Documentation Guidance for Flutter

Introduction

This document contains points to consider before submitting a documentation PR, general documentation rules, and pull request guidance for various documentation change proposals for anyone who wishes to contribute or make changes to Flutter's current list of documentation.

Table of Contents

Points to consider

The Flutter team sincerely welcomes anyone who wishes to contribute or make changes to our current list of documentation. Not only it helps us to identify potential flaws that may exist in our current documentation but also lets us know alternative ways that we can change our documentation to further benefit our users and developers.

Before you submit a documentation pull request for us to review, please consider the following points.

Are there existing documentation that can be utilized?

When proposing a new documentation, are there existing Flutter documentation that can be utilized? It is always better to revise existing documentation to make it more useful to the user rather than replicating duplicate materials.

Have you considered the general rules for documentation listed below?

In general, we use “dartdoc" for our Dart documentation, and similar technologies for the documentation of our APIs in other languages, such as ObjectiveC and Java. All public members in Flutter libraries should have a documentation. We typically follow the Dart documentation guide except where that would contradict this page.

We always encourage people to document immediately the answer they found when working on Flutter and finding themselves still asking for the same question even when looking at those answers so that we can get a good sense of how the answers work for real questions. Additionally, it is fine that if your otherwise-unrelated PR has a bunch of documentation fixes in it to answer questions you had while you were working on your PR.

However, there is one exception. It is better not to document something in our API than to document it poorly. Even if you don’t document it, it will still appear on our list of things to document. Feel free to contact us to remove documentation that violates this. See sections “Answer your own questions straight away” and “Avoid useless documentation” below for further details.

Is the proposed change a major update that is prevalent throughout Flutter documentation?

If the proposed change is a major update/change that is prevalent throughout Flutter documentation (e.g., proposing adding medium sized samples throughout the Flutter API documentation), please use the Flutter design doc template instead and follow the instructions from there.

For more info on design document and design document examples, please visit the section called Design documents under our wiki page.

Is the proposed change a document on the website rather than API documents (api.flutter.dev)?

If the change that you would like to make is a change for the flutter website documents or to add in additional documents for flutter website, please make the appropriate pull request to Flutter/website repo and proposed the change there. We also suggest viewing the “general rules to consider for documentation” below even when proposing document changes for the website. For additional details, please refer to contributing guidance for flutter website.

General rules to consider for documentation

Answer your own questions straight away

When working on Flutter, if you find yourself asking a question about our systems, please place whatever answer you subsequently discover into the documentation in the same place where you first looked for the answer. That way, the documentation will consist of answers to real questions, where people would look to find them. Do this right away; it’s fine if your otherwise-unrelated PR has a bunch of documentation fixes in it to answer questions you had while you were working on your PR.

We try to avoid reliance on "oral tradition". It should be possible for anyone to begin contributing without having had to learn all the secrets from existing team members. To that end, all processes should be documented (typically on the wiki), code should be self-explanatory or commented, and conventions should be written down, e.g. in our style guide.

There is one exception: it’s better to *not* document something in our API docs than to document it poorly. This is because if you don’t document it, it still appears on our list of things to document. Feel free to remove documentation that violates the rules below (especially the next one), so as to make it reappear on the list.

Avoid useless documentation

If someone could have written the same documentation without knowing anything about the class other than its name, then it’s useless.

Avoid checking in such documentation, because it is no better than no documentation but will prevent us from noticing that the identifier is not actually documented.

Example (from CircleAvatar):

```
// BAD:

/// The background color.
final Color backgroundColor;

/// Half the diameter of the circle.
final double radius;


// GOOD:

/// The color with which to fill the circle. Changing the background
/// color will cause the avatar to animate to the new color.
final Color backgroundColor;

/// The size of the avatar. Changing the radius will cause the
/// avatar to animate to the new size.
final double radius;
```

Writing prompts for good documentation

If you are having trouble coming up with useful documentation, here are some prompts that might help you write more detailed prose:

- If someone is looking at this documentation, it means that they have a question which they couldn’t answer by guesswork or by looking at the code. What could that question be? Try to answer all questions you can come up with.
- If you were telling someone about this property, what might they want to know that they couldn’t guess? For example, are there edge cases that aren’t intuitive?

- Consider the type of the property or arguments. Are there cases that are outside the normal range that should be discussed? e.g. negative numbers, non-integer values, transparent colors, empty arrays, infinities, NaN, null? Discuss any that are non-trivial.

- Does this member interact with any others? For example, can it only be non-null if another is null? Will this member only have any effect if another has a particular range of values? Will this member affect whether another member has any effect, or what effect another member has?

- Does this member have a similar name or purpose to another, such that we should point to that one, and from that one to this one? Use the `See also:` pattern.

- Are there timing considerations? Any potential race conditions?

- Are there lifecycle considerations? For example, who owns the object that this property is set to? Who should `dispose()` it, if that's relevant?
- What is the contract for this property/method? Can it be called at any time? Are there limits on what values are valid? If it's a `final` property set from a constructor, does the constructor have any limits on what the property can be set to? If this is a constructor, are any of the arguments not nullable?
- If there are `Future` values involved, what are the guarantees around those? Consider whether they can complete with an error, whether they can never complete at all, what happens if the underlying operation is canceled, and so forth.

Avoid empty prose

It's easy to use more words than necessary. Avoid doing so where possible, even if the result is somewhat terse.

```
// BAD:

/// Note: It is important to be aware of the fact that in the
/// absence of an explicit value, this property defaults to 2.

// GOOD:

/// Defaults to 2.
```

In particular, avoid saying "Note:". It adds nothing.

Leave breadcrumbs in the comments

This is especially important for documentation at the level of classes.

If a class is constructed using a builder of some sort or can be obtained via some mechanism other than merely calling the constructor, then include this information in the documentation for the class.

If a class is typically used by passing it to a particular API, then include that information in the class documentation also.

If a method is the main mechanism used to obtain a particular object or is the main way to consume a particular object, then mention that in the method's description.

Typedefs should mention at least one place where the signature is used.

These rules result in a chain of breadcrumbs that a reader can follow to get from any class or method that they might think is relevant to their task all the way up to the class or method they actually need.

Example:

```
// GOOD:

/// An object representing a sequence of recorded graphical operations.
///
/// To create a [Picture], use a [PictureRecorder].
///
/// A [Picture] can be placed in a [Scene] using a [SceneBuilder], via
/// the [SceneBuilder.addPicture] method. A [Picture] can also be
/// drawn into a [Canvas], using the [Canvas.drawPicture] method.
abstract class Picture ...
```

You can also use "See also" links, is in:

```
/// See also:
///
/// * [FooBar], which is another way to peel oranges.
/// * [Baz], which quuxes the wibble.
```

Each line should end with a period. Prefer "which..." rather than parentheticals on such lines. There should be a blank line between "See also:" and the first item in the bulleted list.

Refactor the code when the documentation would be incomprehensible

If writing the documentation proves to be difficult because the API is convoluted, then rewrite the API rather than trying to document it.

Canonical terminology

The documentation should use consistent terminology:

- *method* - a member of a class that is a non-anonymous closure
- *function* - a callable non-anonymous closure that isn't a member of a class
- *parameter* - a variable defined in a closure signature and possibly used in the closure body.
- *argument* - the value passed to a closure when calling it.

Prefer the term "call" to the term "invoke" when talking about jumping to a closure.

Prefer the term "member variable" to the term "instance variable" when talking about variables associated with a specific object.

Typedef dartdocs should usually start with the phrase "Signature for...".

Use correct grammar

Avoid starting a sentence with a lowercase letter.

```
// BAD

/// [foo] must not be null.

// GOOD

/// The [foo] argument must not be null.
```

Similarly, end all sentences with a period.

Use the passive voice; recommend, do not require

Never use "you" or "we". Avoid the imperative voice. Avoid value judgements.

Rather than telling someone to do something, use "Consider", as in "To obtain the foo, consider using [bar].".

In general, you don’t know who is reading the documentation or why. Someone could have inherited a terrible codebase and be reading our documentation to find out how to fix it; by saying "you should not do X" or "avoid Y" or "if you want Z", you will put the reader in a defensive state of mind when they find code that contradicts the documentation (after all, they inherited this codebase, who are we to say that they’re doing it wrong, it’s not their fault).

Provide sample code

Sample code helps developers learn your API quickly. Writing sample code also helps you think through how your API is going to be used by app developers.

Sample code should go in a section of the documentation that begins with `{@tool snippet}`, and ends with `{@end-tool}`. This will then be checked by automated tools, and extracted and formatted for display on the API documentation web site docs.flutter.io. For details on how to write sample code, see the API documentation documentation. For example, below is the sample code for building an infinite list of children with the ListView widget, as it would appear in the Flutter source code for the ListView widget:

```
/// A scrollable list of widgets arranged linearly.
///
/// ...
///
/// {@tool snippet}
/// An infinite list of children:
///
/// ```dart
/// ListView.builder(
///   padding: EdgeInsets.all(8.0),
///   itemExtent: 20.0,
///   itemBuilder: (BuildContext context, int index) {
///     return Text('entry $index');
///   },
/// )
/// ```
/// {@end-tool}
class ListView {
  // ...
```

Provide full application samples.

Our UI research has shown that developers prefer to see examples that are in the context of an entire app. So, whenever it makes sense, provide an example that can be presented as part of an entire application instead of just a simple sample like the one above.

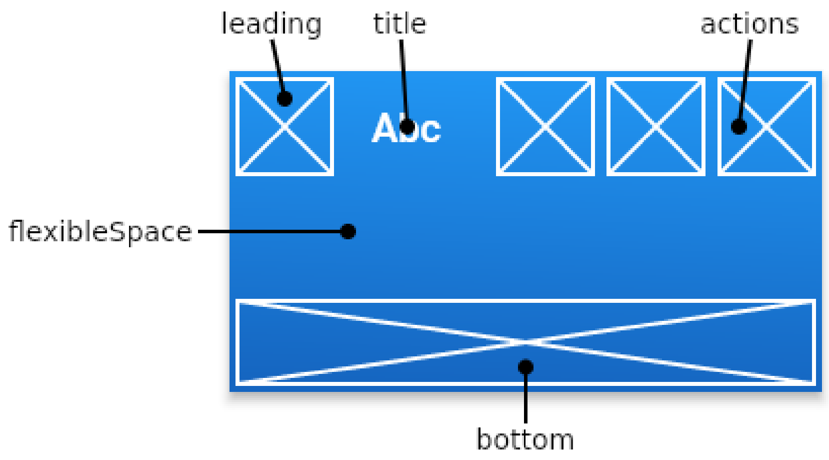
This can be done using the `{@tool dartpad --template=<template>} ... {@end-tool}` or `{@tool sample --template=<template>} ... {@end-tool}` dartdoc indicators, where `<template>` is the name of a template that the given blocks of dart code can be inserted into. See here for more details about writing these kinds of examples, and here for a list and description of the available templates.
Dartpad examples (using the dartdoc `dartpad` indicator) will be presented on the API documentation website as an in-page executable and editable example. This allows developers to interact with the example right there on the page, and is the preferred form of example.
For examples that don't make sense in a web page (for example, code that interacts with a particular platform feature), application examples (the dartdoc `sample` indicator) are preferred, and will be presented on the API documentation website along with information about how to instantiate the example as an application that can be run.
IDEs viewing the Flutter source code using the Flutter plugin also offer the option of creating a new project with either kind of example.

Provide illustrations, diagrams or screenshots

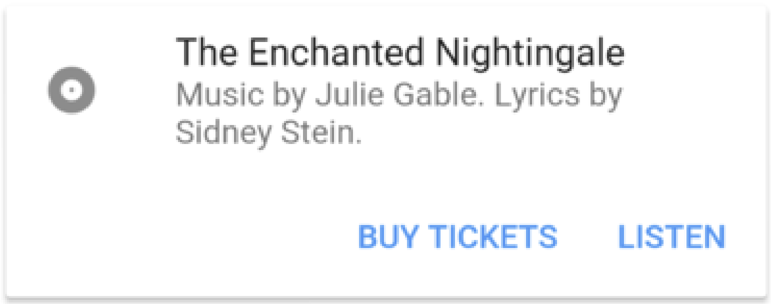
For any widget that draws pixels on the screen, showing how it looks like in its API doc helps developers decide if the widget is useful and learn how to customize it. All illustrations should be easily reproducible, e.g. by running a Flutter app or a script.

Examples:

- A diagram for the AppBar widget



- A screenshot for the Card widget


The Enchanted Nightingale
Music by Julie Gable. Lyrics by Sidney Stein.
BUY TICKETS
LISTEN

When creating diagrams, make sure to provide alternative text as described in the HTML specification.

Clearly mark deprecated APIs

We have conventions around deprecation. See the Tree Hygiene page for more details.

Use `///` for public-quality private documentation

In general, private code can and should also be documented. If that documentation is of good enough quality that we could include it verbatim when making the class public (i.e. it satisfies all the style guidelines above), then you can use `///` for those docs, even though they're private.
Documentation of private APIs that is not of sufficient quality should only use `//`. That way, if we ever make the corresponding class public, those documentation comments will be flagged as missing, and we will know to examine them more carefully.
Feel free to be conservative in what you consider "sufficient quality". It's ok to use `//` even if you have multiple paragraphs of documentation; that's a sign that we should carefully rereview the documentation when making the code public.

Dartdoc templates and macros

Dartdoc supports creating templates that can be reused in other parts of the code. They are defined like so:

```
/// {@template <id>}
/// ...
/// {@endtemplate}
```

and used via:

```
/// {@macro <id>}
```

The <id> should be a unique identifier that is of the form flutter.library.Class.member[.optionalDescription].
For example:

```
// GOOD:
/// {@template flutter.rendering.Layer.findAnnotations.aboutAnnotations}
/// Annotations are great!
/// {@endtemplate

// BAD:
/// {@template the_stuff!}
/// This is some great stuff!
/// {@endtemplate}
```

The optionalDescription component of the identifier is only necessary if there is more than one template defined in one Dartdoc block. If a symbol is not part of a library, or not part of a class, then just omit those parts from the ID.

Dartdoc-specific requirements

The first paragraph of any dartdoc section must be a short self-contained sentence that explains the purpose and meaning of the item being documented. Subsequent paragraphs then must elaborate. Avoid having the first paragraph have multiple sentences. (This is because the first paragraph gets extracted and used in tables of contents, etc, and so has to be able to stand alone and not take up a lot of room.)

When referencing a parameter, use backticks. However, when referencing a parameter that also corresponds to a property, use square brackets instead. (This contradicts the Dart style guide, which says to use square brackets for both. We do this because of dartdoc issue 1486. Currently, there’s no way to unambiguously reference a parameter. We want to avoid cases where a parameter that happens to be named the same as a property despite having no relationship to that property gets linked to the property.)

```
// GOOD

  /// Creates a foobar, which allows a baz to quux the bar.
  ///
  /// The [bar] argument must not be null.
  ///
  /// The `baz` argument must be greater than zero.
  Foo({ this.bar, int baz }) : assert(bar != null), assert(baz > 0);
```

Avoid using terms like "above" or "below" to reference one dartdoc section from another. Dartdoc sections are often shown alone on a Web page, the full context of the class is not present.

Documentation pull request

1. If there is an existing document that you would like to make change to, when making your pull request, please include

- Goal and short description of the request
- The URL or screenshot of where you would like to make the changes
- What changes you are making
- The reason for the change
- Submit the changed file

Please also make sure to review the pull_request_template doc and go through the checklist before submission.

2. If there is a new document that you would like to contribute to Flutter, when making your pull request, please include

- Goal and short description of the request
- The reason for the new document
- Submit the file

Please also make sure to review the pull_request_template doc and go through the checklist before submission.

3. If the documentation that you would like to change is part of the wiki pages listed, in your pull request, please indicate the following:

- Goal and short description of the request
- The place where you would like to make changes (you can use screenshots and best to add in URL link in addition to your description)
- The change you'd like to make
- The reason for the change (e.g., typo etc.)

Please also make sure to review the pull_request_template doc and go through the checklist before submission.

4. If the change is for API documents (on api.flutter.dev), you should follow step 1 and edit the relevant .dart files in the packages folder under src to be submitted for PR.

5. If the documents that you would like to change are those on the website (i.e., flutter.dev), you should go to flutter/website and make the appropriate documentation pull request.

You can find the docs relevant to the flutter website on the repository through src/docs or if you are unable to find them, you can simply include the URL to where you find the document and suggest the change in the pull request.

They are also happy to accept your PRs to the Flutter website repo, even if it's just to fix a typo. The following is the contributing guidance for the Flutter website: CONTRIBUTING.md

If you need additional help, consider asking for advice on the #hackers-new channel on Discord.