If you haven't already read it, start with the repository structure overview to familiarize yourself with the types of packages discussed below.

Types of Tests

Because of the complexities of having native code, plugins have many more types of tests than most other Flutter packages. A plugin will generally have some combination of the following: - **Dart unit tests**. All plugin packages should have these, with the exception of a federated implementation package that contains only native code (i.e., one that only implements a shared method channel). In a plugin, Dart unit tests would cover: - The Dart code in a monolithic plugin (generally using a mock method channel). - The Dart code in an app-facing plugin package (using a mock implementation of the platform interface). - The Dart code in a platform interface package that contains a shared method channel implementation (using a mock method channel). - The Dart code of an implementation package, if any. Some federated implementations are pure Dart, and would be largely tested via unit tests, and some have Dart code binding to a package-internal method channel.

These should live in `test/` - **Integration tests**. These use the `integration_test` package. Unlike Dart unit tests, integration tests run in the context of a Flutter application (the `example/` app), and can therefore load and exercise native plugin code. Almost all plugin packages other than the platform interface package should have these. The only exceptions would be: - Plugins that need native UI tests (see below) - Plugins that have pure Dart implementations which can be comprehensively tested with Dart unit tests.

These should live in `example/integration_tests/` - **Native unit tests**. Most plugins that have native code should have unit tests for that native code. (Currently, many do not; fixing that is currently a priority for plugins work.) They are written as: - Android: **JUnit** - These should live in `android/src/test/` - iOS: **XCTest** - These should live in `example/ios/RunnerTests/` (**Note**: These are in the example directory, not the main package directory, because they are run via the example app's project) - macOS: **XCTest** - These should live in `example/macos/RunnerTests/` (**Note**: These are in the example directory, not the main package directory, because they are run via the example app's project) - Linux: **Google Test** - These should live in `linux/test/`, and be named `*_test.cc`. - Windows: **Google Test** - These should live in `windows/test/`, and be named `*_test.cpp`. - **Native UI tests**. Some plugins show native UI that the test must interact with (e.g., `image_picker`). For these normal integration tests won't work, as there is not way to drive the native UI from Dart. They are written as: - Android: **Espresso**, via the `espresso` plugin - These should live in `example/android/app/src/androidTest/` - iOS: **XCUITest** - These should live in `example/ios/RunnerUITests/` - macOS: **XCUITest** - These should live in `example/macos/RunnerUITests/` - Windows and Linux: **TBD**, see #70233 (Windows) and #70235 (Linux)

Running Tests

Dart unit tests

These can be run like any other Flutter unit tests, either from your preferred Flutter IDE, or using `flutter test`.

Unit tests are located in the `test` directory of the plugin package.

Integration tests

To run the integration tests using Flutter driver (while in a plugin directory):

```
cd example
flutter drive --driver test_driver/integration_test.dart --target integration_test/<name_of_
```

(optionally including a `-d <device>` flag to select the target device), or use the plugin tools `drive-examples` command from the root of the repository:

```
dart run ./script/tool/bin/flutter_plugin_tools.dart drive-examples --plugins=<name_of_plugi
```

To run integration tests as instrumentation tests on a local Android device:

```
cd example
flutter build apk
cd android && ./gradlew -Ptarget=$(pwd)/../test_driver/<name_of_plugin>_test.dart app:connec
```

Native tests

From a terminal (while at the repository root), use the plugin tools `native-test` command. Examples:

```
# Unit and integration (UI) tests, multiple platforms
dart run ./script/tool/bin/flutter_plugin_tools.dart native-test --android --ios --plugins=<
# iOS, integration tests only
dart run ./script/tool/bin/flutter_plugin_tools.dart native-test --ios --no-unit --plugins=<
# Android, unit tests only
dart run ./script/tool/bin/flutter_plugin_tools.dart native-test --android --no-integration
```

It's also possible to run them from native IDEs:

JUnit Run from Android Studio once the example app is opened as an Android project

XCTest and XCUITest Run from Xcode once the example app is opened as an Xcode project.

Google Test For Windows plugins, Visual Studio should auto-detect the tests and allow running them as usual.

Web Tests

Most web tests are written as Integration Tests because they need a web browser (like Chrome) to run. Web integration tests are located in the `example` directory of the `<plugin_name_web>` package.

To run web integration tests:

1. Check what version of Chrome is running on the machine you're running tests on.
2. Download and install the ChromeDriver for that version from here:
   - https://chromedriver.chromium.org/downloads
3. Start ChromeDriver with:

```
chromedriver --port=4444
```

4. Run tests:
   - **All:** from the root `plugins` directory, run:

   ```
   dart ./script/tool/bin/flutter_plugin_tools.dart drive-examples --plugins=<plugin_name>
   ```

   - **One:** `cd` into the `example` directory of the package, then run:

   ```
   flutter drive -d web-server --web-port 7357 --browser-name chrome --driver test_driver/
   ```

All web packages contain a standard `test` directory in the root of the package that can be run with `flutter test`. In most cases, the test there will direct users to the Integration Tests that live in the `example` directory. In some cases (like `file_selector_web`), the directory contains actual Unit Tests that are not web-specific and can run in the Dart VM.

Mocks Some packages (like `google_maps_flutter_web`) contain `.mocks.dart` files alongside the test files that use them.

Mock files are generated by `package:mockito`. The contents of mock files can change with how the mocks are used within the tests, in addition to actual changes in the APIs they're mocking.

Mock files must be updated by running the following command:

```
flutter pub run build_runner build
```

Adding tests

A PR changing a plugin should add tests. Which type(s) of tests you should add will depend on what exactly you are changing. See below for some high-level guidance, but if you're not sure please ask in your PR or in `#hackers-ecosystem` on Discord. Hopefully the scaffolding to run the necessary kinds of tests are already in place for that plugin, in which case you just add tests to the existing files in the locations referenced above. If not, see below for more information about adding test scaffolding.

FAQ: Do I need to add tests even if the part of the code I'm changing isn't already tested?

Yes. Much of the plugin code predates the current strict policy about testing, and as a result the coverage is not as complete as it should be. The goal of having a strict policy going forward is to ensure that the situation improves.

What types of tests to add

The type(s) of tests to add depends on the details of the PR, and there is not always one right answer. Some high level principles to keep in mind: - Your tests should cover all of the new code you've added whenever possible. For instance, if your change has an error path you should test both the normal path and the error path. - Unit tests are more reliable, faster to run, and more precise at isolating bugs than integration tests, so it is generally good to have unit testing of any non-trivial logic. - New functionality should generally include integration tests that validate that the feature works end to end. (In some cases asserting that the native code has done the right thing is impractical in a full integration test, however.)

Some specific guidelines about different test types: - **Dart unit tests**: If you are changing Dart code, you should almost always have Dart unit tests as well. They are fast to run, and generally easy to write and maintain. Even if your Dart changes are just wiring a new parameter through an app-facing package to the platform interface, or a platform interface implementation to a method channel, Dart unit tests can easily validate that they are being passed through correctly at each step. - Unless your change has no native code changes, remember that Dart unit tests **are not sufficient by themselves**. No native code is run as part of Dart unit tests. - **Dart integration tests**: These are the only fully end-to-end tests for plugins, so should be written when possible. This is the only kind of test that can actually test the entire flow as it will be run by plugin clients. Some comment cases where they won't work, however: - The test requires interacting with native UI (e.g., pushing a button in a modal dialog before the test can complete). - The test requires inspecting native UI (e.g., changing the properties of a view inside a platform view provided by the native code—although in some cases it may be viable to write a special platform interface just for testing, that can query those properties from Dart via calls to native code). - The test requires specific hardware state. - **Native unit tests**: Native unit tests are the only way to unit-test native code; if your native code is non-trivial, then it should probably have native unit tests. In particular: - It is often much easier to get complete coverage of edge cases in native unit tests than via full integration tests. - If you need to mock out system components (such as hardware state), you'll need native unit tests. - **Native integration tests**: These are usually the most complex to maintain (being both per-platform, and more likely to be flaky than unit tests), so use with caution. However, they are the only way to get end-to-end testing when interacting with native UI.

Some patterns to consider given all of the above: - If Dart integration tests are viable, use that to ensure end-to-end behavior of the overall feature, and add Dart and/or native unit testing to cover specific implementation details. - If the test requires interacting with native UI (e.g., image or file selection): - use native integration tests to validate the overall flow, plus unit testing of specific details, or - use native unit tests that are set up to test the native portion end to end (i.e., calling into the method channel entry point with a synthesized call, and checking the method call response) with the UI APIs stubbed out, plus Dart unit tests that everything is plumbed through from the Dart API surface to the method call, or - use the first to validate basic functionality, and the second for edge cases that can't be easily tested that way. - If the tests require mocking device state (e.g., camera), use native unit tests that are set up to test the native portion end to end (i.e., calling into the method channel entry point with a synthesized call, and checking the method call response) with the UI APIs stubbed out, plus Dart unit tests that everything is plumbed through from the Dart API surface to the method call.

Adding test scaffolding

If a plugin is missing test scaffolding for the type of tests you want to add, you have several options: - If it's simple to enable them, and you are comfortable making the changes, you can enable them as part of your PR. - If it's non-trivial to enable them, but you are comfortable doing it, you can make a new PR that enables them with some minimal test, and ask for that to be reviewed and landed before continuing with your PR. - If you aren't comfortable making the change, reach out via Discord and ask in `#hackers-ecosystem`.

Regardless of the approach you use, please reach out on Discord if a PR that sets up a new kind of test doesn't get reviewed within two weeks. Filling in the gaps in test scaffolding is a priority for the team, so we want to review such PRs quickly whenever possible.

See below for instructions on bringing up test scaffolding in a plugin (*does not yet cover all types of tests*):

Enabling XCTests or XCUITests

1. Open `<path_to_plugin>/example/ios/Runner.xcworkspace` using Xcode. (For macOS, replace `ios` with `macos`.)
2. Create a new "Unit Testing Bundle" or "UI Testing Bundle", depending on the type of test.
3. In the target options window, populate details as following, then click on "Finish".
    - In the "product name" field, type "RunnerTests" or "RunnerUITests", depending on the type of test.
    - In the "Team" field, select "None".

- Set the Organization Identifier to "dev.flutter.plugins".
- In the Language field, select "Objective-C" for iOS, or "Swift" for macOS.
- In the Project field, select the xcodeproj "Runner" (blue color).
- In the Target to be Tested, select xcworkspace "Runner" (white color).
- In the Build Settings tab, remove most of the target-level overrides that are generated by the template. In particular:
  - `CLANG_WARN_QUOTED_INCLUDE_IN_FRAMEWORK_HEADER` is currently incompatible with Flutter.
  - `IPHONEOS_DEPLOYMENT_TARGET` and `TARGETED_DEVICE_FAMILY` may cause issues running tests.
  - The compiler settings (`CLANG_*`, `GCC_*`, and `MTL_*`) shouldn't be needed.

4. For XCTests, edit `example/ios/Podfile` (`example/macos/Podfile` for macOS) to add the following to the `target 'Runner' do` block:

```
target 'RunnerTests' do
  inherit! :search_paths
  pod 'OCMock', '3.5'
end
```

   The `OCMock` line is only necessary if your tests use OCMock.

5. A RunnerTests/RunnerUITests folder should be created and you can start hacking in the added `.m`/`.swift` file.

Enabling Android UI tests

1. Duplicate the `DartIntegrationTests.java` file from another plugin to 'example/android/app/src/androidTest/java/io/flutter/plugins/DartIntegrationTest.java

2. Create a file under `example/android/app/src/androidTest/java/` with a sub-path corresponding to the example app's package identifier from `example/android/app/src/main/AndroidManifest.xml`. The file should be called `FlutterActivityTest.java` (or if the example uses a custom MainActivity as its `android:name` in `AndroidManifest.xml`, `MainActivityTest.java`).

   - For example, if `AndroidManifest.xml` uses `io.flutter.plugins.fooexample` as the package identifier, and `io.flutter.embedding.android.FlutterActivity` as its `android:name`, the file should be `example/android/app/src/androidTest/java/io/flutter`

   The file should look like: "' package io.flutter.plugins.fooexample;

   import androidx.test.rule.ActivityTestRule; import dev.flutter.plugins.integration_test.FlutterTestRunner; import io.flutter.embedding.android.FlutterActivity; import io.flutter.plugins.DartIntegrationTest; import org.junit.Rule; import org.junit.runner.RunWith;

   @DartIntegrationTest @RunWith(FlutterTestRunner.class) public class

FlutterActivityTest { @Rule public ActivityTestRule rule = new ActivityTestRule<>(FlutterActivity.class); } “‘

Note:

- Update the `package` to match the actual package.
- If using a custom `MainActivity`, replace the `FlutterActivity` references with `MainActivity`.

3. Ensure that `example/android/app/build.gradle`'s `defaultConfig` section contains: `testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"`