NZGOAL Software Extension - Guidance Note 2 (GN2)

2016-11-9

Table of Contents

## Technical practices for developers working with open source projects in government

These guidance notes cover technical practices for developers working on government maintained open source software projects or when working with external open source projects with other developers as part of a wider open source community. It follows on from the NZGOAL Software Extension open source licensing policy for the New Zealand Government as during the public consultation on this policy it was raised as a need to help with the actual technical aspects of releasing open source licensed software.

It’s important to note that alongside the technical considerations in this guidance note, you should also familiarise yourself with the communication and community management aspects of open source which are available in a separate guidance note. Part of developing an open licensed project requires the ability to engage in dialogue about your work, code out on the open, give and receive feedback and interact with contributors in public while using modern software development practices to produce and maintain the project’s code base over time.

If you wish to contribute improvements and further information to these guidance notes you can do so via the [Open Data NZ GitHub repository](https://github.com/opendatanz/nzgoal-se).

## Getting started

### Don’t repeat yourself

A key concept of using or maintaining an open source project is the "Don’t repeat yourself" (DRY) rule. Here are a few guidelines to help you through the process of finding software that already exists that you can make use off before developing from scratch.

* When building a new feature for an existing project check to see if someone has already done the same and reuse it. A quick search using the name of the software and a short description of the feature you are looking to develop will usually help you discover existing code.
* You might also search in other public software directories such as:
  + [GitHub](https://github.com/) - Social code repository
  + [GitHub Government](https://government.github.com/community/) - Find other government code releases around the world
  + [GitLab (SaaS hosted)](https://gitlab.com/explore)
  + [OpenHub](https://www.openhub.net/)
  + [The Free Software Directory](http://directory.fsf.org/wiki/Main_Page)
  + [New Zealand Open Source Society](https://git.nzoss.org.nz/explore/projects)
  + [SilverStripe Addons catalogue](http://addons.silverstripe.org/) - SilverStripe plugins (compatible with Common Web Platform)
  + [Packagist](https://packagist.org/) - PHP code libraries
  + [Ruby Gems](https://rubygems.org/) - Ruby code libraries
  + [PyPi](https://pypi.python.org/pypi) - Python code packages
* Keep an open mind, you may find that parts of your feature exist already in existing libraries. If you find part of what you intended to build ask yourself if you could reuse it in part part and build further on top of it to develop the feature you need.
* Before selecting existing code to reuse, examine how well maintained it already is:
  + Does the maintainer answer comments on the code?
  + Has there been a commit to the code recently?
  + Is there a tagged stable version?
  + Is there any documentation?
  + Are issues or support requests going unanswered?
  + It’s a good practice to base your new developments on maintained existing code, however if the code is not well maintained and you find it still solves your problem you may want to consider maintaining it yourself as a derivative version.
* In the case of an abandoned repository of code, try and at least contact the maintainer.
  + You may be able to adopt the original repository and become its new maintainer (or co-maintainer with others). This is preferable over "forking" the code (taking a copy of it and developing it in parallel).
  + The ideal is for improvements to end up in the original repository however sometimes you have no choice (for example if the code has been abandoned or not actively maintained).

### Where and how to publicly release your code?

Part of the open source process involves putting your code online in an openly accessible way (not just a zip file on your agency website). You will want to ensure there is a process for others (and your own team) to contribute improvements to this code, raise issues and control the collaborative development process. Agencies should look to make use of existing code management tools wherever possible (NZGOAL-SE advocates for this) to enable others to find and reuse the code also to avoid the operational costs of running such an ICT system. There are online repositories (both free and paid) as well as self-hosted options available (should you wish to operate your own code repository).

Some of the available options include:

* [GitHub](https://github.com/)
* [GitLab](https://gitlab.com/)(as a service or self-hosted)
* [Bitbucket](https://bitbucket.org/)
* [Common Web Platform](https://gitlab.cwp.govt.nz/explore/projects) (for New Zealand Government users of this shared service)
* [NZOSS](https://git.nzoss.org.nz/) (free community run GitLab repositories from the New Zealand Open Source Society)
* [Drupal.org](https://www.drupal.org/node/648898) (for Drupal themes and modules)

Which ever option you choose, you’ll want to ensure that it is easy for external parties to interact with. If a contributor has an important fix to your software and cannot submit it or find the contact details for you as the maintainer, then that work may never see the light of day and the benefit of community contributions of open source is lost.

#### Setting up an organisation account

When setting up a code repository account to make your public open source releases, agencies should avoid signing up for an individual user account. It is recommended to set up an organisational account (in some code repositories these are called "groups"). You can then assign access to individual developer accounts within that organisation.

Organisation account have the advantage of:

* Being able to organise a number of open source projects in one central location.
* Managing the permissions of many individual developer accounts (internal teams, contractors or external community members) to an organisation's open source projects.
* Ensure open source projects are not tied to one individual and are available into the future (even as staff come and go from the projects).

For example, the [govt.nz website team have a GitHub organisation](https://github.com/GOVTNZ) grouping all the open source licensed code they have released in one place, they assign individual developer access to these projects when needed.

When working with external contractors or across agencies, giving them read or write access to your repository is encouraged. It facilitates the flow of ideas and builds stronger community support around your open source projects.

Add the additional and removal of developer access to your staff/contractor induction and exit processes. Ideally, have someone like a team lead or IT manager that is savvy with code repository account management hold the administrator access to the organisation account so that they can add or remove staff when required (otherwise you may need to document the process and make available to your HR team). Don’t give admin rights to the organisation account to anyone but your agency staff.

#### Individual user accounts

Government and contractor developers should use individual accounts when committing their code changes and responding to contributors. They should assign their ".govt.nz" email address to the account. This helps to keep an audit trail of the work history and shows others which code has come from the official government maintainer of the project. It’s important to maintain transparency when developing code in the open and this helps everyone contributing to the project know who to contact when working with an open source project. If you already have an account with the code management tool in use for the project, rather than making a new account you can often associate your “.govt.nz” email address to your account as a secondary email.

If you are signing up for a new account:

1. Sign up and associate your ".govt.nz" email as the primary address for the account.
2. Complete your profile including your image, bio etc so others can get to know you better when working on open source projects with you.
3. If available, set up two-factor authentication to keep your account secure.
4. Request your team to be add you to the organisation account that has been set up.
5. Ensure that any notifications from the project go to your ".govt.nz“ email .
6. Respect any agency or project specific code of conduct.
7. Don’t store any sensitive information in the code repository. See "[Safe configuration practices](working-in-the-open#safe-configuration-practices)".

### What to include in your first code release

* Your software code
* A licence (it’s not open source without a licence!)
* Supporting documentation. See ["What documentation should I include with my code?](documenting-you-code#what-documentation-should-i-include-with-my-code).
* Any supporting project metadata files (note a metadata standard and discovery portal has not yet been set to catalogue NZ government open source projects. This may develop in the future, if so we’ll update these guidance notes when applicable).
* Tag a version (See [Semantic Versioning](working-in-the-open#semantic%20versioning) or best practices established already in the project or framework your work relates to).

## Working in the open

### Safe configuration practices

When releasing your code as open source software, ensure that you have not included any sensitive credentials in the publicly released code itself.

For example:

* Database logins
* API keys
* Configuration setting specific to your production deployment
* User account details

A issue here is that if you are using version control, once one of these details has been commented to the repository, it stays in the history forever (unless you remove the project history and start with a fresh commit of code but you will lose a wealth of project history information doing this).

A good practice to adhere to here is building a configuration layer into the software that you write. In your software code you can then write in references to the variables contained within the configuration file without exposing the values themselves. This make the software code much safer to share under an open source licence. You might also provide an example configuration file that users can duplicate before they configure their own copy of the software.

### Release early and often

Part of the value of developing code in the open is to 'release early, and often'[[1]](#footnote-51). When working on code in public, it is acceptable to have code that is not yet ready for production (work in progress, often [indicated by the version number](#semantic-versioning). This is how bugs can be caught and fixed early, improving the quality of the code before it is put into production.

Government open source projects have potential to be harmed by waiting for a particular event to occur before they are made publicly available. The longer the project is run closed prior to open sourcing it, the more difficult it will be to open.

At each step in a project, developers face a choice:

* do something in a manner compatible with a hypothetical future open sourcing; or
* do it in a manner incompatible with open sourcing (eg hard-coding an API key).

Every time the latter is chosen, the project gets just a little bit harder to open source. Sometimes you can't help choosing the latter occasionally. It's very difficult to give a future event the same present-day weight as, say, fixing the incoming bugs reported or finishing a new feature.

Everything in an open source project is part of the version history and exposed when you finally open source it. Cleaning this up prior to releasing takes time and resources that might have been avoided if the project had been open from the beginning.

### Version Control

Version control is like track-changes for documents but much more useful. It allows multiple people to work on the same code simultaneously and avoids having to pass the work back and forth in order for individuals to make their changes.

Using version control allows you to:

* Iteratively build up your software over time.
* Revert to an earlier version whenever you want to.
* Record your changes and the reasons why you made them, to help future developers understand the process.
* Work on changes in parallel as a team before merging them together (See [Working in branches](#working-in-branches)).
* Tag stable releases of code using meaningful version numbers (See [Semantic Versioning](#semantic-versioning))

### Distributed version control systems

[Git](https://git-scm.com/) has become one of the most widely used version control systems. Git is a distributed version (DVCS) as it allows many developers to work simultaneously on the same project code by allowing multiple, distributed copies of the work and incremental improvements to contributed to an agreed copy of the code (usually in a publicly accessible code repository).

It's a good idea to use a DVCS for government open source projects as they are useful in managing both internal work and wider external contributors using the same set of software development processes.

This is because:

* everyone involved in the process has a full copy of the code and its history
* it’s easier for developers to create ‘branches’ in their code to explore new features or approaches without encroaching on others work
* developers can continue to work when the network is unavailable, making small incremental commits, merging their changes back with everyone else’s at a later date

### Getting up to speed with Git

If you are new to Git version control we recommend working through one of the online learning courses available at:

* <https://try.github.io>
* <https://www.codecademy.com/learn/learn-git>

### Making commits

#### Make commits small and self-contained

When committing new code to your version control, make sure commits only relate to one feature or specific set of changes. Keep the commits as small as possible (while still achieving the outcome of the patch). An ideal commit is a cross-section of work including: the code for the patch, tests for the change (if relevant), documentation relating to that change. As a government open source project maintainer, set a good example in your commits and ask other developers that contribute to do the same. It will help with peer review and maintenance of the project over time[[2]](#footnote-60).

#### What makes a good commit message?

Your commit messages act as an audit trail for your code so be sure that if you looked back over these in a year's time, they would still make sense to you. More importantly, would they make sense to any new team members that may join in the future?

Write a commit message like you’re trying to explain it to your future self. Generally a good commit message contains the following[[3]](#footnote-63):

* Short subject line (max. 50 characters).
* Capitalised without a trailing full stop.
* Written in the imperative (i.e. an instruction on *how* it was done), for example "*Fix bug*" (correct) vs. “*Fixed bug*” (incorrect).
* If you need to explain more in your commit message, use the commit body text.

### Working in branches

It's a good rule of thumb to keep your "master" branch in a production ready state. At any time you should be able to safely deploy or install the "master" and have the current stable version of your software running. This means that practices such as contributing new code directly to the master branch should be avoided. Use branches and pull requests to achieve code improvements.

New features can be tested in separate branches without interrupting the main project releasability. People need the freedom to try new things without feeling like they're interfering with other's' work.There are times when code needs to be isolated from the main project branch, for example getting a bug fixed or stabilising a new release without worrying about tracking the moving target of daily development activity. Developers need to be able to review and comment on experimental work, treating branches as first-class, publishable objects makes this possible.

**Quick tips for branches**

* Use branches liberally.
* Encourage others on your team or contributing to the project externally to use them.
* Avoid long running branches, that is, build your feature in small iterations and merge back frequently.
* Watch out for branches becoming a mechanism for dividing the development community's efforts (the eventual goal of branches should be to merge changes back into the master and disappear, as soon as possible not hang around for a long time).
* Use a branching approach that doesn't add undue overhead to getting work completed. For example a large project might make use of a branching standard called "GitFlow"[[4]](#footnote-66), this might add too much undue process if the project is small (in which case you might simply use feature branches[[5]](#footnote-67) and pull requests to merge to master).

### Peer reviewing

#### Why peer review in public?

Peer reviewing is an important process for teams building software. The benefits when peer review is carried out in public are magnified as the potential pool of people available to give constructive feedback is wider. This helps in a number of ways.

* It can improve the quality of the software in a measurable way eg reduced bugs per month by a percentage because of peer reviewing.
* Keeps a record of past software architecture decisions (you can go back and discover why something was done, this is good for new team members to get up to speed on projects).
* Keeps your software releasable, less chance of avoidable bugs making it into your master branch if the peer review picks it up before merging.
* The team and wider community gets to learn about new code as it’s being written.
* Helps to establish and maintain code standards in your open source projects.
* Feedback more likely to be applied if made in public.

#### Carrying out a technical code review

There are some things you can do to make the process of technical peer review smoother (both in your team and out in the wider open source community). These include:

* Making sure you [write good commit messages](#what-makes-a-good-commit-message) to help the reviewer know what they are reviewing.
* Don’t peer review your own work, make sure to engage your team and wider open source community..
* Open a pull request as a way to initiate peer review (allows comments directly on code).
* Use "@mentions" to alert others to consider peer reviewing your change.
* Keep your reviewable code in a separate branch.
* Peer review occurs directly before anything gets merged into master.
* Break down peer reviews into small pieces to help the reviewer.
* Developers can leave comments for reviewers in the source code to help them.
* Build a culture where catch bugs is a positive thing, it’s about the software having less defects and serving users better.
* Use testing tools like [Continuous Integration](#automating-your-testing) to automate some of the review process.

#### Making peer review request in an open source project in public

The following is a high level technical walk-through the process of raising a pull request on an open source project in a public repository for peer review.

1. Find the project repository you want to contribute to.
2. Raise an "issue" outlining the feature you intend to work on.
3. Use this issues as the place to dialogue and work out details of the new proposed feature. This avoids disappointment of your patch being rejected by the maintainers.
4. ‘Fork' a copy of the code repository into your own editable public repository.
5. Checkout your copy of the code to your local development environment.
6. Make a new branch to work in for your feature.
7. Once work is completed locally, push the branch of code to your publicly accessible 'fork'.
8. Raise a 'pull request', a proposal of code changes, on the original project repository.
9. There may be further online discussion while the code is peer reviewed as part of the pull request.
10. Once agreed, the maintainer will accept the pull request to merge branch into the project and the new feature will now be available, ready for a release.

#### When someone reviews your code

* Be grateful for the reviewer's suggestions avoid discounting them outright.
* Don't take it personally. The review is of the code, not you.
* Provide reasoning for why the code exists and be open to change.
* Extract refactoring into new issues to be tackled in the future.
* Do not squash commit until the branch is ready to merge so reviewers can read individual updates based on earlier feedback.
* Seek to understand the reviewer's perspective.
* Try to respond to every comment.
* If you’re using Continuous Integration wait until it tells you the tests pass before merging.
* Merge once you feel confident in the code and its impact on the project.

#### When you review someone else's code

* Seek to understand why the change is necessary (fixes a bug, improves the user experience, refactoring of existing code).
* Communicate which ideas you feel strongly about and those you don't.
* Identify ways to simplify the code while still solving the problem.
* If discussions turn too philosophical or academic, move the discussion offline. In the meantime, let the author make the final decision on alternative implementations.
* Offer alternative implementations, and assume the author already considered them.
* Seek to understand the author's perspective.
* Sign off on the pull request with a "+1" or "Ready to merge" comment once you’re happy with the change.

### Coordinated disclosure of security issues

A coordinated disclosure is designed to improve security of software systems and protect the public. It does this by ensuring that reporters of these issues are treated respectfully when they inform agencies of issues and that both parties (your agency and the reporter) work together in a responsible manner, in good faith and within the law to correct the issue and then let others know about it so they can update their own systems.

It’s important to ensure that external contributors know how to correctly contact you if they discover a security vulnerability in your open source software. Your organisation should respond in good faith to the reporter and encourage these type of reports being made safely for all involved.

You should have a process in place for this in the form of a Coordinated Disclosure policy which informs contributors how they should let you know about security related issues (rather than publicly disclosing them) and how you will work with them to resolve the issue that they have disclosed to you.

Some suggestions when you publish your open source software:

* Read over the [New Zealand Internet Task Force Coordinated Disclosure guidance](http://www.nzitf.net.nz/pdf/NZITF_Disclosure_Guidelines_2014.pdf).
* Put in place a coordinated security policy at your agency.[[6]](#footnote-76)
* Publish your policy on your agency’s website and link to it in your software code repository documentation to ensure contributors can contact you about security related issues.
* Include a summary of what contributors should do when they find a security issue in your code in your CONTRIBUTING.md documentation alongside your code. A sample insert for this is provided below.
* Set up an email such as "security@AGENCY.govt.nz" that can be used to report issues (you may also look into ensuring this can be done securely by providing a public encryption key[[7]](#footnote-78)
* Respond promptly and in good faith to reporters of security issues (within 2 days is suggested), inform them as to what the next steps in the process are to get the issue resolved. Remember they have gone to the effort of letting you know about something important to your security and privacy, work with them to get the issue resolved and both parties should take a "do no harm" approach.
* Once you agree with the reporter that there is an issue to be fixed, keep the reporter informed as your work on the fix (a simple update email once a week to the reporter would be appropriate) to show you’re acting and haven’t gone silent on the matter. Try and stick to any timeframes you set to build trust with the reporter.
* Check your system for any signs of it being compromised and then fix the issue.
* NZGOAL-SE suggests informing other agencies known to be using your software, the CGIO, the GCPO and the Office of the Privacy Commissioner so the widest number of users get the opportunity to apply the fix to the software before it becomes publicly disclosed.
* Lastly, release your fix directly to your publicly accessible code repository (and consider applying a security patch version number as discussed in the [section on Semantic Versioning](#semantic-versioning) along with an advisory note or CHANGELOG in your project documentation informing users of the known issue and prompting them to update.
* Publicly acknowledge the reporter for helping with the fix (if they agree).

#### Communicating security disclosures to contributors

Use the following as a starting point to include in your CONTRIBUTING file regarding security issues (replace the content in upper case with your own details):

### Security Issues

Please don't file security issues in the bugtracker or publicly disclose them without first contacting us. We operate a Coordinated Disclosure policy and will work with you in good faith to responsibly fix and publicly disclose security issues appropriately.

Please read over our coordinated disclosure policy at: POLICY\_URL

#### Checklist for reporting

* Don’t disclose the issue publicly
* Document the issue to help us understand how to fix it
* Read over our Coordinated Disclosure policy
* Report the security issue to us directly via security@AGENCY.govt.nz
* Be professional and clear in your communications
* Wait to hear back from us
* Provide any follow up information if requested to help resolve the issue
* Work with us on timing and publishing of the security issue.

##### Providing encryption for secure communication

There are tools available to generate private/public key pairs, encrypt and decrypt email communication when discussing software security related issues. You will also want to look at learning how to accept encrypted mail and decrypt it at your end.[[8]](#footnote-83)

* For Windows: <https://www.gpg4win.org/>
* For Mac: <https://gpgtools.org/>
* For Linux: <https://www.gnupg.org/index.html>

If you have set up a encryption to allow reporters to security communicate with you you could also add the following in your CONTRIBUTING file (you might also include the generated public key itself in the documentation):

You can communicate securely to us by encrypting your email, please use our public key available at: PUBLICKEY\_URL.

### Semantic Versioning

In order to indicate to other developers the status of publicly released code, your project can adopt [**Semantic Versioning (SemVer)**](https://semver.org), that is, version numbers that have a meaning.

SemVer is a three point versioning open standard in the format of "X.Y.Z". For example, [CKAN](http://ckan.org/)[[9]](#footnote-90) is version [**2.5.2**](https://github.com/ckan/ckan/blob/master/CHANGELOG.rst#v252-2016-03-31) using the SemVer standard.

Let's detail what these numbers mean.

The **"Z"** refers to a “security” version which indicates a bug or security fix has been added to the code and you should look to upgrade as soon as possible. Security releases should be backwards compatible with the current minor version, that is, they should be able to be installed without interrupting any additional customisations you may have made to a minor version of the software. The security release version changes the most frequently of the 3 levels. In our CKAN example, there have been 2 security patches applied to the current released version as you can see from the number on the right of the version (X.Y.2).

The\*\* "Y"\*\* indicates a “minor” version and indicates that new features have been added and they will be backwards compatible with previous minor versions. For the CKAN example here, you can read the middle number and quickly find out that this is the 5th minor version of the software (X.5.Z) and this version should be compatible with version 2.4.0 all the way back to 2.0.0 or any future versions such as 2.6.0 for example.

The **"X"** indicates a "major" version and changes if the code undergoes a major rewrite with backwards *incompatible* changes. Major versions indicate to developers that they will need to do some work when upgrading the version of the software if they have made any customisations. From the CKAN version number on the left (2.Y.Z), you can find out that there have been 2 major versions of the software released.

Putting out CKAN example together we can quickly get an understanding as a developer that there have been have 2 major versions, 5 minor versions of the major version and 2 security patches to the latest minor version - hence version **2.5.2.**

When first releasing an open source project publicly it is common to use version 0.1.0 and begin to increment from this point forward. This indicates that there is no major stable version yet ready for production. Over time as incremental contributions are made and the code is improved, tested and deemed ready to use in projects the semantic version would increment to 1.0.0.

SemVer tagging is widely practiced standard in the wider open source software community and recommended for government open source projects to help communicate to external contributors valuable information about the project status.

### Testing in open source projects

Testing in any software project is useful as it helps ensure that software does what it is supposed to do. In context of an open source project, this is even more important as contributors to the project may be external and have less context about the inner workings of the software than the original maintainers.

While we won’t talk specifically about how to write tests for specific applications (the scope depends on your project and is too wide to cover here), manually testing can be very time consuming so it is a good practice to set up an automated suite of tests. This helps determine if any potential changes to the code will cause it stop functioning as it is supposed to *before* changes make it into the released version of the code.

This is called often referred to as Regression testing (and there are a number of other types you may want to look into for your project including Unit testing and Behavioural/functional testing).

Regression testing becomes useful as the size and complexity of your project grows. There is a trade off of the time to write the tests versus code for the application itself. Here you need to use judgement and test appropriately for the size and importance of the features in your open source project.

It is common practice in open source projects to require that contributors ensure any existing tests pass before their changes are accepted. It’s also preferred that they contribute tests for new code they write and submit. Again, in this case be careful that requiring tests with every contribution doesn’t bottleneck the process of contribution itself (you want to encourage not hinder people efforts). You might want to describe what type of code changes do or do not require testing explicitly in your project documentation so contributors know when they need to write these.

#### Automating your testing

Many existing public code repositories include or can integrate with automated regression testing tools (referred Continuous Integration or CI). These can often be set up free of charge for open source projects and help external contributors see when a change they propose might affect the software working as intended.

If you are using branches and pull requests as part of your development workflow, most CI tools run your tests whenever a new change is proposed via a branch or pull request. CI tools can also block the change from being merged into the code if the tests are failing which avoids bugs making their way into the software to begin with.

The following are some available tools often used for automated testing in open source projects.

##### Travis CI

<https://travis-ci.org/>

Travis CI integrates with GitHub repositories to run your tests automatically, whenever new code is pushed to a branch. To set up your project to work with GitLab CI you include a ".travis.yml" file in your project and then follow their setup guide at: <https://docs.travis-ci.com/user/getting-started/>

##### GitLab CI

<https://about.gitlab.com/gitlab-ci/>

GitLab CI is integrated into the GitLab code repository tool (from version 8.0 and up). It’s a good choice if you are using GitLab to host your open source project. To set up your project to work with GitLab CI you include a ".gitlab-ci.yml" file in your project and then follow their setup guide at: <https://docs.gitlab.com/ce/ci/quick_start/README.html>

##### Jenkins

<https://jenkins.io/>

Jenkins is a self hosted, open source CI tool. It’s useful when you are testing code internally or if you are hosting your own code repository tools for your projects. Assuming you have an instance of Jenkins running, to set up your project to work you include "JenkinsFile" file in your project and then follow their setup guide at: <https://jenkins.io/doc/book/pipeline/jenkinsfile/#creating-a-jenkinsfile>

{::options footnote\_nr="10" /}

## Documenting your code

Documentation is an important part of releasing your work as open source. This helps users get started, understand what your code does and how they might make use of it. Good documentation clearly answers questions that users have - either by informing or by instructing them.[[10]](#footnote-104)

### General points about technical documentation

* Document your code for your own future reference (because it’s in the open others will benefit too), chances are one day you’ll thank yourself (and your team).
* Some documentation is better than none. Leave documentation "stubs" (pages with headings of known areas not documented), this helps by providing a clear place to contribute documentation and shows you are aware of what is and isn’t complete. It is common practice in open source to accept things as incomplete as long as this is pointed out to users.
* Understand that documentation is never really finished, it will evolve as your code does.[[11]](#footnote-106)
* You cannot write a Frequently Asked Questions from scratch; as it says on the tin, they require you to actually answer questions from real users that come up often with real world use of your project. Document as you learn.
* Bundle your documentation alongside your code in a standard format ([Markdown format is useful here](#markdown)); this makes it easy to update as code is improved.
* Always include at least a README document in the root of your project.
* Talk to your users if you can to find out what they need before writing the documentation.

### Types of documentation

There are generally 3 types of documentation that either inform or instruct your users.[[12]](#footnote-108)

1. **Tutorials** - Step-by-step guides that instruct your user how to do something specific with your code. This might be a detailed walkthrough of getting started or how to accomplish a common coding task. They should be short, easy to complete (and not too easy that learning doesn’t occur) and should demonstrate what it’s like to work with your project’s code.
2. **Guides** - These will form the main bulk of your documentation. They are used to describe and inform users about the concepts and features you’ve included in your software. These might also include some code snippets to help explain concepts. Aim for completeness, a user should be able to feel like they have a good grasp on a particular aspect of your software after reading these guides.
3. **Reference material** - This details your softwares public APIs (methods and classes) that developers can make use of when adapting to suit their needs. Developers use these guides to look up the arguments and implementation specifics of your code so the audience is likely more technical. Reference material is not a substitute for good well written guides and tutorials.

### Markdown documentation

Use a open standard format when writing your documentation so that anyone can read it. A standard format widely used for documentation is called Markdown.[[13]](#footnote-110) It’s easy to learn , small in file size and captures both the content and formatting. Markdown can be used for outputting many other formats, for example you can use it to produce HTML for websites, PDF and Docx for print, or even Epub for e-readers while keeping a single source of truth, the Markdown source.

Markdown documentation is treated exactly the same as code in your version control system - you can track the changes over time, many developers are able to work on it at the same time and merge changes together once ready to release.

Markdown is a text file and saved with the file extension ".md", you write your content using plain text and use a set of plain text “marks” to represent the format of your document.

#### Example of markdown format

An example of Markdown looks like this when writing it:

# My documentation  
  
 This is a paragraph of text talking about my work and how you might make use of it as a developer. You’ll note paragraphs need no extra formatting.  
  
 \* Here I’ll make a bullet point  
  
 \* And another  
  
 \* One more  
  
 1. An ordered list of numbers  
  
 2. Another number  
  
 ## Here is a sub heading  
  
 ### And a sub, sub-heading  
  
 [Here is a link](https://ict.govt.nz)

Markdown is rendered into human readable html pages. For example the above Markdown would render as follows when viewed online as part of your code repository:

# **My documentation**

This is a paragraph of text talking about my work and how you might make use of it as a developer. You’ll note paragraphs need no extra formatting.

* Here I’ll make a bullet point
* And another
* One more

1. An ordered list of numbers
2. Another number

## **Here is a sub heading**

### **And a sub, sub-heading**

[Here is a link](https://ict.govt.nz/)

There are several [guides](https://daringfireball.net/projects/markdown/) and [cheat sheets](https://guides.github.com/features/mastering-markdown/) that show the full range of formatting options.

We recommend learning and making use of Markdown for government open source documentation and if a document copy (PDF/Docx) is required then these can be generated from the source material using several [open](https://github.com/alanshaw/markdown-pdf) [source](http://kevin.deldycke.com/2012/01/how-to-generate-pdf-markdown/) [software](https://github.com/walle/gimli) [applications](https://atom.io/packages/markdown-pdf)and [tools](http://pandoc.org/).

#### Markdown Metadata

Another great aspect of Markdown format is you can also include metadata directly within it. You might use this to provide additional information to users about the nature of the documentation contained in the document. Markdown metadata is 3 dashes followed by key/value pairs at the top of your document before you start the first paragraph, closed with a final 3 dashes. Any metadata standard can be represented, it would be suggested to use of a few commons attributes like "title" and “description” and you might also implement an existing open metadata standard such as [Dublin Core](http://dublincore.org/documents/usageguide/elements.shtml).

For example:

---  
title: My document  
description: A description about my document here this might be the same as the introduction paragraph in the body of the document.  
creator: A. Person  
publisher: Land Information New Zealand  
subject: keywords, here, with, commas,  
date: 2016-10-15  
language: en  
rights: CC BY 4.0 [https://creativecommons.org/licenses/by/4.0/](https://creativecommons.org/licenses/by/4.0/)  
---

### What documentation should I include with my code?

* **LICENSE.md** - This should include the text of your chosen open source licence (from NZGOAL Software Extension this will usually be MIT or GPL v3).
* **README.md** - This will likely be the first piece of documentation your users read, depending on the size of your project it may be the only documentation you include. See below for what to include in your readme file.
* **CONTRIBUTING.md** - This file includes information to potential contributors about any code of conduct and processes that users need to be aware of if they contribute to the project such as security disclosures and licensing considerations. The use of a contributing file is mentioned in [NZGOAL Software Extension contribution principles](https://www.ict.govt.nz/guidance-and-resources/open-government/new-zealand-government-open-access-and-licensing-nzgoal-framework/nzgoal-se/nzgoal-se-policy-principles/#contributions).
* **CHANGELOG.md** - Includes a list of changes you make to your code each time you release new versions. You might provide a link to an external changelog tracker in this file rather than store the changes directly. You can also make use of [automated changelog generator tools](https://github.com/skywinder/Github-Changelog-Generator) to produce this file.
* **docs/en/** - Your more detailed documentation should reside in a docs folder and a language code subfolder "en" for English in this example. Within this folder you can include as many Markdown pages and sub-folders needed to best structure your full documentation.

### What to include in your README file

* Name of your project.
* Short description of what the code or software does (2-3 sentences up to a paragraph).
* Requirements - of servers, infrastructure to run the software and any other software packages needed to run the code and any technical know-how such as the programming languages used.
* Installation instructions and "Getting started" section to get the software setup or configured.
* The licence the code is released for reuse under and link to the LICENSE file.
* Documentation - a small project might include this in the README directly or you might provide a link to the full documentation in your "/docs/en/" sub-folder.
* A link to your CHANGELOG file.
* Contact details for the maintainer (you might include a team email address or point of contact).
* Describe how and where users can log issues and bugs, ie in your bug tracker.
* How others can contribute to the project - include a link to your CONTRIBUTING file.

### Your first release of documentation

Documenting a new project can be daunting, ideally you can split up this task by documenting features as you code them though this is not always practical. For you first set of documentation we recommend that you limit the scope to a few key aspects:

* Your README (as described above)
* Create one tutorial that walks new users through a common task or use of the software. This helps show the user what can be done and gets them thinking about how they might use your code.
* Describe what you know is missing from the documentation (you might also created "stub" pages for these).

You don’t need to set any expectations as to when these will get written, it’s acceptable in open source to simply ensure users are aware of any shortcomings and then they can choose you use your code anyway and they are free to contribute any improved documentation.

### Structuring your documentation pages

It’s a good idea to use a consistent structure when you have documentation that spans several separate pages and topics. A skeleton structure for a section of your documentation might look like this:

* Metadata.
* Heading (descriptive enough to tell the reader what topic is covered).
* Short summary of what the reader will get out of reading the document.
* Main body of the content, using sub-headings and a style guide to make it easy to read.
* Short summary of points or review at the end.
* Optional "related" documents or link to the next topic the user should read.

### Writing style guide

To keep your documentation easy to use and consistent the use of a writing guide is a good idea. Your agency may already have its own set of writing guides in which case you might use this. Alternatively you could look to reuse the [Govt.nz writing style guide](https://www.govt.nz/about/our-style-guide/) as it is a good example of ensuring writing is accessable and in plain English while allowing the use of technical terms which are unavoidable in software documention.

**Other general points for technical writing styles are:**

* Important information goes first.
* Use an active voice - tell the reader what you want them to do or what they are learning.
* Know and write for your audience.
* Keep sentences short.
* Avoid superlatives, keep technical writing clear and concise it’s about delivering information to your users.
* Use callouts for really important information.

**Things to avoid:**

* Losing the reader by not getting to the point quickly.
* Making the reader feel stupid by using lots of jargon or technical terms without first defining them (unless you're certain of your audience).
* Not providing enough context to aid understanding.
* Not stating what the reader will learn.
* Not updating your documentation as your code changes over time.

### Licensing your documentation

Your software documentation is a separate copyright work to your software code. As such you need to license it for reuse separately. Open source software licenses should not be used for documentation.

A Creative Commons licence is more in line with existing government policy. NZGOAL Software Extension includes a [section on how to licence your software documentation](https://www.ict.govt.nz/guidance-and-resources/open-government/new-zealand-government-open-access-and-licensing-nzgoal-framework/nzgoal-se/nzgoal-se-review-and-release-process/#licensing-of-accompanying-documentation) appropriately and in line with current policy.

In summary, use Creative Commons Attribution (CC BY 4.0) licence if you have licensed your software under an MIT licences and Creative Commons Attribution Sharealike (CC BY-SA 4.0) if you have released your software under a GPL license.

For more detailed information on the open licensing of New Zealand Government copyright works see the original [NZGOAL](https://www.ict.govt.nz/guidance-and-resources/open-government/new-zealand-government-open-access-and-licensing-nzgoal-framework/)framework.

{::options footnote\_nr="14" /}

## Understanding copyright and licensing

As a developer working with open source projects, it’s important to have an understanding of copyright and licensing, how they differ and how you use them for open source release.

This section will give a simple overview of copyright, licensing and some considerations when contributing to open source projects or accepting contributions into your own government open source projects.

This section is not a substitute for robust legal advice and every edge case. For a fuller understanding of open source licensing please read over the [NZGOAL Software Extension](https://www.ict.govt.nz/guidance-and-resources/open-government/new-zealand-government-open-access-and-licensing-nzgoal-framework/nzgoal-se/).

### Copyright

Copyright is a set of property rights that automatically come into existence when you write software code. Copyright applies to the actual product of the work, not the idea to produce the work. Open source projects are grounded in copyright, there is an owner and this owner can decide what happens to the work, for example, licensing it under open source terms.

When you create work while employed or are commissioned to write software (as a contractor) then the owner of the copyright is the entity that you did the work on behalf of, you don’t personally hold the copyright in this situation.

In terms of code created in or on behalf of the New Zealand Government (the "Crown"), unless contracted out of ( that is, arrangements made in the contract to alter the default):

"the Crown is the first owner of any copyright subsisting in any work created by a person who is employed or engaged by the Crown, under a contract of service, apprenticeship, or a contract for services" (Crown Copyright, 2014).

This is called ["Crown Copyright"](https://en.wikipedia.org/wiki/Crown_copyright#New_Zealand) and means the NZ Government as a whole owns the copyright in the code (and the agencies look after it on the Crown’s behalf).

Other agencies could make use of this Crown Copyright code, however Crown Copyright doesn’t apply to some organisations in Government. Crown Entities and State-owned Enterprises for example are excluded from Crown Copyright when they create works (in this case the Crown Entity itself is the copyright owner). In this situation , in order for the widest reuse across NZ Government in the simplest way, you can license the code using open source licences as set out in NZGOAL Software Extension.

### Licensing

Licensing is related to copyright but is a distinct concept. Licensing concerns itself with two things: who "owns" the code, and who has the right to use it. The author of software code holds the code's copyright (in terms of government this is usually Crown Copyright as mentioned above), and are the copyright holder. If they would like to let others reuse the code, they need to grant them a "license" to do so.

Open source licenses grant a set of rights normally reserved for the copyright holder (hence the term "all rights reserved" you often see in association with copyright) to another person. For instance, the right to use, copy, modify, or redistribute.

This license is provided in writing along with the code (in the LICENSE file) and states the terms under which the code may be used to the world at large, you don’t have to ask permission specifically. Most common licence types are the "copyleft" and the “permissive” and there are many licences you can use that fit into these two categories.

For government open source projects we recommend the use of one of two licenses, the GPL or the MIT. See [NZGOAL Review and release process](https://www.ict.govt.nz/guidance-and-resources/open-government/new-zealand-government-open-access-and-licensing-nzgoal-framework/nzgoal-se/nzgoal-se-review-and-release-process/) which sets out in more detail the legal aspects of selecting and applying an open source licence.

### Accepting contributions from the community

Project documentation should clearly state the license under which code is distributed and that any code publicly contributed to (e.g. pull requests, code comments or patches) must be licensed under the same terms. This is not uncommon in open source projects and is referred to as "terms of contribution" in a CONTRIBUTING file alongside the code.

Under such terms, either the individual contributor will retain the copyright, and their publicly distributed code is licensed under the same terms as the code they are contributing to. NZGOAL Software Extension recommends this approach and suggests the [following insert you can use in your CONTRIBUTING file](https://www.ict.govt.nz/guidance-and-resources/open-government/new-zealand-government-open-access-and-licensing-nzgoal-framework/nzgoal-se/#contributions):

The source code for this [name of project or software] is licensed under the [insert name of applicable free and open source software licence]. By contributing source code to this [name of project or software], you are agreeing to license your contributions under and on the terms of the [insert name of same licence]. Please note that your licence is irrevocable and royalty-free. You or your licensors retain any copyright in your contributions while allowing others to re-use the source code in any way they like as long as they meet the requirements of the licence.

Alternatively, the project maintainer may request that copyright be assigned to them at such time they will license the code under the same terms as the repository so the contributor can still make use of their contributions along with all other users. Note, NZGOAL Software Extension **doesn’t recommend this second approach** as it can been seen as "over-reaching" in its position as copyright holder of externally contributed code.

Either way the copyright is managed for contributions, the rights and freedoms of the shared code remain the same, all can reuse, copy, modify and redistribute because of the overall open source licence the project is released under allows this.

### What do contractors need to know about copyright and licensing?

As a contractor doing work for New Zealand Government agencies, it's important to be aware that unless you have explicitly stated in your services contract that you will be the owner of any copyright material you produce while working for the New Zealand public sector then the agency that commissions you to do the work, owns the copyright in the work you produce for them for the duration of the contract.

This is stated in legislation under the New Zealand Copyright Act 1994:

*Where—*

1. *a person commissions, and pays or agrees to pay for ... the making of a computer program … ; and*
2. *the work is made in pursuance of that commission,—*

*that person is the first owner of any copyright in the work.*

*Subsections (2) and (3) apply subject to any agreement to the contrary.*[[14]](#footnote-145)

This is referred to as the *"commissioning rule"*

This is also mirrored in the New Zealand Government Model Contracts (GMC) that are commonly used when agencies hire contractors. The Intellectual Property section of this says:

*12.2 New Intellectual Property Rights in the Deliverables become the property of the Buyer when they are created.*[[15]](#footnote-146)

As a contractor then you need to ensure you have the agencies permission to release the code you work on under an open source licence. Remember, you need to own the copyright in order to grant freedoms under an open source licence.

### Alternative licensing options to NZGOAL-SE

[NZGOAL-SE puts forward a default set](https://www.ict.govt.nz/guidance-and-resources/open-government/new-zealand-government-open-access-and-licensing-nzgoal-framework/nzgoal-se/nzgoal-se-review-and-release-process/#stage-3-select-a-foss-licence) of well used open source software licenses for the release of government open source code. These are the GNU General Public Licence (GPL) and the Massachusetts Institute of Technology (MIT) licence. When licensing most new software as open source these two licences should cover most use cases.

There may be special cases where circumstances may require the use of an alternative open source software licence. NZGOAL-SE section 69 allows for this scenario by putting forward two alternative licences; the Lesser General Public Licence and the Affero General Public Licence; suggesting particular use cases as to when you might use these. You can also make use of any of the available open source licenses so long as you can justify why you would use these in place of the recommended default set (GPL or MIT) from NZGOAL-SE. Below are short descriptions of some of the alternative open source software licences you may opt to use and links to find out more information.

#### GNU Lesser General Public Licence (LGPL)

<https://opensource.org/licenses/LGPL-3.0>

Permissions of this copyleft license are conditioned on making available complete source code of licensed works and modifications under the same license (LGPL) or the GPL. Copyright and license notices must be preserved. Contributors provide an express grant of patent rights. However, a larger work using the licensed work through interfaces provided by the licensed work may be distributed under different terms and without source code for the larger work.

An example use of LGPL is if the source code to be released is a library of code and the agency knows that there are proprietary software suppliers who wish to use the agency's particular library within, or to link the library to, their own proprietary source code without having to release (or it being argued that they have to release) their proprietary code if they distribute their software. The GNU Lesser General Public Licence (LGPL) will allow this kind of use without exposing the downstream user to risk. The LGPL requires software licensed under it to be modifiable by end users through source code availability and does not require proprietary code that is linked to the LGPL-licensed library to be made available under the GPL.

#### GNU Affero General Public Licence (AGPL)

<https://opensource.org/licenses/AGPL-3.0>

This strong copyleft license is focused on making available complete source code of licensed works and modifications, which include larger works using a licensed work, under the same license (AGPL). Copyright and license notices must be preserved in the source code. Contributors provide an express grant of patent rights.

The sharealike obligation in the GPL (i.e., the obligation to license one's adaptations under the GPL) is triggered upon distribution of an adaptation. Where software is installed on a server to provide a service to end users (e.g., in an application service provider or cloud services context), the sharealike obligation is not triggered. To some people this seemed like a loophole.

The GNU Affero General Public Licence (AGPL) was designed to close that loophole. It is based on the GPL and states that if source code is adapted then the "public use of a modified version, on a publicly accessible server, gives the public access to the source code of the modified version".

There may be cases in which an agency is open source licensing software that is used on servers where the agency would like to achieve this result or where communities of interest ask the agency to achieve this result. If so, the agency may wish to select the GNU Affero General Public Licence (AGPL) instead of the GPL.

#### Apache Licence 2.0

<https://opensource.org/licenses/Apache-2.0>

A permissive license whose main conditions require preservation of copyright and license notices. Contributors provide an express grant of patent rights. Licensed works, modifications, and larger works may be distributed under different terms and without source code.

#### Mozilla Public Licence 2.0

<https://opensource.org/licenses/MPL-2.0>

Permissions of this weak copyleft license are conditioned on making available source code of licensed files and modifications of those files under the same license (or in certain cases, one of the GNU licenses). Copyright and license notices must be preserved. Contributors provide an express grant of patent rights. However, a larger work using the licensed work may be distributed under different terms and without source code for files added in the larger work.

#### Other license options and tools

<https://opensource.org/licenses/category>

[http://choosealicense.com/](http://choosealicense.com/licenses/)

## Creative commons attributions

* Content adapted for the NZ Government context from "[Producing open source software](http://producingoss.com/)" by Karl Fogel licensed under [CC BY-SA 4.0](https://creativecommons.org/licenses/by-sa/4.0/)
* Content adapted for general NZ Government open source guidance from "[Open Source in the NZ public sector](https://github.com/camfindlay/opensource-nzgovt)" by Cam Findlay licensed under [CC BY-SA 4.0](https://creativecommons.org/licenses/by-sa/4.0/)
* Contains public sector information licensed under the [Open Government Licence v3.0.](https://www.nationalarchives.gov.uk/doc/open-government-licence/version/3/)
* Setup and rules adapted from the "[18F Handbook](https://handbook.18f.gov/)" licensed under [CC0 1.0](https://creativecommons.org/publicdomain/zero/1.0/)
* Adapted ["Peer review"](https://github.com/thoughtbot/guides/tree/master/style) by Thoughtbot licenced under [CC BY 3.0](https://creativecommons.org/licenses/by/3.0/us/)
* Adapted "[Licenses](http://choosealicense.com/licenses/)" by [GitHub](https://github.com/) licenced under [CC BY 3.0](https://creativecommons.org/licenses/by/3.0/us/)

## Community contributors

Thanks to community contributions from:

* Rich Churcher
* David Craig

1. The Cathedral and the Bazaar, 2015 http://www.catb.org/esr/writings/cathedral-bazaar/ [↑](#footnote-ref-51)
2. See Costa, T. (2016). [Maintaining open source software.](https://thoughtbot.com/maintaining-open-source-projects-sample.pdf) [↑](#footnote-ref-60)
3. See [more detailed explanation of commit messages](http://chris.beams.io/posts/git-commit/#why-not-how) [↑](#footnote-ref-63)
4. See http://nvie.com/posts/a-successful-git-branching-model [↑](#footnote-ref-66)
5. See https://git-scm.com/book/en/v2/Git-Branching-Branching-Workflows#Topic-Branches [↑](#footnote-ref-67)
6. See the Office of the [Office of the Privacy Commissioner’s policy for a good template](https://www.privacy.org.nz/news-and-publications/guidance-resources/vulnerability-disclosure). (Available under Creative Commons Attributions 3.0 NZ). [↑](#footnote-ref-76)
7. [Introduction to email encryption] (https://support.mozilla.org/en-US/kb/digitally-signing-and-encrypting-messages). [↑](#footnote-ref-78)
8. See https://support.mozilla.org/en-US/kb/digitally-signing-and-encrypting-messages [↑](#footnote-ref-83)
9. CKAN is the application used by beta.data.govt.nz open data portal is as of Oct 2016. [↑](#footnote-ref-90)
10. See Instructing and Informing at http://slash7.com/2006/11/15/tech-writing-the-five-sins/ [↑](#footnote-ref-104)
11. See http://producingoss.com/en/producingoss.html#documentation [↑](#footnote-ref-106)
12. See https://jacobian.org/writing/what-to-write/ [↑](#footnote-ref-108)
13. See https://en.wikipedia.org/wiki/Markdown [↑](#footnote-ref-110)
14. See http://www.legislation.govt.nz/act/public/1994/0143/latest/DLM345930.html [↑](#footnote-ref-145)
15. See http://www.procurement.govt.nz/procurement/pdf-library/agencies/gmcs/gmc-form-1-services-\_-schedule-2-2nd-edition-feb-2012-update [↑](#footnote-ref-146)