This page covers additional information that is specific to contributing to flutter/plugins and flutter/packages. If you aren't already familiar with the general guidance on Flutter contribution, start with Tree hygiene.

## Version and CHANGELOG updates

### Version

Any change that needs to be published in order to take effect must update the version in `pubspec.yaml`. There are very few exceptions:

- PRs that only affect tests.
- PRs that only affect unpublished parts of example apps.
- PRs that only affect local development (e.g., changes to ignored lints).
- Breaking change batching (see below).

This is because the packages in flutter/plugins and flutter/packages use a continuous release model rather than a set release cadence. This model gets improvements to the community faster, makes regressions easier to pinpoint, and simplifies the release process.

### CHANGELOG

All version changes must have an accompanying CHANGELOG update. Even version-exempt changes should generally update CHANGELOG by adding a special `NEXT` entry at the top of `CHANGELOG.md`:

```
## NEXT

* Description of the change.

## X.Y.Z
```

The next release will change `NEXT` to the new version.

This policy exists both to make it easier for maintainers to see a record of all changes to a package, and because some changes (e.g., updates to examples) that do not need to be published may still be of interest to clients of a package.

#### CHANGELOG style

For consistency, all CHANGELOG entries should follow a common style:

- Use `##` for the version line. A version line should have a blank line before and after it.
- Use `*` for individual items.
  - Exception: When editing an existing CHANGELOG that uses `-`, use that instead for local consistency.
- Entries should use present tense indicative for verbs, with "this version" as an implied subject. For example, "Adds cool new feature.", not "Add" or "Added".
- Entries should end with a `.`.
- Breaking changes should be introduced with `**BREAKING CHANGE**:`, or `**BREAKING CHANGES**:` if there is a sub-list of changes.

Example:

```
## 2.0.0

* Adds the ability to fetch data from the future.
* **BREAKING CHANGES**:
  * Removes the deprecated `neverCallThis` method.
  * URLs parameters are now `Uri`s rather than `String`s.

## 1.0.3

* Fixes a crash when the device teleports during a network operation.
```

### FAQ

**Do I need to update the version if I'm just changing the README?** Yes. Most people read the README on pub.dev, not GitHub, so a README change is not very useful unless it is published.

**Do I need to update the version if I'm just changing comments?** If the comment is intended for clients of the package (a `///` comment on anything exported by the package), then yes, since what developers using the package will see in their IDE will come from the published version. If the comment is only useful for someone working on the package (such as an implementation comment within a method, or a comment in a non-exported file), then no.

**What do I do if I there are conflicts to those changes before or during review?** This is common. You can leave the conflicts until you're at the end of the review process to avoid needing to resolve frequently. Including the version changes at the beginning despite the likelihood of conflicts makes it much harder to forget that step, and also means that a reviewer can easily fix it from the GitHub UI just before landing.

### Breaking changes

Because we prefer to minimize breaking changes to packages after 1.0, breaking changes do not always follow the normal one-version-change-per-PR approach. Instead, when making a breaking change consider whether there are other breaking changes that should be made at the same time, and discuss with other regular contributors. If there are multiple changes to make, they can be batched as follows:

1. File an issue tracking all of the breaking changes to include.
2. Prepare PRs for all of the changes, without version changes. Ensure that they have all been reviewed, but do not land them yet. (They should fail CI due to the lack of version change, preventing accidental landing.)
3. Land a PR that temporarily adds `publish_to: none` to the package, with a comment referencing the issue from step 1.
4. Land all of the breaking change PRs. This step should be done in a relatively short window of time (thus the advance preparation above) to avoid having the plugin be unpublishable for longer than necessary. The PRs will pass CI once rebased, since the version check is disabled for unpublishable packages.
5. Once all breaking changes have landed, land a final PR to update the version and remove `publish_to: none`.

## Platform Support

The goal is to have any plugin feature work on every platform on which that feature makes sense; having a lot of features that are only partially implemented across platforms is often confusing and frustrating for developers trying to use those plugins. However, a single developer will not always have the expertise to implement a feature across all supported platforms.

Given that, we welcome PRs that only implement a feature for a subset of platforms, including just one. To set expectations for how such PRs will be handled:

- They will not be fully reviewed until there's an understanding of what support would look like across other platforms, for several reasons:
    - API for features that will be permanently platform-specific may structured in ways that make that limitation more clear, so knowing if other platforms can support it will affect the review process.
    - We want to avoid over-fitting the plugin APIs to a single platform's API. It is often the case that several platforms can implement a feature, but the behavior is different enough across platforms that we need to design the API in a way that covers those variations in a cohesive way. This means that knowing at a high level what the implementation on other platform also affects the review process.
    - Features that are missing implementations on some platforms need to be clearly documented as such in the API, and those comments should clearly express whether those platform are temporarily missing implementations, or are not expected to ever have implementations due to platform limitations.

  The PR author isn't necessarily responsible for answering these questions. These cases will be noted in comments during PR triage; if the PR author, or others in the community, can contribute that information, that will certainly help. If not, that investigation will be part of the review process (in which case the review will likely take longer).

- In some cases, a reviewer may wait on approving a PR for landing until there is a plan in place for landing implementations for other platforms. This is not a hard rule, and will be up to reviewer judgement. This could take a number of forms:
    - Waiting for other PRs from the community that implement the feature for other platforms, and then moving forward with all of them at once.
    - Finding resources within the Flutter team for implementing other platforms before moving forward.
    - Landing the platform interface change and the platform implementations that are done, but waiting on one of the options above before adding the API to the app-facing package's API (allowing developers the option of drilling down to the platform implementation to use the feature on some platforms before it's ready everywhere).

  "Other platforms" might not always include all other platforms. E.g., a feature might be something that's much more likely to be useful on mobile than desktop, or the reverse, and so we might only wait for implementations of that subset. Again, this will be up to reviewer judgement.

In the case where a PR is put on hold for the reasons above, it should be clearly noted in the PR and on the associated issue. We encourage anyone interested in contributing more platform implementation PRs to comment in the bug in such cases.

## Changing federated plugins

Many of the plugins in flutter/plugins are federated. Because a logical plugin consists of multiple packages, and our CI tests using published package dependencies—in order to ensure that every PR can be published without breaking the ecosystem—changes that span multiple packages will need to be done in multiple PRs. This is common when adding new features.

We are investigating ways to streamline this, but currently the process for a multi-package PR is:

1. Create a PR that has all the changes, and update the pubspec.yamls to have path-based dependency overrides. This can be done with the plugin repo tool's `make-deps-path-based` command, targeting any dependency packages changed in the PR. For instance, for an Android-specific change to `video_player` that required platform interface changes as well:

   ```
   $ dart script/tool/bin/flutter_plugin_tools.dart make-deps-path-based --target-dependencies=video_player_platform_interface,video_player_android
   ```

2. Upload that PR and get it reviewed and into a state where the only test failure is the one complaining that you can't publish a package that has dependency overrides.

   - The overall review is completed first to prevent situations where part of the overall change (usually a platform interface change) lands but the rest is never landed, or where the review of the changes in a higher-level part of the change (the app-facing and/or implementation packages) identifies issues in the lower-level parts.

3. Create a new PR that has only the platform interface package changes from the PR above, and ask the reviewers of the main package to review that.

4. Once it has been reviewed (which should be trivial given the review above), landed, and published, update the initial PR to:

   - remove the changes that are part of the other PR,
   - replace the dependency overrides on the platform interface package with a dependency on the published version, and
   - merge in (or rebase to) the latest version of `master`.

5. If there are any dependency overrides remaining, repeat the previous two steps with those packages. There should never be interdependencies between platform implementation packages, so all implementations should be able to be handled in a single new PR.

6. Once there are no dependency overrides, ask the reviewer to land the main PR.

### Breaking changes to plugin platform interfaces

Breaking changes to platform interfaces (any package ending in `_platform_interface`) are strongly discouraged:

- They require each platform implementation to adopt the new version, and the app-facing package can't pick up any of those changes until all implementations have been updated. This could cause situations where bug fixes for one platform are held up on another platform adopting a feature change.
- They require a series of changes to CI to selectively disable testing the latest versions of all packages, then later re-enable them.
- They temporarily lock out unendorsed implementations, until their developers can update.

Because platform interfaces are not expected to be called by clients of the plugins, we favor backward compatibility over having a clean API at that layer.

In order to avoid accidental breaking changes that are missed in review, CI will by default fail for any breaking change to the platform interface. If you believe you need to make a breaking change, and you have discussed it with the reviewer and they agree, you must add the following to the PR description:

```
## Breaking change justification

```

```
<Insert good reason for breaking change here.>
```

The format of the header line must match exactly in order for CI to pass.

### Changing platform interface method parameters

Because platform implementations are subclasses of the platform interface and override its methods, almost *any* change to the parameters of a method is a breaking change. In particular, adding an optional parameter to a platform interface method *is* a breaking change even though it doesn't break callers of the the method.

The least disruptive way to make a parameter change is:

1. Add a new method to the platform interface with the new parameters, whose default implementation calls the existing method. This is not a breaking change.
    1. Strongly consider replacing some or all of the parameters with a parameter object (see `AuthenticationOptions` for `authenticate` as an example), as this will allow adding other parameters in the future without breaking changes or new methods.
    2. Include a comment that the old method is deprecated, and will be removed in a future update. Don't actually mark it as `@Deprecated` since that will create unnecessary warnings, and consumers of a package shouldn't be directly using these methods anyways.
2. Update the implementations to override both methods.
3. Update the app-facing package to call the new method.

At some later point the deprecated method can be cleaned up by:

1. Making a breaking change in the platform interface package to remove any deprecated methods.
2. Updating the app-facing package to allow either the new major version or the previous version of the platform interface (to minimize version lock issues with implementations).
3. Updating all the implementations to use the new major version, removing the override of the deprecated methods.

This cleanup step is low priority though; deprecated methods in a platform interface should be largely harmless, as it's an API with very few direct customers. It's actually preferable to wait quite some time before doing this, as it gives any unendorsed third-party implementations that may exist more time to adapt to the change without disruption.

## Supported Flutter versions

flutter/plugins and flutter/packages has a general policy of supporting the latest `stable` version of Flutter, as well as the current `master`. In practice, many packages often support older versions as well, as minimum version requirements are generally only updated when there is a specific need, such as using a new Flutter feature.

- One exception is new packages which require features that aren't yet available on `stable`. In this rare case, discuss with `#hackers-ecosystem` as it requires adding conditional logic to the CI scripts.

Most CI only runs against those two version. There are only minimal tests of versions older that current `stable` (currently analysis only, for the previous two stable versions of Flutter).

### When to update the required version

- If you know your change requires a feature of Flutter that was added recently, you should update the minimum version accordingly. In particular, if your change was originally written against `master` and had to wait until a change reached `stable`, that means you need to update the minimum version. (Adding a constraint update as soon as the PR fails on `stable` tests is a good way to make sure you don't forget.)
  - If your change requires a newer version of Flutter than is available on `stable`, and it can't wait until that feature reaches `stable`, ask in `#hackers-ecosystem` to see if there's a good solution.
- If your change fails a `legacy-*` CI test, update the minimum version accordingly.

The ecosystem team may also mass-update minimum versions from time to time, to reduce the potential of breaking untested versions (see below). In general, this should use a minimum version somewhat older than the current `stable`, but in some special cases may use the current stable version instead.

### Handling breakage in old versions

Sometimes a change to a package accidentally breaks older version of Flutter; because the full CI test suite does not run anything older than the current stable, such breakage will not necessarily be caught by CI, and will only be discovered via issue reports. When that happens there are several options:

- If the breakage is found quickly enough that retraction is still possible:
  - Retract the latest release.
  - Release a new version that is identical except for updating the minimum Flutter version. This minimizes disruption to developers using old versions of Flutter, with no effect on people using current versions.
- Otherwise, there are several options; discuss with `#hackers-ecosystem` to decide which makes sense for the specific case:
  1. Release a new update that restores compatibility with the old version. This should generally only be done if it's trivial to do so.
  2. Release a revert, then release a new version that reverts the revert but with a constraint update. Consider this option if the number of people affected is likely to be large (e.g., a popular plugin is broken for the previous stable version of Flutter shortly after a stable release). This has essentially the same outcome as retraction, but can be done at any time.
  3. Document the need to pin an old version of the package in the issue, and close it (along with making a `## NEXT` PR for the package that updates its minimum version, to document the correct reality). This will require all affected users to find the issue to learn how to fix it, so should generally only be done if the number of people affected is likely to be small (e.g., when it only affects versions of Flutter that are several stable releases behind).

## Plugin architecture conventions

The plugins in flutter/plugins should follow the following conventions. Note that existing plugins do not currently always follow those conventions because they predate them. PRs that update plugins to follow conventions are welcome.

### Federation

All plugins should be fully federated. This is to ensure that:

- Unofficial federated implementations can be created for any of our plugins. This allows for alternate implementations, as well as supporting unofficial embeddings.

    - Our development processes for federated plugins also helps ensure that we don't accidentally break any such implementations. For instance, our federated safety checks help ensure that we don't make breaking changes to the platform interface without changing the major version.
- We are eating our own dogfood with federation. Federation adds non-trivial complexity to maintaining a plugin, and best practices for federation aren't always obvious. Using federation ourselves means that we are aware of potential issues, and encourages us to create documentation and tooling to improve the developer experience of using federation.

### In-package platform channels

All implementations should use in-package platform channels, for the reasons outlined in the proposal document. Most plugins predate this policy and thus have a legacy "shared method channel" default implementation in the platform interface package, but it should not be used by any first-party implementations.

### Platform exception handling

Having consistent exceptions across platforms is an important part of providing a usable cross-platform API surface. Maintaining consistent errors directly from the native implementations is challenging since there is no easy way to share constants for error code strings across languages, nor any clear reference point for what the possible errors are without reading all of the other implementations.

To ensure that the native errors are a coherent part of the interface, plugins that throw exceptions should follow these best practices:

- The platform interface package should define a plugin-specific `Exception` class, including constants or enums for known error types (e.g., permission failures).
- App-facing packages should `export` that definition, and should include it in relevant API documentation.
- Implementation packages should, in general, have Dart code to catch any `PlatformExceptions` that the native implementation is likely to throw, and convert them to the appropriate interface-defined exceptions.
    - This means that the string constants for error codes returned from native need only be consistent within that platform implementation package, since it won't be passed out of the package.

This means that in general, clients of a plugin should not be expected to see raw `PlatformException`s created from error responses in native code. (This is not a strict rule; failure cases that are so obscure that clients would be unlikely to actually have specific handlers for them don't necessarily need to be converted to a common exception type.)

**Note:** Existing `PlatformException`s are a de-facto part of the API, so updating plugins to follow this practice should be done as a breaking change.