

Get started

Samples & tutorials

Development

User interface

Introduction to widgets

Building layouts

Splash screens

Adding interactivity

Assets and images

Navigation & routing

Animations

Advanced UI

Widget catalog

Data & backend

Accessibility & internationalization

Platform integration

Packages & plugins

Add Flutter to existing app

Tools & techniques

Migration notes

Testing & debugging

Performance & optimization

Deployment

Resources

Reference

Widget index

API reference

Package site

# Adding interactivity to your Flutter app

Docs > Development > UI > Adding interactivity

## Contents

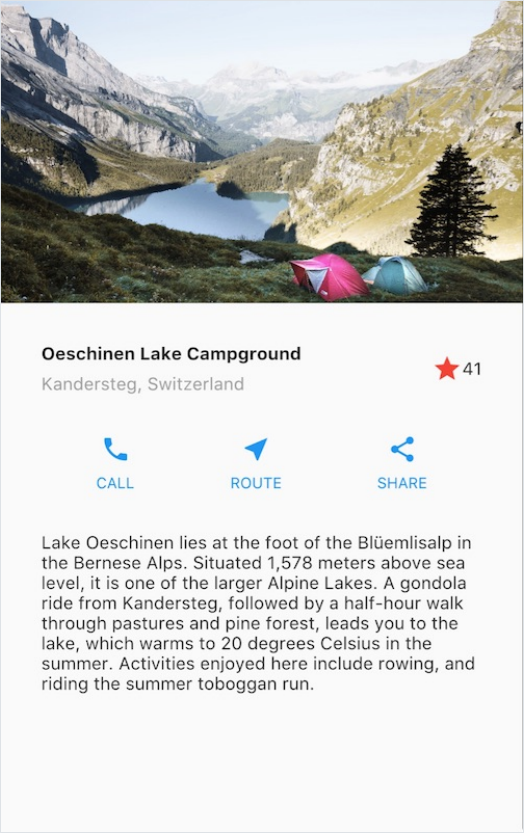
- Stateful and stateless widgets
- Creating a stateful widget
  - Step 0: Get ready
  - Step 1: Decide which object manages the widget’s state
  - Step 2: Subclass StatefulWidget
  - Step 3: Subclass State
  - Step 4: Plug the stateful widget into the widget tree
  - Problems?
- Managing state
  - The widget manages its own state
  - The parent widget manages the widget’s state
  - A mix-and-match approach
- Other interactive widgets
  - Standard widgets
  - Material Components
- Resources

## What you’ll learn

- How to respond to taps.
- How to create a custom widget.
- The difference between stateless and stateful widgets.

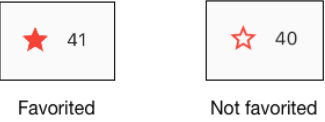
How do you modify your app to make it react to user input? In this tutorial, you’ll add interactivity to an app that contains only non interactive widgets. Specifically, you’ll modify an icon to make it tappable by creating a custom stateful widget that manages two stateless widgets.

Layout tutorial showed you how to create the layout for the following screenshot.





The layout tutorial app

When the app first launches, the star is solid red, indicating that this lake has previously been favorited. The number next to the star indicates that 41 people have favorited this lake. After completing this tutorial, tapping the star removes its favorited status, replacing the solid star with an outline and decreasing the count. Tapping again favorites the lake, drawing a solid star and increasing the count.



To accomplish this, you’ll create a single custom widget that includes both the star and the count, which are themselves widgets. Tapping the star changes state for both widgets, so the same widget should manage both.

<a href="#">Get started</a>	▼
<a href="#">Samples &amp; tutorials</a>	▼
<a href="#">Development</a>	▼
▼ <a href="#">User interface</a>	
<a href="#">Introduction to widgets</a>	
▶ <a href="#">Building layouts</a>	
▶ <a href="#">Splash screens</a>	
▶ <a href="#">Adding interactivity</a>	
▶ <a href="#">Assets and images</a>	
▶ <a href="#">Navigation &amp; routing</a>	
▶ <a href="#">Animations</a>	
▶ <a href="#">Advanced UI</a>	
▶ <a href="#">Widget catalog</a>	
▶ <a href="#">Data &amp; backend</a>	
▶ <a href="#">Accessibility &amp; internationalization</a>	
▶ <a href="#">Platform integration</a>	
▶ <a href="#">Packages &amp; plugins</a>	
▶ <a href="#">Add Flutter to existing app</a>	
▶ <a href="#">Tools &amp; techniques</a>	
▶ <a href="#">Migration notes</a>	
<a href="#">Testing &amp; debugging</a>	▼
<a href="#">Performance &amp; optimization</a>	▼
<a href="#">Deployment</a>	▼
<a href="#">Resources</a>	▼
<a href="#">Reference</a>	▼
<a href="#">Widget index</a>	
<a href="#">API reference</a> 	
<a href="#">Package site</a> 	

You can get right to touching the code in [Step 2: Subclass StatefulWidget](#). If you want to try different ways of managing state, ski [Managing state](#).

# Stateful and stateless widgets

A widget is either stateful or stateless. If a widget can change—when a user interacts with it, for example—it’s stateful.

A *stateless* widget never changes. [Icon](#), [IconButton](#), and [Text](#) are examples of stateless widgets. Stateless widgets subclass [StatelessWidget](#).

A *stateful* widget is dynamic: for example, it can change its appearance in response to events triggered by user interactions or wh it receives data. [Checkbox](#), [Radio](#), [Slider](#), [InkWell](#), [Form](#), and [TextField](#) are examples of stateful widgets. Stateful widgets subclass [StatefulWidget](#).

A widget’s state is stored in a [State](#) object, separating the widget’s state from its appearance. The state consists of values that ca change, like a slider’s current value or whether a checkbox is checked. When the widget’s state changes, the state object calls `setState()`, telling the framework to redraw the widget.

## Creating a stateful widget

### What's the point?

- A stateful widget is implemented by two classes: a subclass of `StatefulWidget` and a subclass of `State`.
- The state class contains the widget’s mutable state and the widget’s `build()` method.
- When the widget’s state changes, the state object calls `setState()`, telling the framework to redraw the widget.

In this section, you’ll create a custom stateful widget. You’ll replace two stateless widgets—the solid red star and the numeric col next to it—with a single custom stateful widget that manages a row with two children widgets: an [IconButton](#) and [Text](#).

Implementing a custom stateful widget requires creating two classes:

- A subclass of `StatefulWidget` that defines the widget.
- A subclass of `State` that contains the state for that widget and defines the widget’s `build()` method.

This section shows you how to build a stateful widget, called `FavoriteWidget`, for the lakes app. After setting up, your first step is choosing how state is managed for `FavoriteWidget`.

## Step 0: Get ready

If you’ve already built the app in [Layout tutorial \(step 6\)](#), skip to the next section.

1. Make sure you’ve [set up](#) your environment.
2. [Create a basic “Hello World” Flutter app](#).
3. Replace the `lib/main.dart` file with [main.dart](#).
4. Replace the `pubspec.yaml` file with [pubspec.yaml](#).
5. Create an `images` directory in your project, and add [lake.jpg](#).

Once you have a connected and enabled device, or you’ve launched the [iOS simulator](#) (part of the Flutter install), you are good to g

## Step 1: Decide which object manages the widget’s state

A widget’s state can be managed in several ways, but in our example the widget itself, `FavoriteWidget`, will manage its own state this example, toggling the star is an isolated action that doesn’t affect the parent widget or the rest of the UI, so the widget can handle its state internally.

Learn more about the separation of widget and state, and how state might be managed, in [Managing state](#).

## Step 2: Subclass StatefulWidget

The `FavoriteWidget` class manages its own state, so it overrides `createState()` to create a `State` object. The framework calls `createState()` when it wants to build the widget. In this example, `createState()` returns an instance of `_FavoriteWidgetState` which you’ll implement in the next step.

```
lib/main.dart (FavoriteWidget)

class FavoriteWidget extends StatefulWidget {
  @override
  _FavoriteWidgetState createState() => _FavoriteWidgetState();
}
```

[Get started](#)

[Samples & tutorials](#)

[Development](#)

▼ [User interface](#)

[Introduction to widgets](#)

▶ [Building layouts](#)

▶ [Splash screens](#)

[Adding interactivity](#)

[Assets and images](#)

[Navigation & routing](#)

▶ [Animations](#)

▶ [Advanced UI](#)

[Widget catalog](#)

▶ [Data & backend](#)

▶ [Accessibility & internationalization](#)

▶ [Platform integration](#)

▶ [Packages & plugins](#)

▶ [Add Flutter to existing app](#)

▶ [Tools & techniques](#)

▶ [Migration notes](#)

[Testing & debugging](#)

[Performance & optimization](#)

[Deployment](#)

[Resources](#)

[Reference](#)

[Widget index](#)

[API reference](#) [↗](#)

[Package site](#) [↗](#)

**Note:** Members or classes that start with an underscore (`_`) are private. For more information, see [Libraries and visibility](#), a section in the [Dart language tour](#).

## Step 3: Subclass State

The `_FavoriteWidgetState` class stores the mutable data that can change over the lifetime of the widget. When the app first launches, the UI displays a solid red star, indicating that the lake has “favorite” status, along with 41 likes. These values are stored in the `_isFavorited` and `_favoriteCount` fields:

```
lib/main.dart (_FavoriteWidgetState fields)

class _FavoriteWidgetState extends State<FavoriteWidget> {
  bool _isFavorited = true;
  int _favoriteCount = 41;
  // ...
}
```

The class also defines a `build()` method, which creates a row containing a red `IconButton`, and `Text`. You use `IconButton` (instead of `Icon`) because it has an `onPressed` property that defines the callback function (`_toggleFavorite`) for handling a tap. You’ll define the callback function next.

```
lib/main.dart (_FavoriteWidgetState build)

class _FavoriteWidgetState extends State<FavoriteWidget> {
  // ...
  @override
  Widget build(BuildContext context) {
    return Row(
      mainAxisAlignment: MainAxisAlignment.min,
      children: [
        Container(
          padding: EdgeInsets.all(0),
          child: IconButton(
            icon: (_isFavorited ? Icon(Icons.star) : Icon(Icons.star_border)),
            color: Colors.red[500],
            onPressed: _toggleFavorite,
          ),
        ),
        SizedBox(
          width: 18,
          child: Container(
            child: Text('$_favoriteCount'),
          ),
        ),
      ],
    );
  }
}
```

**Tip:** Placing the `Text` in a `SizedBox` and setting its width prevents a discernible “jump” when the text changes between the values of 40 and 41 — a jump would otherwise occur because those values have different widths.

The `_toggleFavorite()` method, which is called when the `IconButton` is pressed, calls `setState()`. Calling `setState()` is critical because this tells the framework that the widget’s state has changed and that the widget should be redrawn. The function argument to `setState()` toggles the UI between these two states:

- A `star` icon and the number 41
- A `star_border` icon and the number 40

```
void _toggleFavorite() {
  setState(() {
    if (_isFavorited) {
      _favoriteCount -= 1;
      _isFavorited = false;
    } else {
      _favoriteCount += 1;
      _isFavorited = true;
    }
  });
}
```

## Step 4: Plug the stateful widget into the widget tree

Add your custom stateful widget to the widget tree in the app’s `build()` method. First, locate the code that creates the `Icon` and `Text`, and delete it. In the same location, create the stateful widget:

Get started

Samples & tutorials

Development

User interface

Introduction to widgets

Building layouts

Splash screens

Adding interactivity

Assets and images

Navigation & routing

Animations

Advanced UI

Widget catalog

Data & backend

Accessibility & internationalization

Platform integration

Packages & plugins

Add Flutter to existing app

Tools & techniques

Migration notes

Testing & debugging

Performance & optimization

Deployment

Resources

Reference

Widget index

API reference

Package site

layout/lakes/{step6 → interactive}/lib/main.dart

```
@@ -10,2 +5,2 @@
10 5 class MyApp extends StatelessWidget {
11 6   @override
   @@ -38,11 +33,7 @@
38 33     ],
39 34     ),
40 35   ),
41 -   Icon(
36 +   FavoriteWidget(),
42 -   Icons.star,
43 -   color: Colors.red[500],
44 -   ),
45 -   Text( '41' ),
46 37   ],
47 38   ),
48 39   );
   @@ -114,3 +105,5 @@
114 105   ),
115 106   ),
116 107   ],
108 +   );
109 + }
```

That’s it! When you hot reload the app, the star icon should now respond to taps.

## Problems?

If you can’t get your code to run, look in your IDE for possible errors. [Debugging Flutter Apps](#) might help. If you still can’t find the problem, check your code against the interactive lakes example on GitHub.

- [lib/main.dart](#)
- [pubspec.yaml](#)
- [lakes.jpg](#)

If you still have questions, refer to any one of the developer [community](#) channels.

The rest of this page covers several ways a widget’s state can be managed, and lists other available interactive widgets.

## Managing state

### What's the point?

- There are different approaches for managing state.
- You, as the widget designer, choose which approach to use.
- If in doubt, start by managing state in the parent widget.

Who manages the stateful widget’s state? The widget itself? The parent widget? Both? Another object? The answer is... it depend: There are several valid ways to make your widget interactive. You, as the widget designer, make the decision based on how you expect your widget to be used. Here are the most common ways to manage state:

- [The widget manages its own state](#)
- [The parent manages the widget’s state](#)
- [A mix-and-match approach](#)

How do you decide which approach to use? The following principles should help you decide:

- If the state in question is user data, for example the checked or unchecked mode of a checkbox, or the position of a slider, t the state is best managed by the parent widget.
- If the state in question is aesthetic, for example an animation, then the state is best managed by the widget itself.

If in doubt, start by managing state in the parent widget.

We’ll give examples of the different ways of managing state by creating three simple examples: TapboxA, TapboxB, and TapboxC The examples all work similarly—each creates a container that, when tapped, toggles between a green or grey box. The `_active` boolean determines the color: green for active or grey for inactive.



These examples use [GestureDetector](#) to capture activity on the Container.

[Get started](#)

[Samples & tutorials](#)

[Development](#)

▼ [User interface](#)

[Introduction to widgets](#)

▶ [Building layouts](#)

▶ [Splash screens](#)

[Adding interactivity](#)

[Assets and images](#)

[Navigation & routing](#)

▶ [Animations](#)

▶ [Advanced UI](#)

[Widget catalog](#)

▶ [Data & backend](#)

▶ [Accessibility & internationalization](#)

▶ [Platform integration](#)

▶ [Packages & plugins](#)

▶ [Add Flutter to existing app](#)

▶ [Tools & techniques](#)

▶ [Migration notes](#)

[Testing & debugging](#)

[Performance & optimization](#)

[Deployment](#)

[Resources](#)

[Reference](#)

[Widget index](#)

[API reference](#)

[Package site](#)

## The widget manages its own state

Sometimes it makes the most sense for the widget to manage its state internally. For example, [ListView](#) automatically scrolls wh its content exceeds the render box. Most developers using ListView don't want to manage ListView's scrolling behavior, so ListView itself manages its scroll offset.

The `_TapboxAState` class:

- Manages state for `TapboxA`.
- Defines the `_active` boolean which determines the box's current color.
- Defines the `_handleTap()` function, which updates `_active` when the box is tapped and calls the `setState()` function to update the UI.
- Implements all interactive behavior for the widget.

```
// TapboxA manages its own state.

//----- TapboxA -----

class TapboxA extends StatefulWidget {
  TapboxA({Key key}) : super(key: key);

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

class _TapboxAState extends State<TapboxA> {
  bool _active = false;

  void _handleTap() {
    setState(() {
      _active = !_active;
    });
  }

  Widget build(BuildContext context) {
    return GestureDetector(
      onTap: _handleTap,
      child: Container(
        child: Center(
          child: Text(
            _active ? 'Active' : 'Inactive',
            style: TextStyle(fontSize: 32.0, color: Colors.white),
          ),
        ),
        width: 200.0,
        height: 200.0,
        decoration: BoxDecoration(
          color: _active ? Colors.lightGreen[700] : Colors.grey[600],
        ),
      ),
    );
  }
}

//----- MyApp -----

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      home: Scaffold(
        appBar: AppBar(
          title: Text('Flutter Demo'),
        ),
        body: Center(
          child: TapboxA(),
        ),
      ),
    );
  }
}
```

## The parent widget manages the widget's state

Often it makes the most sense for the parent widget to manage the state and tell its child widget when to update. For example, [IconButton](#) allows you to treat an icon as a tappable button. IconButton is a stateless widget because we decided that the parent widget needs to know whether the button has been tapped, so it can take appropriate action.

In the following example, TapboxB exports its state to its parent through a callback. Because TapboxB doesn't manage any state, subclasses StatelessWidget.

The ParentWidgetState class:



Get started

Samples & tutorials

Development

User interface

Introduction to widgets

Building layouts

Splash screens

Adding interactivity

Assets and images

Navigation & routing

Animations

Advanced UI

Widget catalog

Data & backend

Accessibility & internationalization

Platform integration

Packages & plugins

Add Flutter to existing app

Tools & techniques

Migration notes

Testing & debugging

Performance & optimization

Deployment

Resources

Reference

Widget index

API reference

Package site

- Manages the `_active` state for `TapboxB`.
- Implements `_handleTapboxChanged()`, the method called when the box is tapped.
- When the state changes, calls `setState()` to update the UI.

The `TapboxB` class:

- Extends `StatelessWidget` because all state is handled by its parent.
- When a tap is detected, it notifies the parent.

```
// ParentWidget manages the state for TapboxB.

//----- ParentWidget -----

class ParentWidget extends StatefulWidget {
  @override
  _ParentWidgetState createState() => _ParentWidgetState();
}

class _ParentWidgetState extends State<ParentWidget> {
  bool _active = false;

  void _handleTapboxChanged(bool newValue) {
    setState(() {
      _active = newValue;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Container(
      child: TapboxB(
        active: _active,
        onChanged: _handleTapboxChanged,
      ),
    );
  }
}

//----- TapboxB -----

class TapboxB extends StatelessWidget {
  TapboxB({Key key, this.active: false, @required this.onChanged})
    : super(key: key);

  final bool active;
  final ValueChanged<bool> onChanged;

  void _handleTap() {
    onChanged(!active);
  }



  Widget build(BuildContext context) {
    return GestureDetector(
      onTap: _handleTap,
      child: Container(
        child: Center(
          child: Text(
            active ? 'Active' : 'Inactive',
            style: TextStyle(fontSize: 32.0, color: Colors.white),
          ),
        ),
        width: 200.0,
        height: 200.0,
        decoration: BoxDecoration(
          color: active ? Colors.lightGreen[700] : Colors.grey[600],
        ),
      ),
    );
  }
}
```

**Tip:** When creating API, consider using the `@required` annotation for any parameters that your code relies on. To use `@required`, import the [foundation library](#) (which re-exports Dart’s [meta.dart](#) library):

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

## A mix-and-match approach

For some widgets, a mix-and-match approach makes the most sense. In this scenario, the stateful widget manages some of the state, and the parent widget manages other aspects of the state.

<a href="#">Get started</a>	▼
<a href="#">Samples &amp; tutorials</a>	▼
<a href="#">Development</a>	▲
▼ <a href="#">User interface</a>	
<a href="#">Introduction to widgets</a>	
▶ <a href="#">Building layouts</a>	
▶ <a href="#">Splash screens</a>	
<a href="#">Adding interactivity</a>	
<a href="#">Assets and images</a>	
<a href="#">Navigation &amp; routing</a>	
▶ <a href="#">Animations</a>	
▶ <a href="#">Advanced UI</a>	
<a href="#">Widget catalog</a>	
▶ <a href="#">Data &amp; backend</a>	
▶ <a href="#">Accessibility &amp; internationalization</a>	
▶ <a href="#">Platform integration</a>	
▶ <a href="#">Packages &amp; plugins</a>	
▶ <a href="#">Add Flutter to existing app</a>	
▶ <a href="#">Tools &amp; techniques</a>	
▶ <a href="#">Migration notes</a>	
<a href="#">Testing &amp; debugging</a>	▼
<a href="#">Performance &amp; optimization</a>	▼
<a href="#">Deployment</a>	▼
<a href="#">Resources</a>	▼
<a href="#">Reference</a>	▲
<a href="#">Widget index</a>	
<a href="#">API reference</a> 	
<a href="#">Package site</a> 	

In the `TapboxC` example, on tap down, a dark green border appears around the box. On tap up, the border disappears and the box's color changes. `TapboxC` exports its `_active` state to its parent but manages its `_highlight` state internally. This example has two State objects, `_ParentWidgetState` and `_TapboxCState`.

The `_ParentWidgetState` object:

- Manages the `_active` state.
- Implements `_handleTapboxChanged()`, the method called when the box is tapped.
- Calls `setState()` to update the UI when a tap occurs and the `_active` state changes.

The `_TapboxCState` object:

- Manages the `_highlight` state.
- The `GestureDetector` listens to all tap events. As the user taps down, it adds the highlight (implemented as a dark green border). As the user releases the tap, it removes the highlight.
- Calls `setState()` to update the UI on tap down, tap up, or tap cancel, and the `_highlight` state changes.
- On a tap event, passes that state change to the parent widget to take appropriate action using the `widget` property.

[Get started](#)

[Samples & tutorials](#)

[Development](#)

▼ [User interface](#)

[Introduction to widgets](#)

▶ [Building layouts](#)

▶ [Splash screens](#)

[Adding interactivity](#)

[Assets and images](#)

[Navigation & routing](#)

▶ [Animations](#)

▶ [Advanced UI](#)

[Widget catalog](#)

▶ [Data & backend](#)

▶ [Accessibility & internationalization](#)

▶ [Platform integration](#)

▶ [Packages & plugins](#)

▶ [Add Flutter to existing app](#)

▶ [Tools & techniques](#)

▶ [Migration notes](#)

[Testing & debugging](#)

[Performance & optimization](#)

[Deployment](#)

[Resources](#)

[Reference](#)

[Widget index](#)

[API reference](#)

[Package site](#)

//----- ParentWidget -----

```
class ParentWidget extends StatefulWidget {
  @override
  _ParentWidgetState createState() => _ParentWidgetState();
}
```

```
class _ParentWidgetState extends State<ParentWidget> {
  bool _active = false;
```

```
  void _handleTapboxChanged(bool newValue) {
    setState(() {
      _active = newValue;
    });
  }
```

```
  @override
  Widget build(BuildContext context) {
    return Container(
      child: TapboxC(
        active: _active,
        onChanged: _handleTapboxChanged,
      ),
    );
  }
}
```

//----- TapboxC -----

```
class TapboxC extends StatefulWidget {
  TapboxC({Key key, this.active: false, @required this.onChanged})
    : super(key: key);
```

```
  final bool active;
  final ValueChanged<bool> onChanged;
```

```
  _TapboxCState createState() => _TapboxCState();
}
```

```
class _TapboxCState extends State<TapboxC> {
  bool _highlight = false;
```

```
  void _handleTapDown(TapDownDetails details) {
    setState(() {
      _highlight = true;
    });
  }
```

```
  void _handleTapUp(TapUpDetails details) {
    setState(() {
      _highlight = false;
    });
  }
```

```
  void _handleTapCancel() {
    setState(() {
      _highlight = false;
    });
  }
```

```
  void _handleTap() {
    widget.onChanged(!widget.active);
  }
```

```
  Widget build(BuildContext context) {
    // This example adds a green border on tap down.
    // On tap up, the square changes to the opposite state.
    return GestureDetector(
      onTapDown: _handleTapDown, // Handle the tap events in the order that
      onTapUp: _handleTapUp, // they occur: down, up, tap, cancel
      onTap: _handleTap,
      onTapCancel: _handleTapCancel,
      child: Container(
        child: Center(
          child: Text(widget.active ? 'Active' : 'Inactive',
            style: TextStyle(fontSize: 32.0, color: Colors.white)),
        ),
        width: 200.0,
        height: 200.0,
        decoration: BoxDecoration(
          color:
            widget.active ? Colors.lightGreen[700] : Colors.grey[600],
          border: _highlight
            ? Border.all(
                color: Colors.teal[700],
                width: 10.0,
              )
        )
      )
    );
  }
}
```



[Get started](#)

[Samples & tutorials](#)

[Development](#)

▼ [User interface](#)

[Introduction to widgets](#)

▶ [Building layouts](#)

▶ [Splash screens](#)

[Adding interactivity](#)

[Assets and images](#)

[Navigation & routing](#)

▶ [Animations](#)

▶ [Advanced UI](#)

[Widget catalog](#)

▶ [Data & backend](#)

▶ [Accessibility & internationalization](#)

▶ [Platform integration](#)

▶ [Packages & plugins](#)

▶ [Add Flutter to existing app](#)

▶ [Tools & techniques](#)

▶ [Migration notes](#)

[Testing & debugging](#)

[Performance & optimization](#)

[Deployment](#)

[Resources](#)

[Reference](#)

[Widget index](#)

[API reference](#) 

[Package site](#) 

```
        : null,  
        ),  
    ),  
);  
}  
}
```

An alternate implementation might have exported the highlight state to the parent while keeping the active state internal, but if you asked someone to use that tap box, they'd probably complain that it doesn't make much sense. The developer cares whether the box is active. The developer probably doesn't care how the highlighting is managed, and prefers that the tap box handles those details.

# Other interactive widgets

Flutter offers a variety of buttons and similar interactive widgets. Most of these widgets implement the [Material Design guideline](#), which define a set of components with an opinionated UI.

If you prefer, you can use [GestureDetector](#) to build interactivity into any custom widget. You can find examples of GestureDetector in [Managing state](#), and in the [Flutter Gallery](#).

 **Tip:** Flutter also provides a set of iOS-style widgets called [Cupertino](#).

When you need interactivity, it's easiest to use one of the prefabricated widgets. Here's a partial list:

## Standard widgets

- [Form](#)
- [FormField](#)

## Material Components

- [Checkbox](#)
- [DropDownButton](#)
- [FlatButton](#)
- [FloatingActionButton](#)
- [IconButton](#)
- [Radio](#)
- [RaisedButton](#)
- [Slider](#)
- [Switch](#)
- [TextField](#)

## Resources

The following resources might help when adding interactivity to your app.

- [Gestures](#), a section in [Introduction to widgets](#)  
How to create a button and make it respond to input.
- [Gestures in Flutter](#)  
A description of Flutter's gesture mechanism.
- [Flutter API documentation](#)  
Reference documentation for all of the Flutter libraries.
- [Flutter Gallery](#)  
Demo app showcasing many Material Components and other Flutter features.
- [Flutter's Layered Design](#) (video)  
This video includes information about state and stateless widgets. Presented by Google engineer, Ian Hickson.

