Documentation Best Practices

"Say what you mean, simply and directly." - [Brian Kernighan] (https://en.wikipedia.org/wiki/The_Elements_

Contents:

Minimum viable documentation

A small set of fresh and accurate docs are better than a sprawling, loose assembly of "documentation" in various states of disrepair.

Write short and useful documents. Cut out everything unnecessary, while also making a habit of continually massaging and improving every doc to suit your changing needs. **Docs work best when they are alive but frequently trimmed, like a bonsai tree**.

This guide encourages engineers to take ownership of their docs and keep them up to date with the same zeal we keep our tests in good order. Strive for this.

- Identify what you really need: release docs, API docs, testing guidelines.
- Delete cruft frequently and in small batches.

Update docs with code

Change your documentation in the same CL as the code change. This keeps your docs fresh, and is also a good place to explain to your reviewer what you're doing.

A good reviewer can at least insist that docstrings, header files, README.md files, and any other docs get updated alongside the CL.

Delete dead documentation

Dead docs are bad. They misinform, they slow down, they incite despair in engineers and laziness in team leads. They set a precedent for leaving behind messes in a code base. If your home is clean, most guests will be clean without being asked.

Just like any big cleaning project, **it's easy to be overwhelmed**. If your docs are in bad shape:

- Take it slow, doc health is a gradual accumulation.

- First delete what you're certain is wrong, ignore what's unclear.
- Get your whole team involved. Devote time to quickly scan every doc and make a simple decision: Keep or delete?
- Default to delete or leave behind if migrating. Stragglers can always be recovered.
- Iterate.

Prefer the good over the perfect

Your documentation should be as good as possible within a reasonable time frame. The standards for an documentation review are different from the standards for code reviews. Reviewers can and should ask for improvements, but in general, the author should always be able to invoke the "Good Over Perfect Rule". It's preferable to allow authors to quickly submit changes that improve the document, instead of forcing rounds of review until it's "perfect". Docs are never perfect, and tend to gradually improve as the team learns what they really need to write down.

Documentation is the story of your code

Writing excellent code doesn't end when your code compiles or even if your test coverage reaches 100%. It's easy to write something a computer understands, it's much harder to write something both a human and a computer understand. Your mission as a Code Health-conscious engineer is to **write for humans first, computers second.** Documentation is an important part of this skill.

There's a spectrum of engineering documentation that ranges from terse comments to detailed prose:

1. **Inline comments**: The primary purpose of inline comments is to provide information that the code itself cannot contain, such as why the code is there.
2. **Method and class comments**:
    - **Method API documentation**: The header / Javadoc / docstring comments that say what methods do and how to use them. This documentation is **the contract of how your code must behave**. The intended audience is future programmers who will use and modify your code.

      It is often reasonable to say that any behavior documented here should have a test verifying it. This documentation details what arguments the method takes, what it returns, any "gotchas" or restrictions, and what exceptions it can throw or errors it can return.

It does not usually explain why code behaves a particular way unless that's relevant to a developer's understanding of how to use the method. "Why" explanations are for inline comments. Think in practical terms when writing method documentation: "This is a hammer. You use it to pound nails."

- **Class / Module API documentation**: The header / Javadoc / docstring comments for a class or a whole file. This documentation gives a brief overview of what the class / file does and often gives a few short examples of how you might use the class / file.

  Examples are particularly relevant when there's several distinct ways to use the class (some advanced, some simple). Always list the simplest use case first.

3. **README.md**: A good README.md orients the new user to the directory and points to more detailed explanation and user guides:

   - What is this directory intended to hold?
   - Which files should the developer look at first? Are some files an API?
   - Who maintains this directory and where I can learn more?

   See the README.md guidelines.