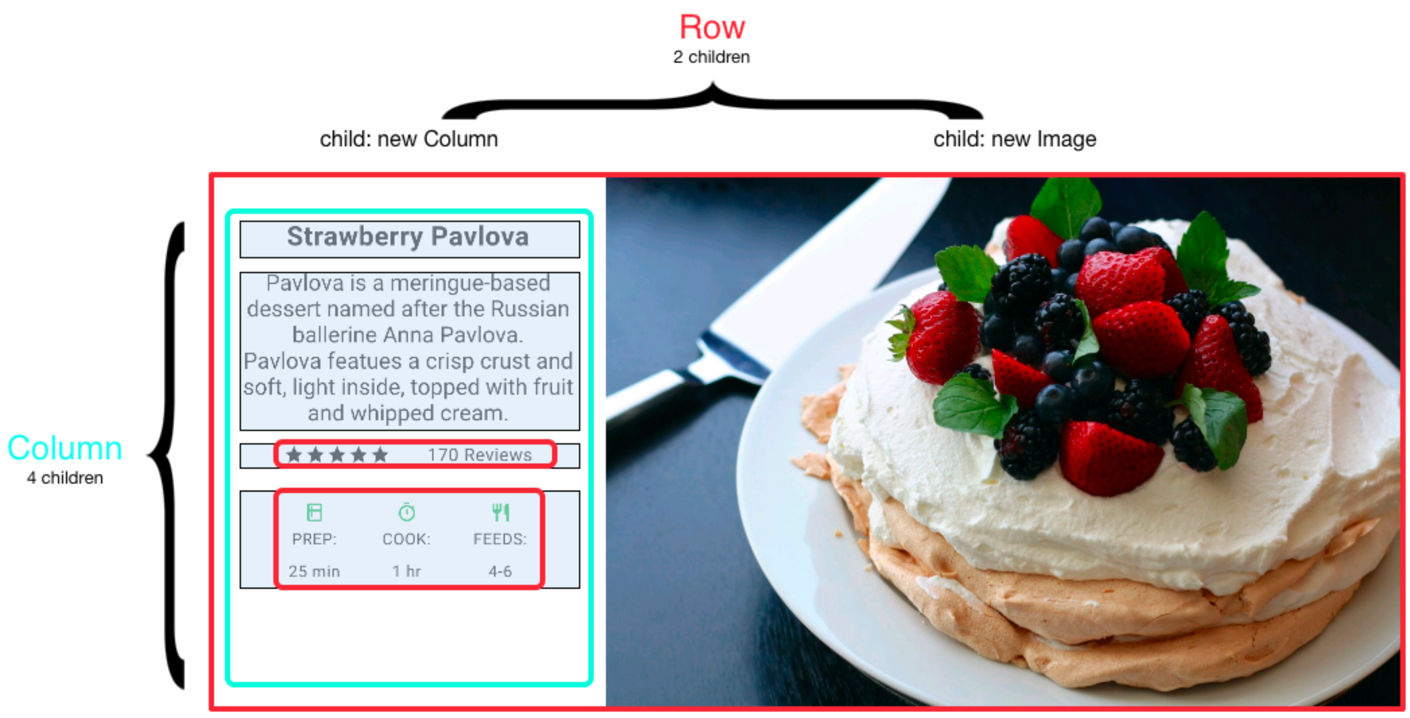
One of the most common layout patterns is to arrange widgets vertically or horizontally. You can use a `Row` widget to arrange widgets horizontally, and a `Column` widget to arrange widgets vertically.

What's the point?

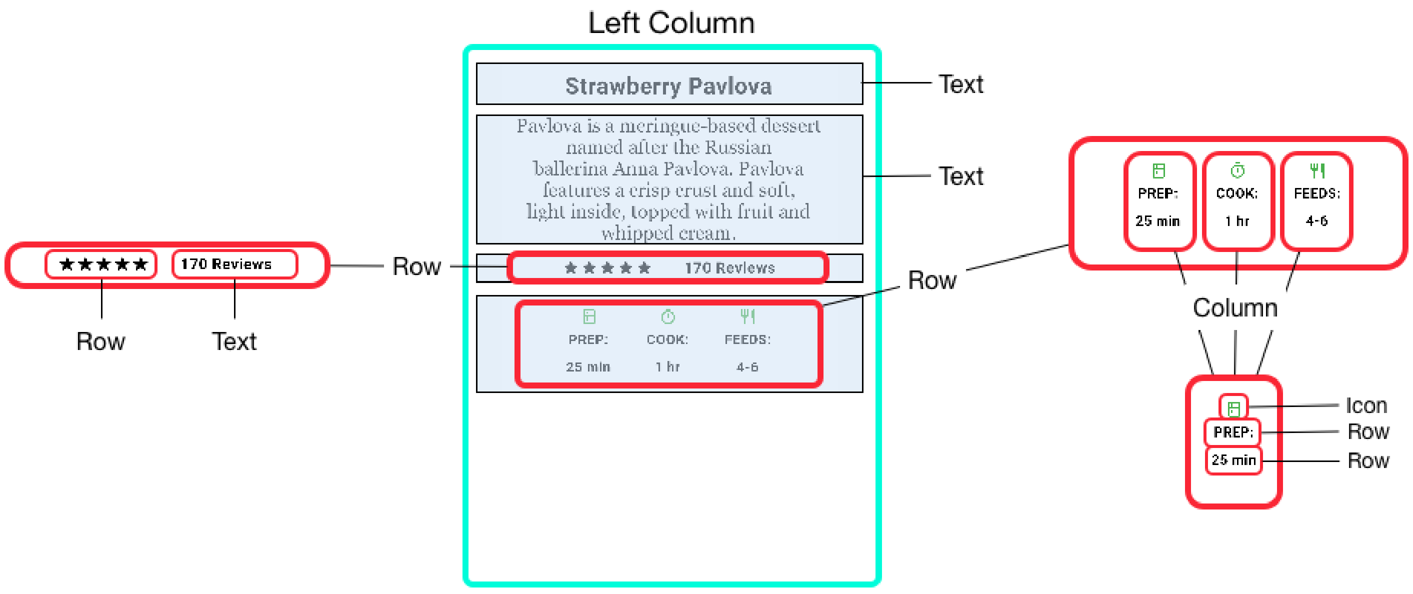
- `Row` and `Column` are two of the most commonly used layout patterns.
- `Row` and `Column` each take a list of child widgets.
- A child widget can itself be a `Row`, `Column`, or other complex widget.
- You can specify how a `Row` or `Column` aligns its children, both vertically and horizontally.
- You can stretch or constrain specific child widgets.
- You can specify how child widgets use the `Row`’s or `Column`’s available space.

To create a row or column in Flutter, you add a list of children widgets to a `Row` or `Column` widget. In turn, each child can itself be a row or column, and so on. The following example shows how it is possible to nest rows or columns inside of rows or columns.

This layout is organized as a `Row`. The row contains two children: a column on the left, and an image on the right:


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

The left column’s widget tree nests rows and columns.

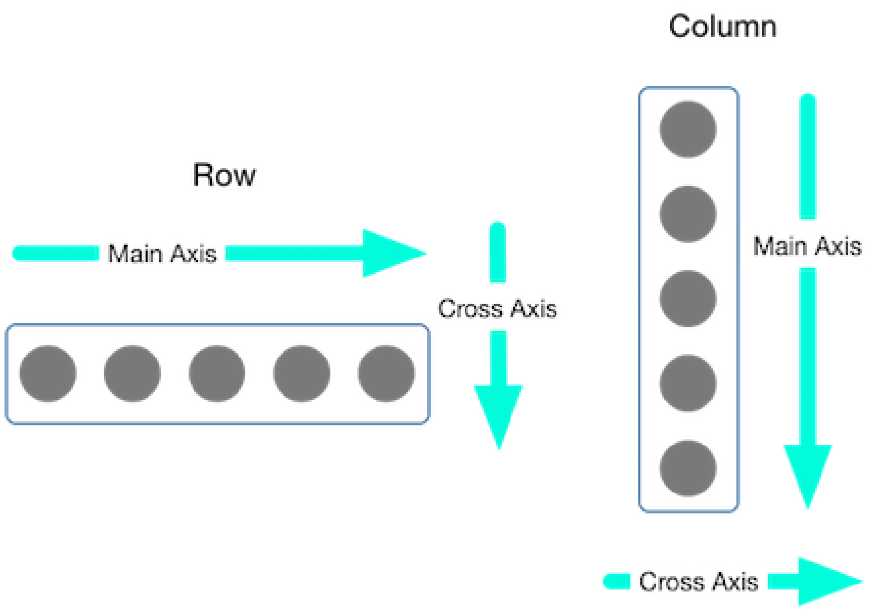

Left Column
Strawberry Pavlova
Pavlova is a meringue-based dessert named after the Russian ballerina Anna Pavlova. Pavlova features a crisp crust and soft, light inside, topped with fruit and whipped cream.
★★★★★ 170 Reviews
PREP: 25 min
COOK: 1 hr
FEEDS: 4-6
Text
Text
Row
Row
Row
Text
Column
Icon
Row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

You’ll implement some of Pavlova’s layout code in Nesting rows and columns.

Note: `Row` and `Column` are basic primitive widgets for horizontal and vertical layouts—these low-level widgets allow for maximum customization. Flutter also offers specialized, higher level widgets that might be sufficient for your needs. For example, instead of `Row` you might prefer [`ListTile`](), an easy-to-use widget with properties for leading and trailing icons, and up to 3 lines of text. Instead of Column, you might prefer [`ListView`](), a column-like layout that automatically scrolls if its content is too long to fit the available space. For more information, see [Common layout widgets]().

Aligning widgets

You control how a row or column aligns its children using the `mainAxisAlignment` and `crossAxisAlignment` properties. For a row, the main axis runs horizontally and the cross axis runs vertically. For a column, the main axis runs vertically and the cross axis runs horizontally.



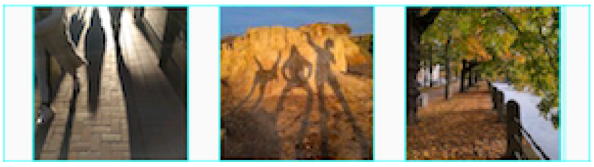
The `MainAxisAlignment` and `CrossAxisAlignment` classes offer a variety of constants for controlling alignment.

Note: When you add images to your project, you need to update the `pubspec.yaml` file to access them—this example uses `Image.asset` to display the images. For more information, see this example's `pubspec.yaml` file or Adding assets and images. You don't need to do this if you're referencing online images using `Image.network`.

In the following example, each of the 3 images is 100 pixels wide. The render box (in this case, the entire screen) is more than 300 pixels wide, so setting the main axis alignment to `spaceEvenly` divides the free horizontal space evenly between, before, and after each image.

content_copy

```
Row(
  mainAxisAlignment: MainAxisAlignment.spaceEvenly,
  children: [
    Image.asset('images/pic1.jpg'),
    Image.asset('images/pic2.jpg'),
    Image.asset('images/pic3.jpg'),
  ],
);
```



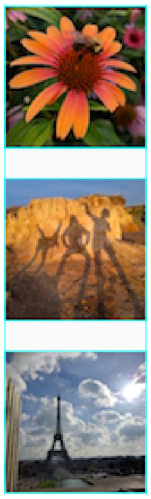
App source: row_column

Columns work the same way as rows. The following example shows a column of 3 images, each is 100 pixels high. The height of the render box (in this case, the entire screen) is more than 300 pixels, so setting the main axis alignment to `spaceEvenly` divides the free vertical space evenly between, above, and below each image.

```
Column(
  mainAxisAlignment: MainAxisAlignment.spaceEvenly,
  children: [
    Image.asset('images/pic1.jpg'),
    Image.asset('images/pic2.jpg'),
    Image.asset('images/pic3.jpg'),
  ],
);
```

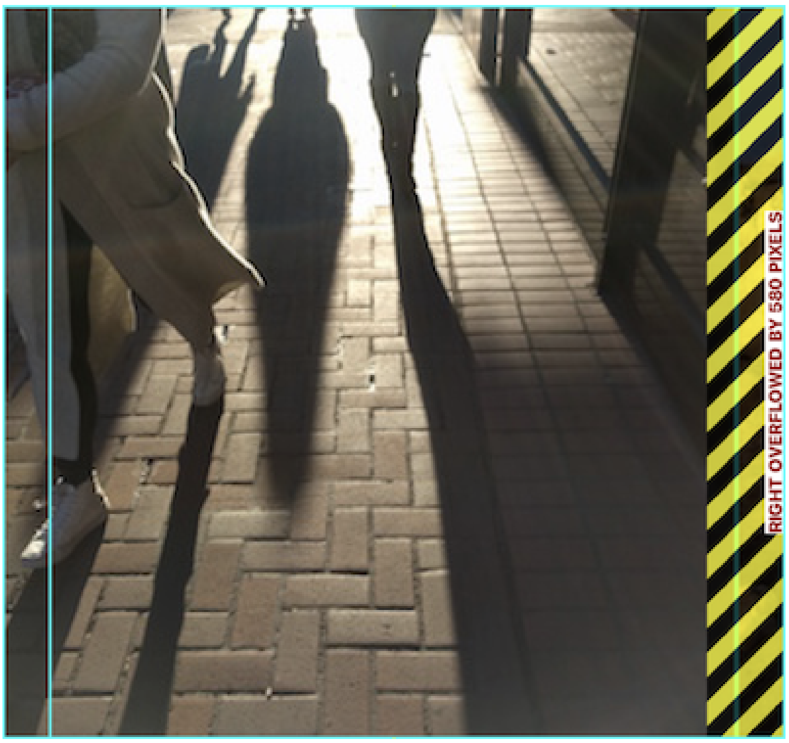
content_copy

App source: row_column



Sizing widgets

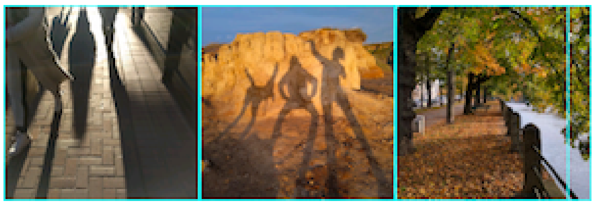
When a layout is too large to fit a device, a yellow and black striped pattern appears along the affected edge. Here is an example of a row that is too wide:



Widgets can be sized to fit within a row or column by using the Expanded widget. To fix the previous example where the row of images is too wide for its render box, wrap each image with an `Expanded` widget.

content_copy

```
Row(
  crossAxisAlignment: CrossAxisAlignment.center,
  children: [
    Expanded(
      child: Image.asset('images/pic1.jpg'),
    ),
    Expanded(
      child: Image.asset('images/pic2.jpg'),
    ),
    Expanded(
      child: Image.asset('images/pic3.jpg'),
    ),
  ],
);
```

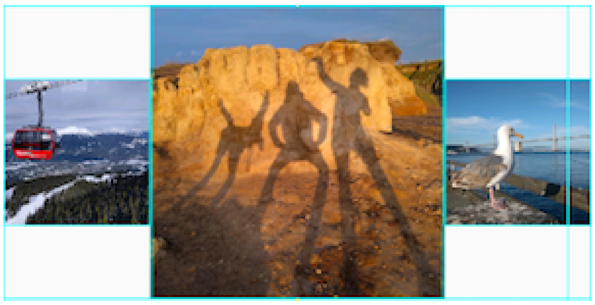


App source: sizing

Perhaps you want a widget to occupy twice as much space as its siblings. For this, use the `Expanded` widget `flex` property, an integer that determines the flex factor for a widget. The default flex factor is 1. The following code sets the flex factor of the middle image to 2:

content_copy

```
Row(
  crossAxisAlignment: CrossAxisAlignment.center,
  children: [
    Expanded(
      child: Image.asset('images/pic1.jpg'),
    ),
    Expanded(
      flex: 2,
      child: Image.asset('images/pic2.jpg'),
    ),
    Expanded(
      child: Image.asset('images/pic3.jpg'),
    ),
  ],
);
```

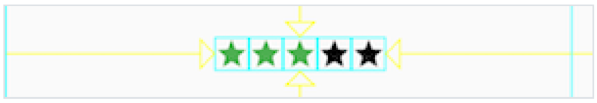


App source: [sizing]()

Packing widgets

By default, a row or column occupies as much space along its main axis as possible, but if you want to pack the children closely together, set its `mainAxisSize` to `MainAxisSize.min`. The following example uses this property to pack the star icons together.

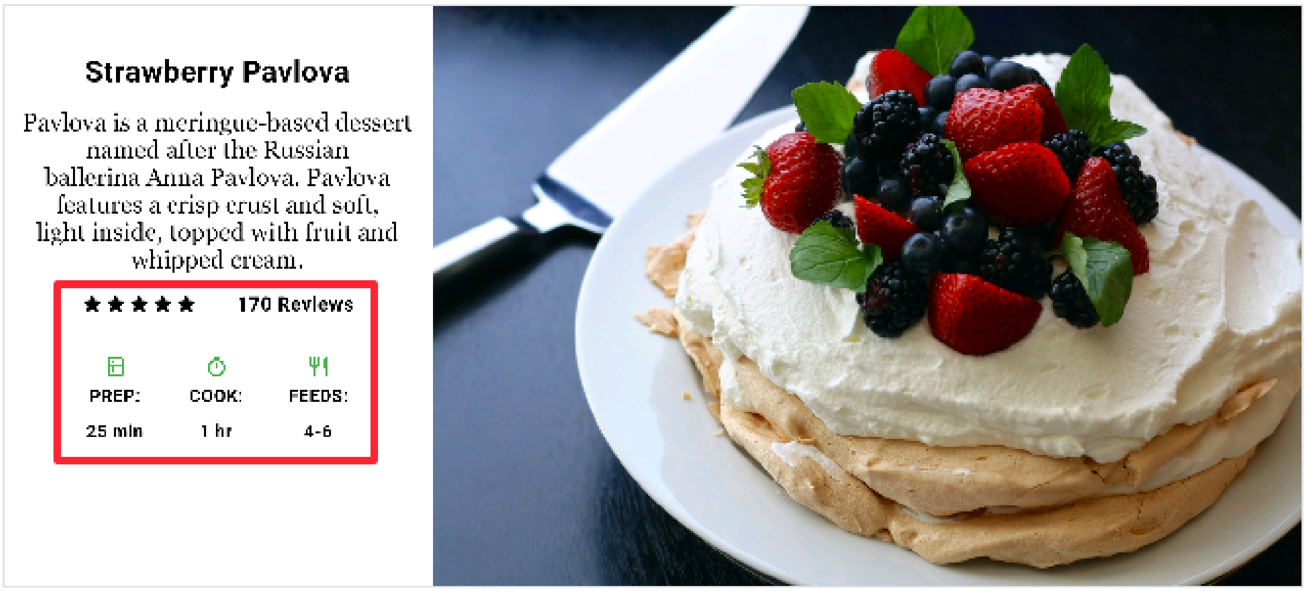
```
Row(                                                    content_copy
  mainAxisSize: MainAxisSize.min,
  children: [
    Icon(Icons.star, color: Colors.green[500]),
    Icon(Icons.star, color: Colors.green[500]),
    Icon(Icons.star, color: Colors.green[500]),
    const Icon(Icons.star, color: Colors.black),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



App source: [pavlova]()

Nesting rows and columns

The layout framework allows you to nest rows and columns inside of rows and columns as deeply as you need. Let’s look at the code for the outlined section of the following layout:



Strawberry Pavlova

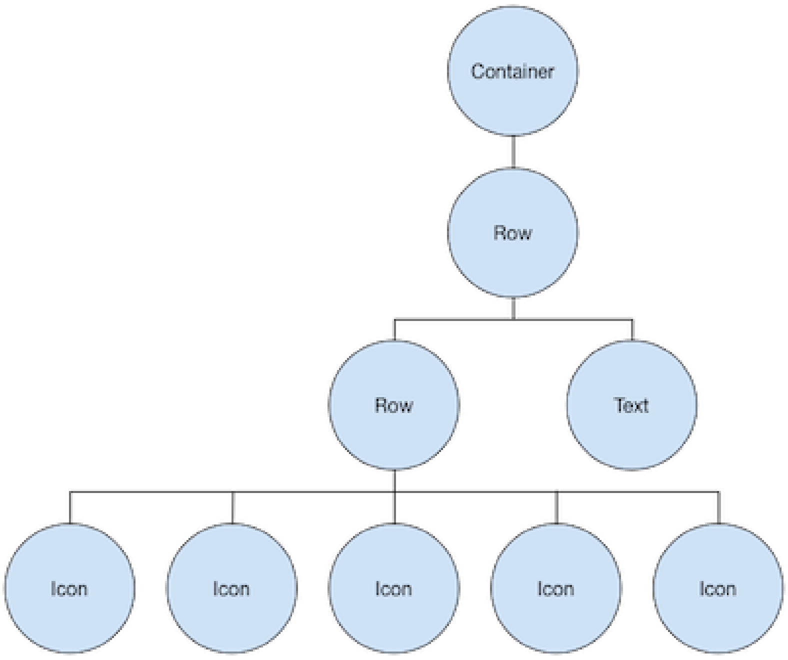
Pavlova is a meringue-based dessert named after the Russian ballerina Anna Pavlova. Pavlova features a crisp crust and soft, light inside, topped with fruit and whipped cream.

★★★★★ 170 Reviews

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

The outlined section is implemented as two rows. The ratings row contains five stars and the number of reviews. The icons row contains three columns of icons and text.

The widget tree for the ratings row:



The `ratings` variable creates a row containing a smaller row of 5 star icons, and text:

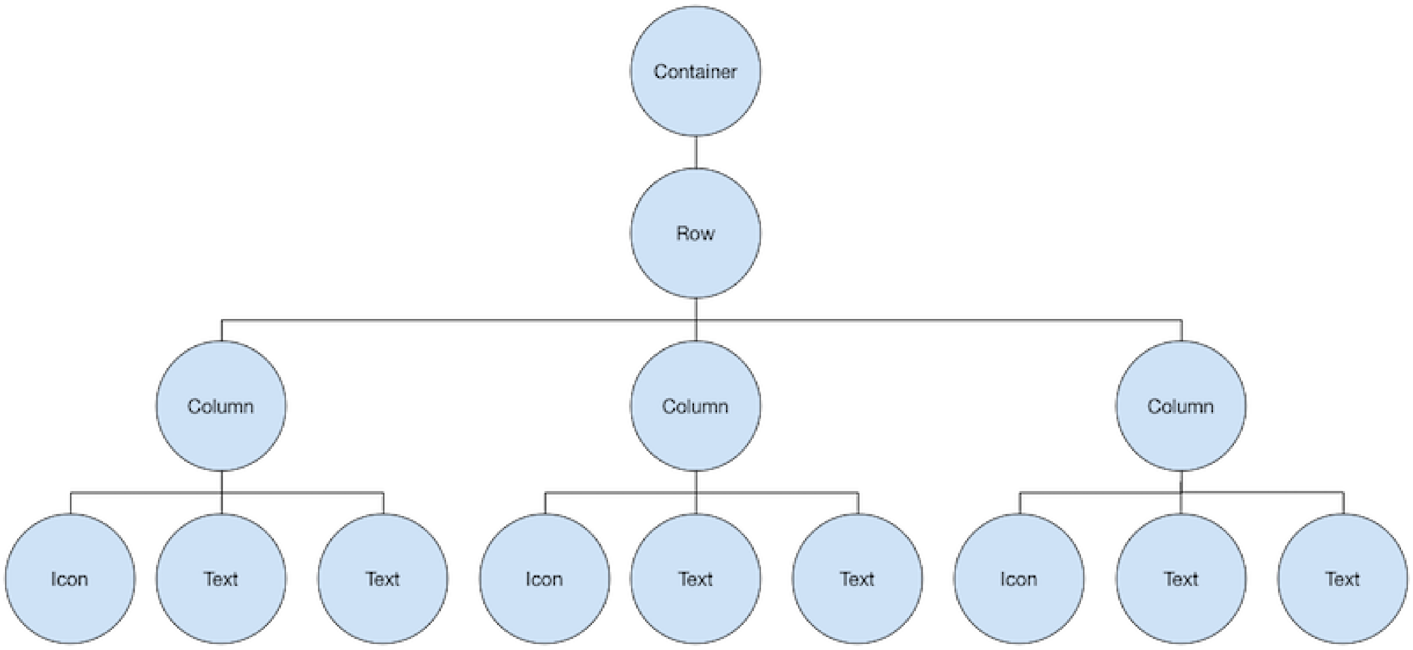
content_copy

```
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),
  ],
);

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,
        ),
      ),
    ],
  ),
);
```

Tip: To minimize the visual confusion that can result from heavily nested layout code, implement pieces of the UI in variables and functions.

The icons row, below the ratings row, contains 3 columns; each column contains an icon and two lines of text, as you can see in its widget tree:



The `iconList` variable defines the icons row:

```
const descTextStyle = TextStyle(
  color: Colors.black,
  fontWeight: FontWeight.w800,
  fontFamily: 'Roboto',
  letterSpacing: 0.5,
  fontSize: 18,
  height: 2,
);

// DefaultTextStyle.merge() allows you to create a default text
// style that is inherited by its child and all subsequent children.
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'),
          ],
        ),
      ],
    ),
  ),
);
```
content_copy

The `leftColumn` variable contains the ratings and icons rows, as well as the title and text that describes the Pavlova:

```
final leftColumn = Container(
  padding: const EdgeInsets.fromLTRB(20, 30, 20, 20),
  child: Column(
    children: [
      titleText,
      subTitle,
      ratings,
      iconList,
    ],
  ),
);
```
content_copy

The left column is placed in a `Container` to constrain its width. Finally, the UI is constructed with the entire row (containing the left column and the image) inside a `Card`.

The Pavlova image is from Pixabay. You can embed an image from the net using `Image.network()` but, for this example, the image is saved to an images directory in the project, added to the pubspec file, and accessed using `Images.asset()`. For more information, see Adding assets and images.

content_copy

```
body: Center(
  child: Container(
    margin: const EdgeInsets.fromLTRB(0, 40, 0, 30),
    height: 600,
    child: Card(
      child: Row(
        crossAxisAlignment: CrossAxisAlignment.start,
        children: [
          SizedBox(
            width: 440,
            child: leftColumn,
          ),
          mainImage,
        ],
      ),
    ),
  ),
),
```

Tip: The Pavlova example runs best horizontally on a wide device, such as a tablet. If you are running this example in the iOS simulator, you can select a different device using the **Hardware > Device** menu. For this example, we recommend the iPad Pro. You can change its orientation to landscape mode using **Hardware > Rotate**. You can also change the size of the simulator window (without changing the number of logical pixels) using **Window > Scale**.

App source: pavlova

Common layout widgets

Flutter has a rich library of layout widgets. Here are a few of those most commonly used. The intent is to get you up and running as quickly as possible, rather than overwhelm you with a complete list. For information on other available widgets, refer to the Widget catalog, or use the Search box in the API reference docs. Also, the widget pages in the API docs often make suggestions about similar widgets that might better suit your needs.

The following widgets fall into two categories: standard widgets from the widgets library, and specialized widgets from the Material library. Any app can use the widgets library but only Material apps can use the Material Components library.

Standard widgets

- `Container`: Adds padding, margins, borders, background color, or other decorations to a widget.
- `GridView`: Lays widgets out as a scrollable grid.
- `ListView`: Lays widgets out as a scrollable list.
- `Stack`: Overlaps a widget on top of another.

Material widgets

- `Card`: Organizes related info into a box with rounded corners and a drop shadow.
- `ListTile`: Organizes up to 3 lines of text, and optional leading and trailing icons, into a row.

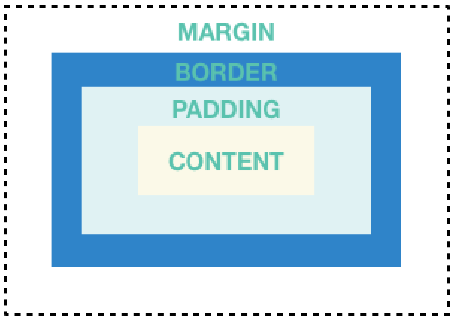
Container

Many layouts make liberal use of `Container`s to separate widgets using padding, or to add borders or margins. You can change the device’s background by placing the entire layout into a `Container` and changing its background color or image.

Summary (Container)

- Add padding, margins, borders
- Change background color or image

- Contains a single child widget, but that child can be a Row, Column, or even the root of a widget tree

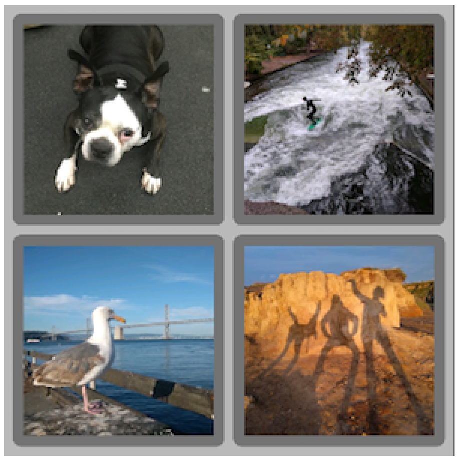


Examples (Container)

This layout consists of a column with two rows, each containing 2 images. A `Container` is used to change the background color of the column to a lighter grey.

content_copy

```
Widget _buildImageColumn() {
  return Container(
    decoration: const BoxDecoration(
      color: Colors.black26,
    ),
    child: Column(
      children: [
        _buildImageRow(1),
        _buildImageRow(3),
      ],
    ),
  );
}
```



A `Container` is also used to add a rounded border and margins to each image:

```
Widget _buildDecoratedImage(int imageIndex) => Expanded(
      child: Container(
        decoration: BoxDecoration(
          border: Border.all(width: 10, color: Colors.black38),
          borderRadius: const BorderRadius.all(Radius.circular(8)),
        ),
        margin: const EdgeInsets.all(4),
        child: Image.asset('images/pic$imageIndex.jpg'),
      ),
    );

Widget _buildImageRow(int imageIndex) => Row(
      children: [
        _buildDecoratedImage(imageIndex),
        _buildDecoratedImage(imageIndex + 1),
      ],
    );
```

content_copy

You can find more `Container` examples in the tutorial and the Flutter Gallery (running app, repo).

App source: container

GridView

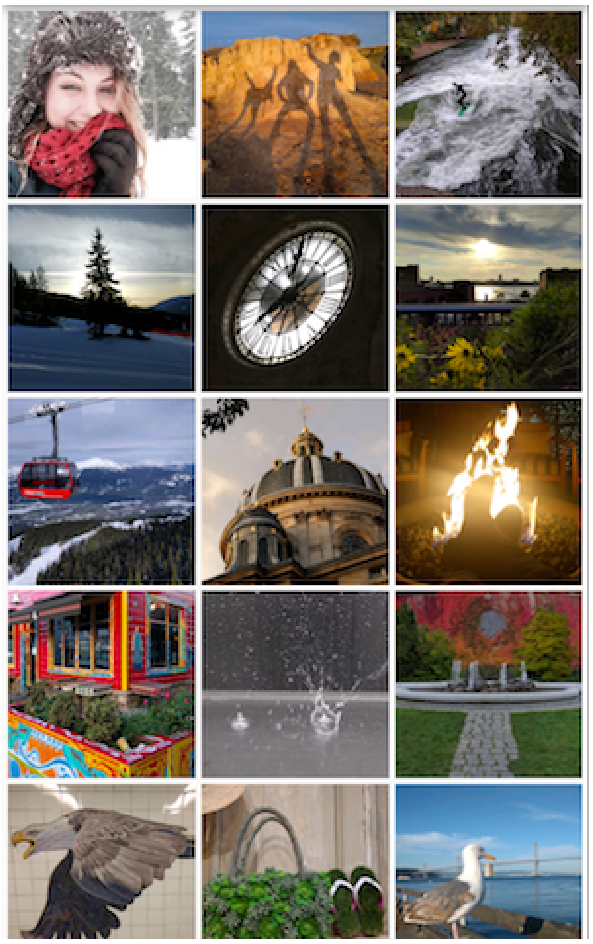
Use `GridView` to lay widgets out as a two-dimensional list. `GridView` provides two pre-fabricated lists, or you can build your own custom grid. When a `GridView` detects that its contents are too long to fit the render box, it automatically scrolls.

Summary (GridView)

- Lays widgets out in a grid
- Detects when the column content exceeds the render box and automatically provides scrolling
- Build your own custom grid, or use one of the provided grids:
  - `GridView.count` allows you to specify the number of columns
  - `GridView.extent` allows you to specify the maximum pixel width of a tile

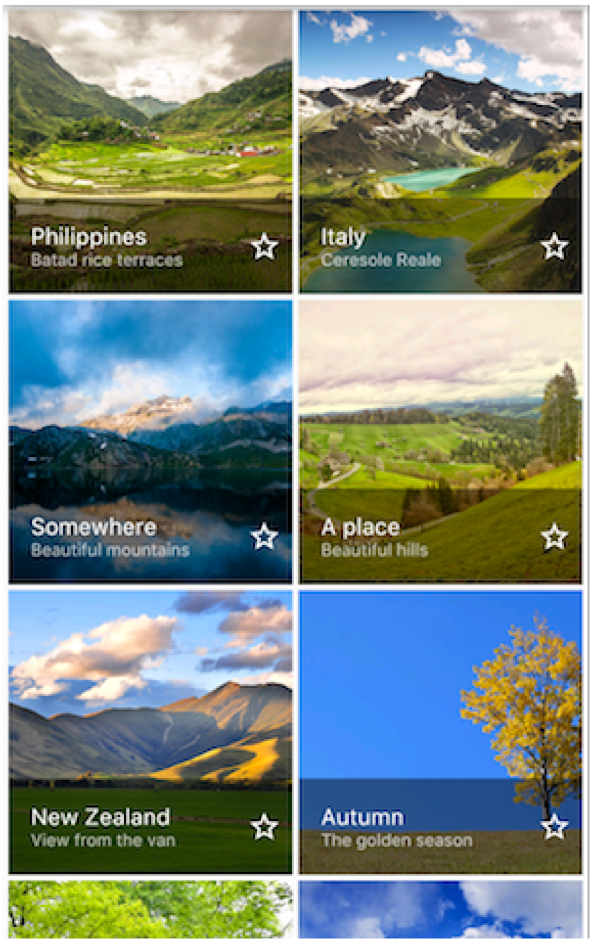
Note: When displaying a two-dimensional list where it's important which row and column a cell occupies (for example, it's the entry in the "calorie" column for the "avocado" row), use `Table` or `DataTable`.

Examples (GridView)



Uses `GridView.extent` to create a grid with tiles a maximum 150 pixels wide.

App source: [grid_and_list]



Uses `GridView.count` to create a grid that's 2 tiles wide in portrait mode, and 3 tiles wide in landscape mode. The titles are created by setting the `footer` property for each `GridTile`.

Dart code: grid_list_demo.dart from the Flutter Gallery

content_copy

```
Widget _buildGrid() => GridView.extent(
    maxCrossAxisExtent: 150,
    padding: const EdgeInsets.all(4),
    mainAxisSpacing: 4,
    crossAxisSpacing: 4,
    children: _buildGridTileList(30));

// The images are saved with names pic0.jpg, pic1.jpg...pic29.jpg.
// The List.generate() constructor allows an easy way to create
// a list when objects have a predictable naming pattern.
List<Container> _buildGridTileList(int count) => List.generate(
    count, (i) => Container(child: Image.asset('images/pic$i.jpg')));
```

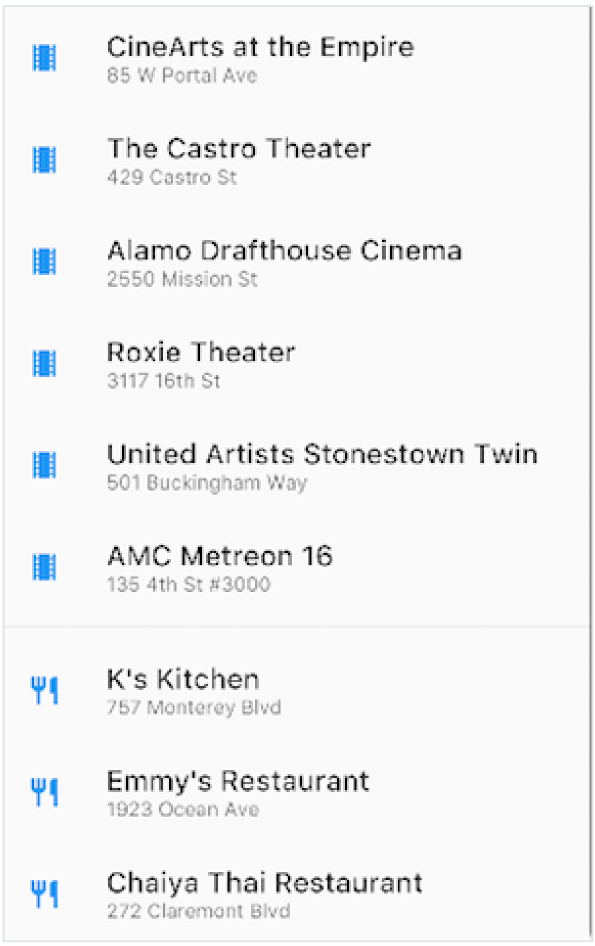
ListView

`ListView`, a column-like widget, automatically provides scrolling when its content is too long for its render box.

Summary (ListView)

- A specialized `Column` for organizing a list of boxes
- Can be laid out horizontally or vertically
- Detects when its content won't fit and provides scrolling
- Less configurable than `Column`, but easier to use and supports scrolling

Examples (ListView)



CineArts at the Empire
85 W Portal Ave

The Castro Theater
429 Castro St

Alamo Drafthouse Cinema
2550 Mission St

Roxie Theater
3117 16th St

United Artists Stonestown Twin
501 Buckingham Way

AMC Metreon 16
135 4th St #3000

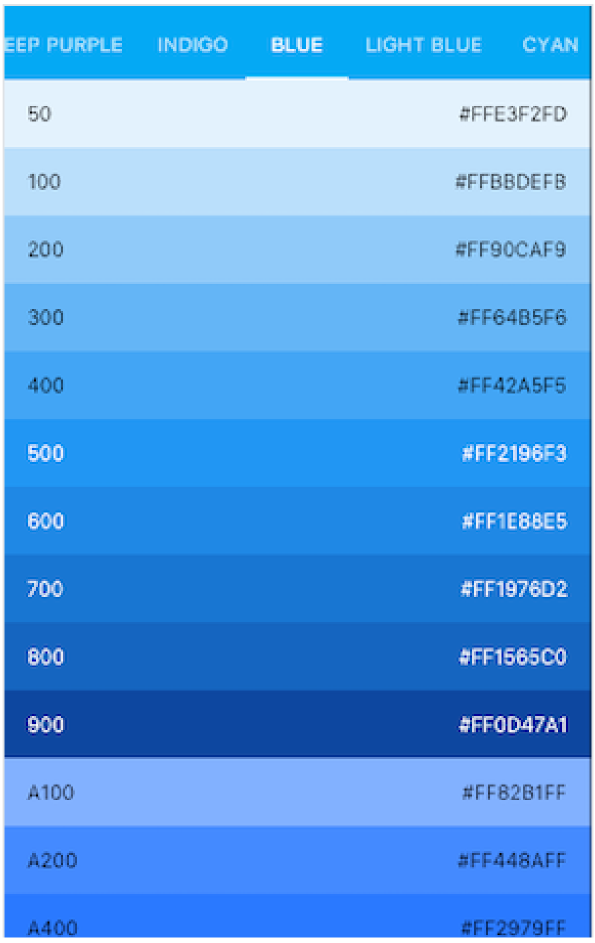
K's Kitchen
757 Monterey Blvd

Emmy's Restaurant
1923 Ocean Ave

Chaiya Thai Restaurant
272 Claremont Blvd

Uses `ListView` to display a list of businesses using `ListTile`s. A `Divider` separates the theaters from the restaurants.

App source: grid_and_list



50	#FFE3F2FD
100	#FFBBDEFB
200	#FF90CAF9
300	#FF64B5F6
400	#FF42A5F5
500	#FF2196F3
600	#FF1E88E5
700	#FF1976D2
800	#FF1565C0
900	#FF0D47A1
A100	#FF82B1FF
A200	#FF448AFF
A400	#FF2979FF

Uses `ListView` to display the `Colors` from the Material Design palette for a particular color family.

Dart code: colors_demo.dart from the Flutter Gallery

content_copy

```
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),
    ],
  );
}

ListTile _tile(String title, String subtitle, IconData icon) {
  return ListTile(
    title: Text(title,
        style: const TextStyle(
          fontWeight: FontWeight.w500,
          fontSize: 20,
        )),
    subtitle: Text(subtitle),
    leading: Icon(
      icon,
      color: Colors.blue[500],
    ),
  );
}
```

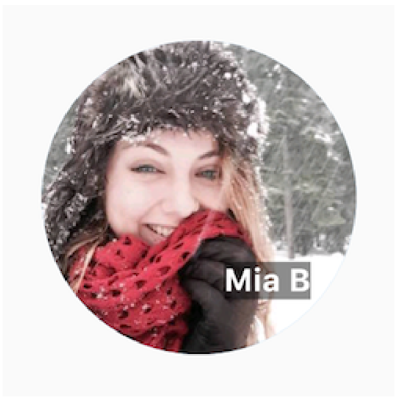
Stack

Use `Stack` to arrange widgets on top of a base widget—often an image. The widgets can completely or partially overlap the base widget.

Summary (Stack)

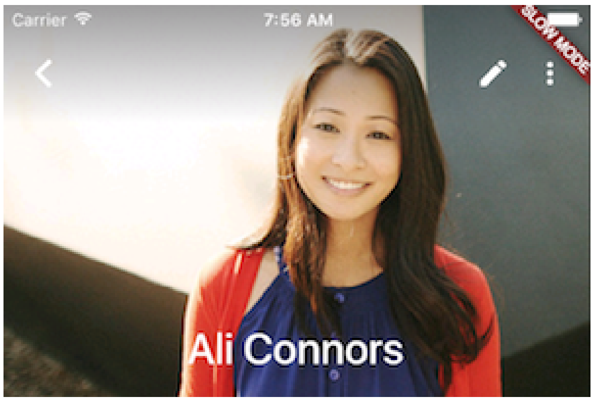
- Use for widgets that overlap another widget
- The first widget in the list of children is the base widget; subsequent children are overlaid on top of that base widget
- A `Stack`'s content can't scroll
- You can choose to clip children that exceed the render box

Examples (Stack)



Uses `Stack` to overlay a `Container` (that displays its `Text` on a translucent black background) on top of a `CircleAvatar`. The `Stack`offsets the text using the `alignment` property and `Alignment`s.

App source: card_and_stack



Uses `Stack` to overlay a gradient to the top of the image. The gradient ensures that the toolbar's icons are distinct against the image.

Dart code: contacts_demo.dart from the Flutter Gallery

content_copy

```
Widget _buildStack() {
  return Stack(
    alignment: const Alignment(0.6, 0.6),
    children: [
      const CircleAvatar(
        backgroundImage: AssetImage('images/pic.jpg'),
        radius: 100,
      ),
      Container(
        decoration: const BoxDecoration(
          color: Colors.black45,
        ),
        child: const Text(
          'Mia B',
          style: TextStyle(
            fontSize: 20,
            fontWeight: FontWeight.bold,
            color: Colors.white,
          ),
        ),
      ),
    ],
  );
}
```

Card

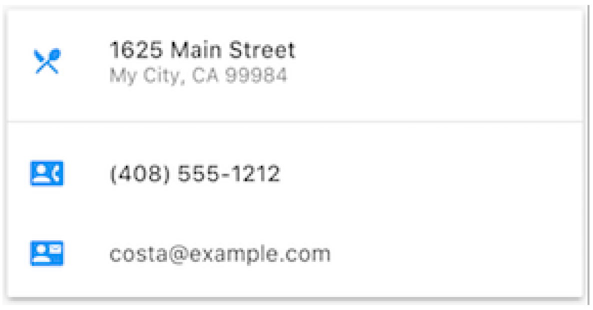
A `Card`, from the Material library, contains related nuggets of information and can be composed from almost any widget, but is often used with `ListTile`. `Card` has a single child, but its child can be a column, row, list, grid, or other widget that supports multiple children. By default, a `Card` shrinks its size to 0 by 0 pixels. You can use `SizedBox` to constrain the size of a card.

In Flutter, a `Card` features slightly rounded corners and a drop shadow, giving it a 3D effect. Changing a `Card`'s `elevation` property allows you to control the drop shadow effect. Setting the elevation to 24, for example, visually lifts the `Card` further from the surface and causes the shadow to become more dispersed. For a list of supported elevation values, see Elevation in the Material guidelines. Specifying an unsupported value disables the drop shadow entirely.

Summary (Card)

- Implements a Material card
- Used for presenting related nuggets of information
- Accepts a single child, but that child can be a `Row`, `Column`, or other widget that holds a list of children
- Displayed with rounded corners and a drop shadow
- A `Card`'s content can't scroll
- From the Material library

Examples (Card)



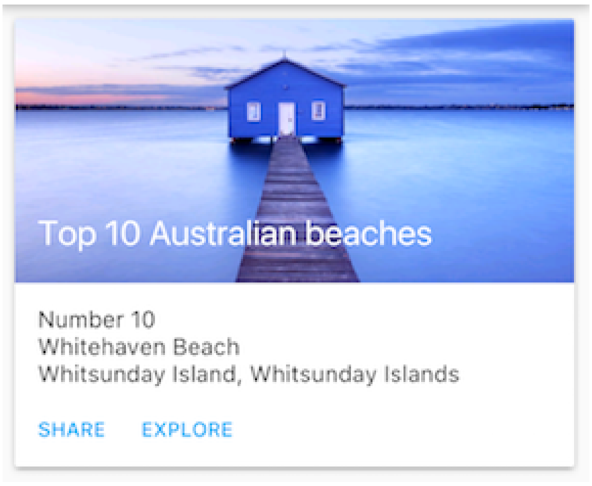
1625 Main Street
My City, CA 99984

(408) 555-1212

costa@example.com

A `Card` containing 3 ListTiles and sized by wrapping it with a `SizedBox`. A `Divider` separates the first and second `ListTiles`.

App source: card_and_stack



Number 10
Whitehaven Beach
Whitsunday Island, Whitsunday Islands

SHARE EXPLORE

A `Card` containing an image and text.

A Card containing an image and text.

Dart code: cards_demo.dart from the Flutter Gallery

content_copy

```
Widget _buildCard() {
  return SizedBox(
    height: 210,
    child: Card(
      child: Column(
        children: [
          ListTile(
            title: const Text(
              '1625 Main Street',
              style: TextStyle(fontWeight: FontWeight.w500),
            ),
            subtitle: const Text('My City, CA 99984'),
            leading: Icon(
              Icons.restaurant_menu,
              color: Colors.blue[500],
            ),
          ),
          const Divider(),
          ListTile(
            title: const Text(
              '(408) 555-1212',
              style: TextStyle(fontWeight: FontWeight.w500),
            ),
            leading: Icon(
              Icons.contact_phone,
              color: Colors.blue[500],
            ),
          ),
          ListTile(
            title: const Text('costa@example.com'),
            leading: Icon(
              Icons.contact_mail,
              color: Colors.blue[500],
            ),
          ),
        ],
      ),
    ),
  );
}
```

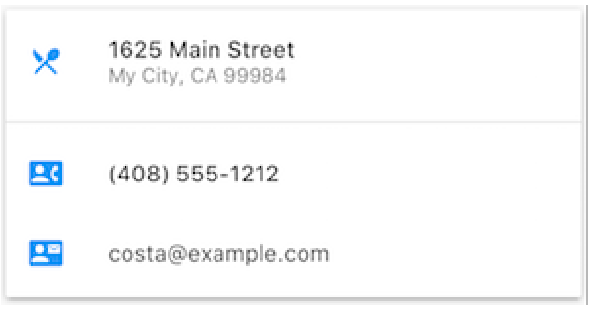
ListTile

Use `ListTile`, a specialized row widget from the Material library, for an easy way to create a row containing up to 3 lines of text and optional leading and trailing icons. `ListTile` is most commonly used in `Card` or `ListView`, but can be used elsewhere.

Summary (ListTile)

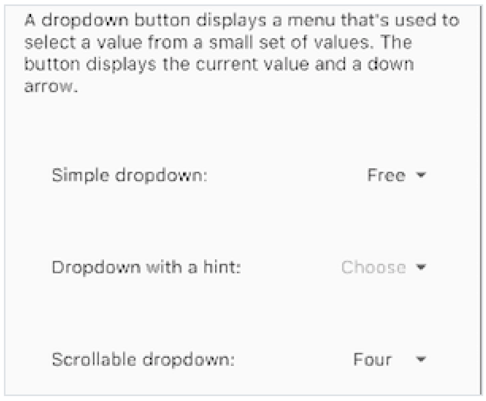
- A specialized row that contains up to 3 lines of text and optional icons
- Less configurable than `Row`, but easier to use
- From the Material library

Examples (ListTile)


1625 Main Street
My City, CA 99984
(408) 555-1212
costa@example.com

A `Card` containing 3 `ListTiles`.

App source: card_and_stack


A dropdown button displays a menu that's used to select a value from a small set of values. The button displays the current value and a down arrow.

Simple dropdown: Free
Dropdown with a hint: Choose
Scrollable dropdown: Four

Uses `ListTile` to list 3 drop down button types.
Dart code: buttons_demo.dart from the Flutter Gallery

Constraints

To fully understand Flutter's layout system, you need to learn how Flutter positions and sizes the components in a layout. For more information, see Understanding constraints.

Videos

The following videos, part of the Flutter in Focus series, explain `Stateless` and `Stateful` widgets.

Flutter in Focus playlist

Each episode of the Widget of the Week series focuses on a widget. Several of them includes layout widgets.

Flutter Widget of the Week playlist

Other resources

The following resources might help when writing layout code.

- Layout tutorial
  Learn how to build a layout.
- Widget catalog
  Describes many of the widgets available in Flutter.
- HTML/CSS Analogs in Flutter
  For those familiar with web programming, this page maps HTML/CSS functionality to Flutter features.
- Flutter Gallery running app, repo
  Demo app showcasing many Material Design widgets and other Flutter features.
- API reference docs
  Reference documentation for all of the Flutter libraries.
- Dealing with Box Constraints in Flutter
  Discusses how widgets are constrained by their render boxes.
- Adding assets and images
  Explains how to add images and other assets to your app's package.
- Zero to One with Flutter
  One person's experience writing his first Flutter app.