Karlsruher Institut für Technologie

SCC Scientific Computing Center

Maintainable up-to-date Documentation with CI

Michele Mesiti, PhD | 10. February 2024

Outline

Disclaimer and Expectation Management

My personal motivation for giving this talk:
I am annoyed when trying to read outdated, slightly off (or outright wrong) documentation.
But

- I am not a documentation expert
- I am not a documentation enthusiast
- Yes, I have written documentation *and* nobody has read it
- And I know it is hard. Documentation requires long term commitment:

  > *... a good documentation system also needs pretty constant care and feeding, reorganizing, and other maintained work. If you view this project as getting the tools in place,* ***without a long term commitment, it will fail*** *...*

(from writethedocs.org)

How do we reduce the necessary workload?

Acknowledgments

This topic was discussed
at the un-deRSE23 un-conference
and the content of this talk
is partially based on the contributions
that were made during that session.

A SHOUTOUT TO ALL THOSE WHO ATTENDED THAT SESSION.
ANY FACTUAL ERRORS/MISUNDERSTANDINGS IN THIS TALK
ARE SOLELY MY RESPONSIBILITY.

Best practices for documentation

How to document software?

- Docs-as-code talk by Jessica Mitchell (part of the HiRSE_PS seminar series)
- writethedocs.org
- How to document your research software by CodeRefinery

... or you can follow the opposite approach: Tutorial Driven Development (Talk) by Christopher Woods

Writing Documentation

Text formats for documentation

- MarkDown (intuitive, but many flavors and implementations - GitHub, GitLab, MyST...) originally by John Gruber

  > *The overriding design goal for Markdown's formatting syntax is to make it as readable as possible. The idea is that a Markdown-formatted document should be publishable as-is, as plain text, without looking like it's been marked up with tags or formatting instructions.*

  Flavours might be converging to a standard? Even JOSS uses it.
- ReSTructured Text (more markup, has an actual learning curve, more features, properly standardized) ReST
- Asciidoc *

  * I never used it and I will not comment further on them

These formats can typically be used in code comments.

Recommendation: in any case, consider using Semantic Line Breaks

Documentation Tools

A list of known tools, without any claim of completeness:

- Doxygen (mainly used for C/C++ - other languages are supported)
- Sphinx - historically used for Python projects
- MkDocs (“Language Agnostic”)
- Documenter.jl/ Literate.jl (Julia)
- Javadoc* (Java)
- FORD* (Fortran)
- hdoc* (C++)
- Quarto* (Python,R,Julia,Observable)

All these systems can extract documentation from comments/docstrings in code in some way (either with plugins or natively)

* I never used these and I will not comment further on them

Doxygen

- Markdown + custom syntax
- Lots of output formats available (including XML - great if you are a machine)
- Can also be used for Fortran (but Fortran projects can also use FORD)

Sphinx

- Supports mainly ReST, via docutils
- MarkDown support with plugin (MyST flavour)
- Output in various formats (not only HTML)
- Target language is usually Python, but there are ways to use it for other languages (e.g., languages supported by Doxygen, through Breathe - see Example by F.Goth and results)
- Vast choice of themes

MkDocs

- Supports only MarkDown
- Vast choice of themes
- Most popular theme/distribution: Material for MkDocs
- Doxygen integration through MkDoxy *

* project is in development

Documenter.jl

“Batteries Included” approach:

- jldoctest - to mark code snippets to test
- client side search
- deploy to github/gitlab pages with a function call

Making Documentation Useful and Convenient

deployment, search, metrics

Deploying docs

- ReadTheDocs
- GitHub pages
- GitLab pages

ReadTheDocs

- integration with GitHub/GitLab/BitBucket via web hooks
- guided set up of integration
- semi-automatic deployment (requires a .readthedocs.yaml config file)

GitHub and GitLab pages

- Host a static site directly on GitHub or on your GitLab instance
- GitHub:
  - the *pages* feature must be explicitly activated in the repository settings
  - an additional workflow will run on a chosen branch (usually *gh-pages*), running a jekyll build (that can be a no-op) and then the actual deployment of the site
  - for best result, any static site generator should run in his workflow and push on the chosen branch (usually *gh-pages*)
- GitLab:
  - the *gitlab pages* feature needs to be activated by the administrator of the instance
  - a `pages` job needs to be configured.
    Configuration is best done using the wizard in the GitLab Web UI (Used to be under Settings, now under Deploy)
  - in the `pages` job, the static website must be generated into a a `public/` directory, mentioned in the `artifacts:` section

Searching the documentation

The ability to search for content in the documentation is very important.
There is a challenge due to the fact that usually these are *static* websites and searching requires some computation.

- readthedocs.io provides server side search
- MkDocs provides a client-side search plugin
- Sphinx also provides a client-side search mechanism for keywords (more limited than server-side search)
- Documenter.jl has client-side search
- Doxygen has a keyword-based client-side search mode and a full-text server-side search mode

Client-side search can also be performed off-line.

Alternatives for Search*

- Site indexing by popular search engines (e.g., Programmable Search Engine)
- Can AI chatbots help with retrieving content from documentation?
  - Through model fine-tuning
  - Through Retrieval-Augmented-Generation (RAG - potentially much cheaper)

* I have no experience with this - included because of relevance

Measurements*

Understanding how, if and how much your documentation is read might be useful.

- Which pages of the documentation do users visit?
- What terms do they search for?

Tools available:

- readthedocs.io has a search and traffic analytics feature
- Web Server logs?
- Google Analytics *et simila*

* I have no experience with this - included because of relevance

Testing Documentation Automatically

Testing Documentation

Keeping the documentation up-to-date with the software (or system) it is supposed to document is a challenge.

This includes making sure that:

- the commands or code in the examples run without errors
- the commands or code in the examples produce the intended output.

Additionally, make sure that:

- the information provided without commands or code is factually correct

Challenges

This brings some challenges, *in general (not all of these might always apply)*:

- define proper equivalence between expected and real output (e.g. use regex patterns or ellipsis)
- setup and tear down of the test cases, (when they do not fit in the flow of the documentation)
- preserving state between code blocks
- define the correct environment for the tests to run (machine, path, user, environment variables, ...)
- extracting the code snippets from the documentation and test them (or associate the test cases with the right code snippets in the documentation)

The Solutions

Ready-available solutions:

- doctest (Python)
- Documenter.jl with jldoctest (Julia)
- rustdoc (rust) *
- jupyter notebooks + nbval (any language there is a jupyter kernel for)

* I have no experience with this - included because of relevance

doctest

- Detects “interactive session examples” in documentation and comments
- Runs them
- Checks that the output coincides with the expected one
- limited ways to deal with non-identical output (e.g., printing a Set)

Documenter.jl

- Runs code blocks appropriately marked
- Labeling code blocks allows to keep state between them
- Has a filtering mechanism that can be used to make sure only the relevant part of the output is checked

Literate programming: jupyter notebooks + nbval

Idea in a nutshell:

- Write docs using jupyter notebooks
- Hidden cells can be used for set up and tear down of test cases (cell hiding relies on nbconvert, done via tags - a bit cumbersome but doable)
- use nbval to check output (can use REGEX output sanitizing)
- works for any language with a suitable jupyter kernel - python, bash, c++, julia (installing the right kernels - c++ can be a little more complex)
- integrate the notebooks in the documentation:
  - Integrate jupyter notebooks into MkDocs
  - Integrate jupyter notebooks into Sphinx docs

Notes/Disclaimers:

- I have done only some basic tests of this, I do not have extensive experience so the devil might be in the details

Accepting defeat: it's hard

- Documentation (as any form of written content) is a liability.
  - Document the strict necessary
  - avoid redundancy (link to other places instead of writing your own explanation).
  - Trade-offs: of course it is nice to have a good, succinct collection of information tailored to a target user
- Write the documentation focusing on testability first:
  - make sure that the code is easy to extract out of the docs with automated tools
  - make sure that the output is easy to compare

  For example, accept the prescriptions of a framework/set of existing tools and prioritize them.

What to do if this is not possible?

- Write your own custom, customizable preprocessor?
- Write tests outside of the documentation, that:
  - Do setup-test-teardown as you need them
  - Check that the code under test is actually mentioned somewhere in the documentation,
  - Check that the output of the code under test is somewhere in the documentation, if needed

  In this way, the documentation is untouched - the tests could even be in another repository.