Expertise
and insight
for the future

Lukas Dagne

Flutter for cross-platform App and SDK development

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

01 May 2019

Author Title Number of Pages Date	Lukas Dagne Flutter for cross-platform App and SDK development 28 pages 10 May 2019
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineering
Instructors	Jarkko Vuori, Principal Lecturer

Mobile application and SDK development involves informed decision making to select the most efficient and fitting toolkit for a project. This thesis aims to determine how Flutter UI toolkit compares to native and other cross-platform options. It investigates the framework's internals as well as its overall architecture to identify its strengths and weaknesses as a competitive cross-platform option.

In contrast to most cross-platform solutions that build on top of native UI frameworks, Flutter provides its own UI components along with innovative rendering mechanism. As a result, the framework poses a strong competition towards native development regardless of its cross-platform aspect.

Based on research and conducted experiments, the author concludes that Flutter is the perfect choice for cross-platform mobile application and SDK development. Its superior fast development cycle, smart techniques of accessing platform services and the resulting smooth user experience makes it stand out.

Some of its weaknesses such as strict and inflexible project structure are outweighed by the overall systematically layered design of the framework and its openness for external contributions. Flutter is at its early stage. Therefore, the above arguments only get stronger as the framework grows.

Keywords	App, SDK, Mobile, Flutter, UI toolkit, iOS, Android

Contents

List of Abbreviations

IBM	International Business Machines. Information technology corporation.
iOS	iPhone Operating System. iPhone and iPad operating system.
UI	User Interface. Set of visual elements displayed on screen which user can interact with.
IDE	Integrated Development Environment. A software suite that contains necessary development tools.
API	Application Program Interface. Method of communication between independent applications or software components.
HTML	Hypertext Markup Language. Protocol or set of rules for transferring variety of files across the world wide web.
CSS	Cascading Style Sheets. Description of how HTTP elements are displayed.
DOM	Document Object Model. Definition of the logical structure of documents.
fps	Frame Per Second. Frequency at which consecutive frames appear on screen.
I/O	Input Output. General term for what a system accepts and produces.
AOT	Ahead-of-Time. Pre-compilation of source code to machine code prior to execution.
JIT	Just-in-time. Compilation of source code just before execution.
VM	Virtual Machine. A software that's made to provide all functionalities of a physical computer or another software.

XML	Extensible Markup Language. A self-descriptive language used to transfer and store data.
JSX	JavaScript XML. A preprocessor step that adds XML syntax to JavaScript.
GPU	Graphics Processing Unit. A computing unit used to render contents visible on screens.
GUI	Graphical User Interface. A form of user interface using graphical objects.
NDK	Native Development Kit. Toolset that lets developers implement using native code.
GPS	Global Positioning System. Satellite based navigation system.

1 Introduction

Smartphone applications offer great value for users, developers and businesses. The early smartphones such as Simon, IBM's (International Business Machines) first smartphone introduced in the early 1990s, came with built-in applications [(1)].

The introduction of App Store, a common place to publish iOS (iPhone Operating System) applications, in 2008 was a great news for many developers [(2)]. Google followed this noble idea by creating Android Market later incorporated in Google Play opening doors to developers and businesses to reach smartphone users with relative ease [(3)].

Mobile platforms, in addition to exclusive application stores, offer development environment, tools and software components that are tied to each platform.

Developers and businesses are required to produce mobile applications tailored for each platform in order to maximize their audience. Cross-platform development solutions aim to facilitate production of mobile applications for multiple platforms from a single codebase. These solutions face different challenges as the underlying platforms significantly differ in features, design and implementations.

Android and iOS, as the biggest mobile platforms, has been the main targets for cross-platform solutions. Therefore, the term cross-platform is used to imply solutions that support the two platforms.

Google introduced Flutter as a portable UI (User Interface) toolkit designed to assist cross-platform application development [(4)]. By providing its own set of interface components, rendering mechanism and fast development cycle, the framework aims to compete with native and cross-platform solutions.

This thesis studies the functionalities of the common cross-platform solutions and Flutter; the framework's internals and overall architecture. In the process, it compares approaches to technical challenges between Flutter, other cross-platform solutions and native application development. In addition, the thesis explores Flutter's usability for SDK (Software Development Kit) development, its portability to existing iOS and Android projects and the techniques that are currently available to achieve flexible use of the framework.

2 Available solutions

2.1 Native development

Android supports Java and Kotlin programming languages with its own interface elements and layout techniques along with several IDEs (Integrated Development Environment) for app development. iOS on the other hand requires apps to be written using Objective C and/or Swift programming languages using interface elements and layout techniques provided by Apple inside the Xcode IDE.

The platforms expose exclusive high-level APIs (Application Program Interface) that are used to implement user interface, I/O (Input output) operations and other features. The result is a tailored look and feel of applications on each platform that most users has become accustomed to.

2.2 Cross-platform

The main appeal to cross-platform solutions over native development is minimizing development team size, time and resources. The other key reason is what motivates developers and businesses to make these applications in the first place; it is an extension to their existing websites. Most companies and startups extend their product or service to mobile applications due to increasing traffic they receive from mobile platforms on their websites. One example is the company Airbnb [(8)].

Cross-platform solutions are usually preferred by web developers as most of these solutions employ web technologies on mobile platforms. Single codebase, hence single development team for applications that run on multiple platforms, is one of the most common pitches for cross-platform solutions.

Due to the far-reaching differences between platforms such as iOS and Android, and the speed at which these platforms grow and evolve, creating cross-platform solution without compromises is challenging and maybe impossible. From varying programming

languages to rendering techniques, there is plenty to unify in order to achieve a cross-platform application that does not become counterproductive in the long run.

2.2.1 Browser-based solutions

Application development platforms such as Ionic and PhoneGap are utilized to create cross-platform Web Apps for any mobile platform with a browser. These applications are built using web technologies such as HTML (Hypertext Markup Language), CSS (Cascading Style Sheets) and JavaScript running on a *WebView* of the platform. The result can be summarized as a modified, app-like, websites that open on mobile browsers.

Most browser-based solutions have fast deployment cycle. For example, Ionic has a live deploy feature that allows developers directly send HTML, CSS and JavaScript updates for bugfixes and missing features without going through App Store [6]. In addition, the familiar web-like project structure is convenient to web developers (see Figure 1).

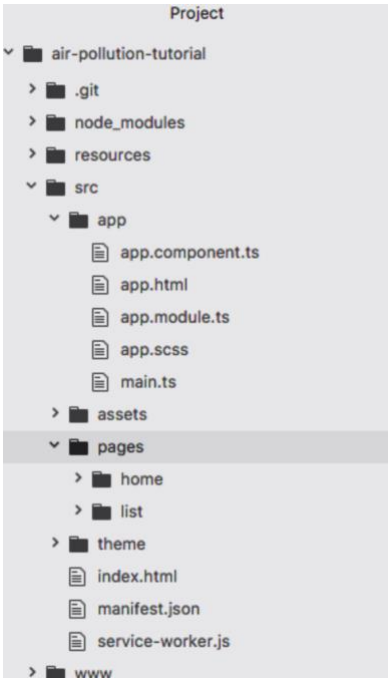

Figure 1. Project structure of Ionic application

2.2.2 Bridging, React Native solution

Facebook's React Native achieves common development environment with a bridging mechanism to produce native components for each platform. Unlike browser-based solutions, this approach creates a layer above the native APIs, hence allowing development of platform independent software.

React Native applications are built using JavaScript. Underneath the JavaScript layer lie a bridge followed by native part of the framework. Native side of React Native framework contains a *RootView* and a *bridge-interface* (written in Objective C for iOS and Java for Android). The *RootView* will be a container for all user interface objects in the app. The *bridge-interface* serves as an intermediator between the native side of the framework and the bridge which is written in C and C++.

The heart of React Native, the bridge, forms a bidirectional asynchronous communication channel between native components inside native frameworks and the JavaScript layer. This means, when creating an instance of a built-in React Native component (e.g. Text component), the bridge is responsible for creating the platform specific interface object (e.g. a subclass of *UIView* for iOS) with identical/close layout and styling information as set on the React Native component. User interactions are received in the native components and propagated to the JavaScript layer via the bridge.

Some interface objects are unique to a platform. For example, *UIAlertController* which is used to present actionable and non-actionable popup interface in iOS does not have an exactly matching counterpart in Android. Android's famous *Toast* interface object does not exist in iOS at all. React Native solves this problem by providing native specific components that developers can use targeting the particular platform.

Differences are not limited to interface objects. Developers often find themselves in need of writing platform specific code or importing platform specific libraries which need to be exposed to the JavaScript layer. React Native offers programming language annotation/tag, *@ReactMetod,* in Android and *RTC_EXPORT_MODULE* and *RTC_EXPORT_METHOD* macros in iOS to assist exposing platform specific source code. An Objective C class is required to conform to the *RTCBridgeModule* protocol in

order to expose its APIs to the JavaScript layer. Listing 2 shows an example of this custom bridging in iOS. [7]

```
// CalendarManager.h
#import <React/RCTBridgeModule.h>

@interface CalendarManager : NSObject <RCTBridgeModule>
@end


// CalendarManager.m
#import "CalendarManager.h"

@implementation CalendarManager

// To export the module, CalendarManager

RCT_EXPORT_MODULE();

//  Or, to export with different name, e.g. AwesomeCalendarManager

RCT_EXPORT_MODULE(AwesomeCalendarManager);


// To expose methods from the module, e.g. addEvent in CalendarManager


RCT_EXPORT_METHOD(addEvent:(NSString *)name
                  location:(NSString *)location) {
  RCTLogInfo(@"Creating event %@ at %@", name, location);
}


@end
```

Listing 2. Exporting native module and methods to the JavaScript layer [7].

Accessing the above Objective C module from the JavaScript layer via the bridge is shown in Listing 3.

```
import {NativeModules} from 'react-native';
var CalendarManager = NativeModules.CalendarManager;
CalendarManager.addEvent('Event', 'Location');
```

Listing 3. Accessing native modules from a JavaScript file [7].

2.3 Advantages of cross-platform solutions

Both browser-based solutions and React-Native's bridging solution allow developers to share a single codebase for multiple mobile platforms. In addition, web developers are able to use their experience in web technologies to develop mobile applications.

Some cross-platform solutions do not need to pass App Store or Google Play approval in order to deploy HTML, CSS and JavaScript bugfixes and updates.

2.4 Limitations of cross-platform solutions

Different cross-platform solutions have different limitations tied to their approach in unifying variances of mobile platforms. Browser-based solutions often fail to achieve the look and feel of native applications due to the wide difference between web interface components and what is offered natively on mobile platforms.

React Native's bridge solution on the contrary does not suffer from that, at least not to the same degree as its underlying interface is made of native components.

However, the approach requires React Native developers to adopt to the frequent evolution of native components on each platform. Therefore, some React Native projects are found to have more unshared platform specific implementations in the long run as pointed out by the Airbnb development team [8]. In those cases, the result is a counterproductive project that is more difficult to maintain as the source code is consisted of lesser shared code in JavaScript and more growing exclusive native implementations in Objective C (or Swift) and Java (or Kotlin) for iOS and Android platforms respectively.

Performance on complex animations and fast scrolls is another limitation for some cross-platform solutions. React Native is not necessarily criticized for luck of performance in practice due to the lightning fast hardware most applications run on these days. Its asynchronous implementation that dedicates the main thread for user interface processes helps produce smooth user experience. React Native applications can achieve indistinguishable user interface by reducing communication via the bridge.

React Native comes with built-in virtual DOM (Document Object Model), light weight JavaScript objects that represent UI. A smart diff algorithm is used to reconcile this representation with the latest actual render. After obtaining the list number of steps needed to achieve the new UI compared to latest drawing, React Native applies only the necessary modifications. This fastens the rendering process hence improving UI and user interactions. (9)

However, in theory, initializing the framework's infrastructure during startup, bridging between native and JavaScript layer when drawing interface objects and handling user interactions via the bridge at runtime create inevitable overhead. It is safe to say therefore, it's not unusual to see React Native apps that struggle to maintain the 60 fps (frame per second) standard. The framework's authors have documented a long list of optimizations developers need to be aware of in order to minimize some common performance hits (10).

3 Flutter

Google defines Flutter as a portable UI toolkit for building beautiful natively compiled applications for mobile, web and desktop from a single codebase (11). Tim Sneath, Group product manager in Google, similarly defines it as "A powerful general-purpose open UI toolkit" (12). The built-in support currently is for iOS and Android mobile platforms (see Figure 2).

```
└── FlutterApplication
    ├── android
    ├── ios
    └── lib
```

Figure 2. Flutter application structure

Flutter approaches cross-platform application development in the most radical fashion. It provides its own set of interface objects, rendering, and an engine that implements flutter's animation, graphics, file and network I/O among the many other core libraries [13].

A flutter project is written in Dart programming language and AOT (Ahead-of-time) compiled to the native platform architecture, hence achieving uncompromised speed.

At the top level, flutter provides widgets that are composed of many widgets to make the most common interface objects that we're used to on iOS and Android platforms. Because flutter follows an open and layered architecture, developers can make their own widgets compositing other widgets at any level of the layered architecture. In fact, that is how the Flutter team made the existing high-level widgets and there is no barrier from the framework for developers to do the same. This customization flexibility is unmatched by UI toolkits from either iOS or Android with hierarchical implementations and limited access levels. [14]

A short summary of how Flutter compares to browser-based solutions and React Native can be as follows:

- Browser-based solutions use *WebView* of the platform to render HTML and CSS
- React Native uses native components of the platform via its bridge
- Flutter renders its own user interface on its own canvas and sends it to its own engine that runs on the platform (see Figure 3).

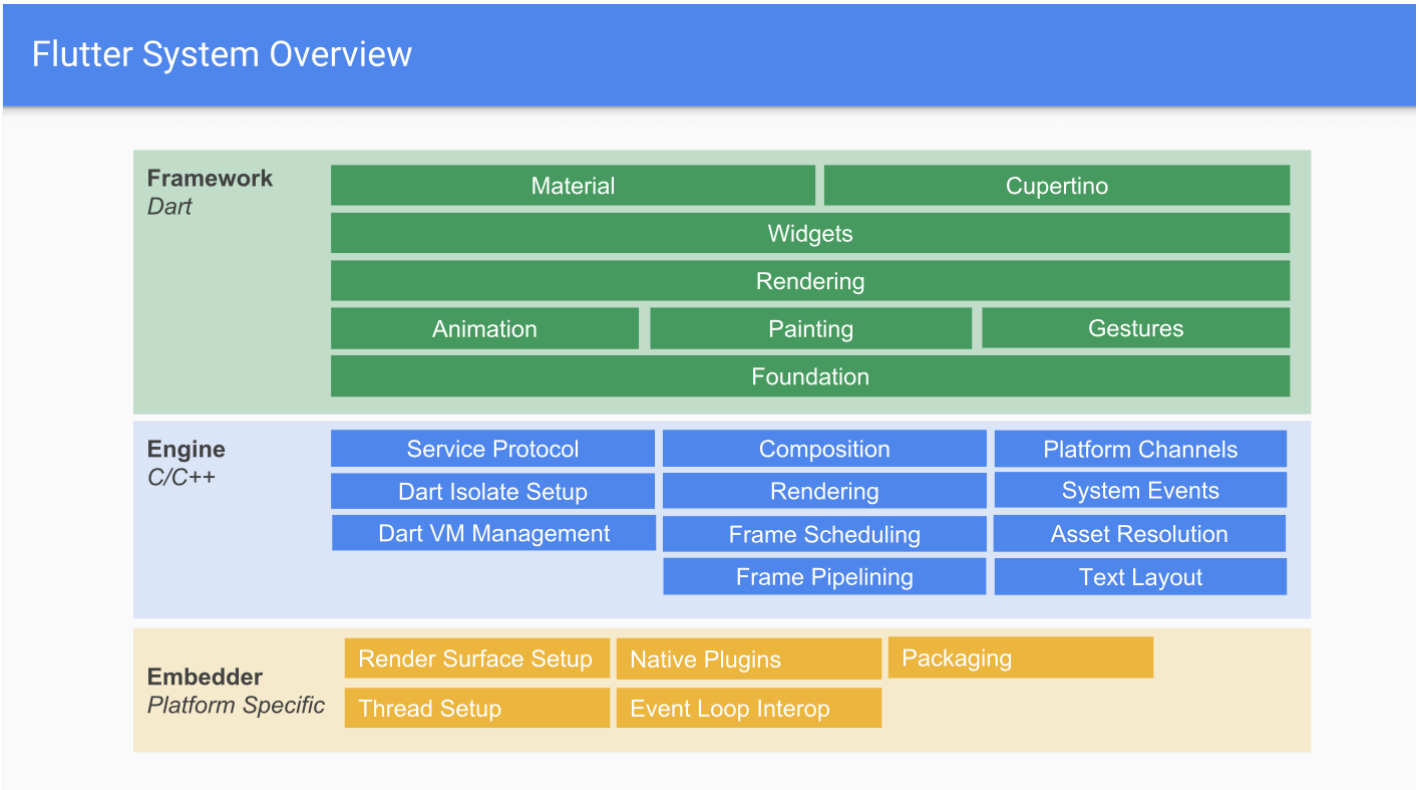

Figure 3. Flutter's system overview

These layers including the Foundation layer in Dart UI framework also called dart:ui are discussed in the later sections.

3.1 Dart

To quote Edsger Wybe Dijkstra, "The tools we use have a profound and devious influence on our thinking habits, and therefore on our thinking abilities" [15]. The Flutter team did not take choosing a programing language for the framework lightly. According to Eric Seidel (lead of the Flutter team in Google), JavaScript was the first choice and dozens of other programming languages were considered before the team settled on Dart, a programming language initially designed for web development [12]. Sample source code written in Dart is shown in Listing 4.

The main evaluation criteria when choosing a programming language for the framework were the author's, developers' and end user's needs [12].

Flutter has a hot reload feature that allows developers to see changes on simulators and emulators without re-running the application. This is a result of Dart VM (Virtual Machine) and its different operation modes for debug and release builds.

Dart VM in debug mode is capable of operating in JIT (Just-in-time) compilation mode dynamically loading and compiling Dart source code to facilitate hot reload, debugging and other features to enhance fast development cycle. [16]

In release mode, Dart VM serves as a runtime library instead of a virtual machine in the traditional sense. The Dart source code is AOT (Ahead of Time) compiled and the Dart VM, as runtime, is used to execute precompiled machine code with fast garbage collection, dynamic method lookup and more *runtime* supports. [16]

Some of the critical strengths of Dart are:

- It is type-safe hence preventing disastrous developer mistakes
- Supports AOT and JIT compilation therefore perfect for hot-reload feature
- Its faster than JavaScript

Dart is not the only language that fulfills these requirements. The Flutter team was able to work closely with the Dart team in Google which made the language a convenient choice. [12]

Dart has allowed creating interface objects dynamically in code rather than relying on a rigid markup system. As a result, building custom interfaces is relatively easier in Flutter. Dart's ability to allocate objects and perform garbage collection without locks plays a big role in achieving 60 fps even for complex animations.

A simple example of creating an app that displays fixed set of words in a list view is shown below (see Listing 4). The user interface is created in code as Dart does not depend on XML (Extensible Markup Language) or JSX (JavaScript XML).

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

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

class App extends StatelessWidget {
 static const titleText = "Word List";

 @override
 Widget build(BuildContext context) {
  Widget body = Center(child: WordsListWidget());
  Widget homeWidget = Scaffold(
    body: body, backgroundColor: Colors.white
  );
  ThemeData theme = ThemeData(primaryColor: Colors.deepPurple);
  return MaterialApp(
    title: titleText, theme: theme, home: homeWidget
  );
 }

}

// A List widget for fixed set of `words`
class WordsListWidget extends StatelessWidget {
 final List<String> words = new List<String>
   .generate(10, (i) => "Word $i");

 @override
 Widget build(BuildContext context) => _makeListView(context);

 // Makes fixed list view populated with `words`
 Widget _makeListView(BuildContext context) {
  Iterable<ListTile> tiles = words.map(
      (String pair) => new ListTile(title: Text(pair)));
  List<Widget> rows = ListTile
    .divideTiles(context: context, tiles: tiles).toList();
  return new ListView(children: rows);
 }

}
```

Listing 4. Dart source code of an app that displays words in a list view

3.2 UI Framework

The top-most layer of Flutter is its Dart UI framework. This layer contains themes (which are widgets), widgets, rendering and the last layer called dart:ui that handles communication down to the flutter engine as shown in Figure 4. The bottom layer, dart:ui, serves as a foundation for the rendering and Widgets library (see Figure 3) and is explored in more detail later.

Figure 4. The Dart UI framework layers

3.2.1 Themes

The framework comes with already made Cupertino (for iOS) and Material (for android) packages. These packages contain interface elements, colors and gesture behaviors specific to the theme or the platform since these themes are designed to resemble iOS and Android platforms. Therefore, in concept, one can implement dynamic setting to switch between Android and iOS themes on a Flutter app regardless of the actual platform the application is running on. [17, 18]

3.2.2 Widgets

Widgets are the key and the only interface components of this framework. The concept of widget is not common to either Android or iOS developers. The term packs much more

implications than the widgets recognized e.g. in Android. The Flutter's documentation defines it as "the way you declare and construct UI" [17].

They are immutable description of every aspect of what is ultimately drawn on the canvas including its appearance, content and position. Take the following simple example flutter app that renders a title at the top (navigation title in iOS or the app bar title in Android) and a button at the center of the screen in Listing 5.

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


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

class App extends StatelessWidget {

 @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),
    ),
   ),
  );
 }

}
```

Listing 5. Sample flutter app

The App itself is a subclass of *Widget* with *build* method the framework calls to determine how the widget and its child widgets should be drawn. Another widget is *Scaffold* which is an abstraction of the general material design screen with top bar, body and other parts [19]. There is also the *Center* widget which is used to define its child widget's position. In contrary to the most common approach where position is a property of the interface object (*View* in Android and *UIView* in iOS), position in flutter is a *Widget* that defines its child's position, reportedly making rendering position updates much faster [12].

Under the hood, tree of widgets is generated and used to draw what should appear on the screen. In contrast to Android's *View* and iOS's *UIView* which are mutable and rendered on screen, Flutter's widgets are very light objects that compose to form other widgets and a widget tree.

3.2.3 Rendering

In Flutter's pipeline lies a fast rendering layer handling layout, paint and composite (see Figure 5). By avoiding complex layout that requires sophisticated calculations in favor of much deeper widget trees and relatively modest algorithms, Flutter's rendering proves that simple is fast and flexible. It uses very simple box constraint model in comparison to the typical sophisticated models used in other systems. For example, *UIKit* (of iOS) uses linear constraint model with a general-purpose constraint solver. (20)

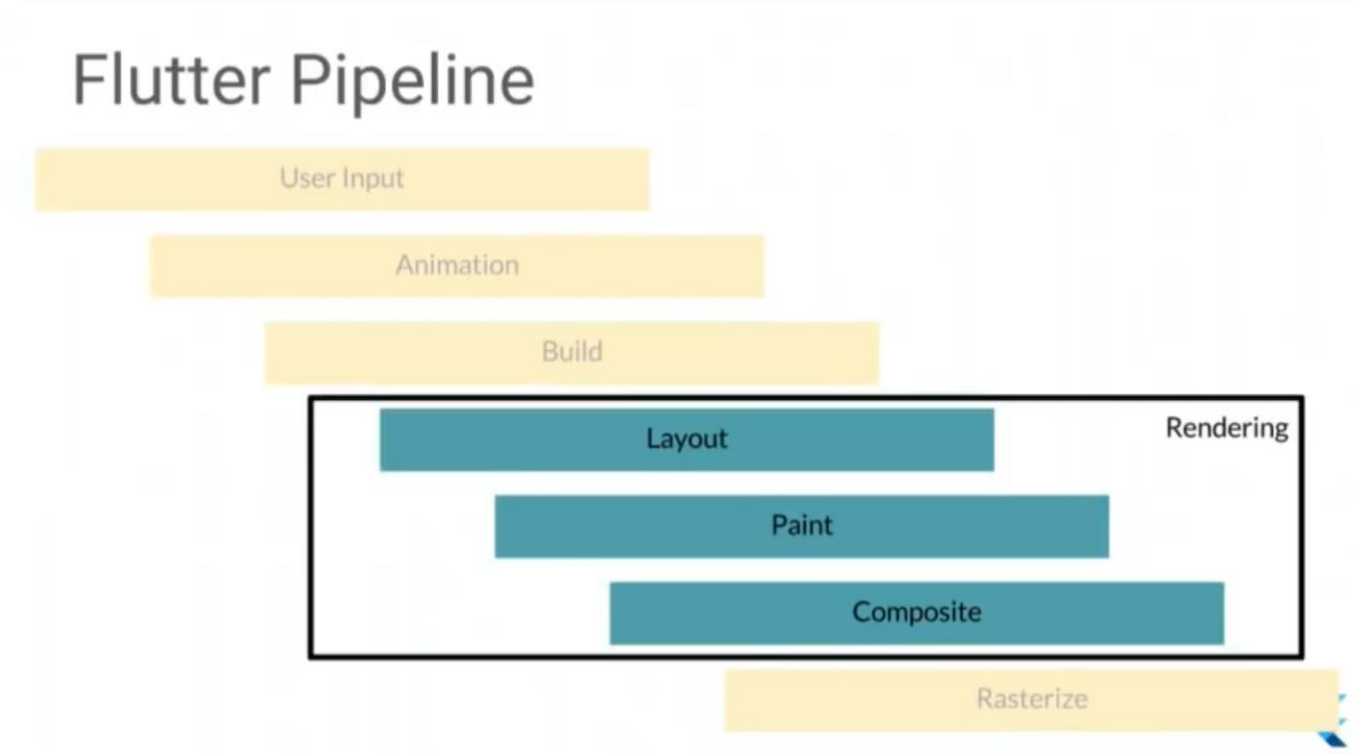

Figure 5. Flutter pipeline

Modern GPUs (Graphics Programming Unit) are great at compositing. Flutter takes advantage of this by applying structural repainting using compositing. It invalidates

subtrees when repaint is necessary as opposed to the traditional method of tracking rectangles that are invalid and in need of repainting. (21)

The design principle of flutter's rendering can be summarized as follows:

- One-pass, linear-time layout and painting
- Simple box constraints, expressive layout
- Structural repainting using compositing

3.2.4 Dart:ui

This layer of the UI framework lies between Flutter's rendering and the C/C++ engine. It contains classes such as *Canvas*, *Paint* and *TextBox*. It lays the foundation for building interface objects. Apps and UI libraries can be made at this layer. In fact, that is what the Flutter team did to support the framework on the web (see Figure 6).

Most apps targeting either iOS or Android platforms build on top of the higher-level UI framework layer. App developers will not need to interact with this low-level implementation directly.

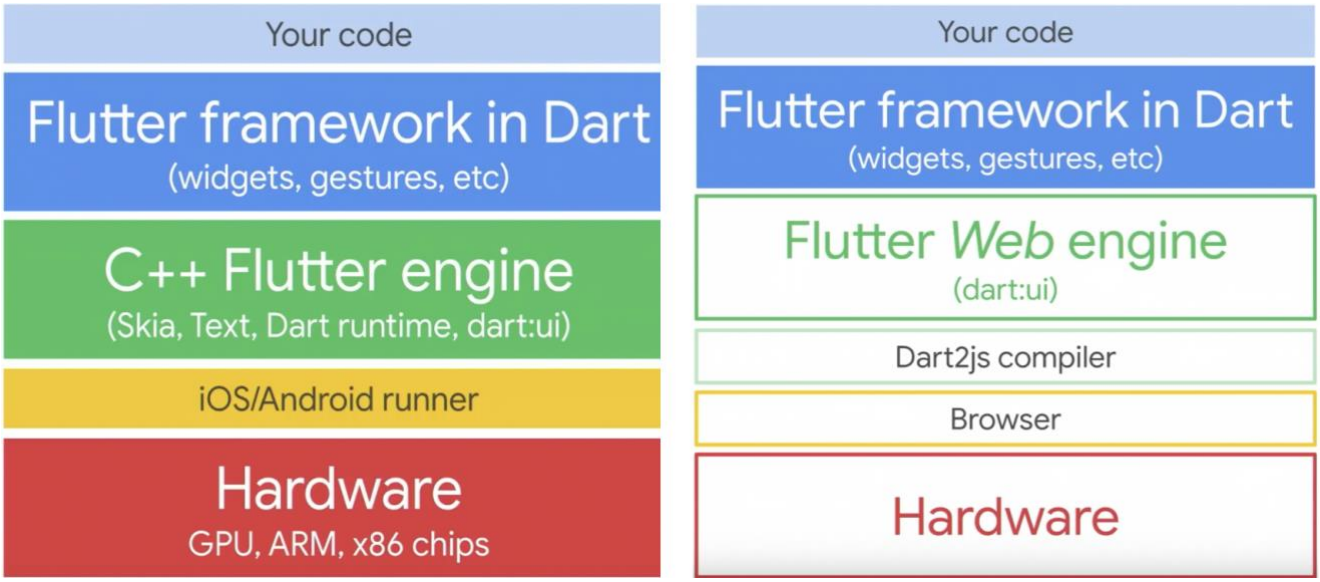

Figure 6. Flutter architecture on mobile and web platforms

3.3 Engine

The engine, placed beneath the Dart UI framework, is a portable runtime that hosts flutter applications. It contains Dart runtime, Skia (an open source 2D graphics library written in C++), file and network I/O, accessibility support and other core libraries. [22]

This thin layer written in C and C++ is compiled with LLVM on iOS and Android's NDK (Native Development Kit). The Dart UI framework and application source code are compiled to ARM libraries in both platforms and are delegated to process event loop, rendering and other tasks. This game-like engine implementation yields fluid user experience regardless of the platform. [14]

Flutter's engine is window toolkit agnostic; its API (written in C) does not contain platform specific dependencies and can be found on GitHub [23]. Having an engine that has no dependency on the underlying platform allows creation of custom embedders for platforms that are not supported out of the box. There are open source libraries that take advantage of this to provide Windows, macOS and Linux support for the Flutter framework [24].

3.4 Embedder

Mediating between the engine and the underlying platform is the Embedder (see Figure 7). This layer contains shells (iOS Shell and Android Shell) that host the Dart VM. Fundamentally the embedder is an implementation of the open source Flutter API [22]. It is responsible for running the Dart VM where precompiled application code executes. It exposes the native API for services and features that are required by the framework.

Higher-level layers of the Flutter framework; the Dart UI library and the engine are platform independent because all crucial interactions with native APIs, such as access to canvas, take place at this layer.

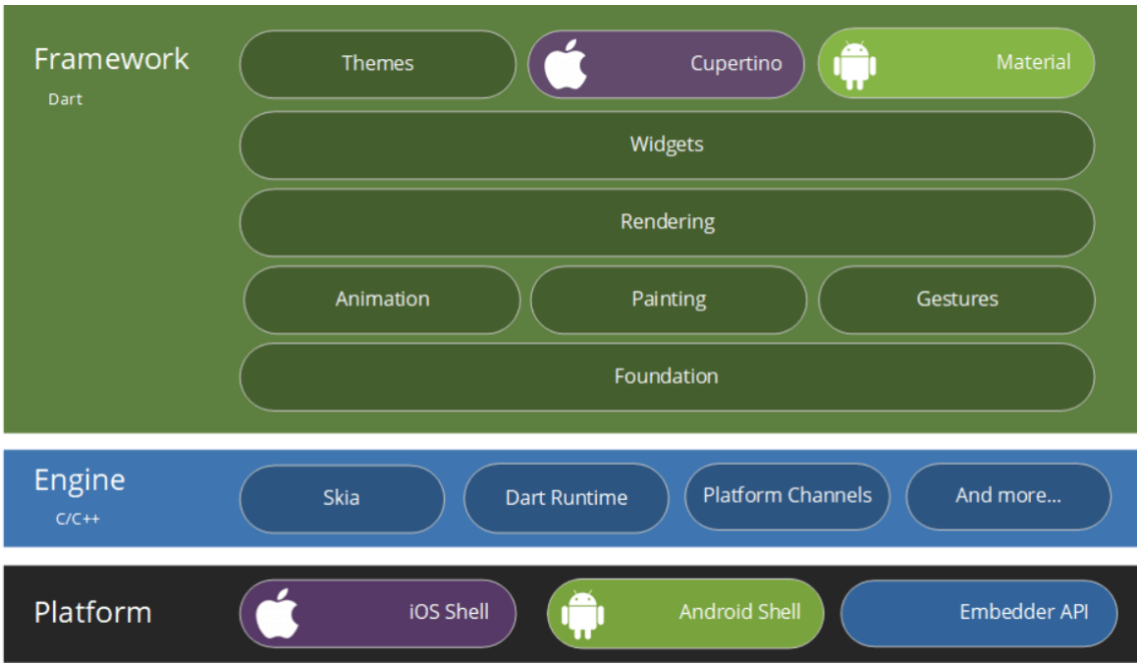

Figure 7. The embedder, engine and Dart UI framework of Flutter [25].

3.5 Platform Channels

One of the key challenges cross-platform solutions face is allowing application code to directly communicate with native APIs. In addition to drawing on screen and processing user interactions, applications sometimes need to reach for native APIs be it to access sensors, camera, GPS (Global Positioning System) etc., or to handle deep links, notifications, app lifecycle and many more.

A naive solution would be wrapping all anticipated interactions with native APIs inside a cross-platform development framework. However, this would require continuous maintenance due to the rapid evolution of native APIs. Furthermore, applications would be forced to contain unused wrappers increasing application size.

Flutter solves the problem by providing platform channels for direct communications with native APIs (see Figure 8).

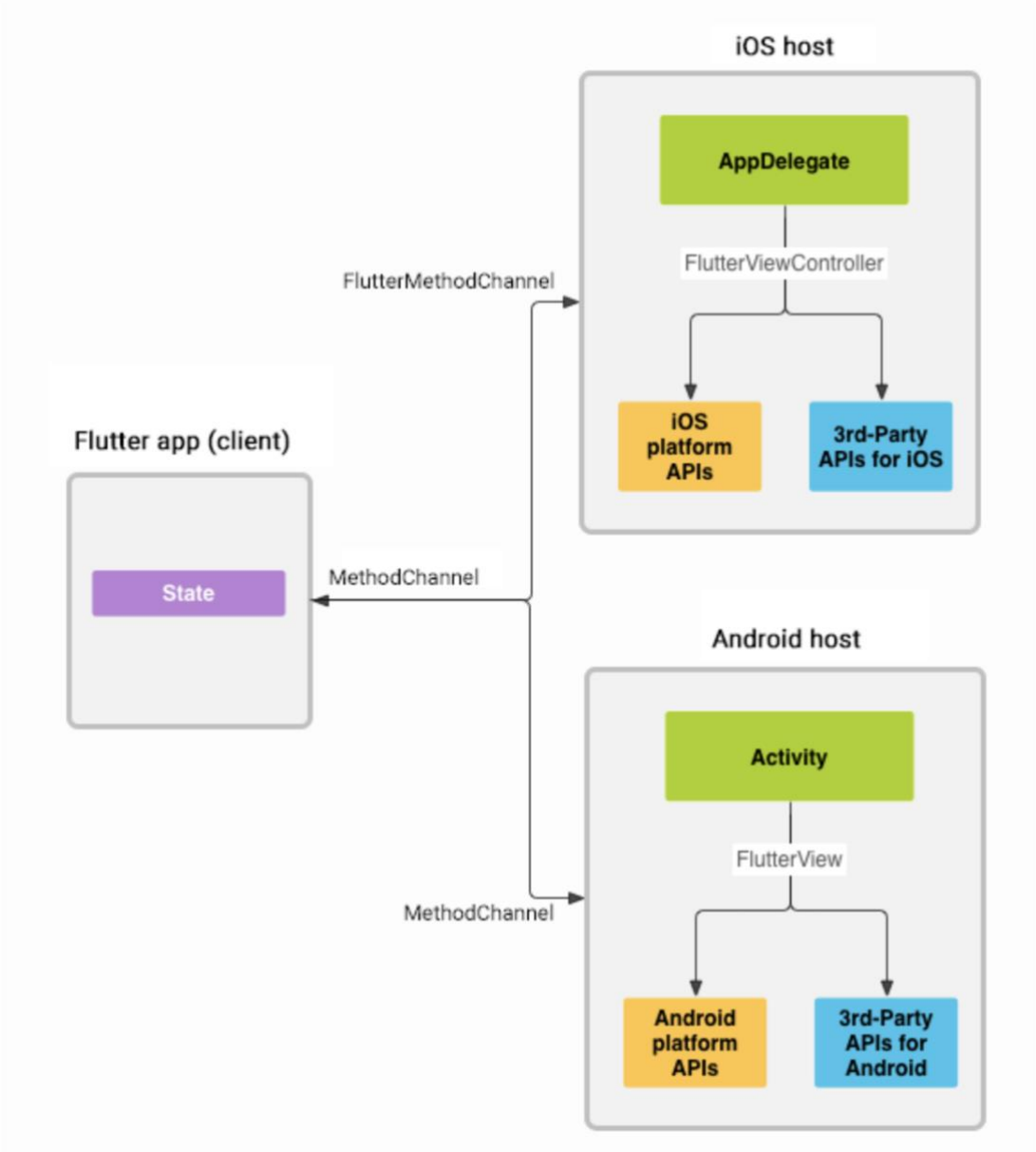

Figure 8. Flutter platform channels [26].

Flutter's interface is hosted inside the platform's standard component. In case of iOS, a *UIViewController* instance, *flutterViewController*, is added to the window at launch. Executing native code before or after presenting Flutter's interface is trivial. The platform specific projects (e.g. .xcodeproj file in iOS) are available inside Flutter's project structure and can be opened using the associated IDE (Integrated Development Environment).

Platform specific tasks that do not need a Dart API can be implemented natively in the platform project with no difference to strictly native projects. On the other hand, in cases where a common Dart API is necessary to native APIs, Flutter channels can be used in conceptually similar manner as React Native's bridge.

Flutter channels are used to access platform specific code inside the Flutter framework. Developers are responsible to selectively expose the native APIs they need in two steps.

- Define the Dart API that sends message to the platform implemented in *services.dart* package

- Expose the native API using Flutter's *FlutterMethodChannel* implemented natively on each platform

This prevents Flutter from containing unnecessary, unused or even outdated wrappers. A simple example of creating asynchronous getter for device battery level is shown in Listing 6. Native implementations are required via the platform channel.

```
// The Dart API

import 'dart:async';
import 'package:flutter/services.dart';

Future<void> _getBatteryLevel() async {
 String batteryLevel;

 try {
  final int result = await platform
     .invokeMethod('getBatteryLevel');
  batteryLevel = 'Battery level at $result % .';
 } on PlatformException catch (e) {
  batteryLevel = "Unavailable: '${e.message}'.";
 }

}
```

Listing 6. Asynchronous getter that accesses native API [26].

The following code shows Swift implementation that exposes device battery level to a Dart API using *FlutterMethodChannel* (see Listing 7).

```
// Dart API native implementation for iOS (Swift)

@UIApplicationMain
@objc class AppDelegate: FlutterAppDelegate {
  override func application(
    _ application: UIApplication,
    didFinishLaunchingWithOptions
    launchOptions: [UIApplicationLaunchOptionsKey: Any]?)
    -> Bool {

    let controller : FlutterViewController =
      window?.rootViewController as! FlutterViewController
    let batteryChannel = FlutterMethodChannel(
      name: "samples.flutter.io/battery",
      binaryMessenger: controller)

    batteryChannel.setMethodCallHandler { [weak self]
      (call: FlutterMethodCall, result: FlutterResult) in
      guard call.method == "getBatteryLevel" else {
        result(FlutterMethodNotImplemented)
        return
      }
      self?.receiveBatteryLevel(result: result)
    }

    GeneratedPluginRegistrant.register(with: self)
```

```
        return super.application(
            application,
            didFinishLaunchingWithOptions: launchOptions)
    }

    private func receiveBatteryLevel(result: FlutterResult) {
        let device = UIDevice.current
        device.isBatteryMonitoringEnabled = true
        if device.batteryState == .unknown {
            result(FlutterError(code: "UNAVAILABLE",
                                message: "unavailable",
                                details: nil))
        } else {
            result(Int(device.batteryLevel * 100))
        }
    }

}
```

Listing 7. Exposing iOS native API to Dart framework via platform channel [26].

This approach has more advantages. It allows separating platform specific code from application code therefore encouraging developers to create Dart APIs backed by Kotlin/Java and Swift/Objective C implementations that can be shared across applications as plugins. It is a good idea to look for existing plugins before implementing one and sharing these plugins with other developers.

4 SDK development

The acronym, SDK, for most mobile application developers is tied to the Android SDK and the iOS SDK that are used to make applications on the platforms.

As the unabbreviated name embraces, SDK is set of software components packed together to aid developers in using a certain system or service. The Firebase SDK for example helps developers easily access services provided by Firebase such as authentication and remote database [27]. A company called Stripe has mobile SDKs for different platforms offering set of UI components and tools to help developers integrate Apple pay and credit card payment systems with relative ease [28]. There are many examples of such services that offer SDKs for the end user; app developers.

Unlike application development, SDK development requires designing and implementing software components that can be integrated in many apps regardless of applications project structure and other domain specific assumptions.

React Native understandably requires developers to integrate such SDKs for each platform separately. Afterall, it is native under the hood and there is no dissimilarity as opposed to doing the same on native projects.

Platform SDKs that provide UI components among other tools are less straightforward to integrate with Flutter applications. SDK authors can increase the popularity of their product by implementing separate dedicated SDKs for Flutter. From the perspective of SDK authors, Flutter is another platform and they need an additional implementation of their SDKs for developers. For SDKs that require native APIs, providers may find it easy to incorporate Dart API along with the native implementations as templates or integration instruction.

5 Building SDKs using Flutter

As the number of Flutter applications on production grow, SDK providers will need to consider supporting these applications in the most optimized and efficient manner. The Flutter team and community on the other hand can support the effort by providing templates and command line tools that can be used to easily create SDKs for Flutter applications.

5.1 Flutter project structure

Flutter has a strict project structure. The command *flutter create* generates a starter project with small sample code. The latest versions of IDEs such as Android Studio, IntelliJ and Visual Studio Code have Flutter plug-ins that need to be installed for Flutter app development. These IDEs with Flutter plug-ins provide GUI (Graphical User Interface) to create a starter project (see Figure 9).

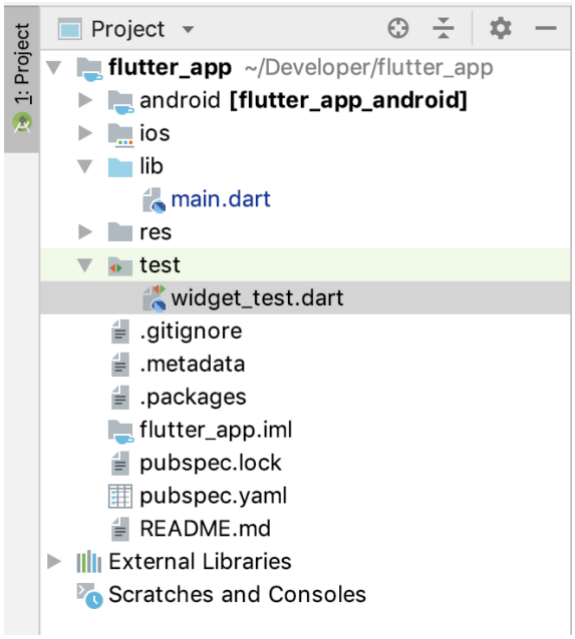

Figure 9. Flutter project generated in Android Studio

Flutter projects generated using these quick tools come with a heavily opinionated project structure that makes Flutter the center of the application [29]. This structure is integral to properly execute application code hence making attempts to modify and customize the project structure is rather unpleasant.

This sturdy structure, in addition to luck of convenience for SDK authors, makes integrating Flutter to existing iOS and Android applications a less straightforward task.

Inside flutter applications are folders named `*ios* ` and `*android*` containing each platform's runner project (see Figure 10). As a simple example of lucking structure flexibility, attempting to rename the iOS generated project from its default name

`*Runner.xcodeproj*` to something more custom such as `*MyApp.xcodeproj*` is nearly impossible [30].

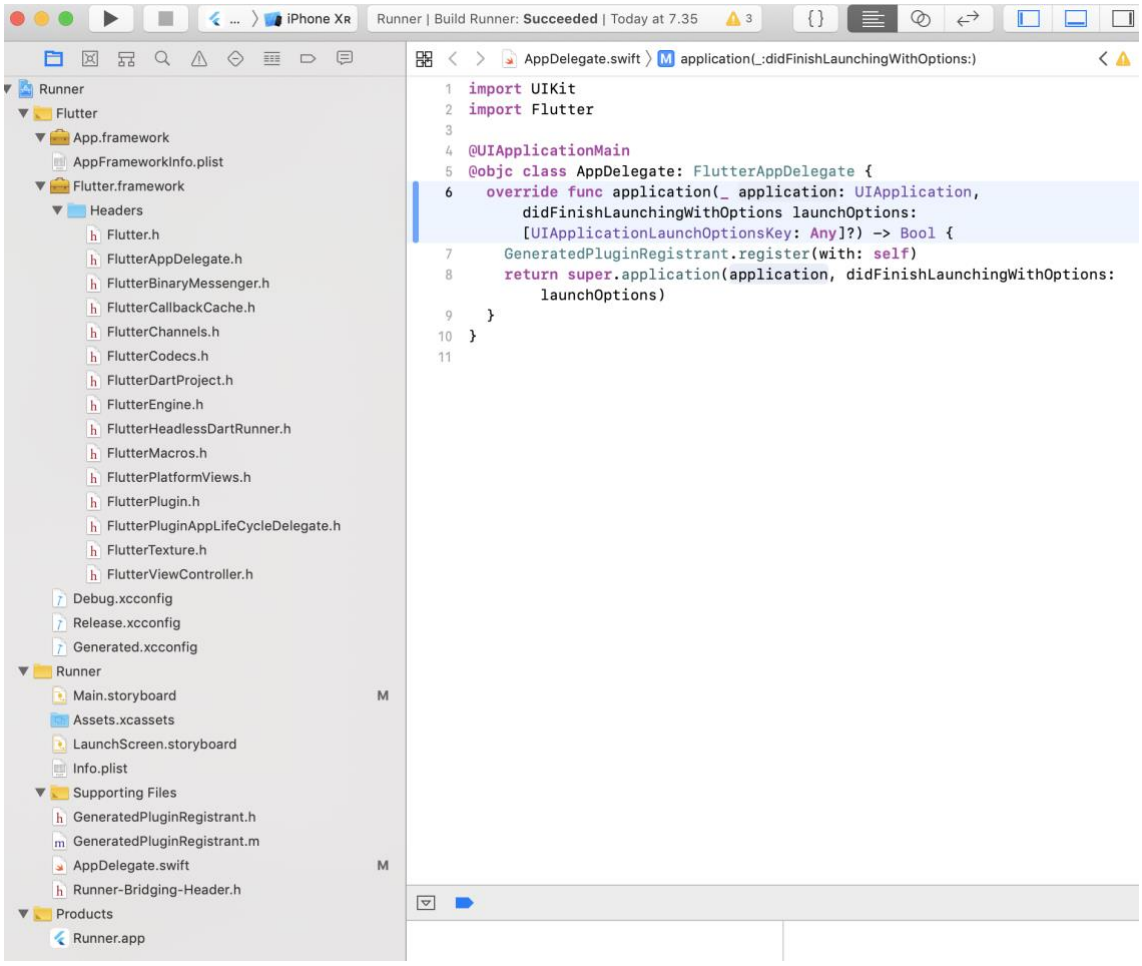

Figure 10. Xcode project generated by flutter create command

A conventional method of adopting new technologies is integrating and testing with existing solutions until benefits of complete replacement is proven. This also gives developers the necessary time to familiarize themselves with the ins and outs of the technology.

There has been recent effort to simplify integration of Flutter framework to existing Android and iOS projects [29]. This currently in progress effort could allow developers to easily implement their new features in Flutter while gradually migrating their existing native applications.

5.2 Git submodule

Git submodule is a convenient tool to assist integration of Flutter framework to existing iOS and Android applications.

It is one of the easiest techniques to manage local dependencies. It has a simple command, *git submodule add*, to add repositories as dependency. The command, *git submodule update -- init -- recursive*, updates existing submodules to latest commits and initializes missing submodules.

Flutter applications in addition to the framework contain iOS and Android source code and other generated files. Git submodules can be used to separate these into three repositories adding the iOS and Android repositories as submodules of the Flutter repository.

Flutter project structure makes the following assumptions:

- The application folder names for iOS and Android projects are `ios` and `android` respectively

- The main android module is named `app` and the iOS runner project is named `Runner.xcodeproj`

It's necessary to rename existing projects appropriately to match these assumptions. Due to the immaturity of the current version of Flutter, other different project specific build settings that should be modified for a successful integration will very likely break in upcoming Flutter releases.

Once embedding Flutter in an existing project, displaying the first Flutter interface is the next step. In Android projects, Flutter requires the *Application* class to extend *FlutterApplication* and in iOS projects, *AppDelegate* to conform to the protocol *FlutterAppDelegate*.

Interface objects, *FlutterActivity* in Android and *FlutterViewController* in iOS can then be displayed on screen using *startActivity* and *present* APIs in Android and iOS respectively. Listing 1 demonstrates the steps for an Android project. (22)

```
// Create a Flutter activity
class MyFlutterActivity : FlutterActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    GeneratedPluginRegistrant.registerWith(this)
  }
}

<!—Register the activity-->
<activity
android:name=".MyFlutterActivity"
android:launchMode="singleTop"
android:configChanges="orientation|keyboardHidden|keyboard|screenSize|locale|layoutDirect
ion|fontScale"
android:hardwareAccelerated="true"
android:windowSoftInputMode="adjustResize">
<meta-data
    android:name="io.flutter.app.android.SplashScreenUntilFirstFrame"
    android:value="true" />
</activity>

// Finally display the activity
startActivity(Intent(this, MyFlutterActivity::class.java))
```

Listing 1. Displaying Flutter interface on Android application

5.3 Flutter Attach

Flutter Attach is an open source project currently working on simplifying the integration of Flutter to existing iOS, Android and Xamarin projects (31). The project aims to simplify attaching existing projects as modules of Flutter template project.

This method strongly relies on the existing iOS and Android project's structure. The first version for example is designed to comply with an iOS project structure generated in Xcode 10.0. Therefore, developers and maintainers need to actively keep up with project structure modifications that come from IDE updates.

6 Conclusion

The goal of this thesis is to offer a comprehensive overview of Flutter UI toolkit and explore the framework's potential for cross-platform SDK and application development. The thesis concludes that Flutter is a strong choice for application development and this argument only gets stronger as its internal technologies mature. Its use for SDK development is less mature and using it with existing iOS and Android projects is not straightforward due to its rigid project structure at the time of writing. However, the Flutter team and the open source community are working to remove this obstacle and SDK authors and application developers can follow and contribute to the progress.

As a cross-platform solution, Flutter's approach to achieving single codebase for multiple platforms is successful without most of the compromises its competitors including React Native suffer from.

In fact, the revolutionary technologies of Flutter's internals and the fast development cycle it introduces makes it a competitive choice over native development regardless of its cross-platform aspect. Its unique rendering techniques and layered architecture makes it an easy choice for innovative interface and animation implementations. The choice of avoiding inheritance in favor of composition at every layer of the framework opens an unpredictable exciting use cases of its components.

There is no doubt that Flutter is exciting for developers and a great choice for businesses. The author of this thesis has learned more about Flutter, React Native and other cross-platform solutions during this research and a number of helpful experimental projects were made with plans to use and contribute to the framework in the future.

References

1. Simon
   URL: https://www.microsoft.com/buxtoncollection/detail.aspx?id=40
   Accessed 28 March 2019

2. The App Store turns 10
   URL: https://www.apple.com/newsroom/2018/07/app-store-turns-10
   Accessed 28 March 2019

3. Introducing Google Play: All your entertainment, anywhere you go
   URL: https://googleblog.blogspot.com/2012/03/introducing-google-play-all-your.html
   Accessed 28 March 2019

4. Flutter
   URL: https://flutter.dev
   Accessed 28 March 2019

5. Why Airbnb is moving off of React Native
   URL: https://softwareengineeringdaily.com/2018/09/24/show-summary-react-native-at-airbnb/ Accessed 1 May 2019

6. Deploy
   URL: https://ionicframework.com/docs/appflow/deploy/intro
   Accessed 15 May 2019

7. Native Modules
   URL: https://facebook.github.io/react-native/docs/native-modules-ios
   Accessed 25 April 2019

8. React Native at Airbnb
   URL: https://medium.com/airbnb-engineering/react-native-at-airbnb-f95aa460be1c Accessed 25 April 2019

9. React's diff algorithm
URL: https://calendar.perfplanet.com/2013/diff/
Accessed 25 April 2019

10. Performance
URL: https://facebook.github.io/react-native/docs/performance
Accessed 25 April

11. Flutter
URL: https://flutter.dev
Accessed 20 May 2019

12. "Flutter: How we're building a UI framework for tomorrow in Google" by Eric Seidel (Lead of Flutter team at Google)
URL: https://www.youtube.com/watch?v=VUiVkDpikDI
Accessed 16 April 2019

13. The Engine Architecture
URL: https://github.com/flutter/flutter/wiki/The-Engine-architecture
Accessed 20 April 2019

14. FAQ
URL: https://flutter.dev/docs/resources/faq
Accessed 20 April 2019

15. Edsger W. Dijkstra (2012). "Selected Writings on Computing: A personal Perspective", p.129, Springer Science & Business Media

16. Introduction to Dart VM
URL: https://mrale.ph/dartvm/
Accessed 16 April 2019

17. Flutter for Android developers
URL: https://flutter.dev/docs/get-started/flutter-for/android-devs
Accessed 20 April 2019

18. Flutter for iOS developers
URL: https://flutter.dev/docs/get-started/flutter-for/ios-devs
Accessed 20 April 2019

19. Scaffold class
URL: https://api.flutter.dev/flutter/material/Scaffold-class.html
Accessed 12 April 2019

20. Rendering in Flutter
URL: https://flutter.dev/docs/resources/rendering
Accessed 12 April 2019

21. Flutter Rendering Pipeline (GoogleTechTalks)
URL: https://www.youtube.com/watch?v=UUfXWzp0-DU
Accessed 12 April 2019

22. Flutter Engine
URL: https://github.com/flutter/engine#architecture-diagram
Accessed 16 April 2019

23. Engine
URL:
https://github.com/flutter/engine/blob/080fbcb1759e5916f0d6cdcdfd945c053320e6b3/shell/platform/embedder/embedder.h
Accessed 16 April 2019

24. Desktop Embedding for Flutter
URL: https://github.com/google/flutter-desktop-embedding
Accessed 17 April 2019

25. How Flutter Works
URL: https://buildflutter.com/how-flutter-works/
Accessed 17 April 2019

26. Writing custom platform-specific code
URL: https://flutter.dev/docs/development/platform-integration/platform-channels
Accessed 17 April 2019

27. Firebase SDK
URL: https://opensource.google.com/projects/firebase-sdk
Accessed 20 April 2019

28. iOS Integration
URL: https://stripe.com/docs/mobile/ios
Accessed 20 April 2019

29. Integrating Flutter into an Existing App – Part One: Flutter with Submodule
URL: https://medium.com/@tpolansk/integrating-flutter-into-an-existing-app-part-one-flutter-with-submodules-9b633ff3cf10
Accessed 2 March 2019

30. Would like to be able to change ios/Runner.xcodeproj to ios/MyName.xcodeproj #9767
URL: https://github.com/flutter/flutter/issues/9767
Accessed 12 April 2019

31. Add Flutter to existing apps
URL: https://github.com/flutter/flutter/wiki/Add-Flutter-to-existing-apps
Accessed 15 May 2019