Flutter for iOS developers

Contents

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

Note: To integrate Flutter code into your iOS app, see Add Flutter to existing app.

Before diving into this doc, you might want to watch a 15-minute video from the Flutter Youtube channelabout the Cupertino package.

Your iOS 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 iOS (and Android) for non-UI tasks. If you're an expert in iOS development, you don't have to relearn everything to use Flutter.

Flutter also already makes a number of adaptations in the framework for you when running on iOS. For a list, see Platform adaptations.

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 UIView in Flutter?

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

On iOS, most of what you create in the UI is done using view objects, which are instances of the `UIView`class. These can act as containers for other `UIView` classes, which form your layout.

In Flutter, the rough equivalent to a `UIView` is a `Widget`. Widgets don't map exactly to iOS 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 `UIView`. 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 iOS view is not recreated when it changes, but rather it's a mutable entity that is drawn once and doesn't redraw until it is invalidated using `setNeedsDisplay()`.

Furthermore, unlike `UIView`, 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. 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?

To update your views on iOS, you directly mutate them. In Flutter, widgets are immutable and not updated directly. Instead, you have to manipulate the widget's state.

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

`StatelessWidgets` are useful when the part of the user interface you are describing does not depend on anything other than the initial configuration information in the widget.

For example, in iOS, this is similar to placing a `UIImageView` with your logo as the `image`. If the logo is not changing during runtime, use a `StatelessWidget` in Flutter.

If you want to dynamically change the UI based on data received after making an HTTP call, use a `StatefulWidget`. After the HTTP call has completed, tell the Flutter framework that the widget's `State` is updated, so it can update the UI.

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.

If you are in doubt, remember this rule: if a widget changes outside of the `build` method (because of runtime user interactions, for example), it's stateful. If the widget never changes, once built, it's stateless. However, even if a widget is stateful, the containing parent widget can still be stateless if it isn't itself reacting to those changes (or other inputs).

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 it subclasses `StatelessWidget`.

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

content_copy

If you look at the code above, you might notice that the `Text` widget carries no explicit state 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:

```
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 Storyboard?

In iOS, you might use a Storyboard file to organize your views and set constraints, or you might set your constraints programmatically in your view controllers. In Flutter, declare your layout in code by composing 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: CupertinoButton(
        onPressed: () {
          setState(() { _pressedCount += 1; });
        },
        child: Text('Hello'),
        padding: EdgeInsets.only(left: 10.0, right: 10.0),
      ),
    ),
  );
}
```

You can add padding to any widget, which mimics the functionality of constraints in iOS.

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

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

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

The following example shows how to toggle between two widgets when the user clicks the `FloatingActionButton`:

content_copy

```dart
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 CupertinoButton(
        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 iOS, you create an animation by calling the `animate(withDuration:animations:)` method on a view. In Flutter, use the animation library to wrap 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

```
class SampleApp 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() {
    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();
        },
      ),
    );
  }

  @override
  dispose() {
    controller.dispose();
    super.dispose();
  }
}
```

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

How do I draw to the screen?

On iOS, you use `CoreGraphics` to draw lines and shapes to the screen. Flutter has a different API based on the `Canvas` class, with two other classes that help you draw: `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 StackOverflow.

content_copy

```
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;
}

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),
    );
  }
}
```

Where is the widget's opacity?

On iOS, everything has .opacity or .alpha. In Flutter, most of the time you need to wrap a widget in an Opacity widget to accomplish this.

How do I build custom widgets?

In iOS, you typically subclass `UIView`, 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).

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`:

content_copy

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

  CustomButton(this.label);

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

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

content_copy

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

Navigation

How do I navigate between pages?

In iOS, to travel between view controllers, you can use a `UINavigationController` that manages the stack of view controllers to display.

Flutter has a similar implementation, using a `Navigator` and `Routes`. 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 a`UIViewController`. The navigator works in a similar way to the iOS `UINavigationController`, in that it can `push()` and `pop()` routes depending on whether you want to navigate to, or back from, a view.

To navigate between pages, you have a couple options:

- Specify a `Map` of route names.
- Directly navigate to a route.

The following example builds a `Map.`

```
void main() {
  runApp(CupertinoApp(
    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 `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 might 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, `pop()` the stack with the result:

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

content_copy

How do I navigate to another app?

In iOS, to send the user to another application, you use a specific URL scheme. For the system level apps, the scheme depends on the app. To implement this functionality in Flutter, create a native platform integration, or use an existing plugin, such as `url_launcher`.

How do I pop back to the iOS native viewcontroller?

Calling `SystemNavigator.pop()` from your Dart code invokes the following iOS code:

```
UIViewController* viewController = [UIApplication
sharedApplication].keyWindow.rootViewController;
  if ([viewController isKindOfClass:[UINavigationController class]]) {
    [((UINavigationController*)viewController) popViewControllerAnimated:NO];
  }
```

content_copy

If that doesn't do what you want, you can create your own platform channel to invoke arbitrary iOS code.

Threading & asynchronicity

How do I write asynchronous code?

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 the iOS main loop—that is, the `Looper` that is attached to the main thread.

Dart's single-threaded model doesn't mean you are required to run everything as a blocking operation that causes the UI to freeze. Instead, use the asynchronous facilities that the Dart language provides, such as `async`/`await`, to perform asynchronous work.

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

content_copy

```
loadData() async {
  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"]}")
    );
  }

  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 iOS.

How do you move work to a background thread?

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 done. 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.

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

content_copy

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

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

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()`. 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.

```
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 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

```dart
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"]}"),
    );
  }

  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 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;
  }
}
```

How do I make network requests?

Making a network call in Flutter is easy when you use the popular `http` package. This abstracts away a lot of the networking that you might normally implement yourself, making it simple to make network calls.

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

content_copy

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

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

content_copy

```
import 'dart:convert';

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

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

In iOS, you typically use a `UIProgressView` while executing a long-running task in the background.

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 example below, 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

```
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"]}"),
    );
  }

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

Project structure, localization, dependencies and assets

How do I include image assets for Flutter? What about multiple resolutions?

While iOS treats images and assets as distinct items, Flutter apps have only assets. Resources that are placed in the `Images.xcasset` folder on iOS, are placed in an assets folder for Flutter. As with iOS, assets are any type of file, not just images. For example, you might have a JSON file located in the `my-assets`folder:

```
my-assets/data.json
```

content_copy

Declare the asset in the `pubspec.yaml` file:

```
assets:
 - my-assets/data.json
```

content_copy

And then access it from code using an [`AssetBundle`]:

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

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

content_copy

For images, Flutter follows a simple density-based format like iOS. Image assets might be `1.0x`, `2.0x`, `3.0x`, or any other multiplier. The so-called [`devicePixelRatio`] expresses the ratio of physical pixels in a single logical pixel.

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.

For example, to add an image called `my_icon.png` to your Flutter project, you might decide to store it in a folder arbitrarily called `images`. Place the base image (1.0x) in the `images` folder, and the other variants in sub-folders named after 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, declare these images in the `pubspec.yaml` file:

```
assets:
 - images/my_icon.png
```
content_copy

You can now access your images using `AssetImage`:

```
return AssetImage("images/a_dot_burr.jpeg");
```
content_copy

or directly in an `Image` widget:

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

For more details, see Adding Assets and Images in Flutter.

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

Unlike iOS, which has the `Localizable.strings` file, Flutter doesn't currently have a dedicated system for handling strings. At the moment, the best practice is to declare your copy text in a class as static fields and access them from there. For example:

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

You can access your strings as such:

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

By default, Flutter only supports US English for its strings. If you need to add support for other languages, include the `flutter_localizations` package. You might also need to add Dart's `intl` package to use i10n machinery, such as date/time formatting.

```
dependencies:
  # ...
  flutter_localizations:
    sdk: flutter
  intl: "^0.15.6"
```

content_copy

To use the `flutter_localizations` package, specify the `localizationsDelegates` and`supportedLocales` on the app widget:

```
import 'package:flutter_localizations/flutter_localizations.dart';

MaterialApp(
 localizationsDelegates: [
   // Add app-specific localization delegate[s] here
   GlobalMaterialLocalizations.delegate,
   GlobalWidgetsLocalizations.delegate,
 ],
 supportedLocales: [
    const Locale('en', 'US'), // English
    const Locale('he', 'IL'), // Hebrew
    // ... other locales the app supports
  ],
  // ...
)
```

content_copy

The delegates contain the actual localized values, while the `supportedLocales` defines which locales the app supports. The above example uses a `MaterialApp`, so it has both a `GlobalWidgetsLocalizations` for the base widgets localized values, and a `MaterialWidgetsLocalizations` for the Material widgets localizations. If you use `WidgetsApp` for your app, you don't need the latter. Note that these two delegates contain "default" values, but you'll need to provide one or more delegates for your own app's localizable copy, if you want those to be localized too.

When initialized, the `WidgetsApp` (or `MaterialApp`) creates a `Localizations` widget for you, with the delegates you specify. The current locale for the device is always accessible from the `Localizations`widget from the current context (in the form of a `Locale` object), or using the `Window.locale`.

To access localized resources, use the `Localizations.of()` method to access a specific localizations class that is provided by a given delegate. Use the `intl_translation` package to extract translatable copy to arb files for translating, and importing them back into the app for using them with `intl`.

For further details on internationalization and localization in Flutter, see the internationalization guide, which has sample code with and without the `intl` package.

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.

What is the equivalent of CocoaPods? How do I add dependencies?

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

While there is a Podfile in the iOS folder in your Flutter project, only use this if you are adding native dependencies needed for per-platform integration. In general, use `pubspec.yaml` to declare external dependencies in Flutter. A good place to find great packages for Flutter is on [pub.dev](pub.dev).

ViewControllers

What is the equivalent to ViewController in Flutter?

In iOS, a `ViewController` represents a portion of user interface, most commonly used for a screen or section. These are composed together to build complex user interfaces, and help scale your application's UI. In Flutter, this job falls to Widgets. As mentioned in the Navigation section, screens in Flutter are represented by Widgets since "everything is a widget!" Use a `Navigator` to move between different `Route`s that represent different screens or pages, or maybe different states or renderings of the same data.

How do I listen to iOS lifecycle events?

In iOS, you can override methods to the `ViewController` to capture lifecycle methods for the view itself, or register lifecycle callbacks in the `AppDelegate`. 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 on Android.

`paused`
The application is not currently visible to the user, is not responding to user input, but is running in the background.

`resumed`
The application is visible and responding to user input.

`suspending`
The application is suspended momentarily. The iOS platform has no equivalent event.

For more details on the meaning of these states, see [`AppLifecycleStatus` documentation]().

Layouts

What is the equivalent of UITableView or UICollectionView in Flutter?

In iOS, you might show a list in either a `UITableView` or a `UICollectionView`. In Flutter, you have a similar implementation using a `ListView`. In iOS, these views have delegate methods for deciding the number of rows, the cell for each index path, and the size of the cells.

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
```
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()),
    );
  }

  _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?

In iOS, you implement the delegate method, `tableView:didSelectRowAtIndexPath:`. In Flutter, use the touch handling provided by the passed-in widgets.

content_copy

```
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()),
    );
  }

  _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 dynamically update a ListView?

In iOS, you update the data for the list view, and notify the table or collection view using the `reloadData`method.

In Flutter, if you update the list of widgets inside a `setState()`, you quickly see that your data doesn't 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 an `==` 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 = [];

  @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.

content_copy

```
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 = [];

  @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 `cellForItemAt` delegate method in an iOS table or collection view, as it takes a position, and returns the cell 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.

What is the equivalent of a ScrollView in Flutter?

In iOS, you wrap your views in a `ScrollView` that allows a user to scroll your content if needed.

In Flutter the easiest way to do this is using the `ListView` widget. This acts as both a `ScrollView` and an iOS `TableView`, as you can layout widgets in a vertical format.

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

For more detailed docs on how to lay out widgets in Flutter, see the layout tutorial.

Gesture detection and touch event handling

How do I add a click listener to a widget in Flutter?

In iOS, you attach a `GestureRecognizer` to a view to handle click events. In Flutter, there are two ways of adding touch listeners:

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

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

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");
          },
        ),
      ),
    );
  }
}
```

How do I handle other gestures on widgets?

Using `GestureDetector` you can listen to a wide range of gestures such as:

- **Tapping**

  `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 tapping**

  `onDoubleTap`
  The user tapped the screen at the same location twice in quick succession.

- **Long pressing**

  `onLongPress`
  A pointer has remained in contact with the screen at the same location for a long period of time.

- **Vertical dragging**

`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 dragging**

  `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.

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();
            }
          },
        ),
      ),
    );
  }
}
```

Theming and text

How do I theme an 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.

To take full advantage of Material Components in your app, 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.

But Flutter is flexible and expressive enough to implement any design language. On iOS, you can use theCupertino library to produce an interface that adheres to the Human Interface Guidelines. For the full set of these widgets, see the Cupertino widgets gallery.

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 {                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 set custom fonts on my Text widgets?

In iOS, you import any `ttf` font files into your project and create a reference in the `info.plist` file. In Flutter, place the font file in a folder and reference it in the `pubspec.yaml` file, similar to how you import images.

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

Then assign the font to your `Text` widget:

content_copy

```
@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'),
      ),
    ),
  );
}
```

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

How do forms work in Flutter? How do I retrieve user input?

Given how Flutter uses immutable widgets with a separate state, you might be wondering how user input fits into the picture. On iOS, you usually query the widgets for their current values when it’s time to submit the user input, or action on it. How does that work in Flutter?

In practice forms are handled, like everything in Flutter, by specialized widgets. If you have a `TextField` or a `TextFormField`, you can supply a `TextEditingController` to retrieve user input:

```
class _MyFormState extends State<MyForm> {
  // Create a text controller and use it to retrieve the current value.
  // of the TextField!
  final myController = TextEditingController();

  @override
  void dispose() {
    // Clean up the controller when disposing of the Widget.
    myController.dispose();
    super.dispose();
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Retrieve Text Input'),
      ),
      body: Padding(
        padding: const EdgeInsets.all(16.0),
        child: TextField(
          controller: myController,
        ),
      ),
      floatingActionButton: FloatingActionButton(
        // When the user presses the button, show an alert dialog with the
        // text the user has typed into our text field.
        onPressed: () {
          return showDialog(
            context: context,
            builder: (context) {
              return AlertDialog(
                // Retrieve the text the user has typed in using our
                // TextEditingController
                content: Text(myController.text),
              );
            },
          );
        },
        tooltip: 'Show me the value!',
        child: Icon(Icons.text_fields),
      ),
    );
  }
}
```

You can find more information and the full code listing in Retrieve the value of a text field, from the Flutter cookbook.

What is the equivalent of a placeholder in a text field?

In Flutter you can easily show a “hint” or a placeholder text for your field by adding an `InputDecoration`object to the decoration constructor parameter for the `Text` widget:

```
body: Center(
  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

```
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 emailString) {
    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(emailString);
  }
}
```

Interacting with hardware, third party services and the platform

How do I interact with the platform, and with platform native code?

Flutter doesn't run code directly on the underlying platform; rather, the Dart code that makes up a Flutter app is run natively on the device, "sidestepping" the SDK provided by the platform. That means, for example, when you perform a network request in Dart, it runs directly in the Dart context. You don't use the Android or iOS APIs you normally take advantage of when writing native apps. Your Flutter app is still hosted in a native app's `ViewController` as a view, but you don't have direct access to the `ViewController` itself, or the native framework.

This doesn't mean Flutter apps cannot interact with those native APIs, or with any native code you have. Flutter provides the platform channel, which communicates and exchanges data with the `ViewController`that hosts your Flutter view. Platform channels are essentially an asynchronous messaging mechanism that bridge the Dart code with the host `ViewController` and the iOS framework it runs on. You can use platform channels to execute a method on the native side, or to retrieve some data from the device's sensors, for example.

In addition to directly using platform channels, you can use a variety of pre-made plugins that encapsulate the native and Dart code for a specific goal. For example, you can use a plugin to access the camera roll and the device camera directly from Flutter, without having to write your own integration. Plugins are found on pub.dev, Dart and Flutter's open source package repository. Some packages might support native integrations on iOS, Android, web, or all of the above.

If you can't find a plugin on pub.dev that fits your needs, you can write your own and publish it on pub.dev.

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_core` for Firebase's Core package
- `firebase_database` for Firebase RTDB
- `firebase_storage` for Firebase Cloud Storage
- `firebase_messaging` for Firebase Messaging (FCM)
- `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.

Databases and local storage

How do I access UserDefault in Flutter?

In iOS, you can store a collection of key-value pairs using a property list, known as the `UserDefaults`.

In Flutter, access equivalent functionality using the Shared Preferences plugin. This plugin wraps the functionality of both `UserDefaults` and the Android equivalent, `SharedPreferences`.

What is the equivalent to CoreData in Flutter?

In iOS, you can use CoreData to store structured data. This is simply a layer on top of an SQL database, making it easier to make queries that relate to your models.

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 iOS, you need to register your app on the developer portal to allow push notifications.

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.