Flutter builds APKs through a long set of steps that ultimately defer to Gradle. The logic lives in many places: the Flutter framework, the Flutter app template, and a mix of both Dart and Gradle scripts. This doc is a high level description of the overall flow, which can be hard to follow given how many different places at once it lives and how asynchronous Gradle builds are.

- The Flutter tool fetches any plugin dependencies of the Flutter app from pub and downloads it to the pub cache. This step follows normal `pub get` behavior and version solving.
- The Flutter tool writes project metadata to some local temporary files. This includes things like the location of the local Flutter SDK, and paths to all the newly downloaded plugins from pub on disk.
- The Flutter tool invokes Gradle to build the APK.
- Gradle reads the app's `build.gradle`, which applies the Flutter framework's `flutter.gradle` plugin.
- `flutter.gradle` adds the Flutter engine and all of its support library dependencies as a `jar` dependency to the Flutter app.
- `flutter.gradle` adds all of the Flutter plugins of an app as "subproject" Gradle dependencies (usually).
- Gradle compiles the Flutter app and all of its subprojects (Flutter plugins) into a root APK.

Those steps are deliberately listed as bullet points and not a numbered list. Gradle invokes a lot of these steps at seemingly random and parallel times. Time is relative.

Plugins are compiled as subprojects (usually)

Flutter plugins are typically included in Flutter apps as source code and compiled by Gradle, the same as the rest of the code in the Flutter app.

This is done by using the Gradle concept of "subprojects", a way of structuring build tasks so that they're marked as being dependent on each other. All Flutter apps are built with one subproject for the actual Android code of the app and other subprojects for the Android code of any plugins.

For example, let's assume that there's a Flutter app that uses two plugins: `foo` and `bar`. Let's also assume `foo` depends on `baz`, so `baz` is included as part of the build at runtime. Running `./gradlew projects` in the app's `android` directory (after first running `flutter build apk` to give the Flutter tool's Dart code an opportunity to set up the build) would reveal the following project structure:

```
$ ./gradlew projects

> Task :projects

------------------------------------------------------------
Root project
```

```
------------------------------------------------------------

Root project 'android'
+--- Project ':app'
+--- Project ':bar'
+--- Project ':baz''
\--- Project ':foo'
```

The how

- The Flutter tool's Dart code writes a `.flutter-plugins` file containing a list of all the plugins used by the app and their location on disk.
- `flutter.gradle` reads from this file and actually adds the plugins as Gradle subproject dependencies of the app.
- This also depends on the Flutter app having a block of code in its template that actually references and builds these subprojects once they're defined, see "hello world" as an example of the app's Gradle script.

Special Considerations

Including plugin dependencies as source code and then compiling them is not something Gradle generally expects apps to be doing. In the past we've hit several issues caused by Gradle tooling around dependency management being built for typical Android patterns (including dependencies as `.aars` or `.jars`) and breaking down for the Flutter use case.

For example:

- Plugins have their own `build.gradle` scripts that may conflict with the app and other plugins in unexpected ways. How Gradle handles these conflicts is situational and sometimes undefined behavior.
- If a plugin has a problem in its Android code, the entire user's app will fail to compile with a stack trace that may or may not really point to the plugin's code on disk.
- If the plugin's code conflicts with the app's Android source code (for example, the plugin has a conflicting transitive dependancy), Gradle will generally fail to compile with difficult to translate stack traces.
- These subprojects are added in an `afterEvaluate` hook in `flutter.gradle`. Generally adding any logic in `afterEvaluate` blocks is discouraged because it runs after so much of Gradle's standard pipeline.

Sometimes plugins are compiled as AARs

Because of the problems with compiling the plugins as subprojects, the Flutter team explored switching the plugins to instead be compiled first as standalone AARs and then included as binary dependencies as is more typical for Android dependencies and expected by Gradle. This effort ended up being blocked, but

is still occasionally automatically used as a fallback attempt when one of the typical problems with building as subprojects is detected as failing a build today.

The basic AAR flow goes like this:

- The Flutter tool fetches any plugin dependencies of the Flutter app from pub and downloads it to the pub cache. This step follows normal `pub get` behavior and version solving.
- The Flutter tool writes project metadata to some local temporary files. This includes things like the location of the local Flutter SDK, and paths to all the newly downloaded plugins from pub on disk.
- The Flutter tool invokes Gradle to build the APK.
- Gradle reads the app's `build.gradle`, which applies the Flutter framework's `flutter.gradle` plugin.
- `flutter.gradle` adds the Flutter engine and all of its support library dependencies as a `jar` dependency to the Flutter app.
- `flutter.gradle` compiles all of the plugins individually into `aars`, and adds all of the Flutter plugins as `aar` dependencies to the Flutter app.
- Gradle compiles the Flutter app and all of its binary dependencies into a root APK.

Why this isn't the standard

There were a couple blocking issues that kept us from following this flow in all cases:

- The implementation of this depends on `TaskInternal.execute()`, a Gradle API that's been removed with no replacement in Gradle 5.
- This style of build is extremely slow compared to the previous build flow.
- The plugins currently in the Flutter ecosystem were written under the previous build system and in many cases depend on the transitive dependency bleed to work. For example, many plugins built this way will fail to compile individually because they're missing an explicit dependency on `androidx.annotation`. There are also other plugins that deliberately expose their own transitive `jar` dependencies in a way where they're automatically included in the app at build time by the subproject system, but apps including them as AARs will fail with "Class not found errors" unless the transitive plugin dependencies from the plugin are manually added into the apps' Gradle dependencies as well.

When this is still used

We still fall back on this as a silent retry if we detect that the failure is caused by one of the likely error cases from the subproject structure: specifically, if we detect that the build has failed because of a likely AndroidX incompatibility issue. Jetifier is one of the Gradle tools that doesn't function normally with the subproject structure, so building the plugins as AARs can sometimes fix errors in that class.