Flutter for Xamarin.Forms developers

Contents

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

Your Android and iOS knowledge and skill set are valuable when building with Flutter, because Flutter relies on the native operating system configurations, similar to how you would configure your native Xamarin.Forms projects. The

Flutter Frameworks is also similar to how you create a single UI, that is used on multiple platforms.

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

Project setup

How does the app start?

For each platform in Xamarin.Forms, you call the `LoadApplication` method, which creates a new application and starts your app.

```
LoadApplication(App());
```
content_copy

In Flutter, the default main entry point is `main` where you load your Flutter app.

```
void main() {
  runApp(MyApp());
}
```
content_copy

In Xamarin.Forms, you assign a `Page` to the `MainPage` property in the `Application` class.

```
public class App: Application
{
    public App()
    {
      MainPage = ContentPage()
                 {
                   Label()
                   {
                     Text="Hello World",
                     HorizontalOptions = LayoutOptions.Center,
                     VerticalOptions = LayoutOptions.Center
                   }
                 };
    }
}
```

content_copy

In Flutter, “everything is a widget”, even the application itself. The following example shows `MyApp`, a simple application `Widget`.

```
class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return Center(
        child: Text("Hello World!", textDirection:
TextDirection.ltr));
  }
}
```

content_copy

How do you create a page?

Xamarin.Forms has many different types of pages; `ContentPage` is the most common. In Flutter, you specify an application widget that holds your root page. You can use a MaterialApp widget, which supports Material Design, or you can use a CupertinoApp widget, which supports an iOS-style app, or you can use the lower level WidgetsApp, which you can customize in any way you want.

The following code defines the home page, a stateful widget. In Flutter, all widgets are immutable, but two types of widgets are supported: stateful and stateless. Examples of a stateless widget are titles, icons, or images.

The following example uses MaterialApp, which holds its root page in the `home` property.

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

content_copy

From here, your actual first page is another `Widget`, in which you create your state.

A stateful widget, such as MyHomePage below, consists of two parts. The first part, which is itself immutable, creates a State object that holds the state of the object. The State object persists over the life of the widget.

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

  final String title;

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

content_copy

The `State` object implements the `build()` method for the stateful widget.

When the state of the widget tree changes, call `setState()`, which triggers a build of that portion of the UI. Make sure to call `setState()` only when necessary, and only on the part of the widget tree that has changed, or it can result in poor UI performance.

content_copy

```dart
class _MyHomePageState extends State<MyHomePage> {
  int _counter = 0;

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        // Take the value from the MyHomePage object that was
created by
        // the App.build method, and use it to set the appbar
title.
        title: Text(widget.title),
      ),
      body: Center(
        // Center is a layout widget. It takes a single child
and positions it
        // in the middle of the parent.
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text(
              'You have pushed the button this many times:',
            ),
            Text(
              '$_counter',
              style: Theme.of(context).textTheme.headline4,
            ),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        tooltip: 'Increment',
        child: Icon(Icons.add),
      ),
    );
  }
}
```

In Flutter, the UI (also known as widget tree), is immutable, meaning you can't change its state once it's built. You change fields in your `State` class, then call `setState()` to rebuild the entire widget tree again.

This way of generating UI is different than Xamarin.Forms, but there are many benefits to this approach.

Views

What is the equivalent of a Page or Element in Flutter?

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

`ContentPage`, `TabbedPage`, `MasterDetailPage` are all types of pages you might in a Xamarin.Forms application. These pages would then hold `Element`s to display the various controls. In Xamarin.Forms an `Entry` or `Button` are examples of an `Element`.

In Flutter, almost everything is a widget. A `Page`, called a `Route` in Flutter, is a widget. Buttons, progress bars, and animation controllers are all widgets. When building a route, you create a widget tree.

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 Xamarin.Forms, each `Page` or `Element` is a stateful class, that has properties and methods. You update your `Element` by updating a property, and this is propagated down to the native control.

In Flutter, `Widget`s are immutable and you can't directly update them by changing a property, instead you have to work with the widget's state.

This is where the concept of Stateful vs 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 Xamarin.Forms, this is similar to placing an `Image` 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 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, as shown in the following 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? What is the equivalent of an XAML file?

In Xamarin.Forms, most developers write layouts in XAML, though sometimes in C#. In Flutter, you write your layouts with a widget tree in code.

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 the layouts that Flutter has to offer in the widget catalog.

How do I add or remove an Element from my layout?

In Xamarin.Forms, you had to remove or add an `Element` in code. This involved either setting the `Content`property or calling `Add()` or `Remove()` if it was a list.

In Flutter, because widgets are immutable there is no direct equivalent. 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

```
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 Xamarin.Forms, you create simple animations using ViewExtensions that include methods such as `FadeTo` and `TranslateTo`. You would use these methods on a view to perform the required animations.

```
<Image Source="{Binding MyImage}" x:Name="myImage" /> content_copy
```

Then in code behind, or a behavior, this would fade in the image, over a 1 second period.

```
myImage.FadeTo(0, 1000);                                content_copy
```

In Flutter, you animate widgets using the animation library by wrapping widgets inside an animated widget. 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() {
    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 draw/paint on the screen?

Xamarin.Forms never had a built in way to draw directly on the screen. Many would use SkiaSharp, if they needed a custom image drawn. In Flutter, you have direct access to the Skia Canvas and can easily draw on screen.

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 StackOverflow.

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;
}
```

Where is the widget's opacity?

On Xamarin.Forms, all `VisualElement`s have an Opacity. In Flutter, you need to wrap a widget in an`Opacity` widget to accomplish this.

How do I build custom widgets?

In Xamarin.Forms, you typically subclass `VisualElement`, or use a pre-existing `VisualElement`, 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 control based off a `Grid` with numerous `VisualElement`s added in, while extending with custom 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

Navigation

How do I navigate between pages?

In Xamarin.Forms, the `NavigationPage` class provides a hierarchical navigation experience where the user is able to navigate through pages, forwards and backwards.

Flutter has a similar implementation, using a `Navigator` and `Routes`. A `Route` is an abstraction for a `Page`of an app, and a `Navigator` is a widget that manages routes.

A route roughly maps to a `Page`. The navigator works in a similar way to the Xamarin.Forms `NavigationPage`, 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. (`MaterialApp`)
- Directly navigate to a route. (`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 pushing its name to the `Navigator`.

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

The `Navigator` is a stack that manages your app's routes. Pushing a route to the stack moves to that route. Popping a route from the stack, returns to the previous route. This is done by awaiting on the `Future`returned by `push()`.

`Async`/`await` is very similar to the .NET implementation and is explained in more detail in Async UI.

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 Xamarin.Forms, to send the user to another application, you use a specific URI scheme, using `Device.OpenUrl("mailto://")`

To implement this functionality in Flutter, create a native platform integration, or use an existing plugin, such as `url_launcher`, available with many other packages on pub.dev.

Async UI

What is the equivalent of Device.BeginOnMainThread() 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.

Dart's single-threaded model doesn't mean you need to run everything as a blocking operation that causes the UI to freeze. Much like Xamarin.Forms, you need to keep the UI thread free. You would use `async`/`await` to perform tasks, where you must wait for the response.

In Flutter, use the asynchronous facilities that the Dart language provides, also named `async`/`await`, to perform asynchronous work. This is very similar to C# and should be very easy to use for any Xamarin.Forms developer.

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

```
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 awaited 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 Android.

How do you move work to a background thread?

Since Flutter is single threaded and runs an event loop, you don't have to worry about thread management or spawning background threads. This is very similar to Xamarin.Forms. 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. This is similar to when you move things to a different thread via `Task.Run()` in Xamarin.Forms.

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

```
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.

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 is a difference between `Task.Run()`. This means you can't access variables from the main thread, or update your UI by calling `setState()`.

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

content_copy

```
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

```
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?

In Xamarin.Forms you would use `HttpClient`. 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`:

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

To make a network request, 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 for a long-running task?

In Xamarin.Forms you would typically create a loading indicator, either directly in XAML or through a 3rd party plugin such as AcrDialogs.

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"]}"),
    );
  }

  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 image files?

Xamarin.Forms has no platform independent way of storing images, you had to place images in the iOS `xcasset` folder or on Android, in the various `drawable` folders.

While Android and iOS treat resources and assets as distinct items, Flutter apps have only assets. All resources that would live in the `Resources/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`:

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
```

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

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

You can then access your images using `AssetImage`:

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

or directly in an `Image` widget:

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

More detailed information can be found in Adding assets and images.

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

Unlike .NET which has `resx` files, 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";
}
```

Then in your code, 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.

Where is my project file?

In Xamarin.Forms you will have a `csproj` file. The closest equivalent in Flutter is pubspec.yaml, which contains package dependencies and various project details. Similar to .NET Standard, files within the same directory are considered part of the project.

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

In the .NET eco-system, native Xamarin projects and Xamarin.Forms projects had access to Nuget and the inbuilt package management system. Flutter apps contain a native Android app, native iOS app and Flutter app.

In Android, you add dependencies by adding to your Gradle build script. In iOS, you add dependencies by adding to your `Podfile`.

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.

In general, use `pubspec.yaml` to declare external dependencies to use in Flutter. A good place to find Flutter packages is on pub.dev.

Application lifecycle

How do I listen to application lifecycle events?

In Xamarin.Forms, you have an `Application` that contains `OnStart`, `OnResume` and `OnSleep`. In Flutter you can instead listen to similar 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 is iOS only.

`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. This event is Android only.

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

Layouts

What is the equivalent of a StackLayout?

In Xamarin.Forms you can create a `StackLayout` with an `Orientation` of horizontal or vertical. Flutter has a similar approach, however you would use the `Row` or `Column` widgets.

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'),
    ],
  );
}
```

What is the equivalent of a Grid?

The closest equivalent of a `Grid` would be a `GridView`. This is much more powerful than what you are used to in Xamarin.Forms. A `GridView` provides automatic scrolling when the content exceeds its viewable space.

```
GridView.count(
  // Create a grid with 2 columns. If you change the
scrollDirection to
  // horizontal, this would produce 2 rows.
  crossAxisCount: 2,
  // Generate 100 widgets that display their index in the List
  children: List.generate(100, (index) {
    return Center(
      child: Text(
        'Item $index',
        style: Theme.of(context).textTheme.headline,
      ),
    );
  }),
);
```

content_copy

You might have used a `Grid` in Xamarin.Forms to implement widgets that overlay other widgets. In Flutter, you accomplish this with the `Stack` widget

This sample creates two icons that overlap each other.

```
child: Stack(
  children: <Widget>[
    Icon(Icons.add_box, size: 24.0, color: const
Color.fromRGBO(0,0,0,1.0)),
    Positioned(
      left: 10.0,
      child: Icon(Icons.add_circle, size: 24.0, color: const
Color.fromRGBO(0,0,0,1.0)),
    ),
  ],
),
```

content_copy

What is the equivalent of a ScrollView?

In Xamarin.Forms, a `ScrollView` wraps around a `VisualElement` and, if the content is larger than the device screen, it scrolls.

In Flutter, the closest match is the `SingleChildScrollView` widget. You simply fill the Widget with the content that you want to be scrollable.

```
@override
Widget build(BuildContext context) {
  return SingleChildScrollView(
    child: Text('Long Content'),
  );
}
```

content_copy

If you have many items you want to wrap in a scroll, even of different `Widget` types, you might want to use a `ListView`. This might seem like overkill, but in Flutter this is far more optimized and less intensive than a Xamarin.Forms `ListView`, which is backing on to platform specific controls.

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

content_copy

How do I handle landscape transitions in Flutter?

Landscape transitions can be handled automatically by setting the `configChanges` property in the AndroidManifest.xml:

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

content_copy

Gesture detection and touch event handling

How do I add GestureRecognizers to a widget in Flutter?

In Xamarin.Forms, `Element`s might contain a click event you can attach to. Many elements also contain a `Command` that is tied to this event. Alternatively you would use the `TapGestureRecognizer`. In Flutter there are two very similar ways:

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?

In Xamarin.Forms you would add a `GestureRecognizer` to the `VisualElement`. You would normally be limited to `TapGestureRecognizer`, `PinchGestureRecognizer` and `PanGestureRecognizer`, unless you built your own.

In Flutter, 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 and adapters

What is the equivalent to a ListView in Flutter?

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

In a Xamarin.Forms `ListView`, you create a `ViewCell` and possibly a `DataTemplateSelector` and pass it into the `ListView`, which renders each row with what your `DataTemplateSelector` or `ViewCell` returns. However, you often have have to make sure you turn on Cell Recycling otherwise you will run into memory issues and slow scrolling speeds.

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()),
    );
  }

  _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 has been clicked?

In Xamarin.Forms, the ListView has an `ItemTapped` method to find out which item was clicked. There are many other techniques you might have used such as checking when `SelectedItem` or `EventToCommand`behaviors change.

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()),
    );
  }

  _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 a ListView dynamically?

In Xamarin.Forms, if you bound the `ItemsSource` property to an `ObservableCollection` you would just update the list in your ViewModel. Alternatively, you could assign a new `List` to the `ItemSource` property.

In Flutter, things work a little differently. If you update the list of widgets inside a `setState()` method, 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 = [];

  @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 = [];

  @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 adds to it.

For more information, see Write your first Flutter app, part 1 and Write your first Flutter app, part 2.

Working with text

How do I set custom fonts on my text widgets?

In Xamarin.Forms, you would have to add a custom font in each native project. Then, in your `Element` you would assign this font name to the `FontFamily` attribute using `filename#fontname` and just `fontname` for iOS.

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

How do I retrieve user input?

Xamarin.Forms `element`s allow you to directly query the `element` to determine the state of any of its properties, or whether it's bound to a property in a `ViewModel`.

Retrieving information in Flutter is handled by specialized widgets and is different than how you are used to. If you have a `TextField` or a `TextFormField`, you can supply a `TextEditingController` to retrieve user input:

content_copy

```dart
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 that the user has typed into our text field.
        onPressed: () {
          return showDialog(
            context: context,
            builder: (context) {
              return AlertDialog(
                // Retrieve the text that the user has entered
using the
                // 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 on an Entry?

In Xamarin.Forms, some `Elements` support a `Placeholder` property that you can assign a value to. For example:

```
<Entry Placeholder="This is a hint">
```

content_copy

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

content_copy

How do I show validation errors?

With Xamarin.Forms, if you wished to provide a visual hint of a validation error, you would need to create new properties and `VisualElement`s surrounding the `Element`s that had validation errors.

In Flutter, you pass through 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

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` or `Activity` as a view, but you don't have direct access to this, or the native framework.

This doesn't mean Flutter apps can't interact with those native APIs, or with any native code you have. Flutter provides platform channels that communicate and exchange data with the `ViewController` or `Activity` that hosts your Flutter view. Platform channels are essentially an asynchronous messaging mechanism that bridges the Dart code with the host `ViewController` or `Activity` and the iOS or Android 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, or Android, or both.

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_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.

Themes (Styles)

How do I theme my app?

Flutter comes with a beautiful, built-in implementation of Material Design, which handles much of the styling and theming needs that you would typically do.

Xamarin.Forms does have a global `ResourceDictionary` where you can share styles across your app. Alternatively, there is Theme support currently in preview.

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 following code, the primary swatch is set to blue and text selection color is red.

content_copy

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

Databases and local storage

How do I access shared preferences or UserDefaults?

Xamarin.Forms developers will likely be familar with the `Xam.Plugins.Settings` plugin.

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

How do I access SQLite in Flutter?

In Xamarin.Forms most applications would use the `sqlite-net-pcl` plugin to access SQLite databases.

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.