tl;dr

- Changes need tests. The only common case that does not need a test is a documentation fix. If you have to ask "can this be exempted from writing a test," the answer will almost certainly be no if you're changing code. A test is a piece of code or build rule that would otherwise fail without your change.
- Tests must pass. Changing existing tests should be done with extreme caution.
- Regressions should be reverted first and questions asked later. Bringing the tree to green is higher priority.
- A breaking change is one that breaks the tests in the flutter/tests repo, and those need a migration guide.
- Expect that a new patch will be reviewed within two weeks, unless it is fixing a P0 bug in which case it should be reviewed the same day. If it has not been reviewed in that timeframe, reach out on [[Chat]]. Remember that reviewers are human beings with additional professional and personal responsibilities.

Introduction

This page covers how to land a PR and other aspects of writing code for Flutter other than the actual writing of the code. For guidance on designing APIs, documenting, and formatting your code, see the [[Style guide for Flutter repo]] document.

Overview

The general process for submitting code to a Flutter repository is as follows:

1. Fork the repository on GitHub (see the contributing guide for advice on doing this and in general setting up your development environment).

2. If there is not already an issue covering the work you are interested in doing, and if you think the work you are doing is likely to be a non-trivial effort (e.g. you are creating a new widget, or otherwise introducing a new API surface, or intending to make a potentially breaking change, or refactoring a library), then file a new bug to describe the issue you are addressing.

3. If the work you are doing is likely to be a non-trivial effort (see above), then discuss your design on the issue. You may find it useful to create a Google Doc to solicit feedback (use the template at flutter.dev/go/template). You may wish to e-mail the mailing list, or discuss the topic on our [[Chat]] channels. The more buy-in you get from the rest of the team (especially the relevant leads), the easier the rest of the process will be. You can put the label "proposal" on your issue to indicate that you have a design up for discussion in the issue.

4. If the work you are doing affects our privacy surface, such as modifying how we collect analytics, crash logs, or the like, then please reach out to a Googler to discuss your changes (you'll want to start a Google Doc to solicit feedback, use the template at flutter.dev/go/template), who will be happy to loop in one of our engineers who explicitly focus on privacy issues so that they're able to give feedback on the work you plan to do.

5. Create a branch off master on your GitHub fork of the repository, and implement your change. Make sure it is tested (see the next section for details).

   You must follow the guidelines described in the [[Style guide for Flutter repo]]. Files must not have trailing spaces. For the engine repository, C, C++, and Objective-C code should be formatted with `clang-format` before submission (use `buildtools/<OS>/clang/bin/clang-format --style=file -i`).

6. Submit this branch as a PR to the relevant Flutter repository.

7. Get your code reviewed (see below). You should probably reach out to the relevant expert(s) for the areas you touched and ask them to review your PR directly. GitHub sometimes recommends specific reviewers; if you're not sure who to ask, that's probably a good place to start.

8. Make sure your PR passes all the pre-commit tests. Consider running some of the post-commit tests locally (see the devicelab directory). If any tests break, especially the `customer_testing` tests, please see the breaking change policy section below for details on how to proceed.

   The `luci-flutter` test isn't checking your PR, it's letting you know whether the tree itself is passing the tests right now (including post- commit tests). If it is red, help out the team in fixing the tree before continuing. It is everyone's job to keep the tree green.

   If the trees or dashboards are showing any regressions, only fixes that improve the situation are allowed to go in.

9. Once everything is green and you have an LGTM from the owners of the code you are affecting (or someone to whom they have delegated), and an LGTM from any other contributor who left comments, add the "waiting for tree to go green" label. A bot will land the patch when it feels like it.

10. Watch the post-commit tests on the dashboard to make sure everything passes. If anything goes wrong, revert your patch and study the problem. You should aim to be the one to revert your patch. You will be racing everyone else on the team who will also be trying to revert your patch.

Changes that break the flutter/plugins CI should also be reverted before the problem is investigated.

See also: [[What should I work on?]]

Tests

Every change in the flutter/engine, flutter/flutter, flutter/plugins, and flutter/packages repos must be tested, except for PRs that:

- only remove code (no modified or added lines) to remove a feature or remove dead code. (Removing code to fix a bug still needs a test.)
- only affect comments (including documentation).
- only affect code inside the `.github` directory, `.cirrus.yml` or `.ci.yaml` config files.
- only affect `.md` files.
- are generated by automated bots (rollers).
- have an *explicit* exemption (a message from Hixie starting with `test-exempt:`).

A bot will comment on your PR if you need an explicit exemption. Do not land a PR that has had such a message from the bot unless it has an explicit exemption from Hixie! The message tells you how to ask for an exemption (please don't just cc Hixie on the PR, he won't see it).

In particular, the following kinds of PRs are *not* automatically exempt and require an explicit comment from Hixie even though the answer may be obvious: refactors with no semantic change (e.g. null safety migrations), configuration changes in the aforementioned repos, PRs that only affect analysis (fixing lints, turning on lints), PRs that only modify test infrastructure, PRs that manually roll a dependency, PRs that are fixing an existing test.

If a reviewer says a PR should have a test, then it needs a test regardless of the exemptions above.

PRs adding data-driven fixes require tests that fall under the test_fixes directory, but are not recognized by the bot as being tested.

Consider using the code coverage tools to check that all your new code is covered by tests (see [[Test coverage for package:flutter]]).

Using git

Assuming your environment has been configured according to the instructions in [[Setting up the Engine development environment]] or [[Setting up the Framework development environment]], follow these steps to start working on a patch:

- `git fetch upstream`
- `git checkout upstream/master -b name_of_your_branch`
- `flutter update-packages`
- Hack away.
- `git commit -a -m "<your informative commit message>"`
- `git push origin name_of_your_branch`

GitHub provides you with a link for submitting the pull request in the message output by `git push`.

Note that `git pull` will often miss tags that are used to define the release of the flutter tool so it is recommended to use `git fetch` typically to avoid version mismatches when running `flutter update-packages`.

Please make sure all your patches have detailed commit messages explaining what the problem was and what the solution is.

You must complete the Contributor License Agreement. You can do this online, and it only takes a minute. If you've never submitted code before, you must add your (or your organization's) name and contact info to the AUTHORS file.

Getting a code review

Every PR must be code-reviewed before check-in, including things like rolling a dependency. Getting a review means that a regular Flutter contributor (someone with commit access; see [[contributor access]] for details) has "approved" the PR in the GitHub UI. We call this "getting an LGTM" ("looks good to me").

If you are not yourself someone with commit access, then a second person with commit access must also review and approve your PR. This ensures that two people with commit access (trusted contributors) agree on every commit.

Why

Code review serves many critical purposes. There's the obvious purpose: catching errors. Even the most experienced engineers frequently make errors that are caught by code review. But there are also many other benefits of code reviews:

- It spreads knowledge among the team. Since every line of code will have been read by two people, it's more likely that once you move on, someone else will understand the code.
- It keeps you honest. Knowing that someone will be reading your code, you are less tempted to cut corners and more motivated to write code you are proud of.
- It exposes you to different modes of thinking. Your code reviewer has probably not thought about the problem in the same way you have, and so may have a fresh perspective and may find you a better way to solve the problem.

We recommend you consider these suggestions for addressing code review comments on your PR.

When

If you're working on a big patch, don't hesitate to get reviews early, before you're ready to check code in. Also, don't hesitate to ask for multiple people to review your code, and don't hesitate to provide unsolicited comments on other people's PRs (although approvals in the GitHub UI should be reserved for those with contributor access). The more reviews the better.

If nobody reviews your PR within two weeks, you can ask for a review via our [[Chat]] channels. Start by asking in #hackers, saying what your patch does and providing a link.

Who

Code should be reviewed by the owner (tech lead) of the area(s) of the codebase that you are changing, or someone to whom they have delegated that authority. (If you're not sure who that is, for the area of code you're dealing with, ask on Discord. See [[Chat]] for details.)

If anyone else leaves comments, please also wait for their approval (LGTM) before landing code.

A reviewer may in some circumstances consider the code satisfactory without having fully reviewed or understood it. If a reviewer has not fully reviewed the code, they admit to this by saying "RSLGTM" rather than just "LGTM". If you feel your code needs a real review, please find someone to actually review it. ("RSLGTM" means "Rubber Stamp Looks Good To Me".)

If you can't figure out who should review your code and GitHub's suggestions aren't useful to you then reach out on our [[Chat]] channels. The `#hackers-new` channel is a good place to ask for help if you're a new contributor.

How

Code review status is managed via GitHub's approval system. PRs should not be merged unless one or more contributors with commit access (at least one of which should be very familiar with the code in question) have approved the PR in the GitHub UI. As such, contributors without commit access are welcome to comment on PRs, but they should NOT be approving them in the GitHub UI. Approvals from contributors without commit access may mislead others to believe that a PR is ready to be merged, even though it has not gone through the official code review process.

Reviewers should carefully read the code and make sure they understand it. A reviewer should check the code for both high level concerns, such as whether the approach is reasonable and whether the code's structure makes sense, as well as lower-level issues like how readable the code is and adherence to the Flutter style guide. Use these best practices when reviewing code and providing comments.

As a reviewer, you are the last line of defense.

1. Take a step back. What problem is the PR trying to solve? Is it a real problem?
2. What other solutions could we consider? What could we do to make this even better?
3. Is it the best API? See our philosophy section. Look for state duplication, synchronous slow work, complecting, global state, overly-specific APIs, API cliffs and API oceans, API design in a vacuum (without a customer). If these terms don't make sense, read the style guide again. :-)
4. Is it the best implementation? Again, see our style guide, in particular its section on good coding patterns. Are there hacks? Are we taking on more technical debt? Think of ways in which the code could break.
5. Is it testable? Is it tested? **All code must be tested.** Are there asserts? Encourage liberal use of assertions.
6. Look for mistakes in indenting the code and other trivial formatting problems.
7. Is new code licensed correctly?

8. Is the documentation thorough and useful? Look for useless documentation, empty prose, and breadcrumbs. See the documentation section of our style guide for what that means.
9. Check for good grammar in API docs and comments. Check that identifiers are named according to our conventions.

Once you are satisfied with the contribution, and *only* once you are satisfied, use the GitHub "Approval" mechanism (an "LGTM" comment is not sufficient). If you feel like you are being worn down, hand the review to someone else. Consider our conflict resolution policy if you feel like you are being forced to agree to something you don't like.

Reviewers should not give an LGTM unless the patch has tests that verify all the affected code, or unless a test would make no sense. If you review a patch, you are sharing the responsibility for the patch with its author. You should only give an LGTM if you would feel confident answering questions about the code.

In general, reviewers should favor approving a PR once it is in a state where it definitely improves the overall code health of the system being worked on, even if the PR isn't perfect.

Reviewers should always feel free to leave comments expressing that something could be better, but if it's not very important, prefix it with something like "Shouldn't block this PR but: " to let the author know that it's just a point of polish that they could choose to ignore in the current PR (these should be documented in TODO comments with a tracking issue).

If you are 100% sure the patch is good and has no issues but you haven't really fully reviewed or understood it, you can give it a rubber-stamp review by marking it RSLGTM. If you mark a patch as RSLGTM, you are still sharing the responsibility for the patch with its author. Reviewing a patch as RSLGTM should be a rare event.

If you are not a regular Flutter contributor (someone with commit access), we very much welcome your reviews on code contributions in the form of comments on the code, but please refrain from approving or LGTM'ing changes, as it confuses PR authors, who may think your approval is authoritative and merge the PR prematurely.

Landing a patch

Once you have submitted your patch and received your LGTM, if you do not have commit access to the repository yet, then wait for one of the project maintainers to submit it for you.

If you do have access, you can just click the green "Merge pull request" button on the GitHub UI of your pull request. Please squash commits (GitHub does this for you by default normally).

Squashing commits

When you squash commits, by default, GitHub will concatenate all your commit messages to form a unified commit message. This often yields an overly verbose commit message with many unhelpful entries (e.g. "fix typo"). Please double-check (and hand-edit if necessary) your commit message before merging such that the message contains a helpful description of the overall change.

Regressions in functionality

If a check-in has caused a regression on master for any of the flutter repositories, revert (roll back) the check-in (even if it isn't yours). When master is broken, it slows down everyone else on the project, so we want to get the tree green again as soon as possible.

If your revert is a straight revert of the latest commit, then you do not need to wait for precommit tests before landing it (since by definition you are bringing the tree back to a known configuration). Just create the revert and land it, then tell the person whose patch you reverted that you reverted their patch, and leave a comment on the PR that you reverted.

If things are broken, the priority of everyone on the team should be helping the team fix the problem. Someone (you, if nobody else has yet taken ownership) should own the issue, and they can delegate responsibilities to others on the team. Once the problem is resolved, write a post-mortem. Postmortems are about documenting what went wrong and how to avoid the problem (and the entire class of problems like it) from recurring in the future. Postmortems are emphatically *not* about assigning blame.

There is no shame in making mistakes.

If the regression still left the tree in a green state (meaning that the failure is one we were not previously testing for), please update the [[Bad Builds]] page to note which builds were affected, so that we don't release a beta build with the regression. Then, write a test for this failure mode! See [[Running and writing tests]] for more details.

Regressions in performance

After each check-in, please monitor the performance dashboards.

If you see a regression (any of the charts increasing after your commit), please follow these steps:

- Comment on the PR acknowledging the regression.
- If the regression is expected and is a desirable trade-off (e.g. disk size increased slightly in exchange for a significant improvement in speed), then rebaseline the relevant benchmarks (log in, then click the magnifying glass at the top right of each chart, then click the button to auto rebaseline and commit).
- If the regression is not expected, and may be a problem in your PR, revert your PR and investigate.
- If the regression is not expected, and is quite severe, revert your PR and investigate.
- If the regression is not expected, and is not severe, and is definitely not a problem in your PR (e.g. you changed a comment and the analyzer performance got worse, or you deleted a README and the rasterizer slowed down), then file a bug, labeled with the "regression", "performance", `P0` labels, and either investigate or delegate to someone to investigate. The investigation should be considered a high priority. It is your responsibility to make sure that the cause is understood within a few days.

Performance regressions are not a problem so long as they are promptly dealt with. Therefore, Flutter considers all unexpected performance regressions to be TODAY, or the highest-priority issues until we have it under control (e.g. we know what caused it and either have a fix under way or have determined it is an acceptable trade-off).

Performance regressions caused by auto-roller commits

Although reverting a normal commit that caused performance regressions is the default behavior, reverting an auto-roller (e.g., an engine-roller commit like https://github.com/flutter/flutter/commit/fdcb57b69eff2162e9aead6dec0f8058788e7608) commit could cause some complications:

1. The auto-roller commit usually include multiple commits of the source repo (e.g., engine-roller commit includes multiple commits of https://github.com/flutter/engine). This can be applied recursively as the engine-roller commit includes a dart-roller commit, or a skia-roller commit. Therefore, a roller commit could actually include a ton of leaf-level commits, which makes it really hard to triage which leaf commit actually caused the regression.

2. The auto-roller will try to roll again as soon as possible that will reland any changes reverted by a Flutter commit revert. So in order to keep the revert effective, one has to either (1) pause the auto-roller, or (2) revert the leaf commit in the source repo.

3. If the auto-roller is paused for a long time (say 1 day), the source repo will accumulate many commits. That makes the next roll very hard to manage: it's difficult to triage a build failure or a new performance regression caused by the next roll, since that roll will include all the commits in the paused period.

Therefore, reverting a roller commit or pausing the auto-roller is *NOT* the default action if it causes a performance regression. The default action should be to file an issue with labels "performance", "regression", and `P0` immediately, and start investigating which leaf-commit caused the regression. Once the leaf-commit is identified, check if it's an expected trade-off. If so, remove the `P0` label and try to see if there's any way to mitigate the regression. If not, revert the leaf commit in the source repo and let the auto-roller apply that revert. Once the revert is rolled into Flutter, close the issue.

Avoid "Revert "Revert "Revert "Revert "Fix foo""" commit messages

Please limit yourself to one "Revert" per commit message, otherwise we won't have any idea what is actually landing. Is it putting us back to where we were before? Is it adding new code? Is it a controversial new feature that actually caused a regression before but is now fixed (we hope)?

Only use "Revert" if you are actually returning us to a known-good state. When you later revert the revert, just land the PR afresh with the original commit message, possibly updated with the information since collected (and ideally, including a link to the original PR and to the revert PR so that people can follow the breadcrumbs later).

Handling breaking changes

In general, we want to avoid making changes to Flutter, our plugins, or our packages, that force developers using Flutter to change their code in order to upgrade to new versions of Flutter. See our compatibility policy.

Sometimes, however, doing this is necessary for the greater good. We want our APIs to be intuitive; if being backwards-compatible requires making an API into something that we would never have designed that way unless forced to by circumstances, then we should instead break the API and make it good.

The process for making breaking changes is as follows:

1. Determine if your change is a breaking change

The first step in making a breaking change is to implement the change you wish to see and run the existing tests against your new code (without having changed the tests first). Changes that break (i.e. require changes to) one or more of the contributed tests are considered "breaking changes".

The "contributed tests" are:

- Those in the `customer_testing` shard on `flutter/flutter` PRs.
- Additional test suites that we have been allowed to run but that are not public. (Notably, Google allows us to run several tens of thousands of proprietary tests on each commit.)

You can contribute tests to the flutter/tests repo by following the instructions on that repo. If you have a significant test suite that you would like to have be considered part of the breaking change definition (one too big to land in the flutter/tests repo), please contact Hixie at ian@hixie.ch.

In cases where we can imagine reasonable scenarios where developers would be affected negatively, by courtesy, once the change has landed, engineers are encouraged to announce the changes by sending an e-mail to flutter-announce@, a message to the `#announcements` channel on our [[Chat]], and tagging the relevant issues with the **severe: API break** label (such that they will be included in our release notes). However, we do not consider these breaking changes. (One reason to do this would be if we see our own tests being significantly affected, even if no contributed test actually fails.)

This definitions is binding. If you think you need an exemption to this policy, please contact Hixie on the #hackers [[Chat]] channel. If a breaking change lands without following this policy and without an explicit exemption from

@Hixie, it must be reverted.

2. Evaluate the breaking change proposal

If you discover that your change would be a breaking change as defined above, we must carefully evaluate it. Create a design document (use this template). You want to describe the change in detail, and ask for feedback. Things you should include are:

- What problem are you solving?
- What does migrating code to your proposed new API look like? Show several before and after examples.
- What other alternatives did you consider?
- A request for feedback. Is the change valuable?

Link to your design document from your issue. Ping @RedBrogdon on the #hackers-devrel channel in [[Chat]] and point him to your design document. Bring it up in the #general channel as well. Send an e-mail to the flutter-dev@ mailing list asking for feedback on your design doc. Allow for several (two or three) days of discussion.

Consider if you really need to make this change. In general, merely renaming a class to make things slightly clearer is insufficient value to justify a breaking change. Such changes leave behind a legacy of old tutorials, YouTube videos, StackOverflow comments, etc, that reference the old name, and so any improvement to the developer experience can be easily offset by the added burden on our ecosystem as a whole. (We record such changes we wish we could make on https://github.com/flutter/flutter/issues/24722, feel free to add it there.)

3. Prepare your change.

Rather than deploying the proposed change as one PR that immediately breaks existing code, adjust your PR so that it introduces the new functionality, API, behavior change, etc, in an opt-in fashion.

For example, rather than replacing a widget with another, introduce the new widget and discourage use of the old one. Rather than changing the order in which a certain argument is processed, provide a flag that selects which order the arguments will be processed in.

When changing the semantics of an API with a temporary opt-in, a three-phase change is needed (adding the new API and opt-in, then removing the old API, then removing the opt-in.)

If possible, avoid four-phase deprecations (adding a new API with a temporary name and deprecating an old API, removing the old API, changing the new API to the old name and deprecating the temporary name, and finally removing the temporary name), because they involve a lot of churn and will irritate our developers.

Stage your change and the documentation for your change. Typically this will be two or more PRs, plus PRs to fix the tests that were broken (see step 1), as well as writing a migration guide as a PR to the Website repository.

If possible, include flutter fixes to aid users in migration. Whether or not the change is supported by flutter fix should be included in the migration guide. To learn about authoring fixes, see Data driven Fixes.

Use our breaking change migration guide template (follow all the instructions in the comments) to create the migration guide that describes the change. Do not land the migration guide at this time. You will need to update it before you land it in the last step.

4. Land your change.

Once you are ready, have received feedback, iterated on your design and your migration guide, land your initial change and start migrating clients. *Do not yet land the migration guide.* Once all the clients are migrated, land your final change. (You may have several iterations here if you have a multiphase roll-out.)

During this process, each individual PR does not break any tests, so it should not block any autorollers.

5. Document the change, including clear documentation for migrating code, with samples, and clear rationales for each change

Once everything has landed:

- update your migration guide based on your experience migrating everyone,
- update the timeline on the guide, and push it to the flutter.dev Web site (don't forget to update the index of that directory as well),
- e-mail a copy to flutter-announce@,
- notify the `#announcements` channel on our [[Chat]], and
- add the **severe: API break** label to the relevant issues, so they get listed in the upcoming Release notes.

Deprecations

Old APIs can be marked as deprecated as part of this process. Deprecation is not a way to avoid making a breaking change; you should consider deprecating an API to be equivalent to removing it, as some of our customers (and we ourselves) consider using a deprecated API to be anathema (triggering a build failure).

The syntax for deprecations must match the following pattern:

```
@Deprecated(
  'Call prepareFrame followed by owner.requestVisualUpdate() instead. '
  'This feature was deprecated after v2.9.0-0.1.pre.'
)
```

In other words:

```
@Deprecated(
  '[description of how to migrate] '
  'This feature was deprecated after [beta version at time of deprecation].'
)
```

To determine the latest beta version, see https://flutter.dev/docs/development/tools/sdk/releases.

When adding a deprecation notice to the framework, a flutter fix should be included with your change. This helps users migrate to the new API as easily as possible. To learn more about authoring fixes, see Data driven Fixes. If a fix cannot be written for the new API, please file an issue in https://github.com/dart-lang/sdk and link to it in your change.

Using this standard form ensures that we can write a script to detect all deprecated APIs and remove them. We have a test that verifies that this syntax is followed.

When deprecating features, be aware that you will not by default be informed when the Flutter code itself uses the deprecated feature (there is a `deprecated_member_use_from_same_package: ignore` line in the root `analysis_options.yaml` file). To find places where the old feature is used, rename its declaration and see where the compiler complains. (You can't just comment out the "ignore" in the `analysis_options.yaml` file because it's hiding hundreds of other warnings...)

Deprecations are removed in a consistent "first-in-first-out" fashion. The lifetime for a Flutter deprecation is 1 year after reaching the stable channel, or after 4 stable releases, whichever is longer. Deprecations are still subject to the policy described on the breaking changes page of the website. Where possible, prepare the dart fix tools and write appropriate migrations guides.

Skipped Tests

Tests can be skipped using the `skip` parameter of `test()`, `group()` and `testWidgets()`. However, they should be kept to a minimum and only done for the following two reasons.

The first is If there is a test that is flaky, we can mark is as temporarily skipped to keep the tree green while a fix for it is developed. For these types of skips you need to file a tracking issue so we can ensure there is follow up to remove the skip. This tracking issue should be tagged with the `skip-test` label. Then in a comment on the same line as the parameter, include a link to this issue:

```
  skip: true, // https://github.com/flutter/flutter/issues/XXXXX
```

The other reason to use the skip parameter is to mark a test that by design doesn't make sense to test under a specific condition. An example would be a test that only tests a feature available on a specific platform or environment. For these cases, include a comment on the same line as the skip parameter with the text `[intended]` and a short description of why the skip is needed:

```
  skip: isBrowser, // [intended] There are no default transitions to test on the
web.
```

If the analyzer script sees a skip without a comment containing either an issue link or an `[intended]` tag, it will report and error and fail the check.