Flutter for Android developers

Contents

This document is meant for Android developers looking to apply their existing Android knowledge to build mobile apps with Flutter. If you understand the fundamentals of the Android framework then you can use this document as a jump start to Flutter development.

> **Note:** To integrate Flutter code into your Android app, see Add Flutter to existing app.

Your Android knowledge and skill set are highly valuable when building with Flutter, because Flutter relies on the mobile operating system for numerous capabilities and configurations. Flutter is a new way to build UIs for mobile, but it

has a plugin system to communicate with Android (and iOS) for non-UI tasks. If you're an expert with Android, you don't have to relearn everything to use Flutter.

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

Views

What is the equivalent of a View in Flutter?

> How is react-style, or *declarative*, programming different than the traditional imperative style? For a comparison, see Introduction to declarative UI.

In Android, the `View` is the foundation of everything that shows up on the screen. Buttons, toolbars, and inputs, everything is a View. In Flutter, the rough equivalent to a `View` is a `Widget`. Widgets don't map exactly to Android views, but while you're getting acquainted with how Flutter works you can think of them as "the way you declare and construct UI".

However, these have a few differences to a `View`. To start, widgets have a different lifespan: they are immutable and only exist until they need to be changed. Whenever widgets or their state change, Flutter's framework creates a new tree of widget instances. In comparison, an Android view is drawn once and does not redraw until `invalidate` is called.

Flutter's widgets are lightweight, in part due to their immutability. Because they aren't views themselves, and aren't directly drawing anything, but rather are a description of the UI and its semantics that get "inflated" into actual view objects under the hood.

Flutter includes the Material Components library. These are widgets that implement the Material Design guidelines. Material Design is a flexible design system optimized for all platforms, including iOS.

But Flutter is flexible and expressive enough to implement any design language. For example, on iOS, you can use the Cupertino widgets to produce an interface that looks like Apple's iOS design language.

How do I update widgets?

In Android, you update your views by directly mutating them. However, in Flutter, `Widget`s are immutable and are not updated directly, instead you have to work with the widget's state.

This is where the concept of `Stateful` and `Stateless` widgets comes from. A `StatelessWidget` is just what it sounds like—a widget with no state information.

`StatelessWidgets` 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.

For example, in Android, this is similar to placing an `ImageView` with your logo. The logo is not going to change during runtime, so use a `StatelessWidget` in Flutter.

If you want to dynamically change the UI based on data received after making an HTTP call or user interaction then you have to work with `StatefulWidget` and tell the Flutter framework that the widget's`State` has been updated so it can update that widget.

The important thing to note here is at the core both stateless and stateful widgets behave the same. They rebuild every frame, the difference is the `StatefulWidget` has a `State` object that stores state data across frames and restores it.

If you are in doubt, then always remember this rule: if a widget changes (because of user interactions, for example) it's stateful. However, if a widget reacts to change, the containing parent widget can still be stateless if it doesn't itself react to change.

The following example shows how to use a `StatelessWidget`. A common `StatelessWidget` is the `Text`widget. If you look at the implementation of the `Text` widget you'll find that it subclasses `StatelessWidget`.

```
Text(
  'I like Flutter!',
  style: TextStyle(fontWeight: FontWeight.bold),
);
```

content_copy

As you can see, the `Text` Widget has no state information associated with it, it renders what is passed in its constructors and nothing more.

But, what if you want to make "I Like Flutter" change dynamically, for example when clicking a `FloatingActionButton`?

To achieve this, wrap the `Text` widget in a `StatefulWidget` and update it when the user clicks the button.

For example:

content_copy

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

void main() {
  runApp(SampleApp());
}

class SampleApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Sample App',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: SampleAppPage(),
    );
  }
}

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

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

class _SampleAppPageState extends State<SampleAppPage> {
  // Default placeholder text
  String textToShow = "I Like Flutter";

  void _updateText() {
    setState(() {
      // update the text
      textToShow = "Flutter is Awesome!";
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Sample App"),
      ),
      body: Center(child: Text(textToShow)),
      floatingActionButton: FloatingActionButton(
        onPressed: _updateText,
        tooltip: 'Update Text',
```

```
          child: Icon(Icons.update),
        ),
      );
    }
  }
```

How do I lay out my widgets? Where is my XML layout file?

In Android, you write layouts in XML, but in Flutter you write your layouts with a widget tree.

The following example shows how to display a simple widget with padding:

content_copy

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text("Sample App"),
    ),
    body: Center(
      child: MaterialButton(
        onPressed: () {},
        child: Text('Hello'),
        padding: EdgeInsets.only(left: 10.0, right: 10.0),
      ),
    ),
  );
}
```

You can view some of the layouts that Flutter has to offer in the widget catalog.

How do I add or remove a component from my layout?

In Android, you call `addChild()` or `removeChild()` on a parent to dynamically add or remove child views. In Flutter, because widgets are immutable there is no direct equivalent to `addChild()`. Instead, you can pass a function to the parent that returns a widget, and control that child's creation with a boolean flag.

For example, here is how you can toggle between two widgets when you click on a `FloatingActionButton`:

content_copy

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

void main() {
  runApp(SampleApp());
}

class SampleApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Sample App',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: SampleAppPage(),
    );
  }
}

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

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

class _SampleAppPageState extends State<SampleAppPage> {
  // Default value for toggle
  bool toggle = true;
  void _toggle() {
    setState(() {
      toggle = !toggle;
    });
  }

  _getToggleChild() {
    if (toggle) {
      return Text('Toggle One');
    } else {
      return MaterialButton(onPressed: () {}, child:
Text('Toggle Two'));
    }
  }

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

```
      appBar: AppBar(
        title: Text("Sample App"),
      ),
      body: Center(
        child: _getToggleChild(),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _toggle,
        tooltip: 'Update Text',
        child: Icon(Icons.update),
      ),
    );
  }
}
```

How do I animate a widget?

In Android, you either create animations using XML, or call the `animate()` method on a view. In Flutter, animate widgets using the animation library by wrapping widgets inside an animated widget.

In Flutter, use an `AnimationController` which is an `Animation<double>` that can pause, seek, stop and reverse the animation. It requires a `Ticker` that signals when vsync happens, and produces a linear interpolation between 0 and 1 on each frame while it's running. You then create one or more `Animation`s and attach them to the controller.

For example, you might use `CurvedAnimation` to implement an animation along an interpolated curve. In this sense, the controller is the "master" source of the animation progress and the `CurvedAnimation`computes the curve that replaces the controller's default linear motion. Like widgets, animations in Flutter work with composition.

When building the widget tree you assign the `Animation` to an animated property of a widget, such as the opacity of a `FadeTransition`, and tell the controller to start the animation.

The following example shows how to write a `FadeTransition` that fades the widget into a logo when you press the `FloatingActionButton`:

content_copy

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

void main() {
  runApp(FadeAppTest());
}

class FadeAppTest extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Fade Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: MyFadeTest(title: 'Fade Demo'),
    );
  }
}

class MyFadeTest extends StatefulWidget {
  MyFadeTest({Key key, this.title}) : super(key: key);
  final String title;
  @override
  _MyFadeTest createState() => _MyFadeTest();
}

class _MyFadeTest extends State<MyFadeTest> with
TickerProviderStateMixin {
  AnimationController controller;
  CurvedAnimation curve;

  @override
  void initState() {
    super.initState();
    controller = AnimationController(
      duration: const Duration(milliseconds: 2000),
      vsync: this,
    );
    curve = CurvedAnimation(parent: controller, curve:
Curves.easeIn);
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
```

```
        ),
        body: Center(
            child: Container(
                child: FadeTransition(
                    opacity: curve,
                    child: FlutterLogo(
                      size: 100.0,
                    )))),
        floatingActionButton: FloatingActionButton(
          tooltip: 'Fade',
          child: Icon(Icons.brush),
          onPressed: () {
            controller.forward();
          },
        ),
      );
    }
  }
```

For more information, see Animation & Motion widgets, the Animations tutorial, and the Animations overview.

How do I use a Canvas to draw/paint?

In Android, you would use the `Canvas` and `Drawable`s to draw images and shapes to the screen. Flutter has a similar `Canvas` API as well, since it is based on the same low-level rendering engine, Skia. As a result, painting to a canvas in Flutter is a very familiar task for Android developers.

Flutter has two classes that help you draw to the canvas: `CustomPaint` and `CustomPainter`, the latter of which implements your algorithm to draw to the canvas.

To learn how to implement a signature painter in Flutter, see Collin's answer on Custom Paint.

content_copy

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

void main() => runApp(MaterialApp(home: DemoApp()));

class DemoApp extends StatelessWidget {
  Widget build(BuildContext context) => Scaffold(body:
Signature());
}

class Signature extends StatefulWidget {
  SignatureState createState() => SignatureState();
}

class SignatureState extends State<Signature> {
  List<Offset> _points = <Offset>[];
  Widget build(BuildContext context) {
    return GestureDetector(
      onPanUpdate: (DragUpdateDetails details) {
        setState(() {
          RenderBox referenceBox = context.findRenderObject();
          Offset localPosition =

referenceBox.globalToLocal(details.globalPosition);
          _points = List.from(_points)..add(localPosition);
        });
      },
      onPanEnd: (DragEndDetails details) => _points.add(null),
      child: CustomPaint(
        painter: SignaturePainter(_points),
        size: Size.infinite,
      ),
    );
  }
}

class SignaturePainter extends CustomPainter {
  SignaturePainter(this.points);
  final List<Offset> points;
  void paint(Canvas canvas, Size size) {
    var paint = Paint()
      ..color = Colors.black
      ..strokeCap = StrokeCap.round
      ..strokeWidth = 5.0;
    for (int i = 0; i < points.length - 1; i++) {
      if (points[i] != null && points[i + 1] != null)
        canvas.drawLine(points[i], points[i + 1], paint);
    }
  }
```

```
  bool shouldRepaint(SignaturePainter other) => other.points !=
points;
}
```

How do I build custom widgets?

In Android, you typically subclass `View`, or use a pre-existing view, to override and implement methods that achieve the desired behavior.

In Flutter, build a custom widget by composing smaller widgets (instead of extending them). It is somewhat similar to implementing a custom `ViewGroup` in Android, where all the building blocks are already existing, but you provide a different behavior—for example, custom layout logic.

For example, how do you build a `CustomButton` that takes a label in the constructor? Create a CustomButton that composes a `RaisedButton` with a label, rather than by extending `RaisedButton`:

```
class CustomButton extends StatelessWidget {
  final String label;

  CustomButton(this.label);

  @override
  Widget build(BuildContext context) {
    return RaisedButton(onPressed: () {}, child: Text(label));
  }
}
```
content_copy

Then use `CustomButton`, just as you'd use any other Flutter widget:

```
@override
Widget build(BuildContext context) {
  return Center(
    child: CustomButton("Hello"),
  );
}
```
content_copy

Intents

What is the equivalent of an Intent in Flutter?

In Android, there are two main use cases for `Intent`s: navigating between Activities, and communicating with components. Flutter, on the other hand, does not have the concept of intents, although you can still start intents through native integrations (using a plugin).

Flutter doesn't really have a direct equivalent to activities and fragments; rather, in Flutter you navigate between screens, using a `Navigator` and `Route`s, all within the same `Activity`.

A `Route` is an abstraction for a "screen" or "page" of an app, and a `Navigator` is a widget that manages routes. A route roughly maps to an `Activity`, but it does not carry the same meaning. A navigator can push and pop routes to move from screen to screen. Navigators work like a stack on which you can `push()` new routes you want to navigate to, and from which you can `pop()` routes when you want to "go back".

In Android, you declare your activities inside the app's `AndroidManifest.xml`.

In Flutter, you have a couple options to navigate between pages:

- Specify a `Map` of route names. (using `MaterialApp`)
- Directly navigate to a route. (using `WidgetsApp`)

The following example builds a Map.

```
void main() {
 runApp(MaterialApp(
   home: MyAppHome(), // becomes the route named '/'
   routes: <String, WidgetBuilder> {
     '/a': (BuildContext context) => MyPage(title: 'page A'),
     '/b': (BuildContext context) => MyPage(title: 'page B'),
     '/c': (BuildContext context) => MyPage(title: 'page C'),
   },
 ));
}
```
content_copy

Navigate to a route by `push`ing its name to the `Navigator`.

```
Navigator.of(context).pushNamed('/b');
```
content_copy

The other popular use-case for `Intent`s is to call external components such as a Camera or File picker. For this, you would need to create a native platform integration (or use an existing plugin).

To learn how to build a native platform integration, see developing packages and plugins.

How do I handle incoming intents from external applications in Flutter?

Flutter can handle incoming intents from Android by directly talking to the Android layer and requesting the data that was shared.

The following example registers a text share intent filter on the native activity that runs our Flutter code, so other apps can share text with our Flutter app.

The basic flow implies that we first handle the shared text data on the Android native side (in our `Activity`), and then wait until Flutter requests for the data to provide it using a `MethodChannel`.

First, register the intent filter for all intents in `AndroidManifest.xml`:

```
<activity                                                    content_copy
  android:name=".MainActivity"
  android:launchMode="singleTop"
  android:theme="@style/LaunchTheme"

android:configChanges="orientation|keyboardHidden|keyboard|scree
nSize|locale|layoutDirection"
  android:hardwareAccelerated="true"
  android:windowSoftInputMode="adjustResize">
  <!-- ... -->
  <intent-filter>
    <action android:name="android.intent.action.SEND" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="text/plain" />
  </intent-filter>
</activity>
```

Then in `MainActivity`, handle the intent, extract the text that was shared from the intent, and hold onto it. When Flutter is ready to process, it requests the data using a platform channel, and it's sent across from the native side:

content_copy

```
package com.example.shared;

import android.content.Intent;
import android.os.Bundle;

import java.nio.ByteBuffer;

import io.flutter.app.FlutterActivity;
import io.flutter.plugin.common.ActivityLifecycleListener;
import io.flutter.plugin.common.MethodCall;
import io.flutter.plugin.common.MethodChannel;
import io.flutter.plugin.common.MethodChannel.MethodCallHandler;
import io.flutter.plugins.GeneratedPluginRegistrant;

public class MainActivity extends FlutterActivity {

  private String sharedText;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    GeneratedPluginRegistrant.registerWith(this);
    Intent intent = getIntent();
    String action = intent.getAction();
    String type = intent.getType();

    if (Intent.ACTION_SEND.equals(action) && type != null) {
      if ("text/plain".equals(type)) {
        handleSendText(intent); // Handle text being sent
      }
    }

    new MethodChannel(getFlutterView(),
"app.channel.shared.data").setMethodCallHandler(
      new MethodCallHandler() {
        @Override
        public void onMethodCall(MethodCall call,
MethodChannel.Result result) {
          if (call.method.contentEquals("getSharedText")) {
            result.success(sharedText);
            sharedText = null;
          }
        }
      });
  }

  void handleSendText(Intent intent) {
    sharedText = intent.getStringExtra(Intent.EXTRA_TEXT);
```

```
    }
  }
```

Finally, request the data from the Flutter side when the widget is rendered:

content_copy

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

void main() {
  runApp(SampleApp());
}

class SampleApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Sample Shared App Handler',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: SampleAppPage(),
    );
  }
}

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

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

class _SampleAppPageState extends State<SampleAppPage> {
  static const platform = const
MethodChannel('app.channel.shared.data');
  String dataShared = "No data";

  @override
  void initState() {
    super.initState();
    getSharedText();
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(body: Center(child: Text(dataShared)));
  }

  getSharedText() async {
    var sharedData = await
platform.invokeMethod("getSharedText");
    if (sharedData != null) {
```

```
        setState(() {
          dataShared = sharedData;
        });
      }
    }
  }
```

What is the equivalent of startActivityForResult()?

The `Navigator` class handles routing in Flutter and is used to get a result back from a route that you have pushed on the stack. This is done by `await`ing on the `Future` returned by `push()`.

For example, to start a location route that lets the user select their location, you could do the following:

```
Map coordinates = await
Navigator.of(context).pushNamed('/location');
```
content_copy

And then, inside your location route, once the user has selected their location you can `pop` the stack with the result:

```
Navigator.of(context).pop({"lat":43.821757,"long":-79.226392});
```
content_copy

Async UI

What is the equivalent of runOnUiThread() in Flutter?

Dart has a single-threaded execution model, with support for `Isolate`s (a way to run Dart code on another thread), an event loop, and asynchronous programming. Unless you spawn an `Isolate`, your Dart code runs in the main UI thread and is driven by an event loop. Flutter's event loop is equivalent to Android's main `Looper`—that is, the `Looper` that is attached to the main thread.

Dart's single-threaded model doesn't mean you need to run everything as a blocking operation that causes the UI to freeze. Unlike Android, which requires you to keep the main thread free at all times, in Flutter, use the asynchronous facilities that the Dart language provides, such as `async`/`await`, to perform asynchronous work. You might be familiar with the `async`/`await` paradigm if you've used it in C#, Javascript, or if you have used Kotlin's coroutines.

For example, you can run network code without causing the UI to hang by using `async`/`await` and letting Dart do the heavy lifting:

```
Future<void> loadData() async {                                    content_copy
  String dataURL = "https://jsonplaceholder.typicode.com/posts";
  http.Response response = await http.get(dataURL);
  setState(() {
    widgets = json.decode(response.body);
  });
}
```

Once the `await`ed network call is done, update the UI by calling `setState()`, which triggers a rebuild of the widget sub-tree and updates the data.

The following example loads data asynchronously and displays it in a `ListView`:

content_copy

```
import 'dart:convert';

import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;

void main() {
  runApp(SampleApp());
}

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

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

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

class _SampleAppPageState extends State<SampleAppPage> {
  List widgets = [];

  @override
  void initState() {
    super.initState();

    loadData();
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Sample App"),
      ),
      body: ListView.builder(
        itemCount: widgets.length,
        itemBuilder: (BuildContext context, int position) {
```

```
            return getRow(position);
          },
        ),
      );
    }

    Widget getRow(int i) {
      return Padding(
        padding: EdgeInsets.all(10.0),
        child: Text("Row ${widgets[i]["title"]}"),
      );
    }

    Future<void> loadData() async {
      String dataURL =
  "https://jsonplaceholder.typicode.com/posts";
      http.Response response = await http.get(dataURL);
      setState(() {
        widgets = json.decode(response.body);
      });
    }
  }
```

Refer to the next section for more information on doing work in the background, and how Flutter differs from Android.

How do you move work to a background thread?

In Android, when you want to access a network resource you would typically move to a background thread and do the work, as to not block the main thread, and avoid ANRs. For example, you might be using an `AsyncTask`, a `LiveData`, an `IntentService`, a `JobScheduler` job, or an RxJava pipeline with a scheduler that works on background threads.

Since Flutter is single threaded and runs an event loop (like Node.js), you don't have to worry about thread management or spawning background threads. If you're doing I/O-bound work, such as disk access or a network call, then you can safely use `async`/`await` and you're all set. If, on the other hand, you need to do computationally intensive work that keeps the CPU busy, you want to move it to an `Isolate` to avoid blocking the event loop, like you would keep *any* sort of work out of the main thread in Android.

For I/O-bound work, declare the function as an `async` function, and `await` on long-running tasks inside the function:

```
Future<void> loadData() async {                                content_copy
  String dataURL = "https://jsonplaceholder.typicode.com/posts";
  http.Response response = await http.get(dataURL);
  setState(() {
    widgets = json.decode(response.body);
  });
}
```

This is how you would typically do network or database calls, which are both I/O operations.

On Android, when you extend `AsyncTask`, you typically override 3 methods, `onPreExecute()`, `doInBackground()` and `onPostExecute()`. There is no equivalent in Flutter, since you `await` on a long running function, and Dart's event loop takes care of the rest.

However, there are times when you might be processing a large amount of data and your UI hangs. In Flutter, use `Isolate`s to take advantage of multiple CPU cores to do long-running or computationally intensive tasks.

Isolates are separate execution threads that do not share any memory with the main execution memory heap. This means you can't access variables from the main thread, or update your UI by calling `setState()`. Unlike Android threads, Isolates are true to their name, and cannot share memory (in the form of static fields, for example).

The following example shows, in a simple isolate, how to share data back to the main thread to update the UI.

content_copy

```
Future<void> loadData() async {
  ReceivePort receivePort = ReceivePort();
  await Isolate.spawn(dataLoader, receivePort.sendPort);

  // The 'echo' isolate sends its SendPort as the first message.
  SendPort sendPort = await receivePort.first;

  List msg = await sendReceive(
    sendPort,
    "https://jsonplaceholder.typicode.com/posts",
  );

  setState(() {
    widgets = msg;
  });
}

// The entry point for the isolate.
static Future<void> dataLoader(SendPort sendPort) async {
  // Open the ReceivePort for incoming messages.
  ReceivePort port = ReceivePort();

  // Notify any other isolates what port this isolate listens
to.
  sendPort.send(port.sendPort);

  await for (var msg in port) {
    String data = msg[0];
    SendPort replyTo = msg[1];

    String dataURL = data;
    http.Response response = await http.get(dataURL);
    // Lots of JSON to parse
    replyTo.send(json.decode(response.body));
  }
}

Future sendReceive(SendPort port, msg) {
  ReceivePort response = ReceivePort();
  port.send([msg, response.sendPort]);
  return response.first;
}
```

Here, `dataLoader()` is the `Isolate` that runs in its own separate execution thread. In the isolate you can perform more CPU intensive processing (parsing a big JSON, for example), or perform computationally intensive math, such as

encryption or signal processing.

You can run the full example below:

content_copy

```
import 'dart:convert';

import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;
import 'dart:async';
import 'dart:isolate';

void main() {
  runApp(SampleApp());
}

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

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

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

class _SampleAppPageState extends State<SampleAppPage> {
  List widgets = [];

  @override
  void initState() {
    super.initState();
    loadData();
  }

  showLoadingDialog() {
    if (widgets.length == 0) {
      return true;
    }

    return false;
  }
```

```
  getBody() {
    if (showLoadingDialog()) {
      return getProgressDialog();
    } else {
      return getListView();
    }
  }

  getProgressDialog() {
    return Center(child: CircularProgressIndicator());
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(
          title: Text("Sample App"),
        ),
        body: getBody());
  }

  ListView getListView() => ListView.builder(
      itemCount: widgets.length,
      itemBuilder: (BuildContext context, int position) {
        return getRow(position);
      });

  Widget getRow(int i) {
    return Padding(
      padding: EdgeInsets.all(10.0),
      child: Text("Row ${widgets[i]["title"]}"),
    );
  }

  Future<void> loadData() async {
    ReceivePort receivePort = ReceivePort();
    await Isolate.spawn(dataLoader, receivePort.sendPort);

    // The 'echo' isolate sends its SendPort as the first
message
    SendPort sendPort = await receivePort.first;

    List msg = await sendReceive(
      sendPort,
      "https://jsonplaceholder.typicode.com/posts",
    );

    setState(() {
      widgets = msg;
```

```
    });
  }

  // the entry point for the isolate
  static Future<void> dataLoader(SendPort sendPort) async {
    // Open the ReceivePort for incoming messages.
    ReceivePort port = ReceivePort();

    // Notify any other isolates what port this isolate listens
to.
    sendPort.send(port.sendPort);

    await for (var msg in port) {
      String data = msg[0];
      SendPort replyTo = msg[1];

      String dataURL = data;
      http.Response response = await http.get(dataURL);
      // Lots of JSON to parse
      replyTo.send(json.decode(response.body));
    }
  }

  Future sendReceive(SendPort port, msg) {
    ReceivePort response = ReceivePort();
    port.send([msg, response.sendPort]);
    return response.first;
  }
}
```

What is the equivalent of OkHttp on Flutter?

Making a network call in Flutter is easy when you use the popular `http` package.

While the http package doesn't have every feature found in OkHttp, it abstracts away much of the networking that you would normally implement yourself, making it a simple way to make network calls.

To use the `http` package, add it to your dependencies in `pubspec.yaml`:

```
dependencies:
  ...
  http: ^0.11.3+16
```

content_copy

To make a network call, call `await` on the `async` function `http.get()`:

```
import 'dart:convert';

import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;
[...]
  Future<void> loadData() async {
    String dataURL =
"https://jsonplaceholder.typicode.com/posts";
    http.Response response = await http.get(dataURL);
    setState(() {
      widgets = json.decode(response.body);
    });
  }
}
```

content_copy

How do I show the progress for a long-running task?

In Android you would typically show a `ProgressBar` view in your UI while executing a long running task on a background thread.

In Flutter, use a `ProgressIndicator` widget. Show the progress programmatically by controlling when it's rendered through a boolean flag. Tell Flutter to update its state before your long-running task starts, and hide it after it ends.

In the following example, the build function is separated into three different functions. If `showLoadingDialog()` is `true` (when `widgets.length == 0`), then render the `ProgressIndicator`. Otherwise, render the `ListView` with the data returned from a network call.

content_copy

```dart
import 'dart:convert';

import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;

void main() {
  runApp(SampleApp());
}

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

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

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

class _SampleAppPageState extends State<SampleAppPage> {
  List widgets = [];

  @override
  void initState() {
    super.initState();
    loadData();
  }

  showLoadingDialog() {
    return widgets.length == 0;
  }

  getBody() {
    if (showLoadingDialog()) {
      return getProgressDialog();
    } else {
      return getListView();
    }
```

```
  }

  getProgressDialog() {
    return Center(child: CircularProgressIndicator());
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(
          title: Text("Sample App"),
        ),
        body: getBody());
  }

  ListView getListView() => ListView.builder(
      itemCount: widgets.length,
      itemBuilder: (BuildContext context, int position) {
        return getRow(position);
      });

  Widget getRow(int i) {
    return Padding(
      padding: EdgeInsets.all(10.0),
      child: Text("Row ${widgets[i]["title"]}"),
    );
  }

  Future<void> loadData() async {
    String dataURL =
"https://jsonplaceholder.typicode.com/posts";
    http.Response response = await http.get(dataURL);
    setState(() {
      widgets = json.decode(response.body);
    });
  }
}
```

Project structure & resources

Where do I store my resolution-dependent image files?

While Android treats resources and assets as distinct items, Flutter apps have only assets. All resources that would live in the `res/drawable-*` folders on Android, are placed in an assets folder for Flutter.

Flutter follows a simple density-based format like iOS. Assets might be `1.0x`, `2.0x`, `3.0x`, or any other multiplier. Flutter doesn't have `dp`s but there are logical pixels, which are basically the same as device-independent pixels. The so-called devicePixelRatio expresses the ratio of physical pixels in a single logical pixel.

The equivalent to Android's density buckets are:

Android density qualifier	Flutter pixel ratio
`ldpi`	`0.75x`
`mdpi`	`1.0x`
`hdpi`	`1.5x`
`xhdpi`	`2.0x`
`xxhdpi`	`3.0x`
`xxxhdpi`	`4.0x`

Assets are located in any arbitrary folder—Flutter has no predefined folder structure. You declare the assets (with location) in the `pubspec.yaml` file, and Flutter picks them up.

Note that before Flutter 1.0 beta 2, assets defined in Flutter were not accessible from the native side, and vice versa, native assets and resources weren't available to Flutter, as they lived in separate folders.

As of Flutter beta 2, assets are stored in the native asset folder, and are accessed on the native side using Android's `AssetManager`:

```
val flutterAssetStream =                                    content_copy
assetManager.open("flutter_assets/assets/my_flutter_asset.png")
```

As of Flutter beta 2, Flutter still cannot access native resources, nor it can access native assets.

To add a new image asset called `my_icon.png` to our Flutter project, for example, and deciding that it should live in a folder we arbitrarily called `images`, you would put the base image (1.0x) in the `images`folder, and all the other variants in sub-folders called with the appropriate ratio multiplier:

```
images/my_icon.png       // Base: 1.0x image
images/2.0x/my_icon.png  // 2.0x image
images/3.0x/my_icon.png  // 3.0x image
```

content_copy

Next, you'll need to declare these images in your `pubspec.yaml` file:

```
assets:
 - images/my_icon.jpeg
```

content_copy

You can then access your images using `AssetImage`:

```
return AssetImage("images/my_icon.jpeg");
```

content_copy

or directly in an `Image` widget:

```
@override
Widget build(BuildContext context) {
  return Image.asset("images/my_image.png");
}
```

content_copy

Where do I store strings? How do I handle localization?

Flutter currently doesn't have a dedicated resources-like system for strings. At the moment, the best practice is to hold your copy text in a class as static fields and accessing them from there. For example:

```
class Strings {
  static String welcomeMessage = "Welcome To Flutter";
}
```

content_copy

Then in your code, you can access your strings as such:

```
Text(Strings.welcomeMessage)
```
content_copy

Flutter has basic support for accessibility on Android, though this feature is a work in progress.

Flutter developers are encouraged to use the intl package for internationalization and localization.

What is the equivalent of a Gradle file? How do I add dependencies?

In Android, you add dependencies by adding to your Gradle build script. Flutter uses Dart's own build system, and the Pub package manager. The tools delegate the building of the native Android and iOS wrapper apps to the respective build systems.

While there are Gradle files under the `android` folder in your Flutter project, only use these if you are adding native dependencies needed for per-platform integration. In general, use `pubspec.yaml` to declare external dependencies to use in Flutter. A good place to find Flutter packages is pub.dev.

Activities and fragments

What are the equivalent of activities and fragments in Flutter?

In Android, an `Activity` represents a single focused thing the user can do. A `Fragment` represents a behavior or a portion of user interface. Fragments are a way to modularize your code, compose sophisticated user interfaces for larger screens, and help scale your application UI. In Flutter, both of these concepts fall under the umbrella of `Widget`s.

To learn more about the UI for building Activities and Fragements, see the community-contributed Medium article, Flutter for Android Developers: How to design Activity UI in Flutter.

As mentioned in the Intents section, screens in Flutter are represented by `Widget`s since everything is a widget in Flutter. Use a `Navigator` to move between different `Route`s that represent different screens or pages, or perhaps different states or renderings of the same data.

How do I listen to Android activity lifecycle events?

In Android, you can override methods from the `Activity` to capture lifecycle methods for the activity itself, or register `ActivityLifecycleCallbacks` on the `Application`. In Flutter, you have neither concept, but you can instead listen to lifecycle events by hooking into the `WidgetsBinding` observer and listening to the `didChangeAppLifecycleState()` change event.

The observable lifecycle events are:

- `inactive` — The application is in an inactive state and is not receiving user input. This event only works on iOS, as there is no equivalent event to map to on Android.
- `paused` — The application is not currently visible to the user, not responding to user input, and running in the background. This is equivalent to `onPause()` in Android.
- `resumed` — The application is visible and responding to user input. This is equivalent to `onPostResume()` in Android.
- `suspending` — The application is suspended momentarily. This is equivalent to `onStop` in Android; it is not triggered on iOS as there is no equivalent event to map to on iOS.

For more details on the meaning of these states, see the `AppLifecycleStatus` documentation.

As you might have noticed, only a small minority of the Activity lifecycle events are available; while `FlutterActivity` does capture almost all the activity lifecycle events internally and send them over to the Flutter engine, they're mostly shielded away from you. Flutter takes care of starting and stopping the engine for you, and

there is little reason for needing to observe the activity lifecycle on the Flutter side in most cases. If you need to observe the lifecycle to acquire or release any native resources, you should likely be doing it from the native side, at any rate.

Here’s an example of how to observe the lifecycle status of the containing activity:

content_copy

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

class LifecycleWatcher extends StatefulWidget {
  @override
  _LifecycleWatcherState createState() =>
_LifecycleWatcherState();
}

class _LifecycleWatcherState extends State<LifecycleWatcher>
with WidgetsBindingObserver {
  AppLifecycleState _lastLifecycleState;

  @override
  void initState() {
    super.initState();
    WidgetsBinding.instance.addObserver(this);
  }

  @override
  void dispose() {
    WidgetsBinding.instance.removeObserver(this);
    super.dispose();
  }

  @override
  void didChangeAppLifecycleState(AppLifecycleState state) {
    setState(() {
      _lastLifecycleState = state;
    });
  }

  @override
  Widget build(BuildContext context) {
    if (_lastLifecycleState == null)
      return Text('This widget has not observed any lifecycle
changes.', textDirection: TextDirection.ltr);

    return Text('The most recent lifecycle state this widget
observed was: $_lastLifecycleState.',
        textDirection: TextDirection.ltr);
  }
}

void main() {
  runApp(Center(child: LifecycleWatcher()));
}
```

Layouts

What is the equivalent of a LinearLayout?

In Android, a LinearLayout is used to lay your widgets out linearly—either horizontally or vertically. In Flutter, use the Row or Column widgets to achieve the same result.

If you notice the two code samples are identical with the exception of the “Row” and “Column” widget. The children are the same and this feature can be exploited to develop rich layouts that can change overtime with the same children.

content_copy

```
  @override
  Widget build(BuildContext context) {
    return Row(
      mainAxisAlignment: MainAxisAlignment.center,
      children: <Widget>[
        Text('Row One'),
        Text('Row Two'),
        Text('Row Three'),
        Text('Row Four'),
      ],
    );
  }
```

content_copy

```
@override
Widget build(BuildContext context) {
  return Column(
    mainAxisAlignment: MainAxisAlignment.center,
    children: <Widget>[
      Text('Column One'),
      Text('Column Two'),
      Text('Column Three'),
      Text('Column Four'),
    ],
  );
}
```

To learn more about building linear layouts, see the community-contributed Medium article Flutter for Android Developers: How to design LinearLayout in Flutter.

What is the equivalent of a RelativeLayout?

A RelativeLayout lays your widgets out relative to each other. In Flutter, there are a few ways to achieve the same result.

You can achieve the result of a RelativeLayout by using a combination of Column, Row, and Stack widgets. You can specify rules for the widgets constructors on how the children are laid out relative to the parent.

For a good example of building a RelativeLayout in Flutter, see Collin's answer on StackOverflow.

What is the equivalent of a ScrollView?

In Android, use a ScrollView to lay out your widgets—if the user's device has a smaller screen than your content, it scrolls.

In Flutter, the easiest way to do this is using the ListView widget. This might seem like overkill coming from Android, but in Flutter a ListView widget is both a ScrollView and an Android ListView.

content_copy

```
@override
Widget build(BuildContext context) {
  return ListView(
    children: <Widget>[
      Text('Row One'),
      Text('Row Two'),
      Text('Row Three'),
      Text('Row Four'),
    ],
  );
}
```

How do I handle landscape transitions in Flutter?

FlutterView handles the config change if AndroidManifest.xml contains:

```
android:configChanges="orientation|screenSize"
```
content_copy

Gesture detection and touch event handling

How do I add an onClick listener to a widget in Flutter?

In Android, you can attach onClick to views such as button by calling the method ‘setOnClickListener’.

In Flutter there are two ways of adding touch listeners:

1. If the Widget supports event detection, pass a function to it and handle it in the function. For example, the RaisedButton has an `onPressed` parameter:

```
@override
Widget build(BuildContext context) {
  return RaisedButton(
      onPressed: () {
        print("click");
      },
      child: Text("Button"));
}
```
content_copy

2. If the Widget doesn't support event detection, wrap the widget in a GestureDetector and pass a function to the `onTap` parameter.

```
class SampleApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
        body: Center(
      child: GestureDetector(
        child: FlutterLogo(
          size: 200.0,
        ),
        onTap: () {
          print("tap");
        },
      ),
    ));
  }
}
```

content_copy

How do I handle other gestures on widgets?

Using the GestureDetector, you can listen to a wide range of Gestures such as:

- Tap
  - `onTapDown` - A pointer that might cause a tap has contacted the screen at a particular location.
  - `onTapUp` - A pointer that triggers a tap has stopped contacting the screen at a particular location.
  - `onTap` - A tap has occurred.
  - `onTapCancel` - The pointer that previously triggered the `onTapDown` won't cause a tap.
- Double tap
  - `onDoubleTap` - The user tapped the screen at the same location twice in quick succession.
- Long press

    - `onLongPress` - A pointer has remained in contact with the screen at the same location for a long period of time.
- Vertical drag
    - `onVerticalDragStart` - A pointer has contacted the screen and might begin to move vertically.
    - `onVerticalDragUpdate` - A pointer in contact with the screen has moved further in the vertical direction.
    - `onVerticalDragEnd` - A pointer that was previously in contact with the screen and moving vertically is no longer in contact with the screen and was moving at a specific velocity when it stopped contacting the screen.
- Horizontal drag
    - `onHorizontalDragStart` - A pointer has contacted the screen and might begin to move horizontally.
    - `onHorizontalDragUpdate` - A pointer in contact with the screen has moved further in the horizontal direction.
    - `onHorizontalDragEnd` - A pointer that was previously in contact with the screen and moving horizontally is no longer in contact with the screen and was moving at a specific velocity when it stopped contacting the screen.

The following example shows a `GestureDetector` that rotates the Flutter logo on a double tap:

content_copy

```
AnimationController controller;
CurvedAnimation curve;

@override
void initState() {
  controller = AnimationController(duration: const
Duration(milliseconds: 2000), vsync: this);
  curve = CurvedAnimation(parent: controller, curve:
Curves.easeIn);
}

class SampleApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
        body: Center(
          child: GestureDetector(
            child: RotationTransition(
                turns: curve,
                child: FlutterLogo(
                  size: 200.0,
                )),
            onDoubleTap: () {
              if (controller.isCompleted) {
                controller.reverse();
              } else {
                controller.forward();
              }
            },
        ),
    ));
  }
}
```

Listviews & adapters

What is the alternative to a ListView in Flutter?

The equivalent to a ListView in Flutter is … a ListView!

In an Android ListView, you create an adapter and pass it into the ListView, which renders each row with what your adapter returns. However, you have to make sure you recycle your rows, otherwise, you get all sorts of crazy visual glitches and memory issues.

Due to Flutter's immutable widget pattern, you pass a list of widgets to your ListView, and Flutter takes care of making sure that scrolling is fast and smooth.

content_copy

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

void main() {
  runApp(SampleApp());
}

class SampleApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Sample App',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: SampleAppPage(),
    );
  }
}

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

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

class _SampleAppPageState extends State<SampleAppPage> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Sample App"),
      ),
      body: ListView(children: _getListData()),
    );
  }

  List<Widget> _getListData() {
    List<Widget> widgets = [];
    for (int i = 0; i < 100; i++) {
      widgets.add(Padding(
        padding: EdgeInsets.all(10.0),
        child: Text("Row $i"),
      ));
    }
    return widgets;
```

```
    }
  }
```

How do I know which list item is clicked on?

In Android, the ListView has a method to find out which item was clicked, ‘onItemClickListener’. In Flutter, use the touch handling provided by the passed-in widgets.

content_copy

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

void main() {
  runApp(SampleApp());
}

class SampleApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Sample App',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: SampleAppPage(),
    );
  }
}

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

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

class _SampleAppPageState extends State<SampleAppPage> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Sample App"),
      ),
      body: ListView(children: _getListData()),
    );
  }

  List<Widget> _getListData() {
    List<Widget> widgets = [];
    for (int i = 0; i < 100; i++) {
      widgets.add(GestureDetector(
        child: Padding(
          padding: EdgeInsets.all(10.0),
          child: Text("Row $i"),
        ),
        onTap: () {
          print('row tapped');
```

```
          },
        ));
      }
      return widgets;
    }
  }
```

How do I update ListView's dynamically?

On Android, you update the adapter and call `notifyDataSetChanged`.

In Flutter, if you were to update the list of widgets inside a `setState()`, you would quickly see that your data did not change visually. This is because when `setState()` is called, the Flutter rendering engine looks at the widget tree to see if anything has changed. When it gets to your `ListView`, it performs a `==` check, and determines that the two `ListView`s are the same. Nothing has changed, so no update is required.

For a simple way to update your `ListView`, create a new `List` inside of `setState()`, and copy the data from the old list to the new list. While this approach is simple, it is not recommended for large data sets, as shown in the next example.

content_copy

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

void main() {
  runApp(SampleApp());
}

class SampleApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Sample App',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: SampleAppPage(),
    );
  }
}

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

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

class _SampleAppPageState extends State<SampleAppPage> {
  List widgets = <Widget>[];

  @override
  void initState() {
    super.initState();
    for (int i = 0; i < 100; i++) {
      widgets.add(getRow(i));
    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Sample App"),
      ),
      body: ListView(children: widgets),
    );
  }
```

```
  Widget getRow(int i) {
    return GestureDetector(
      child: Padding(
        padding: EdgeInsets.all(10.0),
        child: Text("Row $i"),
      ),
      onTap: () {
        setState(() {
          widgets = List.from(widgets);
          widgets.add(getRow(widgets.length + 1));
          print('row $i');
        });
      },
    );
  }
}
```

The recommended, efficient, and effective way to build a list uses a `ListView.Builder`. This method is great when you have a dynamic `List` or a `List` with very large amounts of data. This is essentially the equivalent of RecyclerView on Android, which automatically recycles list elements for you:

content_copy

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

void main() {
  runApp(SampleApp());
}

class SampleApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Sample App',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: SampleAppPage(),
    );
  }
}

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

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

class _SampleAppPageState extends State<SampleAppPage> {
  List widgets = <Widget>[];

  @override
  void initState() {
    super.initState();
    for (int i = 0; i < 100; i++) {
      widgets.add(getRow(i));
    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(
          title: Text("Sample App"),
        ),
        body: ListView.builder(
            itemCount: widgets.length,
            itemBuilder: (BuildContext context, int position) {
              return getRow(position);
```

```
            }));
  }

  Widget getRow(int i) {
    return GestureDetector(
      child: Padding(
        padding: EdgeInsets.all(10.0),
        child: Text("Row $i"),
      ),
      onTap: () {
        setState(() {
          widgets.add(getRow(widgets.length + 1));
          print('row $i');
        });
      },
    );
  }
}
```

Instead of creating a “ListView”, create a `ListView.builder` that takes two key parameters: the initial length of the list, and an `ItemBuilder` function.

The `ItemBuilder` function is similar to the `getView` function in an Android adapter; it takes a position, and returns the row you want rendered at that position.

Finally, but most importantly, notice that the `onTap()` function doesn’t recreate the list anymore, but instead `.add`s to it.

Working with text

How do I set custom fonts on my Text widgets?

In Android SDK (as of Android O), you create a Font resource file and pass it into the FontFamily param for your TextView.

In Flutter, place the font file in a folder and reference it in the `pubspec.yaml` file, similar to how you import images.

```
fonts:
   - family: MyCustomFont
     fonts:
       - asset: fonts/MyCustomFont.ttf
       - style: italic
```

content_copy

Then assign the font to your `Text` widget:

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text("Sample App"),
    ),
    body: Center(
      child: Text(
        'This is a custom font text',
        style: TextStyle(fontFamily: 'MyCustomFont'),
      ),
    ),
  );
}
```

content_copy

How do I style my Text widgets?

Along with fonts, you can customize other styling elements on a `Text` widget. The style parameter of a `Text` widget takes a `TextStyle` object, where you can customize many parameters, such as:

- color
- decoration
- decorationColor
- decorationStyle
- fontFamily
- fontSize
- fontStyle
- fontWeight

- hashCode
- height
- inherit
- letterSpacing
- textBaseline
- wordSpacing

Form input

For more information on using Forms, see Retrieve the value of a text field, from the Flutter cookbook.

What is the equivalent of a “hint” on an Input?

In Flutter, you can easily show a “hint” or a placeholder text for your input by adding an InputDecoration object to the decoration constructor parameter for the Text Widget.

```
body: Center(                                                    content_copy
  child: TextField(
    decoration: InputDecoration(hintText: "This is a hint"),
  )
)
```

How do I show validation errors?

Just as you would with a “hint”, pass an InputDecoration object to the decoration constructor for the Text widget.

However, you don’t want to start off by showing an error. Instead, when the user has entered invalid data, update the state, and pass a new `InputDecoration` object.

content_copy

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

void main() {
  runApp(SampleApp());
}

class SampleApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Sample App',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: SampleAppPage(),
    );
  }
}

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

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

class _SampleAppPageState extends State<SampleAppPage> {
  String _errorText;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Sample App"),
      ),
      body: Center(
        child: TextField(
          onSubmitted: (String text) {
            setState(() {
              if (!isEmail(text)) {
                _errorText = 'Error: This is not an email';
              } else {
                _errorText = null;
              }
            });
          },
          decoration: InputDecoration(
```

```
            hintText: "This is a hint",
            errorText: _getErrorText(),
          ),
        ),
      ),
    );
  }

  _getErrorText() {
    return _errorText;
  }

  bool isEmail(String em) {
    String emailRegexp =
        r'^(([^<>()[\]\\.,;:\s@\"]+(\.[^<>()[\]\\.,;:\s@\"]+)*)|
(\".+\"))@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\])|
(([a-zA-Z\-0-9]+\.)+[a-zA-Z]{2,}))$';

    RegExp regExp = RegExp(emailRegexp);

    return regExp.hasMatch(em);
  }
}
```

Flutter plugins

How do I access the GPS sensor?

Use the geolocator community plugin.

How do I access the camera?

The image_picker plugin is popular for accessing the camera.

How do I log in with Facebook?

To Log in with Facebook, use the `flutter_facebook_login` community plugin.

How do I use Firebase features?

Most Firebase functions are covered by first party plugins. These plugins are first-party integrations, maintained by the Flutter team:

- `firebase_admob` for Firebase AdMob
- `firebase_analytics` for Firebase Analytics
- `firebase_auth` for Firebase Auth
- `firebase_database` for Firebase RTDB
- `firebase_storage` for Firebase Cloud Storage
- `firebase_messaging` for Firebase Messaging (FCM)
- `flutter_firebase_ui` for quick Firebase Auth integrations (Facebook, Google, Twitter and email)
- `cloud_firestore` for Firebase Cloud Firestore

You can also find some third-party Firebase plugins on pub.dev that cover areas not directly covered by the first-party plugins.

How do I build my own custom native integrations?

If there is platform-specific functionality that Flutter or its community Plugins are missing, you can build your own following the developing packages and plugins page.

Flutter's plugin architecture, in a nutshell, is much like using an Event bus in Android: you fire off a message and let the receiver process and emit a result back to you. In this case, the receiver is code running on the native side on Android or iOS.

How do I use the NDK in my Flutter application?

If you use the NDK in your current Android application and want your Flutter application to take advantage of your native libraries then it's possible by building a custom plugin.

Your custom plugin first talks to your Android app, where you call your `native` functions over JNI. Once a response is ready, send a message back to Flutter and render the result.

Calling native code directly from Flutter is currently not supported.

Themes

How do I theme my app?

Out of the box, Flutter comes with a beautiful implementation of Material Design, which takes care of a lot of styling and theming needs that you would typically do. Unlike Android where you declare themes in XML and then assign it to your application using AndroidManifest.xml, in Flutter you declare themes in the top level widget.

To take full advantage of Material Components in your app, you can declare a top level widget `MaterialApp` as the entry point to your application. MaterialApp is a convenience widget that wraps a number of widgets that are commonly required for applications implementing Material Design. It builds upon a WidgetsApp by adding Material specific functionality.

You can also use a `WidgetsApp` as your app widget, which provides some of the same functionality, but is not as rich as `MaterialApp`.

To customize the colors and styles of any child components, pass a `ThemeData` object to the `MaterialApp`widget. For example, in the code below, the primary swatch is set to blue and text selection color is red.

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

content_copy

Databases and local storage

How do I access Shared Preferences?

In Android, you can store a small collection of key-value pairs using the SharedPreferences API.

In Flutter, access this functionality using the Shared_Preferences plugin. This plugin wraps the functionality of both Shared Preferences and NSUserDefaults (the iOS equivalent).

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

void main() {
  runApp(
    MaterialApp(
      home: Scaffold(
        body: Center(
          child: RaisedButton(
            onPressed: _incrementCounter,
            child: Text('Increment Counter'),
          ),
        ),
      ),
    ),
  );
}

_incrementCounter() async {
  SharedPreferences prefs = await
SharedPreferences.getInstance();
  int counter = (prefs.getInt('counter') ?? 0) + 1;
  print('Pressed $counter times.');
  prefs.setInt('counter', counter);
}
```

content_copy

How do I access SQLite in Flutter?

In Android, you use SQLite to store structured data that you can query using SQL.

In Flutter, access this functionality using the SQFlite plugin.

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.

Notifications

How do I set up push notifications?

In Android, you use Firebase Cloud Messaging to setup push notifications for your app.

In Flutter, access this functionality using the Firebase Messaging plugin. For more information on using the Firebase Cloud Messaging API, see the `firebase_messaging` plugin documentation.