Developing packages & plugins

Contents

The plugin API has been updated and now supports federated plugins that enable separation of different platform implementations. You can also now indicate which platforms a plugin supports, for example web and macOS.

Eventually, the old plugin APIs will be deprecated. In the short term, you will see a warning when the framework detects that you are using an old-style plugin. For information on how to upgrade your plugin, see Supporting the new Android plugins APIs.

Package introduction

Packages enable the creation of modular code that can be shared easily. A minimal package consists of the following:

`pubspec.yaml`
A metadata file that declares the package name, version, author, and so on.

`lib`
The `lib` directory contains the public code in the package, minimally a single `<package-name>.dart` file.

Note: For a list of dos and don’ts when writing an effective plugin, see the Medium article by Mehmet Fidanboylu, Writing a good plugin.

Package types

Packages can contain more than one kind of content:

Dart packages
General packages written in Dart, for example the `path` package. Some of these might contain Flutter specific functionality and thus have a dependency on the Flutter framework, restricting their use to Flutter only, for example the `fluro` package.

Plugin packages
A specialized Dart package that contains an API written in Dart code combined with one or more platform-specific implementations.
Plugin packages can be written for Android (using Kotlin or Java), iOS (using Swift or Objective-C), web, macOS, Windows, or Linux, or any combination thereof.

A concrete example is the `url_launcher` plugin package. To see how to use the `url_launcher` package, and how it was extended to implement support for web, see the Medium article by Harry Terkelsen, How to Write a Flutter Web Plugin, Part 1.

Developing Dart packages

The following instructions explain how to write a Flutter package.

Step 1: Create the package

To create a Flutter package, use the `--template=package` flag with `flutter create`:

```
$ flutter create --template=package hello
```

content_copy

This creates a package project in the `hello` folder with the following content:

LICENSE
A (mostly) empty license text file.

test/hello_test.dart
The unit tests for the package.

hello.iml
A configuration file used by the IntelliJ IDEs.

.gitignore
A hidden file that tells Git which files or folders to ignore in a project.

.metadata
A hidden file used by IDEs to track the properties of the Flutter project.

pubspec.yaml
A yaml file containing metadata that specifies the package's dependencies. Used by the pub tool.

README.md

A starter markdown file that briefly describes the package's purpose.

lib/hello.dart
A starter app containing Dart code for the package.

.idea/modules.xml, **.idea/workspace.xml**
A hidden folder containing configuration files for the IntelliJ IDEs.

CHANGELOG.md
A (mostly) empty markdown file for tracking version changes to the package.

Step 2: Implement the package

For pure Dart packages, simply add the functionality inside the main `lib/<package name>.dart` file, or in several files in the `lib`directory.

To test the package, add unit tests in a `test` directory.

For additional details on how to organize the package contents, see the Dart library package documentation.

Developing plugin packages

If you want to develop a package that calls into platform-specific APIs, you need to develop a plugin package.

The API is connected to the platform-specific implementation(s) using a platform channel.

Federated plugins

Federated plugins are a way of splitting support for different platforms into separate packages. So, a federated plugin can use one package for iOS, another for Android, another for web, and yet another for a car (as an example of an IoT device). Among other benefits, this approach allows a domain expert to extend an existing plugin to work for the platform they know best.

A federated plugin requires the following packages:

app-facing package
The package that plugin users depend on to use the plugin. This package specifies the API used by the Flutter app.

platform package(s)
One or more packages that contain the platform-specific implementation code. The app-facing package calls into these packages—they aren't included into an app, unless they contain platform-specific functionality accessible to the end user.

platform interface package
The package that glues the app-facing packing to the platform package(s). This package declares an interface that any platform package must implement to support the app-facing package. Having a single package that defines this interface ensures that all platform packages implement the same functionality in a uniform way.

Endorsed federated plugin

Ideally, when adding a platform implementation to a federated plugin, you will coordinate with the package author to include your implementation. In this way, the original author *endorses* your implementation.

For example, say you write a `foobar_windows` implementation for the (imaginary) `foobar` plugin. In an endorsed plugin, the original `foobar` author adds your Windows implementation as a dependency in the pubspec for the app-facing package. Then, when a developer includes the `foobar` plugin in their Flutter app, the Windows implementation, as well as the other endorsed implementations, are automatically available to the app.

Non-endorsed federated plugin

If you can't, for whatever reason, get your implementation added by the original plugin author, then your plugin is *not* endorsed. A developer can still use your implementation, but must manually add the plugin to the app's pubspec file. So, the developer must include both the `foobar` dependency *and* the `foobar_windows` dependency in order to achieve full functionality.

For more information on federated plugins, why they are useful, and how they are implemented, see the Medium article by Harry Terkelsen, [How To Write a Flutter Web Plugin, Part 2]().

Specifying a plugin's supported platforms

Plugins can specify the platforms they support by adding keys to the `platforms` map in the `pubspec.yaml` file. For example, the following pubspec file shows the `flutter:` map for the `hello` plugin, which supports only iOS and Android:

content_copy

```
flutter:
  plugin:
    platforms:
      android:
        package: com.example.hello
        pluginClass: HelloPlugin
      ios:
        pluginClass: HelloPlugin

environment:
  sdk: ">=2.1.0 <3.0.0"
  # Flutter versions prior to 1.12 did not support the
  # flutter.plugin.platforms map.
  flutter: ">=1.12.0"
```

When adding plugin implementations for more platforms, the `platforms` map should be updated accordingly. For example, here's the map in the pubspec file for the `hello` plugin, when updated to add support for macOS and web:

content_copy

```
flutter:
  plugin:
    platforms:
      android:
        package: com.example.hello
        pluginClass: HelloPlugin
      ios:
        pluginClass: HelloPlugin
      macos:
        pluginClass: HelloPlugin
      web:
        pluginClass: HelloPlugin
        fileName: hello_web.dart

environment:
  sdk: ">=2.1.0 <3.0.0"
  # Flutter versions prior to 1.12 did not support the
  # flutter.plugin.platforms map.
  flutter: ">=1.12.0"
```

Step 1: Create the package

To create a plugin package, use the `--template=plugin` flag with `flutter create`.

As of Flutter 1.20.0, Use the `--platforms=` option followed by a comma separated list to specify the platforms that the plugin supports. Available platforms are: `android`, `ios`, `web`, `linux`, `macos`, and `windows`. If no platforms are specified, the resulting project doesn't support any platforms.

Use the `--org` option to specify your organization, using reverse domain name notation. This value is used in various package and bundle identifiers in the generated plugin code.

Use the `-a` option to specify the language for android or the `-i` option to specify the language for ios. Please choose **one** of the following:

content_copy

```
$ flutter create --org com.example --template=plugin --platforms=android,ios -a
kotlin hello
```

content_copy

```
$ flutter create --org com.example --template=plugin --platforms=android,ios -a
java hello
```

content_copy

```
$ flutter create --org com.example --template=plugin --platforms=android,ios -i
objc hello
```

```
$ flutter create --org com.example --template=plugin --platforms=android,ios -i
swift hello
```
content_copy

This creates a plugin project in the `hello` folder with the following specialized content:

`lib/hello.dart`
The Dart API for the plugin.

`android/src/main/java/com/example/hello/HelloPlugin.kt`
The Android platform-specific implementation of the plugin API in Kotlin.

`ios/Classes/HelloPlugin.m`
The iOS-platform specific implementation of the plugin API in Objective-C.

`example/`
A Flutter app that depends on the plugin, and illustrates how to use it.

By default, the plugin project uses Swift for iOS code and Kotlin for Android code. If you prefer Objective-C or Java, you can specify the iOS language using `-i` and the Android language using `-a`. For example:

```
$ flutter create --template=plugin --platforms=android,ios -i objc hello
```
content_copy

```
$ flutter create --template=plugin --platforms=android,ios -a java hello
```
content_copy

Step 2: Implement the package

As a plugin package contains code for several platforms written in several programming languages, some specific steps are needed to ensure a smooth experience.

Step 2a: Define the package API (.dart)

The API of the plugin package is defined in Dart code. Open the main `hello/` folder in your favorite Flutter editor. Locate the file `lib/hello.dart`.

Step 2b: Add Android platform code (.kt/.java)

We recommend you edit the Android code using Android Studio.

Then use the following steps:

1. Launch Android Studio.
2. Select **Open an existing Android Studio Project** in the **Welcome to Android Studio** dialog, or select **File > Open** from the menu, and select the `hello/example/android/build.gradle` file.
3. In the **Gradle Sync** dialog, select **OK**.
4. In the **Android Gradle Plugin Update** dialog, select **Don't remind me again for this project**.

The Android platform code of your plugin is located in `hello/java/com.example.hello/HelloPlugin`.

You can run the example app from Android Studio by pressing the run (▶) button.

Step 2c: Add iOS platform code (.swift/.h+.m)

We recommend you edit the iOS code using Xcode.

Before editing the iOS platform code in Xcode, first make sure that the code has been built at least once (in other words, run the example app from your IDE/editor, or in a terminal execute `cd hello/example; flutter build ios --no-codesign`).

Then use the following steps:

1. Launch Xcode.
2. Select **File > Open**, and select the `hello/example/ios/Runner.xcworkspace` file.

The iOS platform code for your plugin is located in `Pods/Development Pods/hello/../../example/ios/.symlinks/plugins/hello/ios/Classes` in the Project Navigator.

You can run the example app by pressing the run (▶) button.

Step 2d: Connect the API and the platform code

Finally, you need to connect the API written in Dart code with the platform-specific implementations. This is done using a platform channel, or through the interfaces defined in a platform interface package.

Add support for platforms in an existing plugin project

To add support for specific platforms to an existing plugin project, run `flutter create` with the `--template=plugin` flag again in the project directory. For example, to add web support in an existing plugin, run:

```
$ flutter create --template=plugin --platforms=web .
```
content_copy

If this command displays a message about updating the `pubspec.yaml` file, follow the provided instructions.

Testing your plugin

We encourage you test your plugin with automated tests, to ensure that functionality does not regress as you make changes to your code. For more information, see Testing your plugin, a section in Supporting the new Android plugins APIs.

Adding documentation

It is recommended practice to add the following documentation to all packages:

1. A `README.md` file that introduces the package
2. A `CHANGELOG.md` file that documents changes in each version
3. A `LICENSE` file containing the terms under which the package is licensed
4. API documentation for all public APIs (see below for details)

API documentation

When you publish a package, API documentation is automatically generated and published to pub.dev/documentation. For example, see the docs for `device_info`.

If you wish to generate API documentation locally on your development machine, use the following commands:

1. Change directory to the location of your package:

   ```
   cd ~/dev/mypackage
   ```
   content_copy

2. Tell the documentation tool where the Flutter SDK is located (change the following commands to reflect where you placed it):

   ```
   export FLUTTER_ROOT=~/dev/flutter  # on macOS or Linux
   set FLUTTER_ROOT=~/dev/flutter     # on Windows
   ```
   content_copy

3. Run the `dartdoc` tool (included as part of the Flutter SDK), as follows:

   ```
   $FLUTTER_ROOT/bin/cache/dart-sdk/bin/dartdoc   # on macOS or Linux
   %FLUTTER_ROOT%\bin\cache\dart-sdk\bin\dartdoc  # on Windows
   ```
   content_copy

For tips on how to write API documentation, see Effective Dart Documentation.

Adding licenses to the LICENSE file

Individual licenses inside each LICENSE file should be separated by 80 hyphens on their own on a line.

If a LICENSE file contains more than one component license, then each component license must start with the names of the packages to which the component license applies, with each package name on its own line, and the list of package names separated from the actual license text by a blank line. (The packages need not match the names of the pub package. For example, a package might itself contain code from multiple third-party sources, and might need to include a license for each one.)

The following example shows a well-organized license file:

content_copy
```
package_1

<some license text>

--------------------------------------------------------------------------------
-
package_2

<some license text>
```

Here is another example of a well-organized license file:

content_copy
```
package_1

<some license text>

--------------------------------------------------------------------------------
-
package_1
package_2

<some license text>
```

Here is an example of a poorly-organized license file:

content_copy
```
<some license text>

--------------------------------------------------------------------------------
-
<some license text>
```

Another example of a poorly-organized license file:

content_copy
```
package_1

<some license text>
--------------------------------------------------------------------------------
-
<some license text>
```

Publishing your package

Tip: Have you noticed that some of the packages and plugins on pub.dev are designated as Flutter Favorites? These are the packages published by verified developers and are identified as the packages and plugins you should first consider using when writing your app. To learn more, see the Flutter Favorites program.

Once you have implemented a package, you can publish it on pub.dev, so that other developers can easily use it.

Prior to publishing, make sure to review the `pubspec.yaml`, `README.md`, and `CHANGELOG.md` files to make sure their content is complete and correct. Also, to improve the quality and usability of your package (and to make it more likely to achieve the status of a Flutter Favorite), consider including the following items:

- Diverse code usage examples
- Screenshots, animated gifs, or videos
- A link to the corresponding code repository

Next, run the publish command in `dry-run` mode to see if everything passes analysis:

```
$ flutter pub publish --dry-run
```
content_copy

The next step is publishing to pub.dev, but be sure that you are ready because publishing is forever:

```
$ flutter pub publish
```
content_copy

For more details on publishing, see the publishing docs on dart.dev.

Handling package interdependencies

If you are developing a package `hello` that depends on the Dart API exposed by another package, you need to add that package to the `dependencies` section of your `pubspec.yaml` file. The code below makes the Dart API of the `url_launcher` plugin available to `hello`:

```
dependencies:
  url_launcher: ^5.0.0
```
content_copy

You can now `import 'package:url_launcher/url_launcher.dart'` and `launch(someUrl)` in the Dart code of `hello`.

This is no different from how you include packages in Flutter apps or any other Dart project.

But if `hello` happens to be a *plugin* package whose platform-specific code needs access to the platform-specific APIs exposed by `url_launcher`, you also need to add suitable dependency declarations to your platform-specific build files, as shown below.

Android

The following example sets a dependency for `url_launcher` in `hello/android/build.gradle`:

```
android {
    // lines skipped
    dependencies {
        compileOnly rootProject.findProject(":url_launcher")
    }
}
```
content_copy

You can now `import io.flutter.plugins.urllauncher.UrlLauncherPlugin` and access the `UrlLauncherPlugin` class in the source code at `hello/android/src`.

For more information on `build.gradle` files, see the Gradle Documentation on build scripts.

iOS

The following example sets a dependency for `url_launcher` in `hello/ios/hello.podspec`:

```
Pod::Spec.new do |s|
  # lines skipped
  s.dependency 'url_launcher'
```
content_copy

You can now `#import "UrlLauncherPlugin.h"` and access the `UrlLauncherPlugin` class in the source code at `hello/ios/Classes`.

For additional details on `.podspec` files, see the CocoaPods Documentation on them.

Web

All web dependencies are handled by the `pubspec.yaml` file like any other Dart package.