Docness

Release 0.1dev

Benoit Bryon

Sep 27, 2017

Contents

Vision, tips and conventions about documentation content and workflows.

CHAPTER 1

Ressources

- online documentation: http://docness.readthedocs.org/
- code repository: https://github.com/benoitbryon/docness
- bugtracker: https://github.com/benoitbryon/docness/issues

CHAPTER 2

Status

This is a proposal.

Although applied on private or small projects, this documentation howto should be considered as a proposal. It is not industry standard and have not been supported by some "big" projects.

However, **give it a try!** and give feedback[1].

[1] https://github.com/benoitbryon/docness/issues

CHAPTER 3

Vision

Context

In software development, when developers contribute to projects, they read and write documentation. Most people agree documentation matters. But everybody has its own culture, habits and vision about documentation. Thus it appears quite difficult to share vision about documentation and then create efficient documentation.

Goal

```
Feature: shared vision and best practices about documentation

  In order to share vision about documentation and create efficient
  documentation material
  As member of a development team
  I want to share best practices about documentation.

  Scenario: Adopt documentation-related vision and practices
    Given a project
    And a team
    When the team documents the project
    Then team members follow guidelines provided at
    http://docness.readthedocs.org/
    And reference it in the project's documentation.
```

Scope

This project mainly deals with documentation you write within a dedicated tool, i.e. not docstrings (documentation within code).

Related work

Best practices and vision are not enough. We need productivity tools:

- Coding standards. Here is a style guide for Sphinx-based documentations[2].
- Templates, snippets and content generators. Here are templates for Sphinx-based documentations[3].

[2] https://github.com/benoitbryon/documentation-style-guide-sphinx
[3] https://github.com/benoitbryon/documentation-templates-sphinx

CHAPTER 4

Usage

- Read and follow documentation best practices[6].
- Reference it in your own's project's documentation.

[6] http://docness.readthedocs.org/

CHAPTER 5

Contribute

Create tickets

Please use the bugtracker[1] **before** starting some work:

- check if the bug or feature request has already been filed. It may have been answered too!
- else create a new ticket.
- if you plan to contribute, tell us, but don't wait for us! So that we are given an opportunity to discuss, join forces or give feedback as soon as possible.

Fork and branch

- Work in forks and branches.
- Prefix your branch with the ticket ID corresponding to the issue. As an example, if you are working on ticket #23 which is about headings convention, name your branch like `23-headings`.

Download and install

System requirements:

- Python[4] version 2.6 or 2.7.

> **Note:** The provided Makefile uses `python` command. So you may use Virtualenv[5] to make sure the active `python` is the adequate one.

[4] http://python.org
[5] http://virtualenv.org

- Access to the Internet.

Execute:

```
git clone git@github.com/benoitbryon/docness.git
cd docness/
make install
```

If you cannot execute the Makefile, read it and adapt the few commands it contains in the `install` section to your needs.

Hack

They said "Eat your own dog food", so follow:

- style guide for Sphinx-based documentations[2]
- documentation best practices[6]

In your commit messages, reference the ticket with some `refs #TICKET-ID` syntax.

Test and build

Build the documentation and review your work before commit.

```
make build-documentation
```

Share

- Push your code
- Submit a pull request

CHAPTER 6

References

CHAPTER 7

Contents

Documentation usage

Guidelines about most valuable documentation usages.

Discover

- Documentation is generally used as an entry point to discover a project.
- Newcomers are heavy users of documentation.
- Documentation quality can make a difference between projects.
- Documentation gives an overview of the project.

Learn

Documentation provides guidelines to go deeper in the project. It points out interesting topics, provides tutorials, examples...

Remember, find

- Users naturally search the documentation for some content they can't remember.
- Search is an important feature.

Make durable

- Some contributors leave, others come...

- Documentation is a good place where current contributors can share with potential contributors. It's an asynchronous communication channel.

Reference

Users check the documentation when they are not sure. But keep in mind that external documentation shouldn't be the reference for everything. As an example it can't replace inline documentation such as docstrings. See *Redirect information to the right communication channel* for details.

Redirect

As an entry point, documentation is a good place to redirect users to other services or ressources. What can't be maintained in documentation is referenced in documentation with hyperlinks or other straightforward redirections.

Share "non productive" content

Some content cannot be stored in scripts, configuration or other "productive" tools. As example:

- vision
- conventions

Documentation can be used for that purpose.

Keep straightforward

Documentation has to be:

- simple to read and understand
- simple to create and maintain.

The best way to achieve this is to keep it minimal. So, make documentation straightforward.

See also:

- *Be directive*: focus on what has been proven functional.
- *Redirect information to the right communication channel*: redirect readers to external ressources.

Be directive

Let's quote the Zen of Python[1]:

- Special cases aren't special enough to break the rules.
- Although practicality beats purity.
- There should be one– and preferably only one –obvious way to do it.

Then apply it to documentation:

- documentation is the obvious place to get information.

[1] http://www.python.org/dev/peps/pep-0020/

- documentation tells the rules.
- it covers the (only) pragmatic way to go.

Don't make me think

- Sometimes users don't need to understand everything. They just want something that works. So give them a straightforward recipe that just works.
- Don't try to explain everything. You will become verbose. Users will feel bored, and if they are in some lazy mood, they may get out.
- You'd better make an user-friendly product than explain its usage in documentation.
- You'd better improve code readability than explain code in documentation.

Support driven documentation

- Focus on what have been proven functional. No matter if it is not perfect or generic. If it works, it's enough.
- Support best practices only. Refuse what breaks conventions.
- If something important is wrong with existing documentation, fix it. But don't lose time and energy to support what has already been documented.

Make choices

Don't try to create a perfect product. You can create and maintain a sufficient one. It's a matter of vision, priorities, choices... and iterations.

Write down these choices in the documentation. The adequate place to discuss and plan these choices is the ticket system (user stories, bugtracker), not the documentation.

References

Documentation is part of the product

Consider the documentation as a component of your product, i.e. don't package it as an external product or as an option.

In software development, your "product" is usually made of the software, which itself is usually based on source code.

Include documentation in production workflow

Since documentation is part of the product, include it in your production workflow:

- add it to the definition of done (see scrum development method[1])
- package and release documentation with code
- a problem in documentation is called a bug, as any problem in code
- changes in documentation can be prioritized as any feature

[1] https://en.wikipedia.org/wiki/Scrum_%28development%29

- a new version of documentation implies a new version of the product.

Synchronize code and documentation

Make sure that, for a given version of your product, you can get the adequate version of the documentation, and vice-versa.

Easiest way to do so is to manage documentation and code in the same version control repository.

Otherwise, at least create releases of documentation when you create releases of the product. And you should automate this task.

Use Sphinx instead of wikis

Several platforms, such as Github, propose a wiki service. It's a true temptation to use a wiki to host documentation:

- it's collaborative,
- it can be versionned,
- it's a good place to receive contributions. It's visible and easy to use.

So it seems suitable to serve documentation, but:

- it's easier to synchronize versions and branches of documentation and code when they live in the same repository. Will you create a branch or version of wiki for each branch or version of code?
- wiki is a kind of "live" content. It's meant to receive continuous and spontaneous contributions. It is not meant to be included in a development workflow with tickets, commits, code reviews, merges, releases and support.

So, for documentation, use tools like Sphinx[2] rather than wiki services.

On the contrary, wikis may be more suitable than "static" documentation for use cases where "live content" or "community-made content" are most valuable features:

- encyclopedia
- community articles about a web application which is live (only current production version is important).

References

Redirect information to the right communication channel

Documentation is a communication channel for development team or users. It is suitable to receive, store and send some information. But there are communication flows you should redirect to other channels.

Here are tips about content you shouldn't put in documentation. These patterns have poor value in documentation.

Scripts

Documentation should not contain list of commands. Scripts are made for that purpose.

As an example, installation procedures are usually written in documentation. But, when they become quite long, they should be simplified with scripts.

[2] http://sphinx.pocoo.org

It doesn't means that you can't write some INSTALL document. It means that INSTALL document should be easy to read and use.

Yes:

```
.. code-block:: sh

  make install
```

No:

```
.. code-block:: sh

  # Install system dependencies.
  # If you are on Debian and have sudo installed:
  sudo aptitude --without-recommends python python-dev virtualenv
  # Get the source.
  git clone git://demo@example.com
  # Create a virtual environment.
  virtualenv demo
  source demo/bin/activate
  cd demo/
  # Install Python packages.
  pip install -r requirements.txt
```

As any piece of code, scripts have to be self-documented, readable, put in the right place...

Some documentation may be generated from scripts, but not the opposite. Documentation is not the place where to maintain scripts.

Code API reference

External documentation is not the place where to maintain code API reference. Code should be self-documented.

It doesn't mean documentation cannot deal with API: it is a suitable place to provide tutorials or recipes, to focus on topics or workflows... It means that external documentation is not intended to replace inline code documentation.

As an example, Sphinx[1] users can generate API reference from code then include it to Sphinx documentation with Sphinx's autodoc extension[2].

References

Configuration

Documentation is not intended to contain configuration. Configuration is meant to be consumed by scripts or similar tools.

As code, configuration should be readable. As code, effective (production) configuration is the reference. Documentation may be generated from it, but not the opposite.

As an example, consider system architecture. In some "Architecture" document, you describe relationships between servers and clients:

- you may put some schemas or diagrams in the documentation, so that you give users an overview of the architecture.

[1] http://sphinx.pocoo.org
[2] http://sphinx.pocoo.org/ext/autodoc.html

- you shouldn't write server names in the documentation. You'd better reference the configuration of your environments.
- utilities you use for deployment, network management or monitoring should provide comprehensive views, or at least entry points so that you could generate comprehensive views.

Yes:

```
Here is an overview of the architecture:

.. image:: /_static/architecture-diagram.svg

Architecture configuration is consumed by deployment tools:

* `Production <https://example.com/deployment>`_
* `Staging <https://staging.example.com/deployment>`_

Monitoring gives you information about servers:

* `Production <https://example.com/monitoring>`_
* `Staging <https://staging.example.com/monitoring>`_
```

No:

```
In production environment:

* ``www.example.com`` is a Debian Squeeze server with, 4Go RAM and 20Go HDD.
  It serves:

  * the frontend, with Django 1.2
  * PostgreSQL server, version 8.4
  * Memcache

* ``static.example.com`` is a FreeBSD server with 256Mo RAM and 500Go HDD.
  It serves static files with Nginx.
```

Configuration will not be maintained in documentation. Thus it is to become obsolete, wrong and could lead to errors, misunderstanding... i.e. it has negative impact.

Templates

Avoid the copy-paste-adapt pattern in documentation. Replace it by interactive scripts, configuration files and templates.

Yes:

```
Configure deployment:

.. code-block:: sh

  make configure
```

Yes:

```
Use paster to generate Buildout configuration:

.. code-block:: sh
```

```
  bin/paster create -t buildout_configuration etc/buildout
```

No:

```
Copy the following content to ``settings.py`` file somewhere in your sys.path
then adapt it to your needs:

.. code-block:: python

  from myproject.default_settings import *

  DEBUG = True
```

TODO

Consider the documentation as code. If it doesn't work or it isn't "finished", it should not be committed. So avoid "maybe" or "todo" patterns in documentation:

- if resolution is important, fix it now.
- if resolution can be postponed, create a ticket (bug or task), so that it can be prioritized. Then remove "todo" from documentation.

As an example, don't use Sphinx's todo extension[1].

References

Declare conventions

The project's documentation is a good place for a team to store some conventions.

Here are some recommandations about conventions.

Feedback over conventions

Whenever you can, associate feedback to conventions. You may even replace conventions by feedback.

Code coverage example

As an example, a team could state about code coverage:

> Developers check their code tests against code coverage.

Good motivation: we have a convention.

But even if the developer checks code coverage, he cannot tell if the result is good or bad.

We need a recipe to get feedback about the convention:

> Tests must cover a minimum ratio of 80% of the code, use the `evaluate-code-coverage` command.

[1] http://sphinx.pocoo.org/ext/todo.html

Better: we can get feedback on demand.

But, it is not restrictive. It belongs to each developer to respect the convention, i.e. developers can ignore it. With time, developers may forget it... new developers won't see it... so the convention becomes unused, and the documentation obsolete.

The team can plug some code coverage tool in their continuous integration service. So that they automatically get feedback:

- they can see code coverage history: does it increase or decrease? When did it break?
- they receive a notification (or see a big red light) when code coverage is less than 80%.

Good! Everybody automatically gets feedback.

Now the whole team owns the convention: if one fails, the team has to perform something or get a big red alert. So, the convention is visible, it is obvious. Nobody can ignore it.

So we may remove the convention from the documentation. The team no longer needs to maintain the convention: it's live!

Let's compare it to the story ot the chicken and the pig:

- by writing down conventions, you get involved.
- by replacing conventions by feedback, you get committed.

More use cases

- "No test, no commit": use commit hooks or a continuous integration service to get feedback when tests get broken.
- "Coding standards": plug a "coding standards reviewer" into a continuous integration service.
- "No TODO in PROD": write a test or a commit hook.
- "No debug in PROD": write a test or a commit hook... where "debug" definition depends on your language. For Python, check "pdb" and "print".

Documentation language

Recommendation:

- Analyze documentation usage
- Choose a language for the documentation
- Write language choice and the reason in the documentation

As an example:

```
`This documentation is written in english, because english is the standard
for technical documentation about software
<http://docness.readthedocs.org/conventions/index.html#english-is-standard-for-
↪software-development>`_.
```

Keep in mind that the language of the documentation is related to documentation usage: who does read it? Who does write it?

English is standard for software development

For technical documentations related to software, english is recommended, because it's the standard in software industry. Programming languages and code are written in english. So should be their documentation.

Guidelines for non english-speaking developers

A common scenario in countries where english is not a natural language:

- a team develops software
- some team members don't speak english, or at least not well enough to be efficient with english.

So english doesn't sound natural to this team... Let's consider other points of interest, so that you can base your choice on pragmatic values.

How much is english a useful skill to develop software?

Since most software documentations, articles, tutorials are written in english, learning english is truly useful. For developers, english language is valuable.

So the question is: in the context of the project, can the team learn english? If the answer is "yes", then english remains a good choice. Some helpers:

- do team members actually are interested in learning english?
- compare estimated learning cost to estimated benefits of english skill.

Notice that, if some developers speak english, they can help others while doing code review or pair programming.

How many foreign-speaking developers contribute to the software?

Think about collaboration and maintenance. May external developers contribute to the project? Which languages could be used for communications?

The documentation language should be one in the list of possibilities.

Notice that, as a universal communication language, english has chances to be in the list. The more "international" or "open" is your project, the more english is a good choice.

How much would be a translation?

Whatever the language choice, you may have to translate the documentation one day. So, if you are not sure about the suitable language, consider the cost of translation.

If the cost is low, you may try a language now, and wonder about languages later, when necessary. Just make sure you defined "when necessary".

As an example, if your documentation is about 200 words, you'd better write it now and translate it later, if necessary, rather than learn english then write the docs. But, the question should be asked again if the documentation reaches 5.000 words.

Multiple languages

If only you have to

Support multiple languages if only you have to.

One use case where you have to support multiple languages is a end-user documentation of an international service:

- you provide a service or product in multiple languages.
- users speak several languages.
- you provide support for this product in multiple languages.
- so the end-user documentation is written in multiple languages.

Then here is an example where multiple languages are not required... You provide a service to developers. Since your service is world-famous, users speak various languages. But your interface is user-friendly and most developers are used to english... so you don't have to localize the end-user documentation.

If only you can

Support multiple languages if only you can.

Keep in mind that supporting several languages means lot of work: translations, coordination of translation teams, maintenance.

As an example, if you often release original documentation, you have to translate often, or translations would be obsolete. And obsolete (i.e. wrong or incomplete) documentation is generally more harmful than untranslated one.

Sometimes, you can't support multiple languages yourself, but commmunity can. As an example, the PHP documentation is available in several languages. The original documentation is english, then translated in several languages. In that case, the community is big enough to support the translation process. In fact, here, original documentation and its translations are managed as distinct products.

As separated documentations

When you have to support multiple languages, you'd better create distinct products (documentations) which are linked, instead of a big product which contains it all.

Several reasons:

- separate maintenance: if a documentation is broken, it doesn't block others.
- separate contributors: original authors create the reference, then translators translate. You shouldn't block the release of original content because a translation is missing.

Adopt documentation style guide

Readability counts. Many programming languages and frameworks provide coding standards, conventions or best practices to increase code readbility. It makes collaboration and maintenance easier.

Apply the same rule to documentation.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search