# Interactivity and Asset

# Learning Outcomes

After completing this lesson you should be able to explain

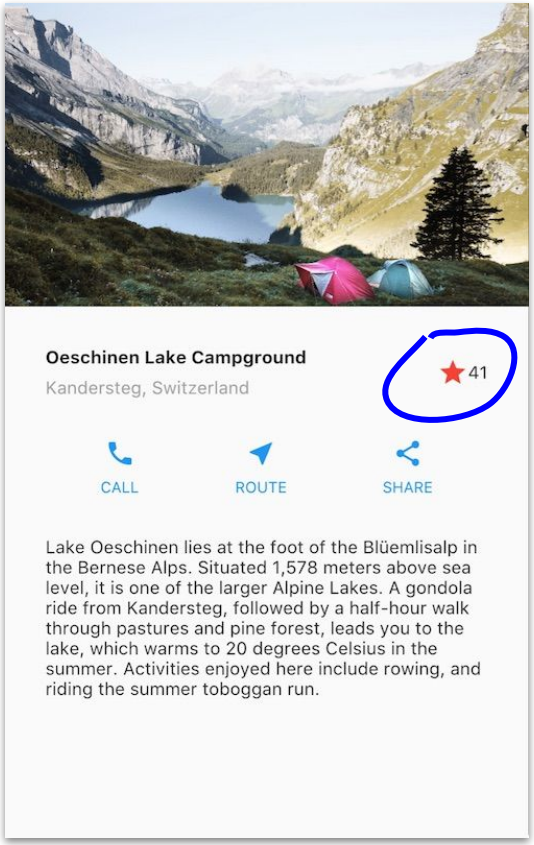
- How to **respond to taps** in Flutter application
- How to create a **custom widget**
- The difference between **stateless and stateful widgets**
- The common approaches used for **managing state** in Flutter
- How to **add assets**

# Adding interactivity to your Flutter app

## What we want to achieve

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.

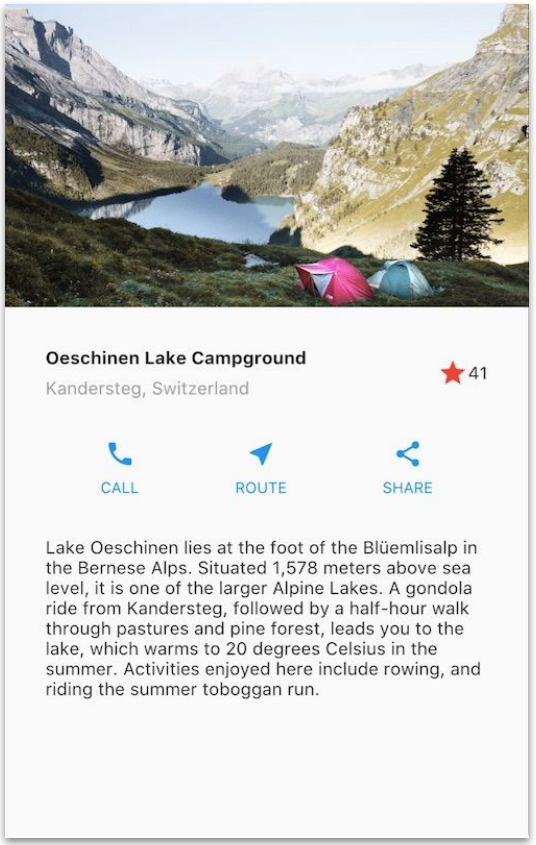



# Adding interactivity to your Flutter app

To achieve this goal

We will 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

# Stateful and Stateless widgets

A widget is either **stateful** or **stateless**

Stateful widget changes state

- `Checkbox`, `Radio`, `Slider`, `InkWell`, `Form`, and `TextField` are examples

A stateless widget never changes

- `Icon`, `IconButton`, and `Text` are examples

Stateless widgets subclass **`StatelessWidget`**

Stateful widgets subclass **`StatefulWidget`**

# Stateful widgets

A widget's state is stored in a `State` object, separating the widget's state from its appearance

The state consists of values that can change, for instance

- 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

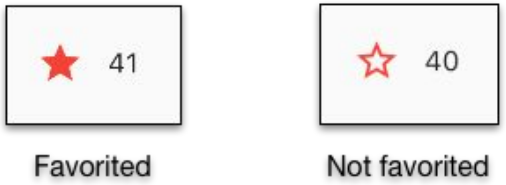
A **stateful** widget is implemented by two classes

a subclass of `StatefulWidget` and

a subclass of **State**

The state class contains the widget's **mutable state** and the widget's `build()` method

# Creating a stateful widget




Let us see how to build a stateful widget, called **`FavoriteWidget`**

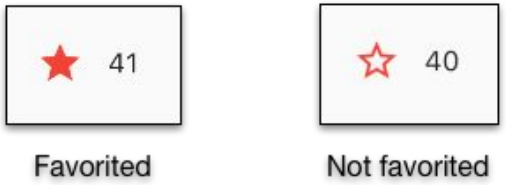
**Step 1:** Decide which object manages the widget's state

A widget's state can be managed in several ways

In this example, toggling the star is an isolated action that doesn't affect the parent widget or the rest of the UI

Therefore, `FavoriteWidget` itself can manage its own state internally

# Creating a stateful widget



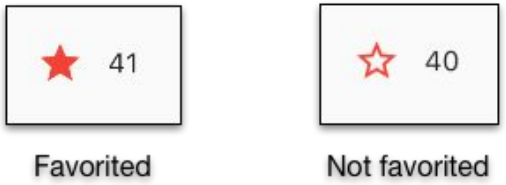

**Step 2**: Subclass `StatefulWidget`

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.

`createState()` returns an instance of `_FavoriteWidgetState`

# Creating a stateful widget



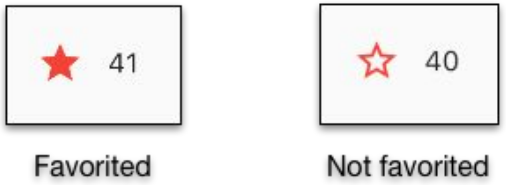

**Step 2**: Subclass **`StatefulWidget`**

```
class FavoriteWidget extends StatefulWidget {

    @override
    _FavoriteWidgetState createState() =>
            _FavoriteWidgetState();

}
```

# Creating a stateful widget

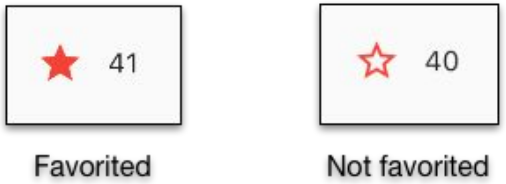



**Step 3:** Subclass **`State`**

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

The **`_FavoriteWidgetState`** class stores the mutable data that can change over the lifetime of the widget

# Creating a stateful widget

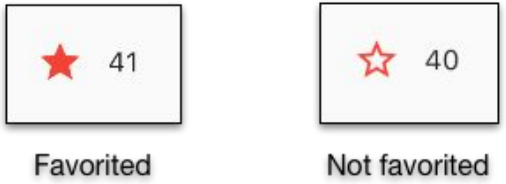



**Step 3:** Subclass `State`

The `_FavoriteWidgetState` class also defines a `build()` method, which creates a row containing a red `IconButton`, and `Text`

The `build()` method is shown in the next slide

# Creating a stateful widget




```
class _FavoriteWidgetState extends State<FavoriteWidget> {
  // ···
  @override
  Widget build(BuildContext context) {
    return Row(
      children: [
       IconButton(
        icon: (_isFavorited ? Icon(Icons.star):Icon(Icons.star_border)),
            color: Colors.red[500],
            onPressed: _toggleFavorite,
         ),
       Text('$_favoriteCount'),
      ],
    );
  }
}
```

The `_toggleFavorite()` method is shown in the next slide

# Creating a stateful widget


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

This method is called when the **`IconButton`** is pressed

Calling **`setState()`** 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 the states highlighted in yellow

# Managing state

Who manages the stateful widget's state?

- The widget itself?
- The parent widget?
- Both?
- Another object?

# Managing state

Keep in mind the following principles while deciding which approach to use

The **parent** should manage the state If the state in question is user data, for example the checked or unchecked mode of a checkbox, or the position of a slider

The **widget itself** should manage the state If the state in question is aesthetic, for example an animation

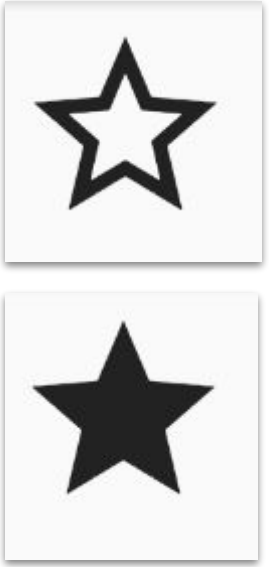
If in doubt, start by managing state in the parent widget

# Managing state: Example

Consider an application which has a star icon that, when tapped, toggles between a **solid** or **outline** star

The `_favorite` boolean variable determines the icon type: **solid** when `_favorite` is true or **outline** if false

We will have three versions of an application called `ToggleFavorite` to explore the three state management approaches



# The widget manages its own state

**Stateful** Widget class

```
class ToggleFavorite extends StatefulWidget {
 @override
 _ToggleFavoriteState createState() {
   return _ToggleFavoriteState();
 }
}
```

# The widget manages its own state

The **state** class

```
class _ToggleFavoriteState extends State<ToggleFavorite> {
 bool _favorite = true;
 // ...
}
```

Here the **`_favorite`** variable holds the state internally

# The widget manages its own state

The **build** method

```
@override
Widget build(BuildContext context) {
  return IconButton(
    iconSize: 100.0,
    icon: _favorite ? Icon(Icons.star) : Icon(Icons.star_border),
    onPressed: _handlePress,
  );
}
```

# The widget manages its own state

The **_handlePress** method

```
_handlePress() {
  setState(() {
    _favorite = !_favorite;
  });
}
```

# The parent widget manages the widget's state

The **parent** stateful widget class

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

# The parent widget manages the widget's state

The **parent** state class

```
class _ParentWidgetState extends State<ParentWidget> {
 bool _favorite = true;
  // ...
}
```

# The parent widget manages the widget's state

The **`build`** method

```
@override
Widget build(BuildContext context) {
  return ToggleFavorite(
    favorite: _favorite,
    onChanged: _handleToggle,
  );
}
```

Notice the highlighted child widget

The state and a callback function are passed as a constructor argument to the child widget **`ToggleFavorite`**

# The parent widget manages the widget's state

The **`_handleToggle`** method

```
_handleToggle(bool newValue) {
  setState(() {
    _favorite = newValue;
  });
}
```

Notice that **`_handleToggle`** accepts an argument

# The parent widget manages the widget's state

The **child stateless** widget class - **`ToggleFavorite`**

```
class ToggleFavorite extends StatelessWidget {
 ToggleFavorite({this.favorite = false, this.onChanged});
 final bool favorite;
 final Function onChanged;

// ...
}
```

Notice how the **`favorite`** an **`onChanged`** fields are initialized by the constructor

ValueChanged is a callback type which report when value changes

# The parent widget manages the widget's state

The **child stateless** widget class - **`build`** method

```
@override
 Widget build(BuildContext context) {
   return IconButton(
     iconSize: 100.0,
     icon: favorite ? Icon(Icons.star) : Icon(Icons.star_border),
     onPressed: _handlePress,
   );
 }
```

Now the state of **`favorite`** is managed at the parent widget

# The parent widget manages the widget's state

The **child stateless** widget class - **`_handlePress`** method

```
_handlePress() {
  onChanged(!favorite);
}
```

**`_handlePress`** calles a callback function **`onChange`** that was passed to its class from the parent

# A mix-and-match approach

In this scenario, the stateful widget manages some of the state, and the parent widget manages other aspects of the state

Exercise:

- Combine the previous two state management approaches
- Let the parent widget manage the toggle state
- Add a color state for the icon which changes together with the icon types
- `ToggleFavorite widget should manage the color state`

# Widgets with built in interactivity

## Standard widgets

- `Form`
- `FormField`

## Material Components

- `Checkbox`
- `DropdownButton`
- `TextButton`
- TextField

## Material Components

- `FloatingActionButton`
- `IconButton`
- `Radio`
- `ElevatedButton`
- `Slider`
- `Switch`

# Adding assets

An `asset` is a file that is bundled and deployed with your app, and is accessible at runtime

**Common types of assets** include

**static data** (for example, JSON files),

**configuration files**,

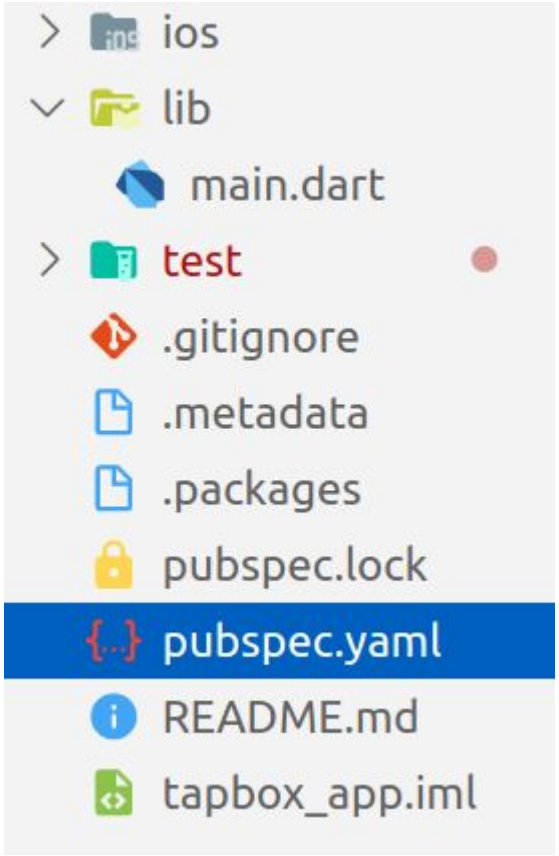
**icons**, and

**images** (JPEG, WebP, GIF, animated WebP/GIF, PNG, BMP, and WBMP)

# Specifying assets

Flutter uses the **`pubspec.yaml`** file, located at the root of your project, to identify assets required by an app

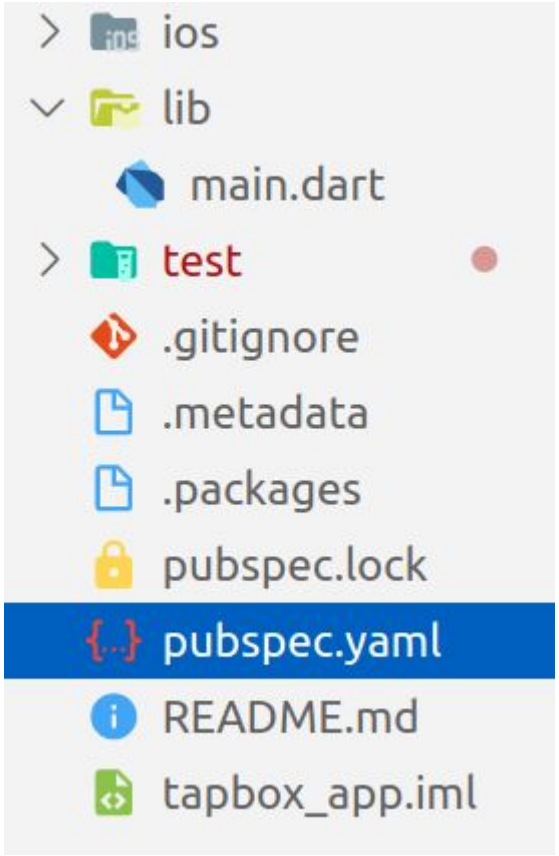
```
flutter:
  assets:
    - assets/my_icon.png
    - assets/background.png
```



# Specifying assets

To include all assets under a directory, specify the directory name with the `/` character at the end:

```
flutter:
  Assets:
    - directory/
    - directory/subdirectory/
```



# Asset bundling

The assets subsection of the flutter section specifies files that should be included with the app

During a build, Flutter places assets into a special archive called the **asset bundle** that apps read from at runtime

# Asset variants

Refers to different versions of an asset that might be displayed in different contexts

When an asset's path is specified in the `assets` section of `pubspec.yaml`, the build process looks for any files with the same name in adjacent subdirectories and include them in the **asset bundle**

Flutter uses asset variants when choosing resolution-appropriate images

# Asset variants

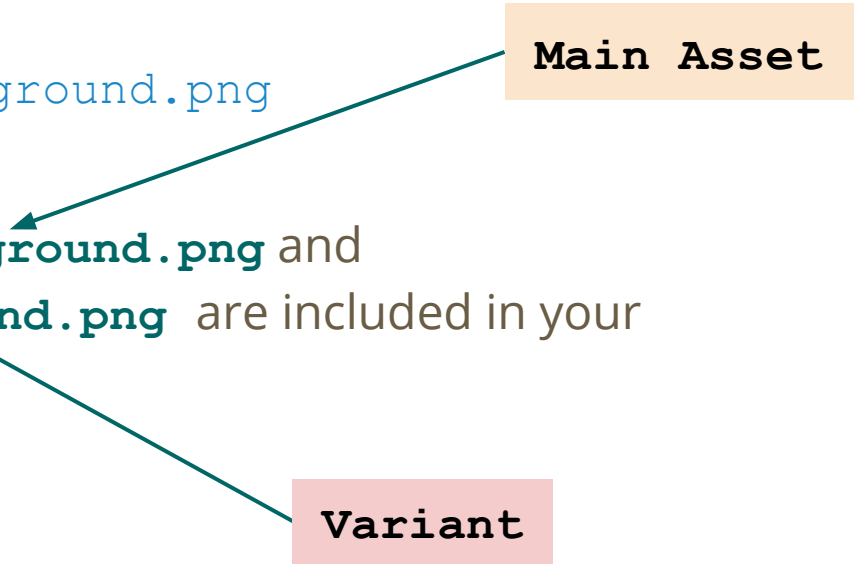
Let us assume you have the following files in your application directory:

```
.../pubspec.yaml
.../graphics/my_icon.png
.../graphics/background.png
.../graphics/dark/background.png
...etc
```

# Asset variants

And your **`pubspec.yaml`** file contains the following

```
flutter:
  assets:
    - graphics/background.png
```




Then both **`graphics/background.png`** and **`graphics/dark/background.png`** are included in your asset bundle

# Asset variants

If your **`pubspec.yaml`** file contains the following

```
flutter:
  assets:
    - graphics/
```

Then the **`graphics/my_icon.png`**, **`graphics/background.png`** and **`graphics/dark/background.png`** files are included in the asset bundle

# Loading assets

Your app can access its assets through an **`AssetBundle`** object

Below are the two main methods that are available in **`AssetBundle`** for loading **`string/text`** asset or an **`image/binary`** asset

**`loadString()`**

**`load()`**

You can us a **logical key** to access the assets

The logical key maps to the path to the asset specified in the **`pubspec.yaml`** file at build time

# Loading text assets

Each Flutter app has a **`rootBundle`** object for easy access to the main asset bundle

It is possible to load assets directly using the **`rootBundle`** global static from **`package:flutter/services.dart`**

```
import 'dart:async' show Future;
import 'package:flutter/services.dart' show rootBundle;

Future<String> loadAsset() async {
  return await rootBundle.loadString('assets/config.json');
}
```

# Loading text assets

However, it's recommended to obtain the `AssetBundle` for the current `BuildContext` using `DefaultAssetBundle`

This approach enables a parent widget to substitute a different `AssetBundle` at runtime, which can be useful for **localization** or **testing** scenarios

Typically, you'll use `DefaultAssetBundle.of()` to indirectly load an asset

# Loading images

Flutter can load **resolution-appropriate** images for the current device pixel ratio

**Declaring resolution-aware image assets**

> `AssetImage` understands how to map a logical requested asset onto one that most closely matches the current device pixel ratio
>
> Assets should be arranged according to a particular directory structure so that `AssetImage` use the appropriate image for the **current device pixel ratio**

# Loading images

To load an image, you can use the **`AssetImage`** class in a widget's **`build()`** method

```
Widget build(BuildContext context) {
  return Image(image: AssetImage('graphics/background.png'));
}
```

# Platform assets

## Updating the app icon for Android

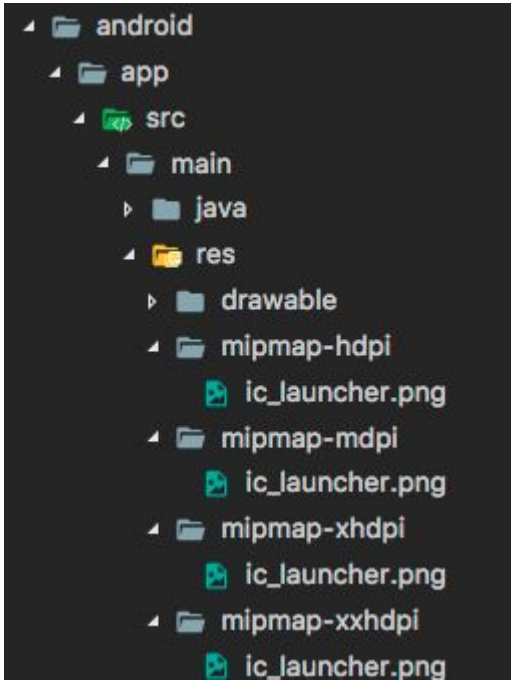

In your Flutter project's root directory, navigate to **`.../android/app/src/main/res`**

The various **bitmap** resource folders such as `mipmap-hdpi` already contain placeholder images named **`ic_launcher.png`**

Replace them with your desired assets respecting the recommended icon size per screen density

# Platform assets

## Updating the app icon for Android

# Platform assets

## Updating the app icon for iOS


In your Flutter project's root directory, navigate to `.../ios/Runner`

The `Assets.xcassets/AppIcon.appiconset` directory already contains placeholder images

Replace them with the appropriately sized images as indicated by their filename

# Platform assets

## Updating the app icon for iOS