Integrate a Flutter module into your Android project

Contents

Flutter can be embedded into your existing Android application piecemeal, as a source code Gradle subproject or as AARs.

The integration flow can be done using the Android Studio IDE with the Flutter plugin or manually.

Warning: Your existing Android app may support architectures such as `mips` or `x86`. Flutter currently only supports building ahead-of-time (AOT) compiled libraries for `x86_64`, `armeabi-v7a` and `arm64-v8a`.

Consider using the `abiFilters` Android Gradle Plugin API to limit the supported architectures in your APK. Doing this avoids a missing `libflutter.so` runtime crash, for example:

```
android {
  //...
  defaultConfig {
    ndk {
      // Filter for architectures supported by Flutter.
      abiFilters 'armeabi-v7a', 'arm64-v8a', 'x86_64'
    }
  }
}
```

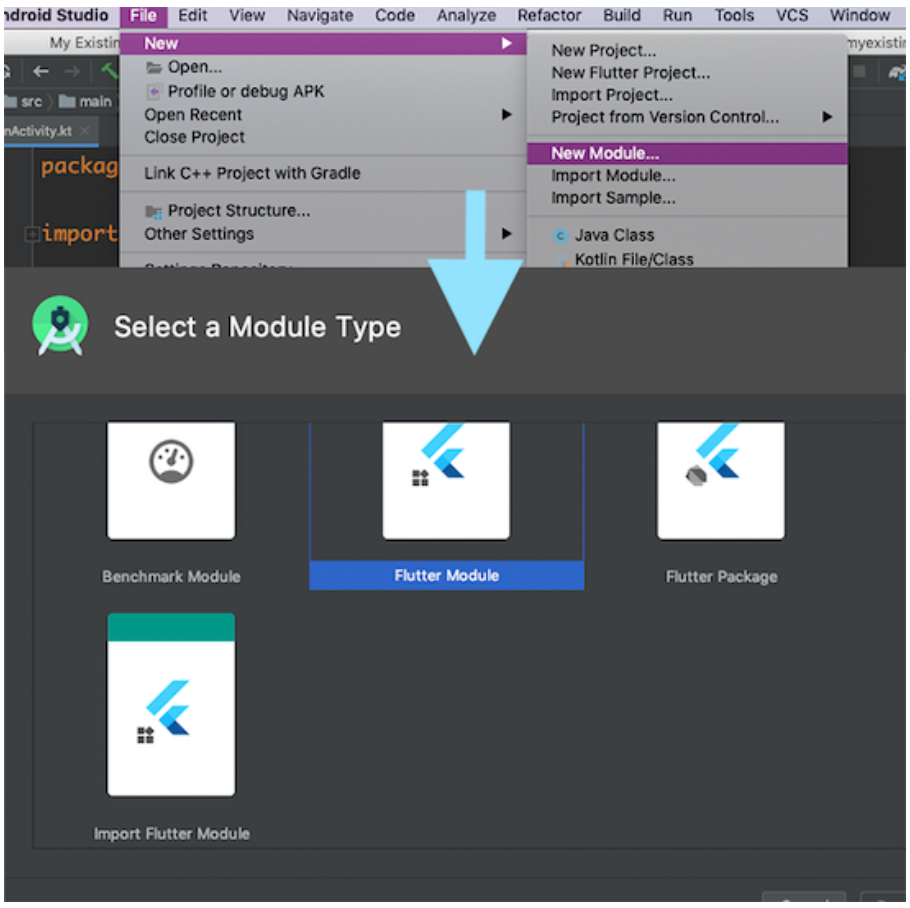
The Flutter engine has an `x86` and `x86_64` version. When using an emulator in debug Just-In-Time (JIT) mode, the Flutter module still runs correctly.

Using Android Studio

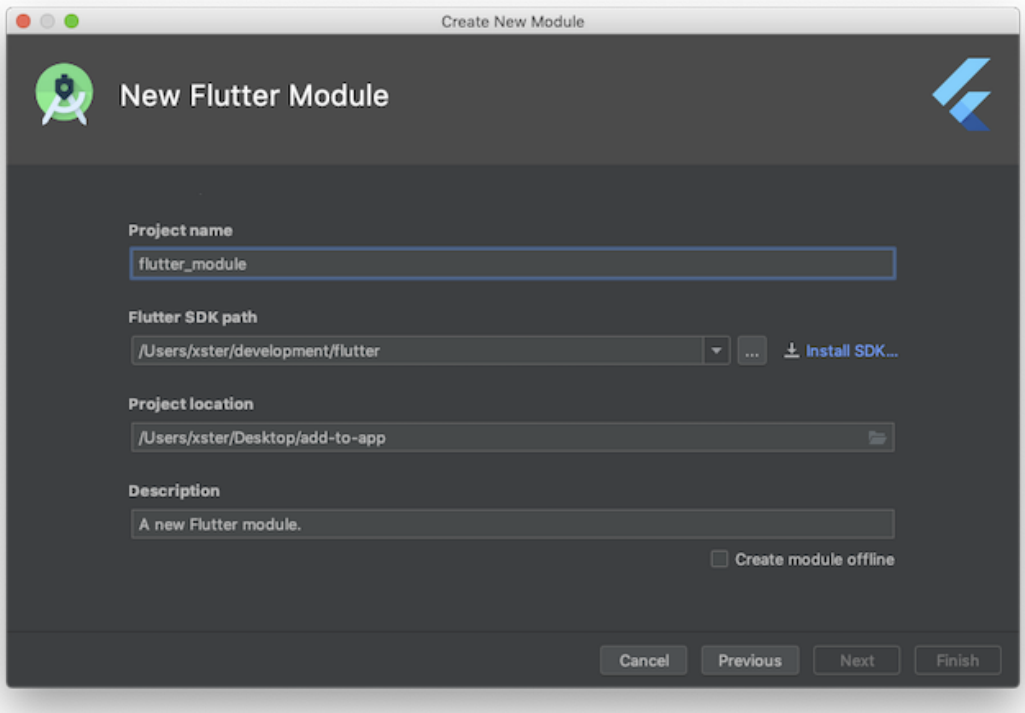
The Android Studio IDE is a convenient way of integrating your Flutter module automatically. With Android Studio, you can co-edit both your Android code and your Flutter code in the same project. You can also continue to use your normal IntelliJ Flutter plugin functionalities such as Dart code completion, hot reload, and widget inspector.

Add-to-app flows with Android Studio are only supported on Android Studio 3.6 with version 42+ of the Flutter IntelliJ plugin. The Android Studio integration also only supports integrating using a source code Gradle subproject, rather than using AARs. See belo for more details on the distinction.

Using the `File > New > New Module...` menu in Android Studio in your existing Android project, you can either create a new Flu module to integrate, or select an existing Flutter module that was created previously.



If you create a new module, you can use a wizard to select the module name, location, and so on.



The Android Studio plugin automatically configures your Android project to add your Flutter module as a dependency, and your ap ready to build.

Note: To see the changes that were automatically made to your Android project by the IDE plugin, consider using source control for your Android project before performing any steps. A local diff shows the changes.

Tip: By default, your project's Project pane is probably showing the 'Android' view. If you can't see your new Flutter files in the Project pane, ensure that your Project pane is set to display 'Project Files', which shows all files without filtering.

Your app now includes the Flutter module as a dependency. You can jump to the API usage documentations to follow the next ste

Manual integration

To integrate a Flutter module with an existing Android app manually, without using Flutter's Android Studio plugin, follow these st

Create a Flutter module

Let's assume that you have an existing Android app at `some/path/MyApp`, and that you want your Flutter project as a sibling:

```
$ cd some/path/
$ flutter create -t module --org com.example my_flutter
```

This creates a `some/path/my_flutter/` Flutter module project with some Dart code to get you started and a `.android/` hidden subfolder. The `.android` folder contains an Android project that can both help you run a barebones standalone version of your Fl module via `flutter run` and it's also a wrapper that helps bootstrap the Flutter module an embeddable Android library.

> **Note:** Add custom Android code to your own existing application's project or a plugin, not to the module in `.android/`. Changes made in your module's `.android/` directory will not appear in your existing Android project using the module.
>
> Do not source control the `.android/` directory since it's autogenerated. Before building the module on a new machine, run `flutter pub get` in the `my_flutter` directory first to regenerate the `.android/` directory before building the Android project using the Flutter module.

Java 8 requirement

The Flutter Android engine uses Java 8 features.

Before attempting to connect your Flutter module project to your host Android app, ensure that your host Android app declares th following source compatibility within your app's `build.gradle` file, under the `android { }` block, such as:

```
android {
  //...
  compileOptions {
    sourceCompatibility 1.8
    targetCompatibility 1.8
  }
}
```

Add the Flutter module as a dependency

Next, add the Flutter module as a dependency of your existing app in Gradle. There are two ways to achieve this. The AAR mechanism creates generic Android AARs as intermediaries that packages your Flutter module. This is good when your downstre app builders don't want to have the Flutter SDK installed. But, it adds one more build step if you build frequently.

The source code subproject mechanism is a convenient one-click build process, but requires the Flutter SDK. This is the mechani used by the Android Studio IDE plugin.

Option A - Depend on the Android Archive (AAR)

This option packages your Flutter library as a generic local Maven repository composed of AARs and POMs artifacts. This option allows your team to build the host app without installing the Flutter SDK. You can then distribute the artifacts from a local or rem repository.

Let's assume you built a Flutter module at `some/path/my_flutter`, and then run:

```
$ cd some/path/my_flutter
$ flutter build aar
```

Then, follow the on-screen instructions to integrate.

More specifically, this command creates (by default all debug/profile/release modes) a local repository, with the following files:

```
build/host/outputs/repo
└── com
    └── example
        └── my_flutter
            ├── flutter_release
            │   ├── 1.0
            │   │   ├── flutter_release-1.0.aar
            │   │   ├── flutter_release-1.0.aar.md5
            │   │   ├── flutter_release-1.0.aar.sha1
            │   │   ├── flutter_release-1.0.pom
            │   │   ├── flutter_release-1.0.pom.md5
            │   │   └── flutter_release-1.0.pom.sha1
            │   ├── maven-metadata.xml
            │   ├── maven-metadata.xml.md5
            │   └── maven-metadata.xml.sha1
            ├── flutter_profile
            │   ├── ...
            └── flutter_debug
                └── ...
```

To depend on the AAR, the host app must be able to find these files.

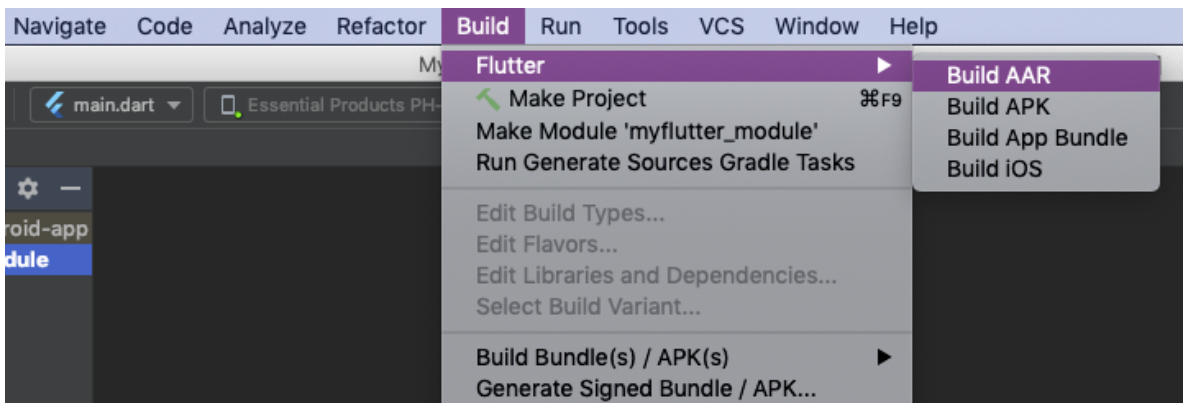
To do that, edit `app/build.gradle` in your host app such as it includes the local repository and the dependency:

```
android {
  // ...
}

repositories {
  maven {
    url 'some/path/my_flutter/build/host/outputs/repo'
    // This is relative to the location of the build.gradle file
    // if using a relative path.
  }
  maven {
    url 'https://storage.googleapis.com/download.flutter.io'
  }
}

dependencies {
  // ...
  debugImplementation 'com.example.flutter_module:flutter_debug:1.0'
  profileImplementation 'com.example.flutter_module:flutter_profile:1.0'
  releaseImplementation 'com.example.flutter_module:flutter_release:1.0'
}
```

Tip: You can also build an AAR for your Flutter module in Android Studio using the `Build > Flutter > Build AAR` menu.



Your app now includes the Flutter module as a dependency. You can follow the next steps in the API usage documentations.

Option B - Depend on the module's source code

This option enables a one-step build for both your Android project and Flutter project. This option is convenient when you work or both parts simultaneously and rapidly iterate, but your team must install the Flutter SDK to build the host app.

Include the Flutter module as a subproject in the host app's `settings.gradle`:

```
include ':app'                                     // assumed existing content
setBinding(new Binding([gradle: this]))                                // new
evaluate(new File(                                                     // new
  settingsDir.parentFile,                                              // new
  'my_flutter/.android/include_flutter.groovy'                         // new
))                                                                     // new
```

Assuming `my_flutter` is a sibling to `MyApp`.

The binding and script evaluation allows the Flutter module to `include` itself (as `:flutter`) and any Flutter plugins used by the module (as `:package_info`, `:video_player`, etc) in the evaluation context of your `settings.gradle`.

Introduce an `implementation` dependency on the Flutter module from your app:

```
dependencies {
  implementation project(':flutter')
}
```

Your app now includes the Flutter module as a dependency. You can follow the next steps in the API usage documentations.