Charles Thao
(with partial collaboration with Huzefa Mandviwala)

EC500 Case Study Project - Flutter

Flutter is an SDK developed by Google for fast and robust cross-platform mobile application development. It is an open-source project used in many applications, and it aims to allow designing and writing interfaces that function well, look sleek and are easy to code. Some of its core features include a fully customizable and flexible UI, a responsive interface, and integration with existing tools like Java, Kotlin, Objective C, and Swift code, platform APIs, as well as third-party SDKs.

Technology and Platform

a. Coding language

Flutter is written in Dart, a programming language also developed by Google. Dart is an object-oriented language which can be compiled to JavaScript or mobile application languages like Objective-C, Swift, Java and Kotlin. According to Flutter's developers, Dart was "designed to be easy to write development tools for, well-suited to modern app development, and capable of high-performance implementations." Outside the scope of Flutter, Dart could be employed to build web and server applications.

Since Flutter debuted in 2017, had it been introduced today, Dart would still be a perfect candidate. First of all, Flutter is a Google project using primarily Google libraries and frameworks (such as Skia), an in-house language like Dart would facilitate integrations and developments by eliminating the barrier between language maintainer/developer and Flutter framework developer. While there exists other more established languages for creating applications, such as JavaScript for React Native, Dart has more features that could be used to create smooth UIs with animations and powerful rendering. Dart includes a rich type system, module system, native testing framework, and a native package manager.

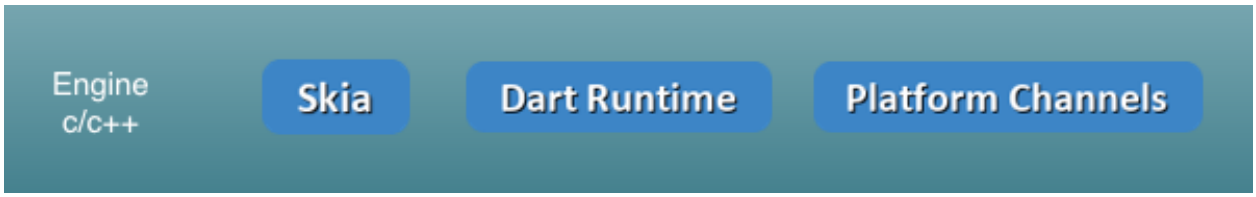
b. **Build system**: Flutter functions like a game engine. Instead of utilizing web view or relying on the device's OEM widgets, Flutter renders every view component using its own high-performance rendering engine. This allows building applications with native-like performance characteristics. The engine's C/C++ code is compiled with NDK on Android and LLVM on iOS. Any Dart code is AOT-compiled to native code during compilation.

In order to develop a Flutter application, users have two options.

1. Visual Studio Code (or any IDEs with Flutter and Dart supports): Flutter codes could be compiled and run on the command line.
2. Android Studio (with Flutter and Dart plug-ins)

c. **Engine and Frameworks.**

Engine: The engine's C and C++ code is compiled with Android's NDK or iOS's LLVM. The Dart code (both the SDK and the App code) is ahead-of-time (AOT) compiled into native ARM library. This library is included in "runner" Android/IOS project, and combined into .apk or .ipa. When launched, the app loads the Flutter Library. Any rendering, input or event handling is delegated to the compiled Flutter and app code.



Framework: The Flutter framework contains everything you need to develop an app. Flutter apps look like native iOS or Android application with Cupertino theme for iOS, and Material for Android.

Flutter is heavily based on materials software design paradigm, a design model by Google. A basic unit in any Flutter application is called a widget. Flutter doesn't have native controls or components. Flutter draws the UI output on a Skia Canvas. This reduces complexity drastically, as Flutter only has Widgets. Widgets are UI controls that you can use in your app. Your entire app will be made up of Stateless or Stateful Widgets.

Flutter also supports using C/C++ libraries, including direct calls from C/C++ to Dart and from Dart to C/C++.

Testing

a. **Testing metrics:**

Flutter enforces a strict testing regiment in order to maintain sanity and working code. Dart tests are written with a "flutter_test" package API, follow a specific .dart naming convention, and placed inside a master/dev/tests/ subdirectory. Metrics during testing includes code coverage and concurrency tests.

Flutter supports regular unit tests, unit tests with golden-file testing, and end-to-end tests. Regular unit tests written with the flutter_test package utilize flutter-specific extensions on top of the Dart test package in order to test against specific Flutter functionality. Golden-file tests involve comparing an interface generated by Flutter code to a screenshot taken manually of the expected output. This comparison is done at a pixel-by-pixel level in order to ensure that the output generated from the code is an exact match to the expected output. The screenshot is known as the "golden-file", because it is the standard to which the generated output must match up to.

```
testWidgets('Golden file testing', (tester) async {
  // RepaintBoundary is needed to optimize the png
  await tester.pumpWidget(RepaintBoundary(child: MyWidget(0)));

  await tester.pumpAndSettle();

  await expectLater(
    find.byType(RepaintBoundary),
    matchesGoldenFile('golden-file.png'),
  );
});
```

While these tests are mandated by Flutter, developers could also include integration tests, manual tests, and missing dependency tests. Each of these can be found in their own directory under master/dev/. All these tests ensure that new features and bugfixes will work with existing code and not disaffect or break any existing code.

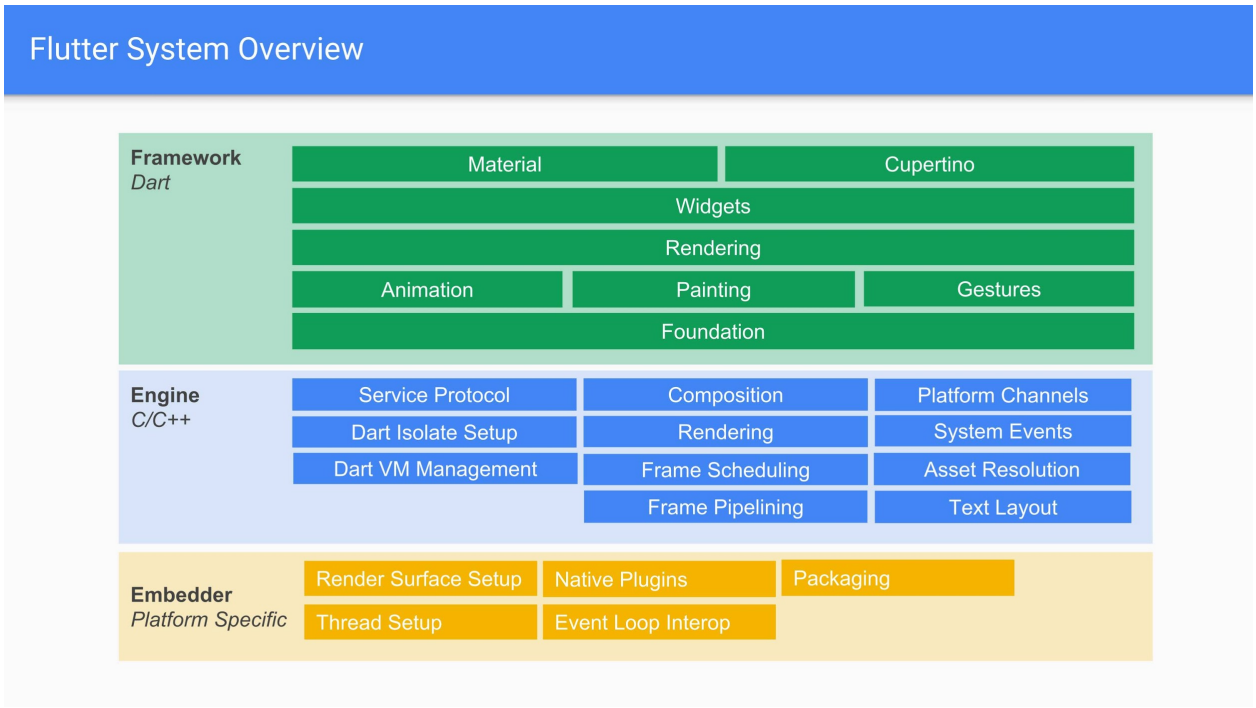
b. <u>Continuous Integration (CI) and test platforms</u>

These tests, all included in the test.dart script for each PR, are run by Flutter's Cirrus Continuous Integration platform. Cirrus is a modern CI, built by Google, that is cloud-enabled in order to perform fast, efficient, and secure testing. Cirrus employs bots to run testing on Flutter tools, framework, rebuilds, and updates. It also uses containers to test the various tasks and builds. The Cirrus configuration can be found in master/.cirrus.yml, and we gathered from this file that Cirrus is testing on Windows, Mac, Linux, Android, JDK, and iOS.

Finally, there are post-commit end-to-end tests that run on a physical lab known as the Devicelab. The Devicelab tests Flutter on physical Android and iOS devices using some scripting tasks, and reports the results back to developers. This final testing stage ensures that Flutter actually works on real devices as a whole package. The Devicelab employs 2 Linux agents, 7 Mac agents, and 2 Windows agents. Each agent runs various tasks in order to give detailed analysis of what exact task might fail on what exact system.

Flutter ensures test coverage using Coveralls, although this feature is currently broken. Instead, they employ a few workarounds to download coverage information. In addition, developers are encouraged to check code coverage of any new features or bugfixes implemented before submitting any PRs.

Software Architecture



a. **<u>Contribution and Usage</u>**

Contributing to Flutter is a regulated process with a few crucial steps. After setting up an engine development environment which uses C++, Java, and Objective C, a developer must also setup a framework development environment which uses Dart. Flutter enforces strict Tree hygiene and Issue hygiene, which describe how to handle changes, failures, triaging/assigning bugs, and Github labeling/naming conventions. Finally, Flutter also has a comprehensive style guide which dictates code format, syntax, and API design.

In general, a workflow for adding a new feature might look like the following:

1. Fork the repo on Github
2. File a new issue (if one does not exist) for the new feature
3. Discuss design on the issue with the Google Flutter team
4. Create a branch on your fork, implement change
    a. Ensure the code is tested
    b. Use code coverage tools to check that new code is covered by tests
    c. Follow style guide for formatting
5. Submit branch as a PR

6. Code review by Flutter experts
7. Ensure PR passes pre-commit tests
    a. Also test post-commit tests locally (Devicelab tests)
    b. Wait for Cirrus to successfully complete CI
8. Merge your new code!
9. Watch post-commit Devicelab tests to ensure everything works
    a. If anything breaks, revert and study problem

Since Flutter is an SDK, it is best used to develop mobile applications and cannot be incorporated in current applications. For example, you simply could not develop a half-Swift, half-Flutter application. This principle applies to other cross-platform development kits such as React Native and Xamarin. Flutter, however, could import packages and software from other applications as plug-ins.

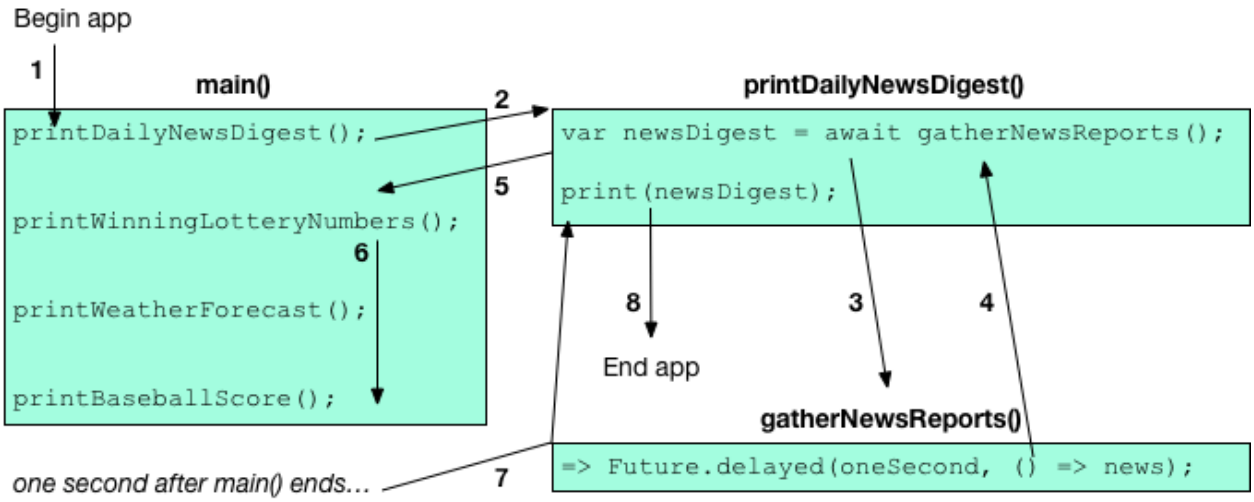
Flutter provides a base project structure to get started with, including a main.dart file, tests directory, and other boilerplate files. The sample application that Flutter provides is a stateless, simple application with a button and some text. You can add widgets, text, and functionality to the sample application to start developing in Dart. Flutter cannot run on its own, because it is a development package. One unique feature of Flutter is the "hot reload", which allows you to load an updated version of an app or widget onto the simulator (e.g. iPhone X emulator) very quickly, rather than restarting the entire emulator just to test a small change in the code.

b. <u>Asynchronous parts of software</u>

Flutter does offer asynchronous programming, with a 'Future<T>' object, which represents an asynchronous operation that produces a result of type T. When a function that returns a future:

1. It queues up work to be done and returns an uncompleted Future object.
2. Later, when the operation is finished, the Future object completes with a value or with an error.

To use a 'Future<T>' object, the developer could use async and await or use the Future API. The async and await keywords are part of the Dart language's asynchronous programming support. They allow you to write asynchronous code that looks like synchronous code and doesn't use the Future API. An async function is one that has the async keyword before its body. The await keyword works only in async functions.



The Future API is an older method before async and await were introduced in Dart 1.9. Like in JavaScript, this API works by using the then() method to register a callback. This callback fires when the Future completes.

c. More on Dart

While developers build Flutter apps in object-oriented style, Dart is a multi-paradigm language supporting scripting, object-oriented, imperative, reflective, functional programming. Developers might find similarities with JavaScript and Swift.

```
class ChatScreenState extends State<ChatScreen> with TickerProviderStateMixin {
  //TickerProviderStateMixin ensures vsync
  final List<ChatMessage> messages = <ChatMessage>[];
  final TextEditingController _textController = new TextEditingController();
  @override
  void dispose() {
    // dispose any unused animation because
    // u only need the animation for the latest text
    for (ChatMessage message in messages) message.animationController.dispose();
    super.dispose();
  }

  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text(
          "Friendlychat",
        ), // Text
        elevation: Theme.of(context).platform == TargetPlatform.iOS ? 0.0 : 4.0,
      ), // AppBar
      body: Container(
        child: new Column(children: <Widget>[
          //flexible widgets just expand all along any axis
          new Flexible(
              //displaying all the messages as a listview
              child: new ListView.builder(
            padding: new EdgeInsets.all(8.0),
            reverse:
                true, //reverse means that the latest item is actually in the botto
            itemBuilder: (_, int index) => messages[index],
            itemCount: messages.length,
          )), // ListView.builder, Flexible
          // just a gap
          new Divider(height: 1.0),
          //the bottom will be the text box with the send button
          new Container(
            decoration: new BoxDecoration(
                color: new Color.fromARGB(100, 239, 237, 236)), // BoxDecoration
            child: _buildTextComposer(),
```

This code example demonstrates a class called ChatScreenState with inheritance features. A developer can use strong type declarations or weak type using ‘var’ keyword.

In addition to object-oriented features, this code also shows elements of scripting languages in the ‘build’ method, where a Scaffold object is returned with nested objects inside of it. Alternatively, the developer can choose to embrace object-oriented programming by declaring these objects serially.

Two Defects

Flutter's contribution guide:

https://github.com/flutter/flutter/blob/master/CONTRIBUTING.md

1. VM crash

Issue link: https://github.com/flutter/flutter/issues/29379

Description: VM crashes while loading and resizing images in Flutter Android app (Multiple mutators entering an isolate / Dart VM is shutting down).

Potential solution: It seems to be a flutter engine problem, specifically with memory issues. The issue could be tied to a bad pointer not freed after reference/usage.

2. iOS compatibility

Issue link: https://github.com/flutter/flutter/issues/29437

https://github.com/flutter/flutter/issues/24140

Description: Various compatibility issues with Running Flutter apps on physical iOS device, especially older iPhone models. Debugging on Android Studio with X simulator seems to be working fine. However, on iPhone 6s - iOS 12.1.4, applications seem to have memory leaks symptoms for consuming too much memory and therefore killed by iOS native operating system.