

Flutter for React Native developers

Contents

- [Introduction to Dart for JavaScript Developers](#)
 - [Entry point](#)
 - [Printing to the console](#)
 - [Variables](#)
 - [Creating and assigning variables](#)
 - [Default value](#)
 - [Checking for null or zero](#)
 - [Functions](#)
 - [Asynchronous programming](#)
 - [Futures](#)
 - [async and await](#)
- [The basics](#)
 - [How do I create a Flutter app?](#)
 - [How do I run my app?](#)
 - [How do I import widgets?](#)
 - [What is the equivalent of the React Native “Hello world!” app in Flutter?](#)
 - [How do I use widgets and nest them to form a widget tree?](#)
 - [How do I create reusable components?](#)
- [Project structure and resources](#)
 - [Where do I start writing the code?](#)
 - [How are files structured in a Flutter app?](#)
 - [Where do I put my resources and assets and how do I use them?](#)
 - [How do I load images over a network?](#)
 - [How do I install packages and package plugins?](#)
- [Flutter widgets](#)
- [Views](#)
 - [What is the equivalent of the View container?](#)
 - [What is the equivalent of FlatList or SectionList?](#)
 - [How do I use a Canvas to draw or paint?](#)
- [Layouts](#)
 - [How do I use widgets to define layout properties?](#)
 - [How do I layer widgets?](#)

- [Styling](#)
 - [How do I style my components?](#)
 - [How do I use Icons and Colors?](#)
 - [How do I add style themes?](#)
- [State management](#)
 - [The StatelessWidget](#)
 - [The StatefulWidget](#)
 - [What are the StatefulWidget and StatelessWidget best practices?](#)
- [Props](#)
- [Local storage](#)
 - [How do I store persistent key-value pairs that are global to the app?](#)
- [Routing](#)
 - [How do I navigate between screens?](#)
 - [How do I use tab navigation and drawer navigation?](#)
 - [Tab navigation](#)
 - [Drawer navigation](#)
- [Gesture detection and touch event handling](#)
 - [How do I add a click or press listeners to a widget?](#)
- [Making HTTP network requests](#)
 - [How do I fetch data from API calls?](#)
- [Form input](#)
 - [How do I use text field widgets?](#)
 - [How do I use Form widgets?](#)
- [Platform-specific code](#)
- [Debugging](#)
 - [What tools can I use to debug my app in Flutter?](#)
 - [How do I perform a hot reload?](#)
 - [How do I access the in-app developer menu?](#)
- [Animation](#)
 - [How do I add a simple fade-in animation?](#)
 - [How do I add swipe animation to cards?](#)
- [React Native and Flutter widget equivalent components](#)

This document is for React Native (RN) developers looking to apply their existing RN knowledge to build mobile apps with Flutter. If you understand the fundamentals of the RN framework then you can use this document as a way to get started learning Flutter development.

This document can be used as a cookbook by jumping around and finding questions that are most relevant to your needs.

Introduction to Dart for JavaScript Developers

Like React Native, Flutter uses reactive-style views. However, while RN transpiles to native widgets, Flutter compiles all the way to native code. Flutter controls each pixel on the screen, which avoids performance problems caused by the need for a JavaScript bridge.

Dart is an easy language to learn and offers the following features:

- Provides an open-source, scalable programming language for building web, server, and mobile apps.
- Provides an object-oriented, single inheritance language that uses a C-style syntax that is AOT-compiled into native.
- Transcompiles optionally into JavaScript.
- Supports interfaces and abstract classes.

A few examples of the differences between JavaScript and Dart are described below.

Entry point

JavaScript doesn't have a pre-defined entry function—you define the entry point.

```
// JavaScript  
function startHere() {  
  // Can be used as entry point  
}
```

content_copy

In Dart, every app must have a top-level `main()` function that serves as the entry point to the app.

```
// Dart  
main() {  
}
```

content_copy

Try it out in [DartPad](#).

Printing to the console

To print to the console in Dart, use `print()`.

```
// JavaScript
console.log('Hello world!');
```

content_copy

```
// Dart
print('Hello world!');
```

content_copy

Try it out in [DartPad](#).

Variables

Dart is type safe—it uses a combination of static type checking and runtime checks to ensure that a variable’s value always matches the variable’s static type. Although types are mandatory, some type annotations are optional because Dart performs type inference.

Creating and assigning variables

In JavaScript, variables cannot be typed.

In [Dart](#), variables must either be explicitly typed or the type system must infer the proper type automatically.

```
// JavaScript
var name = 'JavaScript';
```

content_copy

```
// Dart
String name = 'dart'; // Explicitly typed as a string.
var otherName = 'Dart'; // Inferred string.
// Both are acceptable in Dart.
```

content_copy

Try it out in [DartPad](#).

For more information, see [Dart’s Type System](#).

Default value

In JavaScript, uninitialized variables are `undefined`.

In Dart, uninitialized variables have an initial value of `null`. Because numbers are objects in Dart, even uninitialized variables with numeric types have the value `null`.

```
// JavaScript
var name; // == undefined
```

content_copy

```
// Dart
var name; // == null
int x; // == null
```

content_copy

Try it out in [DartPad](#).

For more information, see the documentation on [variables](#).

Checking for null or zero

In JavaScript, values of 1 or any non-null objects are treated as true.

```
// JavaScript
var myNull = null;
if (!myNull) {
  console.log('null is treated as false');
}
var zero = 0;
if (!zero) {
  console.log('0 is treated as false');
}
```

content_copy

In Dart, only the boolean value `true` is treated as true.

content_copy

```
// Dart
var myNull = null;
if (myNull == null) {
  print('use "==" null" to check null');
}
var zero = 0;
if (zero == 0) {
  print('use "==" 0" to check zero');
}
```

Try it out in [DartPad](#).

Functions

Dart and JavaScript functions are generally similar. The primary difference is the declaration.

content_copy

```
// JavaScript
function fn() {
  return true;
}
```

content_copy

```
// Dart
fn() {
  return true;
}
// can also be written as
bool fn() {
  return true;
}
```

Try it out in [DartPad](#).

For more information, see the documentation on [functions](#).

Asynchronous programming

Futures

Like JavaScript, Dart supports single-threaded execution. In JavaScript, the Promise object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

Dart uses [Future](#) objects to handle this.

```
// JavaScript
class Example {
  _getIPAddress() {
    const url = 'https://httpbin.org/ip';
    return fetch(url)
      .then(response => response.json())
      .then(responseJson => {
        const ip = responseJson.origin;
        return ip;
      });
  }
}

function main() {
  const example = new Example();
  example
    ._getIPAddress()
    .then(ip => console.log(ip))
    .catch(error => console.error(error));
}

main();
```

content_copy

```
// Dart
import 'dart:convert';
import 'package:http/http.dart' as http;

class Example {
  Future<String> _getIPAddress() {
    final url = 'https://httpbin.org/ip';
    return http.get(url).then((response) {
      String ip = jsonDecode(response.body)['origin'];
      return ip;
    });
  }
}

main() {
  final example = new Example();
  example
    ._getIPAddress()
    .then((ip) => print(ip))
    .catchError((error) => print(error));
}
```

For more information, see the documentation on [Future](#) objects.

async and await

The `async` function declaration defines an asynchronous function.

In JavaScript, the `async` function returns a `Promise`. The `await` operator is used to wait for a `Promise`.


```
// JavaScript
class Example {
  async function _getIPAddress() {
    const url = 'https://httpbin.org/ip';
    const response = await fetch(url);
    const json = await response.json();
    const data = await json.origin;
    return data;
  }
}

async function main() {
  const example = new Example();
  try {
    const ip = await example._getIPAddress();
    console.log(ip);
  } catch (error) {
    console.error(error);
  }
}

main();
```

In Dart, an `async` function returns a `Future`, and the body of the function is scheduled for execution later. The `await` operator is used to wait for a `Future`.

```
// Dart
import 'dart:convert';
import 'package:http/http.dart' as http;

class Example {
  Future<String> _getIPAddress() async {
    final url = 'https://httpbin.org/ip';
    final response = await http.get(url);
    String ip = jsonDecode(response.body)['origin'];
    return ip;
  }
}

main() async {
  final example = new Example();
  try {
    final ip = await example._getIPAddress();
    print(ip);
  } catch (error) {
    print(error);
  }
}
```

For more information, see the documentation for [async and await](#).

The basics

How do I create a Flutter app?

To create an app using React Native, you would run `create-react-native-app` from the command line.

```
$ create-react-native-app <projectname>
```

To create an app in Flutter, do one of the following:

- Use an IDE with the Flutter and Dart plugins installed.
- Use the `flutter create` command from the command line. Make sure that the Flutter SDK is in your PATH.

```
$ flutter create <projectname>
```

content_copy

For more information, see [Getting started](#), which walks you through creating a button-click counter app. Creating a Flutter project builds all the files that you need to run a sample app on both Android and iOS devices.

How do I run my app?

In React Native, you would run `npm run` or `yarn run` from the project directory.

You can run Flutter apps in a couple of ways:

- Use the “run” option in an IDE with the Flutter and Dart plugins.
- Use `flutter run` from the project’s root directory.

Your app runs on a connected device, the iOS simulator, or the Android emulator.

For more information, see the Flutter [Getting started](#) documentation.

How do I import widgets?

In React Native, you need to import each required component.

```
//React Native
import React from 'react';
import { StyleSheet, Text, View } from 'react-native';
```

content_copy

In Flutter, to use widgets from the Material Design library, import the `material.dart` package. To use iOS style widgets, import the Cupertino library. To use a more basic widget set, import the Widgets library. Or, you can write your own widget library and import that.

```
import 'package:flutter/material.dart';
import 'package:flutter/cupertino.dart';
import 'package:flutter/widgets.dart';
import 'package:flutter/my_widgets.dart';
```

content_copy

Whichever widget package you import, Dart pulls in only the widgets that are used in your app.

For more information, see the [Flutter Widget Catalog](#).

What is the equivalent of the React Native “Hello world!” app in Flutter?

In React Native, the `HelloWorldApp` class extends `React.Component` and implements the render method by returning a view component.

```
// React Native
import React from 'react';
import { StyleSheet, Text, View } from 'react-native';

export default class App extends React.Component {
  render() {
    return (
      <View style={styles.container}>
        <Text>Hello world!</Text>
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center'
  }
});
```

content_copy

In Flutter, you can create an identical “Hello world!” app using the `Center` and `Text` widgets from the core widget library. The `Center` widget becomes the root of the widget tree and has one child, the `Text` widget.

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

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

The following images show the Android and iOS UI for the basic Flutter “Hello world!” app.

Hello world app on Android

Android

Hello world app on iOS

iOS

Now that you’ve seen the most basic Flutter app, the next section shows how to take advantage of Flutter’s rich widget libraries to create a modern, polished app.

How do I use widgets and nest them to form a widget tree?

In Flutter, almost everything is a widget.

Widgets are the basic building blocks of an app's user interface. You compose widgets into a hierarchy, called a widget tree. Each widget nests inside a parent widget and inherits properties from its parent. Even the application object itself is a widget. There is no separate “application” object. Instead, the root widget serves this role.

A widget can define:

- A structural element—like a button or menu
- A stylistic element—like a font or color scheme
- An aspect of layout—like padding or alignment

The following example shows the “Hello world!” app using widgets from the Material library. In this example, the widget tree is nested inside the `MaterialApp` root widget.

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

void main() => runApp(MyApp());

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

content_copy

The following images show “Hello world!” built from Material Design widgets. You get more functionality for free than in the basic “Hello world!” app.

Hello world app on Android

Android

Hello world app on iOS

iOS

When writing an app, you’ll use two types of widgets: [StatelessWidget](#) or [StatefulWidget](#). A [StatelessWidget](#) is just what it sounds like—a widget with no state. A [StatelessWidget](#) is created once, and never changes its appearance. A [StatefulWidget](#) dynamically changes state based on data received, or user input.

The important difference between stateless and stateful widgets is that [StatefulWidget](#)s have a [State](#) object that stores state data and carries it over across tree rebuilds, so it’s not lost.

In simple or basic apps it’s easy to nest widgets, but as the code base gets larger and the app becomes complex, you should break deeply nested widgets into functions that return the widget or smaller classes. Creating separate functions and widgets allows you to reuse the components within the app.

How do I create reusable components?

In React Native, you would define a class to create a reusable component and then use [props](#) methods to set or return properties and values of the selected elements. In the example below, the [CustomCard](#) class is defined and then used inside a parent class.

```
// React Native
class CustomCard extends React.Component {
  render() {
    return (
      <View>
        <Text> Card {this.props.index} </Text>
        <Button
          title="Press"
          onPress={() => this.props.onPress(this.props.index)}
        />
      </View>
    );
  }
}

// Usage
<CustomCard onPress={this.onPress} index={item.key} />
```

In Flutter, define a class to create a custom widget and then reuse the widget. You can also define and call a function that returns a reusable widget as shown in the `build` function in the following example.


```
// Flutter
class CustomCard extends StatelessWidget {
  CustomCard({@required this.index, @required
    this.onPress});

  final index;
  final Function onPress;

  @override
  Widget build(BuildContext context) {
    return Card(
      child: Column(
        children: <Widget>[
          Text('Card $index'),
          FlatButton(
            child: const Text('Press'),
            onPressed: this.onPress,
          ),
        ],
      ),
    );
  }
}

...
// Usage
CustomCard(
  index: index,
  onPress: () {
    print('Card $index');
  },
)

...
```

In the previous example, the constructor for the `CustomCard` class uses Dart's curly brace syntax `{ }` to indicate named [optional parameters](#).

To require these fields, either remove the curly braces from the constructor, or add `@required` to the constructor.

The following screenshots show an example of the reusable `CustomCard` class.

 Custom cards on Android

 Custom cards on iOS

Android

iOS

Project structure and resources

Where do I start writing the code?

Start with the `lib/main.dart` file. It's autogenerated when you create a Flutter app.

```
// Dart
void main(){
  print('Hello, this is the main function.');
```

content_copy

In Flutter, the entry point file is `'projectname'/lib/main.dart` and execution starts from the `main` function.

How are files structured in a Flutter app?

When you create a new Flutter project, it builds the following directory structure. You can customize it later, but this is where you start.

content_copy

```
└─ projectname
  └─
    └─ android      - Contains Android-specific files.
    └─ build        - Stores iOS and Android build files.
    └─ ios          - Contains iOS-specific files.
    └─ lib          - Contains externally accessible Dart source
files.
      └─
        └─ src      - Contains additional source files.
        └─ main.dart - The Flutter entry point and the start of a new
app.
This is generated automatically when you create
a Flutter
project.
It's where you start writing your Dart code.
└─ test          - Contains automated test files.
└─ pubspec.yaml - Contains the metadata for the Flutter app.
This is equivalent to the package.json file in
React Native.
```

Where do I put my resources and assets and how do I use them?

A Flutter resource or asset is a file that is bundled and deployed with your app and is accessible at runtime. Flutter apps can include the following asset types:

- Static data such as JSON files
- Configuration files
- Icons and images (JPEG, PNG, GIF, Animated GIF, WebP, Animated WebP, BMP, and WBMP)

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
```

content_copy

The `assets` subsection specifies files that should be included with the app. Each asset is identified by an explicit path relative to the `pubspec.yaml` file, where the asset file is located. The order in which the assets are declared does not matter. The actual directory used (`assets` in this case) does not matter. However, while assets can be placed in any app directory, it's a best practice to place them in the `assets` directory.

During a build, Flutter places assets into a special archive called the *asset bundle*, which apps read from at runtime. 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. These files are also included in the asset bundle along with the specified asset. Flutter uses asset variants when choosing resolution-appropriate images for your app.

In React Native, you would add a static image by placing the image file in a source code directory and referencing it.

```
<Image source={require('./my-icon.png')} />
```

content_copy

In Flutter, add a static image to your app using the `AssetImage` class in a widget's build method.

```
image: AssetImage('assets/background.png'),
```

content_copy

For more information, see [Adding Assets and Images in Flutter](#).

How do I load images over a network?

In React Native, you would specify the `uri` in the `source` prop of the `Image` component and also provide the size if needed.

In Flutter, use the `Image.network` constructor to include an image from a URL.

```
// Flutter
body: Image.network(
  'https://flutter.io/images/owl.jpg',
```

content_copy

How do I install packages and package plugins?

Flutter supports using shared packages contributed by other developers to the Flutter and Dart ecosystems. This allows you to quickly build your app without having to develop everything from scratch. Packages that contain platform-specific code are known as package plugins.

In React Native, you would use `yarn add {package-name}` or `npm install --save {package-name}` to install packages from the command line.

In Flutter, install a package using the following instructions:

1. Add the package name and version to the `pubspec.yaml` dependencies section. The example below shows how to add the `google_sign_in` Dart package to the `pubspec.yaml` file. Check your spaces when working in the YAML file because **white space matters!**

```
dependencies:
  flutter:
    sdk: flutter
  google_sign_in: ^3.0.3
```

content_copy

1. Install the package from the command line by using `flutter pub get`. If using an IDE, it often runs `flutter pub get` for you, or it might prompt you to do so.
2. Import the package into your app code as shown below:

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

content_copy

For more information, see [Using Packages](#) and [Developing Packages & Plugins](#).

You can find many packages shared by Flutter developers in the [Flutter packages](#) section of [pub.dev](#).

Flutter widgets

In Flutter, you build your UI out of widgets that describe what their view should look like given their current configuration and state.

Widgets are often composed of many small, single-purpose widgets that are nested to produce powerful effects. For example, the `Container` widget consists of several widgets responsible for layout, painting, positioning, and sizing. Specifically, the `Container` widget includes the `LimitedBox`, `ConstrainedBox`, `Align`, `Padding`, `DecoratedBox`, and `Transform` widgets. Rather than subclassing `Container` to produce a customized effect, you can compose these and other simple widgets in new and unique ways.

The `Center` widget is another example of how you can control the layout. To center a widget, wrap it in a `Center` widget and then use layout widgets for alignment, row, columns, and grids. These layout widgets do not have a visual representation of their own. Instead, their sole purpose is to control some aspect of another widget's layout. To understand why a widget renders in a certain way, it's often helpful to inspect the neighboring widgets.

For more information, see the [Flutter Technical Overview](#).

For more information about the core widgets from the `widgets` package, see [Flutter Basic Widgets](#), the [Flutter Widget Catalog](#), or the [Flutter Widget Index](#).

Views

What is the equivalent of the `View` container?

In React Native, `View` is a container that supports layout with `Flexbox`, style, touch handling, and accessibility controls.

In Flutter, you can use the core layout widgets in the [Widgets](#) library, such as [Container](#), [Column](#), [Row](#), and [Center](#). For more information, see the [Layout Widgets](#) catalog.

What is the equivalent of `FlatList` or `SectionList`?

A [List](#) is a scrollable list of components arranged vertically.

In React Native, `FlatList` or `SectionList` are used to render simple or sectioned lists.

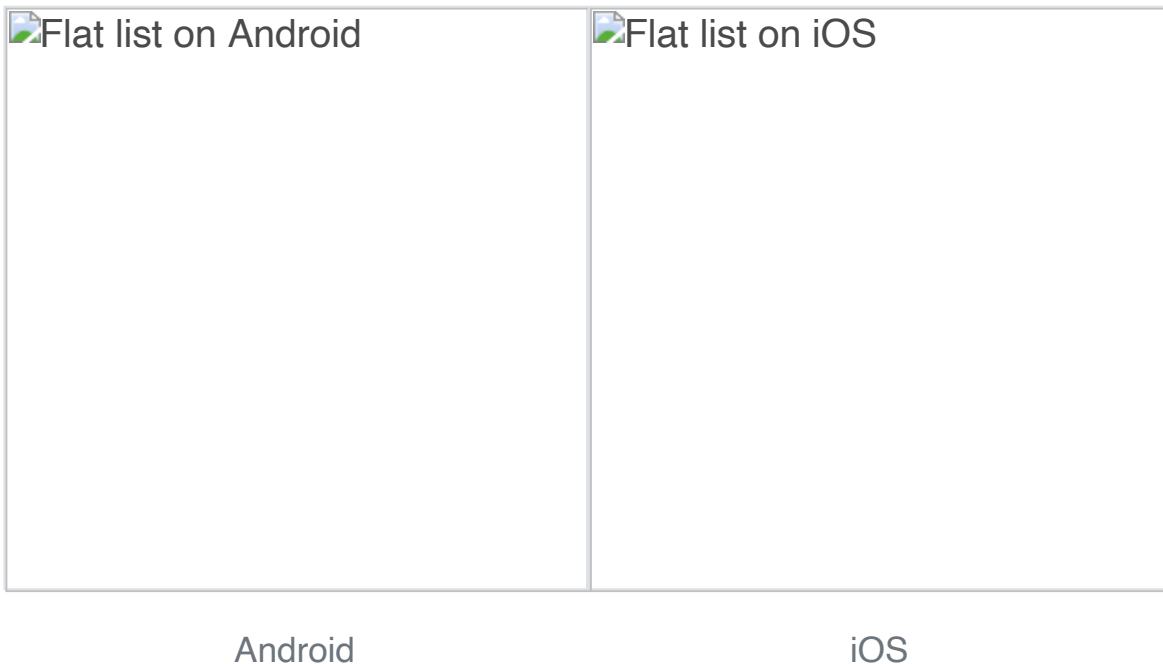
```
// React Native
<FlatList
  data={[ ... ]}
  renderItem={({ item }) => <Text>{item.key}</Text>}
/>
```

[content_copy](#)

[ListView](#) is Flutter's most commonly used scrolling widget. The default constructor takes an explicit list of children. [ListView](#) is most appropriate for a small number of widgets. For a large or infinite list, use `ListView.builder`, which builds its children on demand and only builds those children that are visible.

```
// Flutter
var data = [ ... ];
ListView.builder(
  itemCount: data.length,
  itemBuilder: (context, int index) {
    return Text(
      data[index],
    );
  },
)
```

[content_copy](#)



To learn how to implement an infinite scrolling list, see the [Write Your First Flutter App, Part 1](#) codelab.

How do I use a Canvas to draw or paint?

In React Native, canvas components aren't present so third party libraries like `react-native-canvas` are used.

```
// React Native
handleCanvas = canvas => {
  const ctx = canvas.getContext('2d');
  ctx.fillStyle = 'skyblue';
  ctx.beginPath();
  ctx.arc(75, 75, 50, 0, 2 * Math.PI);
  ctx.fillRect(150, 100, 300, 300);
  ctx.stroke();
};

render() {
  return (
    <View>
      <Canvas ref={this.handleCanvas} />
    </View>
  );
}
```

content_copy

In Flutter, you can use the [CustomPaint](#) and [CustomPainter](#) classes to draw to the canvas.

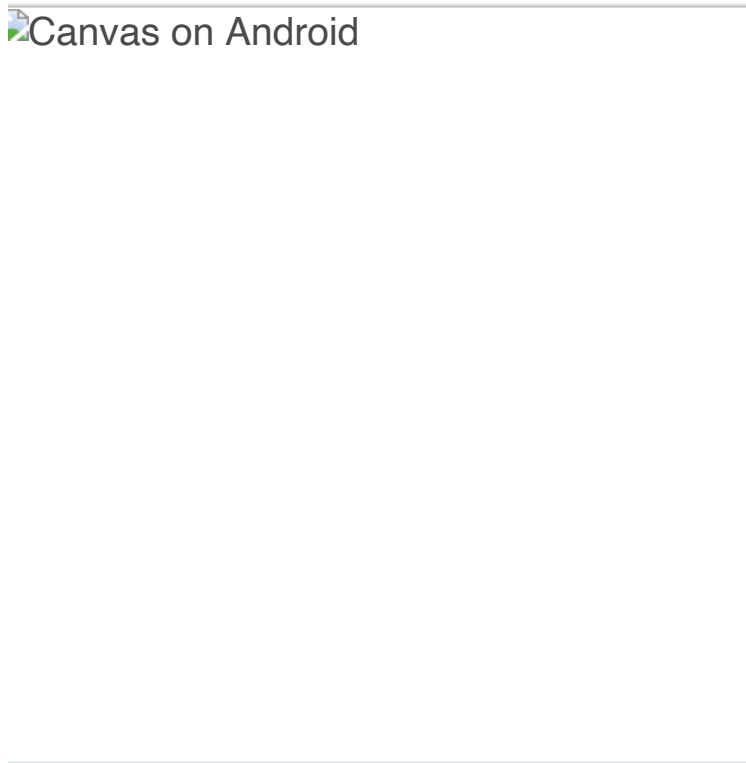
The following example shows how to draw during the paint phase using the [CustomPaint](#) widget. It implements the abstract class, [CustomPainter](#), and passes it to [CustomPaint](#)'s painter property. [CustomPaint](#) subclasses must implement the [paint\(\)](#) and [shouldRepaint\(\)](#) methods.

```
// Flutter content_copy
class MyCanvasPainter extends CustomPainter {

  @override
  void paint(Canvas canvas, Size size) {
    Paint paint = Paint();
    paint.color = Colors.amber;
    canvas.drawCircle(Offset(100.0, 200.0), 40.0, paint);
    Paint paintRect = Paint();
    paintRect.color = Colors.lightBlue;
    Rect rect = Rect.fromPoints(Offset(150.0, 300.0),
Offset(300.0, 400.0));
    canvas.drawRect(rect, paintRect);
  }

  bool shouldRepaint(MyCanvasPainter oldDelegate) => false;
  bool shouldRebuildSemantics(MyCanvasPainter oldDelegate) =>
false;
}
class _MyCanvasState extends State<MyCanvas> {

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: CustomPaint(
        painter: MyCanvasPainter(),
      ),
    );
  }
}
```

A large, empty rectangular area representing a canvas for Android.A large, empty rectangular area representing a canvas for iOS.

Android

iOS

Layouts

How do I use widgets to define layout properties?

In React Native, most of the layout can be done with the props that are passed to a specific component. For example, you could use the `style` prop on the `View` component in order to specify the flexbox properties. To arrange your components in a column, you would specify a prop such as: `flexDirection: "column"`.

```
// React Native
<View
  style={{
    flex: 1,
    flexDirection: 'column',
    justifyContent: 'space-between',
    alignItems: 'center'
  }}
>
```

content_copy

In Flutter, the layout is primarily defined by widgets specifically designed to provide layout, combined with control widgets and their style properties.

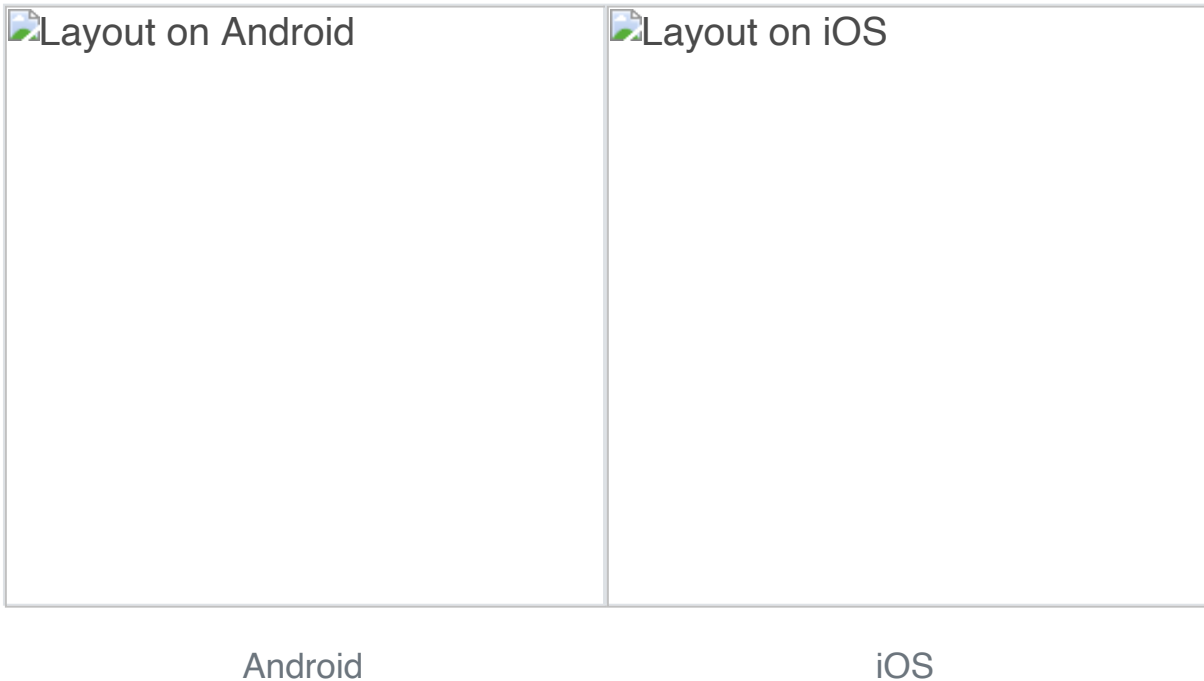
For example, the [Column](#) and [Row](#) widgets take an array of children and align them vertically and horizontally respectively. A [Container](#) widget takes a combination of layout and styling properties, and a [Center](#) widget centers its child widgets.

```
// Flutter
Center(
  child: Column(
    children: <Widget>[
      Container(
        color: Colors.red,
        width: 100.0,
        height: 100.0,
      ),
      Container(
        color: Colors.blue,
        width: 100.0,
        height: 100.0,
      ),
      Container(
        color: Colors.green,
        width: 100.0,
        height: 100.0,
      ),
    ],
  ),
)
```

content_copy

Flutter provides a variety of layout widgets in its core widget library. For example, [Padding](#), [Align](#), and [Stack](#).

For a complete list, see [Layout Widgets](#).



How do I layer widgets?

In React Native, components can be layered using [absolute](#) positioning.

Flutter uses the [Stack](#) widget to arrange children widgets in layers. The widgets can entirely or partially overlap the base widget.

The [Stack](#) widget positions its children relative to the edges of its box. This class is useful if you simply want to overlap several children widgets.

```
// Flutter
Stack(
  alignment: const Alignment(0.6, 0.6),
  children: <Widget>[
    CircleAvatar(
      backgroundImage: NetworkImage(
        'https://avatars3.githubusercontent.com/u/14101776?v=4'),
    ),
    Container(
      decoration: BoxDecoration(
        color: Colors.black45,
      ),
      child: Text('Flutter'),
    ),
  ],
)
```

content_copy

The previous example 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` coordinates.

 Stack on Android

 Stack on iOS

Android

iOS

For more information, see the [Stack](#) class documentation.

Styling

How do I style my components?

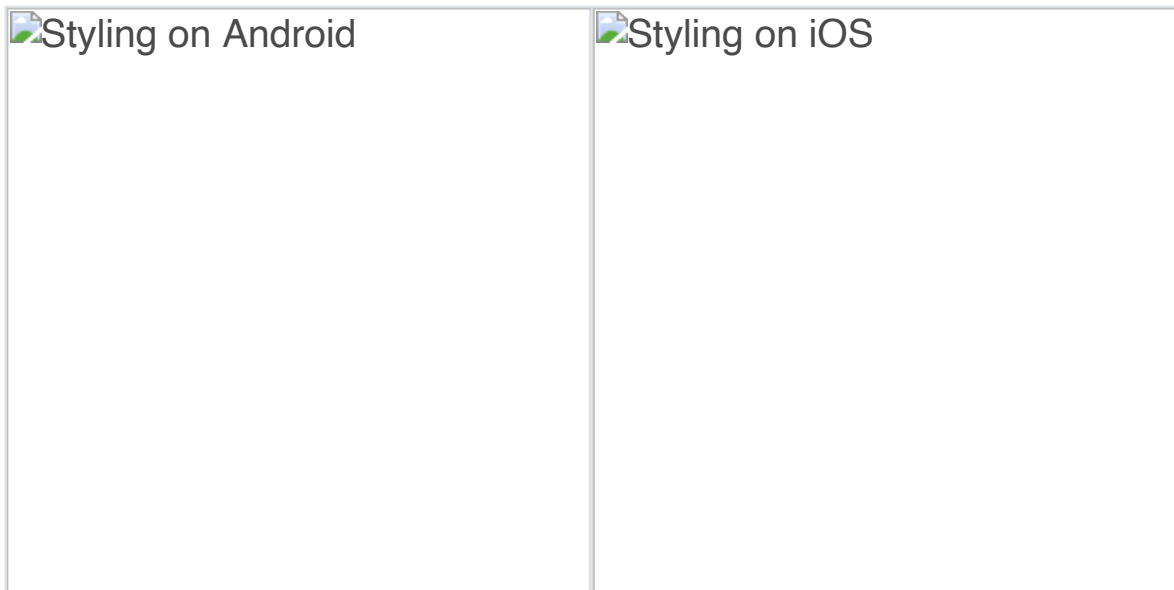
In React Native, inline styling and `stylesheets.create` are used to style components.

```
// React Native
<View style={styles.container}>
  <Text style={{ fontSize: 32, color: 'cyan', fontWeight: '600' }}>
    This is a sample text
  </Text>
</View>

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center'
  }
});
```

In Flutter, a `Text` widget can take a `TextStyle` class for its style property. If you want to use the same text style in multiple places, you can create a `TextStyle` class and use it for multiple `Text` widgets.

```
// Flutter
var textStyle = TextStyle(fontSize: 32.0, color: Colors.cyan,
fontWeight:
  FontWeight.w600);
...
Center(
  child: Column(
    children: <Widget>[
      Text(
        'Sample text',
        style: textStyle,
      ),
      Padding(
        padding: EdgeInsets.all(20.0),
        child: Icon(Icons.lightbulb_outline,
          size: 48.0, color: Colors.redAccent)
        ),
    ],
  ),
)
```



Android

iOS

How do I use **Icons** and **Colors**?

React Native doesn't include support for icons so third party libraries are used.

In Flutter, importing the Material library also pulls in the rich set of [Material icons](#) and [colors](#).

```
Icon(Icons.lightbulb_outline, color: Colors.redAccent) content_copy
```

When using the **Icons** class, make sure to set `uses-material-design: true` in the project's `pubspec.yaml` file. This ensures that the **MaterialIcons** font, which displays the icons, is included in your app.

```
name: my_awesome_application content_copy
flutter: uses-material-design: true
```

Flutter's [Cupertino \(iOS-style\)](#) package provides high fidelity widgets for the current iOS design language. To use the **CupertinoIcons** font, add a dependency for `cupertino_icons` in your project's `pubspec.yaml` file.

```
name: my_awesome_application content_copy
dependencies:
 /cupertino_icons: ^0.1.0
```

To globally customize the colors and styles of components, use [ThemeData](#) to specify default colors for various aspects of the theme. Set the theme property in [MaterialApp](#) to the [ThemeData](#) object. The [Colors](#) class provides colors from the Material Design [color palette](#).

The following example sets the primary swatch to [blue](#) and the text selection to [red](#).

```
class SampleApp extends StatelessWidget {                                     content_copy
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Sample App',
      theme: ThemeData(
        primarySwatch: Colors.blue,
        textSelectionColor: Colors.red
      ),
      home: SampleAppPage(),
    );
  }
}
```

How do I add style themes?

In React Native, common themes are defined for components in stylesheets and then used in components.

In Flutter, create uniform styling for almost everything by defining the styling in the [ThemeData](#) class and passing it to the theme property in the [MaterialApp](#) widget.

```
@override                                                                 content_copy
Widget build(BuildContext context) {
  return MaterialApp(
    theme: ThemeData(
      primaryColor: Colors.cyan,
      brightness: Brightness.dark,
    ),
    home: StylingPage(),
  );
}
```


A [Theme](#) can be applied even without using the [MaterialApp](#) widget. The [Theme](#) widget takes a [ThemeData](#) in its `data` parameter and applies the [ThemeData](#) to all of its children widgets.

```
@override
Widget build(BuildContext context) {
  return Theme(
    data: ThemeData(
      primaryColor: Colors.cyan,
      brightness: brightness,
    ),
    child: Scaffold(
      backgroundColor: Theme.of(context).primaryColor,
      ...
    ),
  );
}
```

content_copy

State management

State is information that can be read synchronously when a widget is built or information that might change during the lifetime of a widget. To manage app state in Flutter, use a [StatefulWidget](#) paired with a `State` object.

For more information on ways to approach managing state in Flutter, see [State management](#).

The StatelessWidget

A [StatelessWidget](#) in Flutter is a widget that doesn't require a state change—it has no internal state to manage.

Stateless widgets are useful when the part of the user interface you are describing does not depend on anything other than the configuration information in the object itself and the [BuildContext](#) in which the widget is inflated.

[AboutDialog](#), [CircleAvatar](#), and [Text](#) are examples of stateless widgets that subclass [StatelessWidget](#).

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

void main() => runApp(MyStatelessWidget(text: 'StatelessWidget
Example to show immutable data'));

class MyStatelessWidget extends StatelessWidget {
  final String text;
  MyStatelessWidget({Key key, this.text}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Center(
      child: Text(
        text,
        textDirection: TextDirection.ltr,
      ),
    );
  }
}
```

The previous example uses the constructor of the `MyStatelessWidget` class to pass the `text`, which is marked as `final`. This class extends `StatelessWidget`—it contains immutable data.

The `build` method of a stateless widget is typically called in only three situations:

- When the widget is inserted into a tree
- When the widget's parent changes its configuration
- When an [InheritedWidget](#) it depends on, changes

The StatefulWidget

A [StatefulWidget](#) is a widget that changes state. Use the `setState` method to manage the state changes for a `StatefulWidget`. A call to `setState()` tells the Flutter framework that something has changed in a state, which causes an app to rerun the `build()` method so that the app can reflect the change.

State is information that can be read synchronously when a widget is built and might change during the lifetime of the widget. It's the responsibility of the widget implementer to ensure that the state object is promptly notified when the state

changes. Use `StatefulWidget` when a widget can change dynamically. For example, the state of the widget changes by typing into a form, or moving a slider. Or, it can change over time—perhaps a data feed updates the UI.

[Checkbox](#), [Radio](#), [Slider](#), [InkWell](#), [Form](#), and [TextField](#) are examples of stateful widgets that subclass `StatefulWidget`.

The following example declares a `StatefulWidget` that requires a `createState()` method. This method creates the state object that manages the widget's state, `_MyStatefulWidgetState`.

```
class StatefulWidget extends StatefulWidget {                                content_copy
  StatefulWidget({Key key, this.title}) : super(key: key);
  final String title;

  @override
  _MyStatefulWidgetState createState() =>
  _MyStatefulWidgetState();
}
```

The following state class, `_MyStatefulWidgetState`, implements the `build()` method for the widget. When the state changes, for example, when the user toggles the button, `setState()` is called with the new toggle value. This causes the framework to rebuild this widget in the UI.

```

class _MyStatefulWidgetState extends State<MyStatefulWidget> {
  bool showtext=true;
  bool toggleState=true;
  Timer t2;

  void toggleBlinkState(){
    setState((){
      toggleState=!toggleState;
    });
    var twenty = const Duration(milliseconds: 1000);
    if(toggleState==false) {
      t2 = Timer.periodic(twenty, (Timer t) {
        toggleShowText();
      });
    } else {
      t2.cancel();
    }
  }

  void toggleShowText(){
    setState((){
      showtext=!showtext;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: Column(
          children: <Widget>[
            (showtext
              ?(Text('This execution will be done before you can
blink.'))
              :(Container()))
            ),
            Padding(
              padding: EdgeInsets.only(top: 70.0),
              child: RaisedButton(
                onPressed: toggleBlinkState,
                child: (toggleState
                  ?( Text('Blink'))
                  :(Text('Stop Blinking'))
                )
              )
            )
          ],
        ),
      ),
    ),
  ),

```

```
}  
}  
) ;
```

What are the StatefulWidget and StatelessWidget best practices?

Here are a few things to consider when designing your widget.

1. Determine whether a widget should be a StatefulWidget or a StatelessWidget.

In Flutter, widgets are either Stateful or Stateless—depending on whether they depend on a state change.

- If a widget changes—the user interacts with it or a data feed interrupts the UI, then it's Stateful.
- If a widget is final or immutable, then it's Stateless.

1. Determine which object manages the widget's state (for a StatefulWidget).

In Flutter, there are three primary ways to manage state:

- The widget manages its own state
- The parent widget manages the widget's state
- A mix-and-match approach

When deciding which approach to use, consider the following principles:

- If the state in question is user data, for example the checked or unchecked mode of a checkbox, or the position of a slider, then the state is best managed by the parent widget.
- If the state in question is aesthetic, for example an animation, then the widget itself best manages the state.
- When in doubt, let the parent widget manage the child widget's state.

1. Subclass StatefulWidget and State.

The `MyStatefulWidget` class manages its own state—it extends `StatefulWidget`, it overrides the `createState()` method to create the `State` object, and the framework calls `createState()` to build the widget. In this example, `createState()` creates an

instance of `_MyStatefulWidgetState`, which is implemented in the next best practice.

```
class MyStatefulWidget extends StatefulWidget {                                content_copy
  MyStatefulWidget({Key key, this.title}) : super(key: key);
  final String title;

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

class _MyStatefulWidgetState extends State<MyStatefulWidget> {

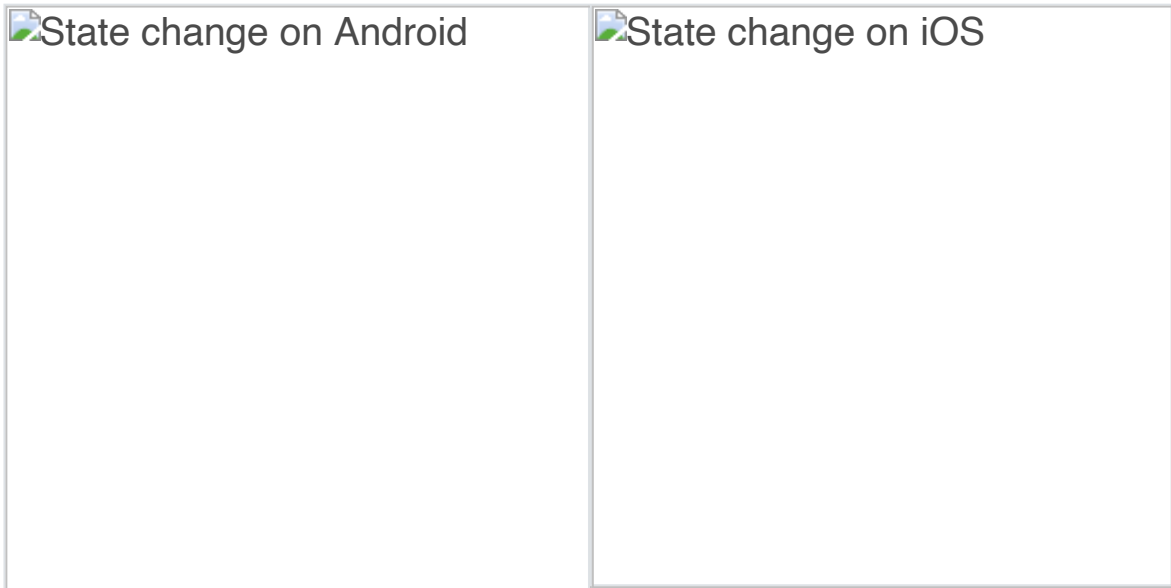
  @override
  Widget build(BuildContext context) {
    ...
  }
}
```

1. Add the `StatefulWidget` into the widget tree.

Add your custom `StatefulWidget` to the widget tree in the app's build method.

```
class MyStatelessWidget extends StatelessWidget {                                content_copy
  // This widget is the root of your application.

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: MyStatefulWidget(title: 'State Change Demo'),
    );
  }
}
```



Android

iOS

Props

In React Native, most components can be customized when they are created with different parameters or properties, called `props`. These parameters can be used in a child component using `this.props`.

```
// React Native
class CustomCard extends React.Component {
  render() {
    return (
      <View>
        <Text> Card {this.props.index} </Text>
        <Button
          title='Press'
          onPress={() => this.props.onPress(this.props.index)}
        />
      </View>
    );
  }
}

class App extends React.Component {

  onPress = index => {
    console.log('Card ', index);
  };

  render() {
    return (
      <View>
        <FlatList
          data={[ ... ]}
          renderItem={({ item }) => (
            <CustomCard onPress={this.onPress} index={item.key} />
          )}
        />
      </View>
    );
  }
}
```

In Flutter, you assign a local variable or function marked `final` with the property received in the parameterized constructor.


```
// Flutter
class CustomCard extends StatelessWidget {

  CustomCard({@required this.index, @required this.onPress});
  final index;
  final Function onPress;

  @override
  Widget build(BuildContext context) {
    return Card(
      child: Column(
        children: <Widget>[
          Text('Card $index'),
          FlatButton(
            child: const Text('Press'),
            onPressed: this.onPress,
          ),
        ],
      ));
  }
}

...
//Usage
CustomCard(
  index: index,
  onPress: () {
    print('Card $index');
  },
)
```

A screenshot of the 'Cards on Android' app, showing a grid of cards.A screenshot of the 'Cards on iOS' app, showing a grid of cards.

Android

iOS

Local storage

If you don't need to store a lot of data and it doesn't require structure, you can use `shared_preferences` which allows you to read and write persistent key-value pairs of primitive data types: booleans, floats, ints, longs, and strings.

How do I store persistent key-value pairs that are global to the app?

In React Native, you use the `setItem` and `getItem` functions of the `AsyncStorage` component to store and retrieve data that is persistent and global to the app.

```
// React Native
await AsyncStorage.setItem( 'counterkey',
json.stringify(++this.state.counter));
AsyncStorage.getItem( 'counterkey' ).then(value => {
  if (value != null) {
    this.setState({ counter: value });
  }
});
```

content_copy

In Flutter, use the [shared_preferences](#) plugin to store and retrieve key-value data that is persistent and global to the app. The [shared_preferences](#) plugin wraps `NSUserDefaults` on iOS and `SharedPreferences` on Android, providing a persistent store for simple data. To use the plugin, add [shared_preferences](#) as a dependency in the `pubspec.yaml` file then import the package in your Dart file.

```
dependencies:
  flutter:
    sdk: flutter
  shared_preferences: ^0.4.3
```

content_copy

```
// Dart
import 'package:shared_preferences/shared_preferences.dart';
```

content_copy

To implement persistent data, use the setter methods provided by the `SharedPreferences` class. Setter methods are available for various primitive types, such as `setInt`, `setBool`, and `setString`. To read data, use the appropriate getter method provided by the `SharedPreferences` class. For each setter there is a corresponding getter method, for example, `getInt`, `getBool`, and `getString`.

```
SharedPreferences prefs = await SharedPreferences.getInstance();
_counter = prefs.getInt( 'counter' );
prefs.setInt( 'counter', ++_counter );
setState(() {
  _counter = _counter;
});
```

content_copy

Routing

Most apps contain several screens for displaying different types of information. For example, you might have a product screen that displays images where users could tap on a product image to get more information about the product on a new screen.

In Android, new screens are new Activities. In iOS, new screens are new ViewControllers. In Flutter, screens are just Widgets! And to navigate to new screens in Flutter, use the Navigator widget.

How do I navigate between screens?

In React Native, there are three main navigators: StackNavigator, TabNavigator, and DrawerNavigator. Each provides a way to configure and define the screens.

```
// React Native
const MyApp = TabNavigator(
  { Home: { screen: HomeScreen }, Notifications: { screen:
tabNavScreen } },
  { tabBarOptions: { activeTintColor: '#e91e63' } }
);
const SimpleApp = StackNavigator({
  Home: { screen: MyApp },
  stackScreen: { screen: StackScreen }
});
export default (MyApp1 = DrawerNavigator({
  Home: {
    screen: SimpleApp
  },
  Screen2: {
    screen: drawerScreen
  }
})));
```

content_copy

In Flutter, there are two main widgets used to navigate between screens:

- A [Route](#) is an abstraction for an app screen or page.
- A [Navigator](#) is a widget that manages routes.

A [Navigator](#) is defined as a widget that manages a set of child widgets with a stack discipline. The navigator manages a stack of [Route](#) objects and provides methods for managing the stack, like [Navigator.push](#) and [Navigator.pop](#). A list of routes might be specified in the [MaterialApp](#) widget, or they might be built on the fly, for example, in hero animations. The following example specifies named routes in the [MaterialApp](#) widget.

```
// Flutter
class NavigationApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      routes: <String, WidgetBuilder>{
        '/a': (BuildContext context) => usualNavscreen(),
        '/b': (BuildContext context) => drawerNavscreen(),
      },
    );
  }
}
```

To navigate to a named route, the [Navigator.of\(\)](#) method is used to specify the [BuildContext](#) (a handle to the location of a widget in the widget tree). The name of the route is passed to the [pushNamed](#) function to navigate to the specified route.

```
Navigator.of(context).pushNamed('/a');
```

You can also use the push method of [Navigator](#) which adds the given [Route](#) to the history of the navigator that most tightly encloses the given [BuildContext](#), and transitions to it. In the following example, the [MaterialPageRoute](#) widget is a modal route that replaces the entire screen with a platform-adaptive transition. It takes a [WidgetBuilder](#) as a required parameter.

```
Navigator.push(context, MaterialPageRoute(builder: (BuildContext context)
=> UsualNavscreen()));
```

How do I use tab navigation and drawer navigation?

In Material Design apps, there are two primary options for Flutter navigation: tabs and drawers. When there is insufficient space to support tabs, drawers provide a good alternative.

Tab navigation

In React Native, `createBottomTabNavigator` and `TabNavigation` are used to show tabs and for tab navigation.

```
// React Native content_copy  
import { createBottomTabNavigator } from 'react-navigation';  
  
const MyApp = TabNavigator(  
  { Home: { screen: HomeScreen }, Notifications: { screen:  
tabNavScreen } },  
  { tabBarOptions: { activeTintColor: '#e91e63' } }  
);
```

Flutter provides several specialized widgets for drawer and tab navigation:

[TabController](#)

Coordinates the tab selection between a `TabBar` and a `TabBarView`.

[TabBar](#)

Displays a horizontal row of tabs.

[Tab](#)

Creates a material design `TabBar` tab.

[TabBarView](#)

Displays the widget that corresponds to the currently selected tab.

```
// Flutter content_copy  
TabController controller=TabController(length: 2, vsync: this);  
  
TabBar(  
  tabs: <Tab>[  
    Tab(icon: Icon(Icons.person)),  
    Tab(icon: Icon(Icons.email)),  
  ],  
  controller: controller,  
),
```

A `TabController` is required to coordinate the tab selection between a `TabBar` and a `TabBarView`. The `TabController` constructor `length` argument is the total number of tabs. A `TickerProvider` is required to trigger the notification whenever a frame

triggers a state change. The `TickerProvider` is `vsync`. Pass the `vsync: this` argument to the `TabController` constructor whenever you create a new `TabController`.

The `TickerProvider` is an interface implemented by classes that can vend `Ticker` objects. Tickers can be used by any object that must be notified whenever a frame triggers, but they're most commonly used indirectly via an `AnimationController`. `AnimationControllers` need a `TickerProvider` to obtain their `Ticker`. If you are creating an `AnimationController` from a `State`, then you can use the `TickerProviderStateMixin` or `SingleTickerProviderStateMixin` classes to obtain a suitable `TickerProvider`.

The `Scaffold` widget wraps a new `TabBar` widget and creates two tabs. The `TabBarView` widget is passed as the `body` parameter of the `Scaffold` widget. All screens corresponding to the `TabBar` widget's tabs are children to the `TabBarView` widget along with the same `TabController`.

// Flutter

content_copy

```
class _NavigationHomePageState extends State<NavigationHomePage>
with SingleTickerProviderStateMixin {
  TabController controller=TabController(length: 2, vsync: this);
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      bottomNavigationBar: Material (
        child: TabBar(
          tabs: <Tab> [
            Tab(icon: Icon(Icons.person),),
            Tab(icon: Icon(Icons.email),),
          ],
          controller: controller,
        ),
        color: Colors.blue,
      ),
      body: TabBarView(
        children: <Widget> [
          home.homeScreen(),
          tabScreen.tabScreen()
        ],
        controller: controller,
      )
    );
  }
}
```

Drawer navigation

In React Native, import the needed react-navigation packages and then use `createDrawerNavigator` and `DrawerNavigation`.

```
// React Native
export default (MyApp1 = DrawerNavigator({
  Home: {
    screen: SimpleApp
  },
  Screen2: {
    screen: drawerScreen
  }
})));
```

content_copy

In Flutter, we can use the `Drawer` widget in combination with a `Scaffold` to create a layout with a Material Design drawer. To add a `Drawer` to an app, wrap it in a `Scaffold` widget. The `Scaffold` widget provides a consistent visual structure to apps that follow the [Material Design](#) guidelines. It also supports special Material Design components, such as `Drawers`, `AppBars`, and `SnackBars`.

The `Drawer` widget is a Material Design panel that slides in horizontally from the edge of a `Scaffold` to show navigation links in an application. You can provide a `RaisedButton`, a `Text` widget, or a list of items to display as the child to the `Drawer` widget. In the following example, the `ListTile` widget provides the navigation on tap.

```
// Flutter
Drawer(
  child: ListTile(
    leading: Icon(Icons.change_history),
    title: Text('Screen2'),
    onTap: () {
      Navigator.of(context).pushNamed('/b');
    },
  ),
  elevation: 20.0,
),
```

content_copy

The `Scaffold` widget also includes an `AppBar` widget that automatically displays an appropriate `IconButton` to show the `Drawer` when a `Drawer` is available in the `Scaffold`. The `Scaffold` automatically handles the edge-swipe gesture to show the `Drawer`.


```
// Flutter
@override
Widget build(BuildContext context) {
  return Scaffold(
    drawer: Drawer(
      child: ListTile(
        leading: Icon(Icons.change_history),
        title: Text('Screen2'),
        onTap: () {
          Navigator.of(context).pushNamed('/b');
        },
      ),
    ),
    elevation: 20.0,
  ),
  appBar: AppBar(
    title: Text('Home'),
  ),
  body: Container(),
);
}
```

 Navigation on Android Navigation on iOS

Android

iOS

Gesture detection and touch event handling

To listen for and respond to gestures, Flutter supports taps, drags, and scaling. The gesture system in Flutter has two separate layers. The first layer includes raw pointer events, which describe the location and movement of pointers, (such as touches, mice, and styli movements), across the screen. The second layer includes gestures, which describe semantic actions that consist of one or more pointer movements.

How do I add a click or press listeners to a widget?

In React Native, listeners are added to components using [PanResponder](#) or the [Touchable](#) components.

```
// React Native
<TouchableOpacity
  onPress={() => {
    console.log('Press');
  }}
  onLongPress={() => {
    console.log('Long Press');
  }}
>
  <Text>Tap or Long Press</Text>
</TouchableOpacity>
```

content_copy

For more complex gestures and combining several touches into a single gesture, [PanResponder](#) is used.

```
// React Native
class App extends Component {

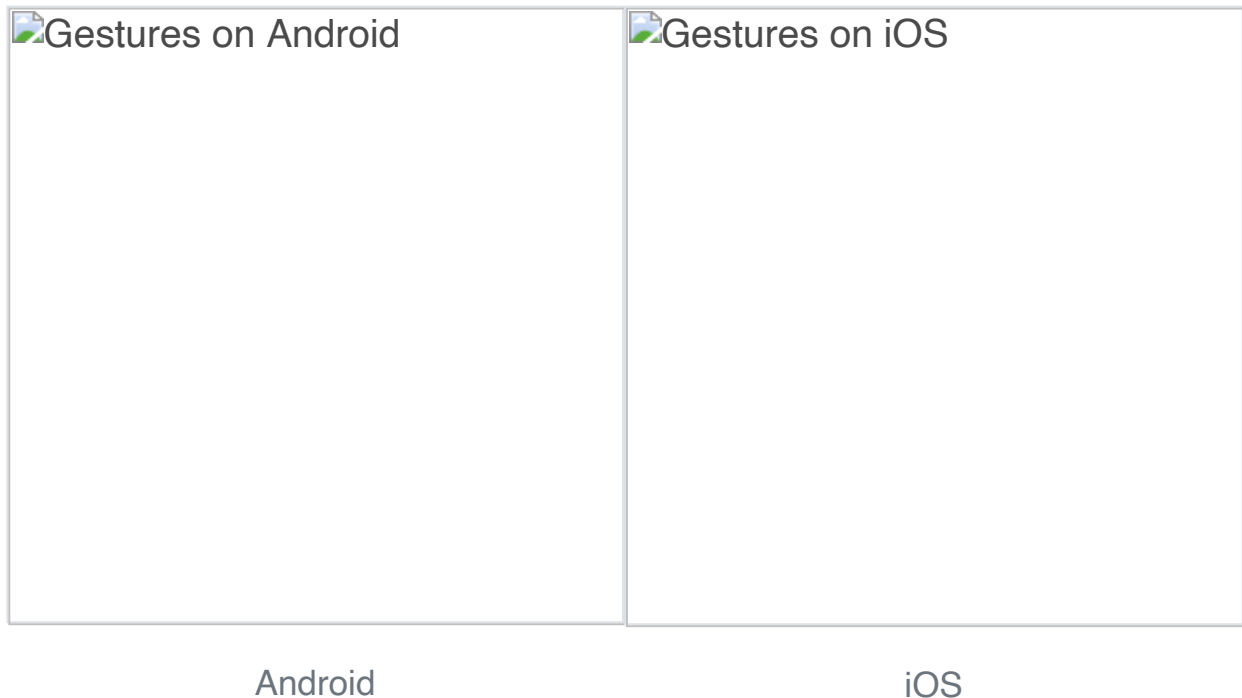
  componentWillMount() {
    this._panResponder = PanResponder.create({
      onMoveShouldSetPanResponder: (event, gestureState) =>
        !!getDirection(gestureState),
      onPanResponderMove: (event, gestureState) => true,
      onPanResponderRelease: (event, gestureState) => {
        const drag = getDirection(gestureState);
      },
      onPanResponderTerminationRequest: (event, gestureState) =>
true
    });
  }

  render() {
    return (
      <View style={styles.container}
{...this._panResponder.panHandlers}>
        <View style={styles.center}>
          <Text>Swipe Horizontally or Vertically</Text>
        </View>
      </View>
    );
  }
}
```

In Flutter, to add a click (or press) listener to a widget, use a button or a touchable widget that has an `onPress`: field. Or, add gesture detection to any widget by wrapping it in a [GestureDetector](#).

```
// Flutter
GestureDetector(
  child: Scaffold(
    appBar: AppBar(
      title: Text('Gestures'),
    ),
    body: Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          Text('Tap, Long Press, Swipe Horizontally or Vertically
'),
        ],
      ),
    ),
  ),
  onTap: () {
    print('Tapped');
  },
  onLongPress: () {
    print('Long Pressed');
  },
  onVerticalDragEnd: (DragEndDetails value) {
    print('Swiped Vertically');
  },
  onHorizontalDragEnd: (DragEndDetails value) {
    print('Swiped Horizontally');
  },
);
```

For more information, including a list of Flutter [GestureDetector](#) callbacks, see the [GestureDetector class](#).



Making HTTP network requests

Fetching data from the internet is common for most apps. And in Flutter, the `http` package provides the simplest way to fetch data from the internet.

How do I fetch data from API calls?

React Native provides the Fetch API for networking—you make a fetch request and then receive the response to get the data.

```
// React Native
_getIPAddress = () => {
  fetch('https://httpbin.org/ip')
    .then(response => response.json())
    .then(responseJson => {
      this.setState({ _ipAddress: responseJson.origin });
    })
    .catch(error => {
      console.error(error);
    });
};
```

content_copy

Flutter uses the [http](#) package. To install the [http](#) package, add it to the dependencies section of our pubspec.yaml.

```
dependencies:
  flutter:
    sdk: flutter
  http: <latest_version>
```

content_copy

Flutter uses the [dart:io](#) core HTTP support client. To create an HTTP Client, import [dart:io](#).

```
import 'dart:io';
```

content_copy

The client supports the following HTTP operations: GET, POST, PUT, and DELETE.

```
// Flutter
final url = Uri.https('httpbin.org', 'ip');
final httpClient = HttpClient();
_getIPAddress() async {
  var request = await httpClient.getUrl(url);
  var response = await request.close();
  var responseBody = await
response.transform(utf8.decoder).join();
  String ip = jsonDecode(responseBody)['origin'];
  setState(() {
    _ipAddress = ip;
  });
}
```

content_copy

 API calls on Android

 API calls on iOS

Form input

Text fields allow users to type text into your app so they can be used to build forms, messaging apps, search experiences, and more. Flutter provides two core text field widgets: [TextField](#) and [TextFormField](#).

How do I use text field widgets?

In React Native, to enter text you use a [TextInput](#) component to show a text input box and then use the callback to store the value in a variable.

```
// React Native content_copy  
<TextInput  
  placeholder="Enter your Password"  
  onChangeText={password => this.setState({ password })}  
/>  
<Button title="Submit" onPress={this.validate} />
```

In Flutter, use the [TextEditingController](#) class to manage a [TextField](#) widget. Whenever the text field is modified, the controller notifies its listeners.

Listeners read the text and selection properties to learn what the user typed into the field. You can access the text in [TextField](#) by the [text](#) property of the controller.

```
// Flutter
final TextEditingController _controller = TextEditingController();

...
TextField(
  controller: _controller,
  decoration: InputDecoration(
    hintText: 'Type something', labelText: 'Text Field '
  ),
),
RaisedButton(
  child: Text('Submit'),
  onPressed: () {
    showDialog(
      context: context,
      child: AlertDialog(
        title: Text('Alert'),
        content: Text('You typed ${_controller.text}'),
      ),
    );
  },
),
)
```

In this example, when a user clicks on the submit button an alert dialog displays the current text entered in the text field. This is achieved using an [alertDialog](#) widget that displays the alert message, and the text from the `TextField` is accessed by the `text` property of the [TextEditingController](#).

How do I use Form widgets?

In Flutter, use the [Form](#) widget where [TextFormField](#) widgets along with the submit button are passed as children. The `TextFormField` widget has a parameter called [onSaved](#) that takes a callback and executes when the form is saved. A `FormState` object is used to save, reset, or validate each `FormField` that is a descendant of this `Form`. To obtain the `FormState`, you can use `Form.of()` with a context whose ancestor is the `Form`, or pass a `GlobalKey` to the `Form` constructor and call `GlobalKey.currentState()`.


```

final formKey = GlobalKey<FormState>();
content_copy

...

Form(
  key: formKey,
  child: Column(
    children: <Widget>[
      TextFormField(
        validator: (value) => !value.contains('@') ? 'Not a valid
email.' : null,
        onSave: (val) => _email = val,
        decoration: const InputDecoration(
          hintText: 'Enter your email',
          labelText: 'Email',
        ),
      ),
      RaisedButton(
        onPressed: _submit,
        child: Text('Login'),
      ),
    ],
  ),
)

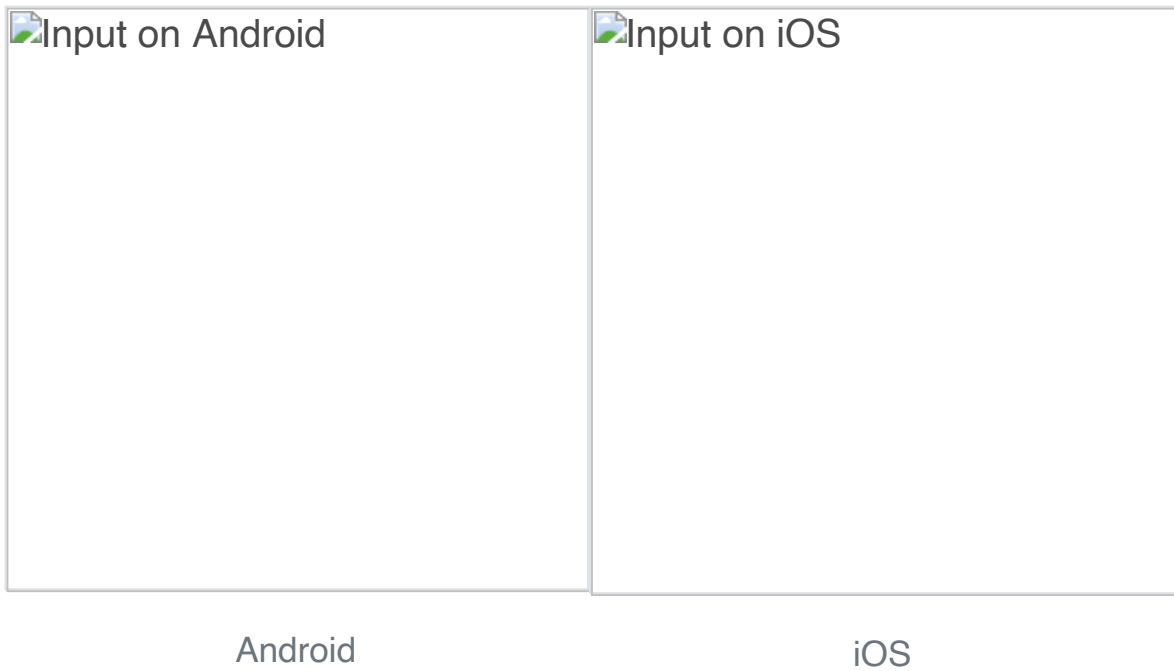
```

The following example shows how `Form.save()` and `formKey` (which is a `GlobalKey`) are used to save the form on submit.

```

void _submit() {
  final form = formKey.currentState;
  if (form.validate()) {
    form.save();
    showDialog(
      context: context,
      child: AlertDialog(
        title: Text('Alert'),
        content: Text('Email: $_email, password: $_password'),
      ),
    );
  }
}
content_copy

```



Platform-specific code

When building a cross-platform app, you want to re-use as much code as possible across platforms. However, scenarios might arise where it makes sense for the code to be different depending on the OS. This requires a separate implementation by declaring a specific platform.

In React Native, the following implementation would be used:

```
// React Native  
if (Platform.OS === 'ios') {  
  return 'iOS';  
} else if (Platform.OS === 'android') {  
  return 'android';  
} else {  
  return 'not recognised';  
}
```

content_copy

In Flutter, use the following implementation:

```
// Flutter
if (Theme.of(context).platform == TargetPlatform.iOS) {
  return 'iOS';
} else if (Theme.of(context).platform == TargetPlatform.android) {
  return 'android';
} else if (Theme.of(context).platform == TargetPlatform.fuchsia) {
  return 'fuchsia';
} else {
  return 'not recognised ';
}
```

Debugging

What tools can I use to debug my app in Flutter?

Use the [DevTools](#) suite for debugging Flutter or Dart apps.

DevTools includes support for profiling, examining the heap, inspecting the widget tree, logging diagnostics, debugging, observing executed lines of code, debugging memory leaks and memory fragmentation. For more information, see the [DevTools](#) documentation.

If you're using an IDE, you can debug your application using the IDE's debugger.

How do I perform a hot reload?

Flutter's Stateful Hot Reload feature helps you quickly and easily experiment, build UIs, add features, and fix bugs. Instead of recompiling your app every time you make a change, you can hot reload your app instantly. The app is updated to reflect your change, and the current state of the app is preserved.

In React Native, the shortcut is ⌘R for the iOS Simulator and tapping R twice on Android emulators.

In Flutter, If you are using IntelliJ IDE or Android Studio, you can select Save All (⌘s/ctrl-s), or you can click the Hot Reload button on the toolbar. If you are running the app at the command line using `flutter run`, type `r` in the Terminal window. You can also perform a full restart by typing `R` in the Terminal window.

How do I access the in-app developer menu?

In React Native, the developer menu can be accessed by shaking your device: ⌘D for the iOS Simulator or ⌘M for Android emulator.

In Flutter, if you are using an IDE, you can use the IDE tools. If you start your application using `flutter run`you can also access the menu by typing `h` in the terminal window, or type the following shortcuts:

Action	Terminal Shortcut	Debug functions
Widget hierarchy of the app	w	debugDumpApp()
Rendering tree of the app	t	debugDumpRenderTree()
Layers	L	debugDumpLayers()
Accessibility	s (traversal order) or u (inverse hit test order)	debugDumpSemantics()
To toggle the widget inspector	i	WidgetsApp.showWidgetInspector()
To toggle the display of construction lines	p	debugPaintSizeEnabled
To simulate different operating systems	o	defaultTargetPlatform

Action	Terminal Shortcut	Debug functions
To display the performance overlay	<code>P</code>	<code>WidgetsApp.showPerformance</code>
To save a screenshot to flutter.png	<code>s</code>	
To quit	<code>q</code>	

Animation

Well-designed animation makes a UI feel intuitive, contributes to the look and feel of a polished app, and improves the user experience. Flutter's animation support makes it easy to implement simple and complex animations. The Flutter SDK includes many Material Design widgets that include standard motion effects and you can easily customize these effects to personalize your app.

In React Native, Animated APIs are used to create animations.

In Flutter, use the [Animation](#) class and the [AnimationController](#) class. [Animation](#) is an abstract class that understands its current value and its state (completed or dismissed).

The [AnimationController](#) class lets you play an animation forward or in reverse, or stop animation and set the animation to a specific value to customize the motion.

How do I add a simple fade-in animation?

In the React Native example below, an animated component, [FadeInView](#) is created using the Animated API. The initial opacity state, final state, and the duration over which the transition occurs are defined. The animation component is added inside the [Animated](#) component, the opacity state [fadeAnim](#) is mapped to the opacity of the [Text](#) component that we want to animate, and then, `start()` is called to start the animation.

```
// React Native
class FadeInView extends React.Component {
  state = {
    fadeAnim: new Animated.Value(0) // Initial value for opacity:
    0
  };
  componentDidMount() {
    Animated.timing(this.state.fadeAnim, {
      toValue: 1,
      duration: 10000
    }).start();
  }
  render() {
    return (
      <Animated.View style={{...this.props.style, opacity:
this.state.fadeAnim }} >
        {this.props.children}
      </Animated.View>
    );
  }
}

...
<FadeInView>
  <Text> Fading in </Text>
</FadeInView>
...
```

To create the same animation in Flutter, create an [AnimationController](#) object named `controller` and specify the duration. By default, an [AnimationController](#) linearly produces values that range from 0.0 to 1.0, during a given duration. The animation controller generates a new value whenever the device running your app is ready to display a new frame. Typically, this rate is around 60 values per second.

When defining an [AnimationController](#), you must pass in a [vsync](#) object. The presence of [vsync](#) prevents offscreen animations from consuming unnecessary resources. You can use your stateful object as the [vsync](#) by adding [TickerProviderStateMixin](#) to the class definition. An [AnimationController](#) needs a [TickerProvider](#), which is configured using the [vsync](#) argument on the constructor.

A [Tween](#) describes the interpolation between a beginning and ending value or the mapping from an input range to an output range. To use a [Tween](#) object with an animation, call the [Tween](#) object's `animate()` method and pass it the [Animation](#) object that you want to modify.

For this example, a [FadeTransition](#) widget is used and the `opacity` property is mapped to the `animation` object.

To start the animation, use `controller.forward()`. Other operations can also be performed using the controller such as `fling()` or `repeat()`. For this example, the [FlutterLogo](#) widget is used inside the `FadeTransition` widget.

```

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

void main() {
  runApp(Center(child: LogoFade()));
}

class LogoFade extends StatefulWidget {
  _LogoFadeState createState() => _LogoFadeState();
}

class _LogoFadeState extends State<LogoFade> with
TickerProviderStateMixin {
  Animation animation;
  AnimationController controller;

  initState() {
    super.initState();
    controller = AnimationController(
      duration: const Duration(milliseconds: 3000), vsync:
this);
    final CurvedAnimation curve =
      CurvedAnimation(parent: controller, curve: Curves.easeIn);
    animation = Tween(begin: 0.0, end: 1.0).animate(curve);
    controller.forward();
  }

  Widget build(BuildContext context) {
    return FadeTransition(
      opacity: animation,
      child: Container(
        height: 300.0,
        width: 300.0,
        child: FlutterLogo(),
      ),
    );
  }

  dispose() {
    controller.dispose();
    super.dispose();
  }
}

```


Flutter fade on AndroidFlutter fade on iOS

Android

iOS

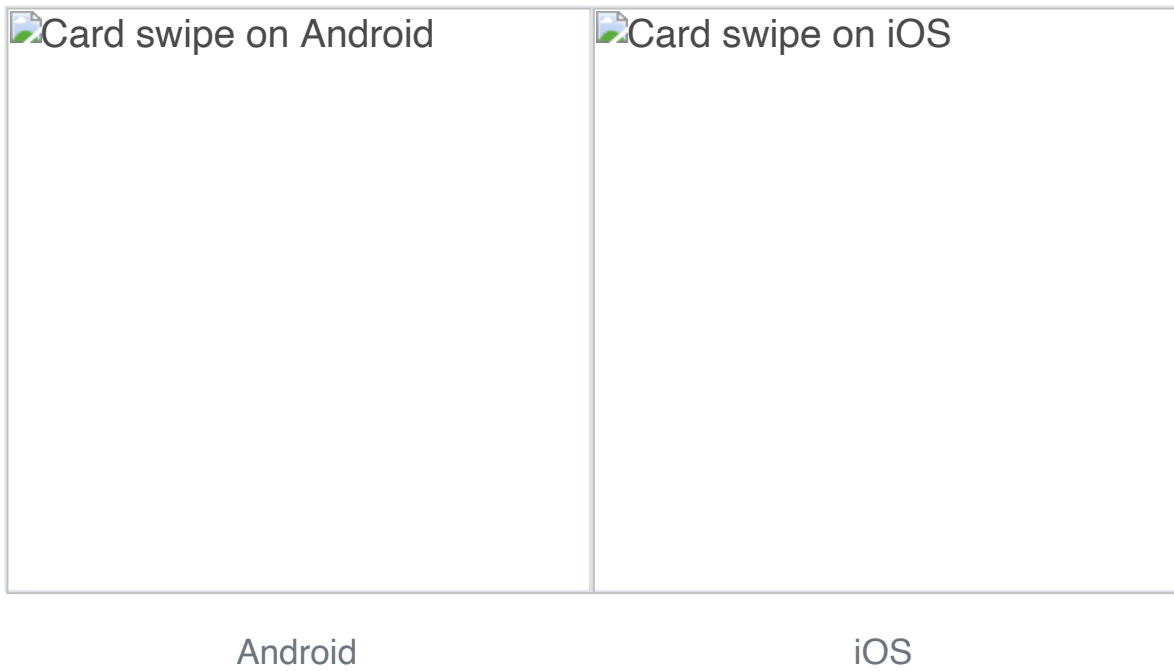
How do I add swipe animation to cards?

In React Native, either the [PanResponder](#) or third-party libraries are used for swipe animation.

In Flutter, to add a swipe animation, use the [Dismissible](#) widget and nest the child widgets.

```
child: Dismissible(  
  key: key,  
  onDismissed: (DismissDirection dir) {  
    cards.removeLast();  
  },  
  child: Container(  
    ...  
  ),  
),
```

content_copy



React Native and Flutter widget equivalent components

The following table lists commonly-used React Native components mapped to the corresponding Flutter widget and common widget properties.

React Native Component	Flutter Widget	Description
Button	RaisedButton	A basic raised button.
	onPressed [required]	The callback when the button is otherwise activated.
	Child	The button's label.
Button	FlatButton	A basic flat button.

React Native Component	Flutter Widget	Description
	onPressed [required]	The callback when the button is otherwise activated.
	Child	The button's label.
ScrollView	ListView	A scrollable list of widgets
	children	(<Widget> []) List of children
	controller	[ScrollController] An object used to control a scrollable widget
	itemExtent	[double] If non-null, force items to have the given extent in the given direction
	scroll Direction	[Axis] The axis along which the list scrolls.
FlatList	ListView.builder	The constructor for a linear list view that are created on demand
	itemBuilder [required]	[IndexedWidgetBuilder] A function that builds children on demand. This function is only called with indices greater than or equal to the first parameter and less than the itemCount
	itemCount	[int] improves the ability to quickly estimate the maximum scroll height

React Native Component	Flutter Widget	Description
Image	Image	A widget that displays an
	image [required]	The image to display.
	Image. asset	Several constructors are p various ways that an imag
	width, height, color, alignment	The style and layout for th
	fit	Inscribing the image into t during layout.
Modal	ModalRoute	A route that blocks interac routes.
	animation	The animation that drives and the previous route's fo
ActivityIndicator	CircularProgressIndicator	A widget that shows progr
	strokeWidth	The width of the line used
	backgroundColor	The progress indicator's b The current theme's <code>ThemeData.backg</code> default.

React Native Component	Flutter Widget	Description
ActivityIndicator	LinearProgressIndicator	A widget that shows progress
	value	The value of this progress
RefreshControl	RefreshIndicator	A widget that supports the “refresh” idiom.
	color	The progress indicator’s foreground color
	onRefresh	A function that’s called when the refresh indicator is pulled down far enough that they want the app to refresh
View	Container	A widget that surrounds a child widget
View	Column	A widget that displays its children in a vertical array.
View	Row	A widget that displays its children in a horizontal array.
View	Center	A widget that centers its child widget

React Native Component	Flutter Widget	Description
View	Padding	A widget that insets its child padding.
	padding [required]	[EdgeInsets] The amount the child.
TouchableOpacity	GestureDetector	A widget that detects gestures
	onTap	A callback when a tap occurs
	onDoubleTap	A callback when a tap occurs at the same location twice in quick succession
TextInput	TextInput	The interface to the system text input control.
	controller	[TextEditingController] and modify text.
Text	Text	The Text widget that displays text with a single style.
	data	[String] The text to display
	textDirection	[TextAlign] The direction the text flows.

React Native Component	Flutter Widget	Description
Switch	Switch	A material design switch.
	value [required]	[boolean] Whether this switch is on or off.
	onChanged [required]	[callback] Called when the switch is turned on or off.
Slider	Slider	Used to select from a range of values.
	value [required]	[double] The current value of the slider.
	onChanged [required]	Called when the user selects a new value for the slider.