Your First Patch

Prerequisites

Now that you have your LF ID, you need to install Git and git-review on your workstation.

Git

Git is a free and open source distributed version control system. See the git downloads page for how to install git on your OS.

Docs: https://git-scm.com/doc

For Debian-based systems:

```
sudo apt-get install git
```

For Centos:

```
sudo yum install git
```

git-review

git-review is a command-line tool for that is used to submit Git branches to Gerrit for review.

For Debian-based systems:

```
sudo apt-get install git-review
```

For Centos:

```
sudo yum install git-review
```

You might need the Python package manager **pip** installed on your system in order to install git-review. See the pip docs for installation instructions.

Install git-review
`pip install git-review`

Docs: git-review usage

Configure git and git-review

```
git config --global user.name "Firstname Lastname"
git config --global user.email "your_email@youremail.com"
git config --global gitreview.username gerritusername
git config --global core.editor <editor> (optional)
```

Your user.name, user.email, and gitreview.username should match the values in gerrit.

git config

Lets you get and set configuration variables that control all aspects of how Git looks and operates. These variables can be stored in three different places:

1. `/etc/gitconfig` file: Contains values for every user on the system and all their repositories. If you pass the option `--system` to `git config`, it reads and writes from this file specifically.
2. `~/.gitconfig` or `~/.config/git/config` file: Specific to your user. You can make Git read and write to this file specifically by passing the `--global` option.
3. `config` file in the Git directory (that is, `.git/config`) of whatever repository you're currently using: Specific to that single repository.

Each level overrides values in the previous level, so values in `.git/config` trump those in `/etc/gitconfig`.

Useful commands:

git config --list	Use git config --list to list all the settings Git can find
git config <key>	Use git config <key> to check the value of a specific key
git config --global --edit	This will open the global .gitconfig file for editing. The path to the file on Linux is /home/<username>/.gitconfig

SSH Key in Gerrit

You need to add your public SSH key to gerrit. Typically SSH keys are stored in your home directory, under `~/.ssh.` If you don't have one:

```
ssh-keygen -t rsa
```

Then copy the content of the public key file onto your clipboard:

```
cat ~/.ssh/id_rsa.pub
```

...and paste it into Gerrit's web interface. Login, find your the drop-down arrow to the R of your name in the upper R corner of the UI. Select Settings, then SSH Keys from the left menu.

Cloning a Repository

If you want to change code and push patches, you must clone the repo using ssh.

```
git clone ssh://<your_username>@gerrit.acumos.org:29418/<repo_name>
```

Submitting a New Patch

cd to the repo directory and create a new development branch with a short topic. The topic could be a Jira ticket number plus the subject of the patch. For example: *295/updateDocs* or *365/fixMemoryLeak.* NEVER WORK ON MASTER. ALWAYS CREATE A LOCAL BRANCH.

```
git checkout -b TOPIC-BRANCH
```

Check to make sure git-review is set up correctly to work with the remote repo:

```
git review -s
```

Once you are done with your changes, do:

```
git add -A
or
git add <list of files to be commited>
```

Check the status of your local branch to make sure you added your changes:

```
git status
```

Commit your changes:

```
git commit -s
```

Note that you haven't pushed anything to the remote repository yet. The *commit* command only does things locally.

Writing Your Commit Message

Now write your commit message:

```
1 liner with a brief description of the patch - 50 characters max
[blank]
JIRA issue associated with the format Issue-ID: ACUMOS-<jira ticket number>
[blank]
Extended description (optional but recommended)
```

Guidelines for a good commit message:

1. Separate subject from body with a blank line
2. Limit the subject line to 50 characters
3. Capitalize the subject line
4. Do not end the subject line with a period
5. Use the imperative mood in the subject line
6. Wrap the body at 72 characters
7. Use the body to explain *what* and *why* vs. *how*
    a. **Do not assume the reviewer understands what the original problem was.** When reading bug reports, after a number of back & forth comments, it is often as clear as mud, what the root cause problem is. The commit message should have a clear statement as to what the original problem is. The bug is merely interesting historical background on /how/ the problem was identified. It should be possible to review a proposed patch for correctness without needing to read the bug ticket.
    b. **Do not assume the reviewer has access to external web services/site.** In 6 months time when someone is on a train/plane/coach /beach/pub troubleshooting a problem & browsing Git history, there is no guarantee they will have access to the online bug tracker, or online blueprint documents. The great step forward with distributed SCM is that you no longer need to be "online" to have access to all information about the code repository. The commit message should be totally self-contained, to maintain that benefit.
    c. **Do not assume the code is self-evident/self-documenting.** What is self-evident to one person, might be clear as mud to another person. Always document what the original problem was and how it is being fixed, for any change except the most obvious typos, or whitespace only commits.
    d. **Describe *why* a change is being made.** A common mistake is to just document how the code has been written, without describing /why/ the developer chose to do it that way. By all means describe the overall code structure, particularly for large changes, but more importantly describe the intent/motivation behind the changes.
    e. **Read the commit message to see if it hints at improved code structure.** Often when describing a large commit message, it becomes obvious that a commit should have in fact been split into 2 or more parts. Don't be afraid to go back and rebase the change to split it up into separate commits.
    f. **Ensure sufficient information to decide whether to review.** When Gerrit sends out email alerts for new patch submissions there is minimal information included, principally the commit message and the list of files changes. Given the high volume of patches, it is not reasonable to expect all reviewers to examine the patches in detail. The commit message must thus contain sufficient information to alert the potential reviewers to the fact that this is a patch they need to look at.
    g. **The first commit line is the most important.** In Git commits the first line of the commit message has special significance. It is used as email subject line, git annotate messages, gitk viewer annotations, merge commit messages and many more places where space is at a premium. As well as summarizing the change itself, it should take care to detail what part of the code is affected. eg if it affects the libvirt driver, mention 'libvirt' somewhere in the first line.
    h. **Describe any limitations of the current code.** If the code being changed still has future scope for improvements, or any known limitations then mention these in the commit message. This demonstrates to the reviewer that the broader picture has been considered and what tradeoffs have been done in terms of short term goals vs. long term wishes.
    i. **Do not include patch set-specific comments.** In other words, if you rebase your change please don't add "Patch set 2: rebased" to your commit message. That isn't going to be relevant once your change has merged. Please *do* make a note of that in Gerrit as a comment on your change, however. It helps reviewers know what changed between patch sets. This also applies to comments such as "Added unit tests", "Fixed localization problems", or any other such patch set to patch set changes that don't affect the overall intent of your commit.

```
Summarize changes in around 50 characters or less

The commit message must contain all the information required to
fully understand & review the patch for correctness.
Less is not more. More is more.


More detailed explanatory text, if necessary. Wrap it to about 72
characters or so. In some contexts, the first line is treated as the
subject of the commit and the rest of the text as the body. The
blank line separating the summary from the body is critical (unless
you omit the body entirely); various tools like `log`, `shortlog`
and `rebase` can get confused if you run the two together.

Explain the problem that this commit is solving. Focus on why you
are making this change as opposed to how (the code explains that).
Are there side effects or other unintuitive consequences of this
change? Here's the place to explain them.

Further paragraphs come after blank lines.

 - Bullet points are okay, too

 - Typically a hyphen or asterisk is used for the bullet, preceded
   by a single space, with blank lines in between, but conventions
   vary here


Issue-ID: ACUMOS-nnnn  <the Acumos Jira ticket number>
Signed-off-by: Your Full Name <your email address in gerrit>
```

Publishing Your Patch

Now it's time to publish your patch remotely. Just type:

```
git review
```

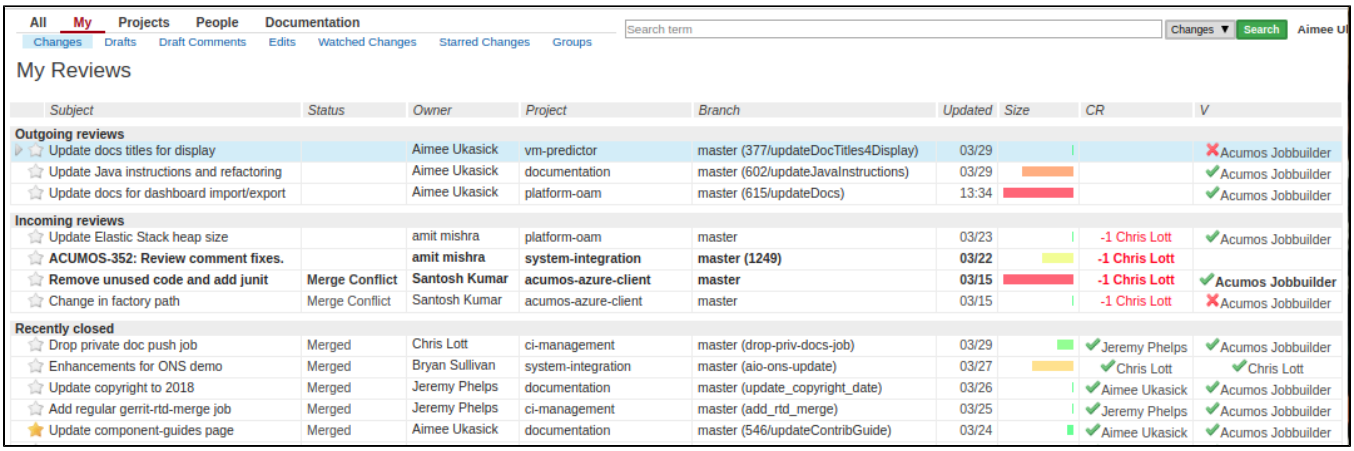
Wait a bit and you will get some output like this:

```
$ git review
remote:
remote: Processing changes: new: 1, refs: 1, done
remote:
remote: New Changes:
remote: https://gerrit.acumos.org/gerrit/1319 Add Swagger annotations
```

Viewing Your Patch on Gerrit

Gerrit is used for repository management. All code must be reviewed before it can be merged. See the Code Reviews page for info on reviewing code.

You can see your patches on the My Changes page. My Changes also lists patches on which you are a Reviewer.



People review and submit comments on your patch using the Gerrit UI.

You can make comments about your code using the "0" number. Please do not give a "+1" to your own code!

NOTE: only people who are Committers to a repo can give a +-2 but anyone, ie Contributors, can give a +-1.

Updating Your Patch

Use the same local directory on your workstation to make updates to your patch. Each new update you push to gerrit is called a *patchset*.

Make code changes then:

```
git commit -a --amend
```

This will enable you to update your commit message to reflect the changes you just made.

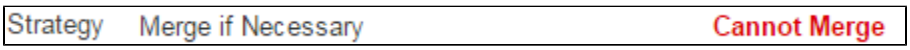
Then push your patch to gerrit:

```
git review
```

Rebasing Your Patch

Sometimes your patch cannot be merged because somebody else changed the same files while you were working on your patch. This is called a **merge conflict**. Git will detect it and will prevent your patch from being merged, which is normal.

You might get something similar to one of the following messages:



Gerrit Code Review
Change could not be merged due to a path conflict.
Please rebase the change locally and upload the rebased commit for review.

You have to resolve the conflict locally on your workstation and amend a *rebased* patch.

When that happens, do:

```
git rebase master
```

It will complain saying something similar like this:

```
CONFLICT (content): Merge conflict in mydirectory/myfile.java
Auto-merging mydirectory/myfile.java
Failed to merge in the changes.
```

Now you have to edit myfile.java or whatever file with a conflict and you will see some lines like:

<<<<<<< 1466a9e5f19c9fff0503a139b22866f716a4692b

>>>>>>> <your commit message>

This is telling you what belongs to the master branch and what belongs to yours. Resolve and modify what you need.

Now:

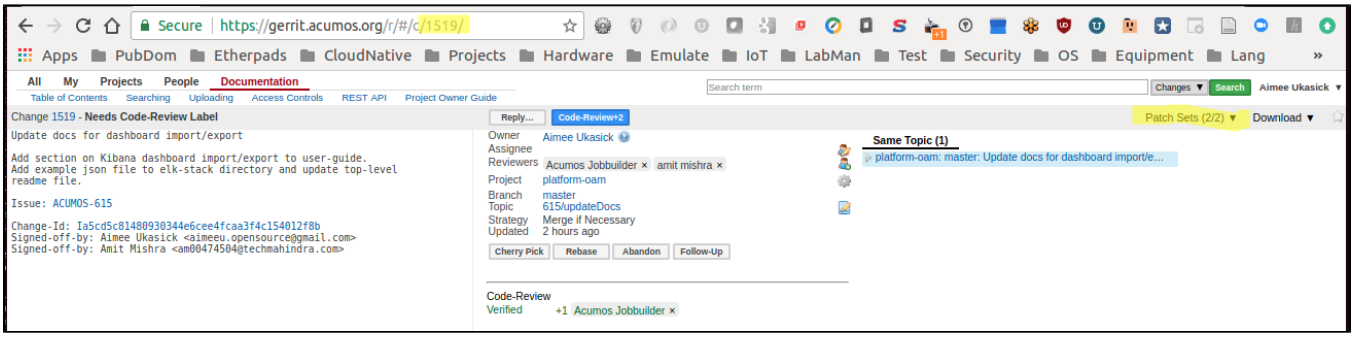
```
git add mydirectory/myfile.java  (or whatever file)
git rebase --continue
git commit --amend
git review
```

This will push a new patch with the resolved conflicts and you will be able to merge it.

Working on Someone Else's Patch.... (or Working on Your Own Patch If You Deleted Your Local Directory)

Clone the repo like you did in the Cloning a Repository step.

Then find the number of the patch you want to work on, as well as the patchset number.



Example:

```
git clone ssh://<username>@gerrit.acumos.org:29418/documentation
cd d*
git review -d 1519/2
```

git review -d <reviewNumber>/<patchsetNumber>

Now you can proceed with changing the code.

Then commit the code, **amending** the commit message:

```
git commit -a --amend
```

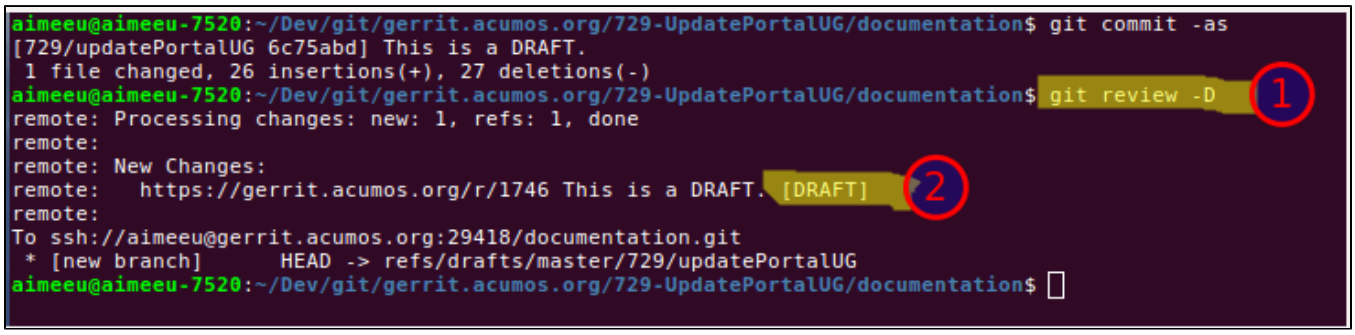
Push the change to gerrit:

```
git review
```

Submitting a Patch as a Draft

A **draft** in Gerrit is a type of change that is not intended (and probably not ready) to be merged into the production repository. Something can be created as a draft in Gerrit for the purpose of getting feedback from others or sharing a concept, testing the build environment, etc.

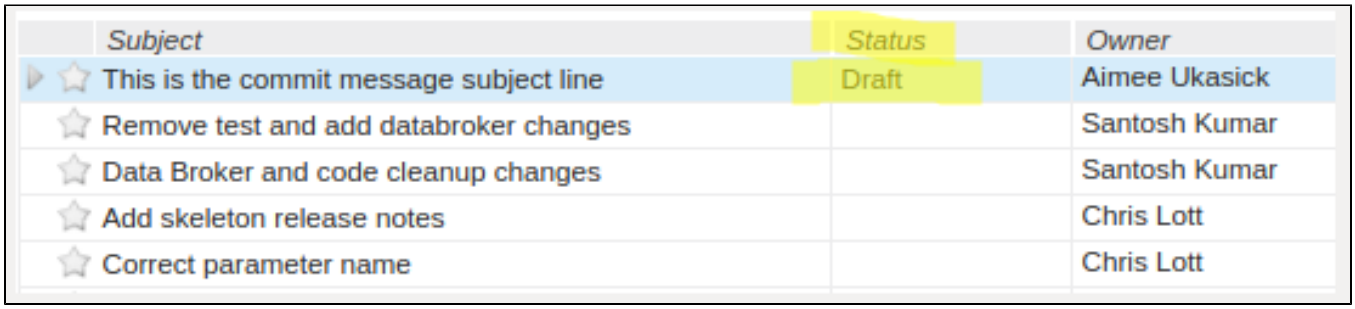
After you have cloned your repository, created your local branch, made your code changes, committed them to git with a commit message, publish your patch using **git review -D.**

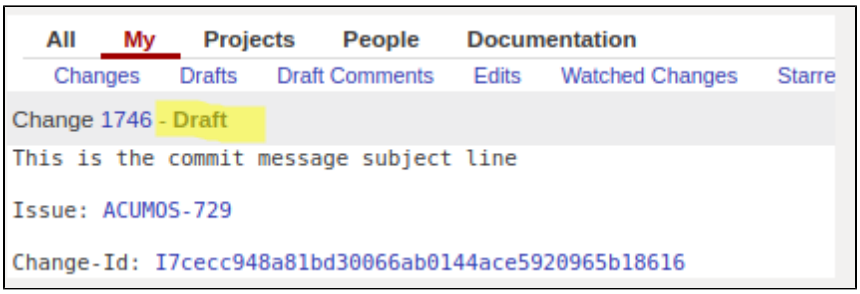


(1) git review -D to submit your patch as a Draft

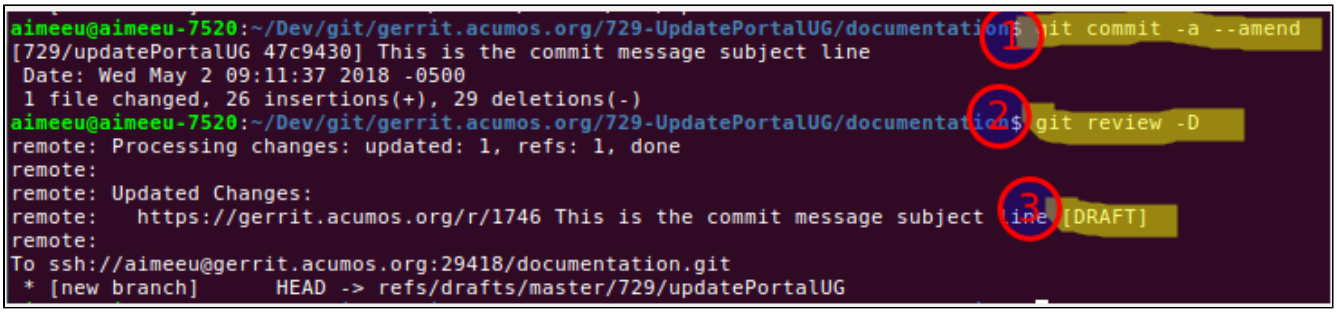
(2) notice the "[DRAFT]" tag in the output

In gerrit, your Draft patch will have a "Draft" status.





Amending a Draft Patch



(1) git commit -a --amend

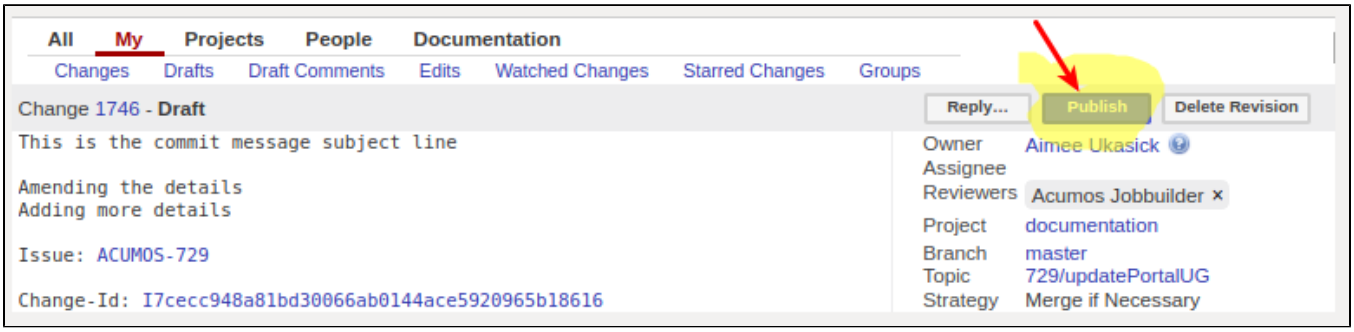
(2) git review -D to submit your patch as a Draft

(3) notice the "[DRAFT]" tag in the output

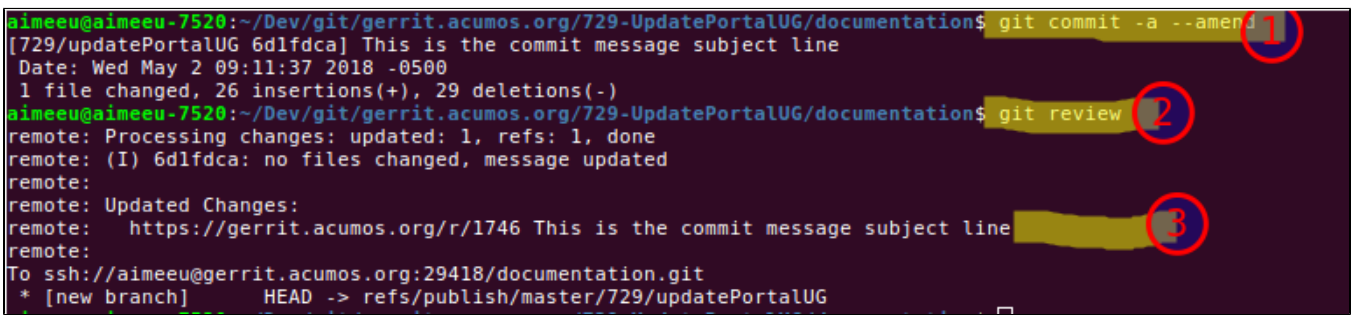
Removing Draft Status

There are two ways to move your patch from Draft to normal status:

1. Use the Publish button in Gerrit



2. Amend your patch and push to change to Gerrit without the "-D" option

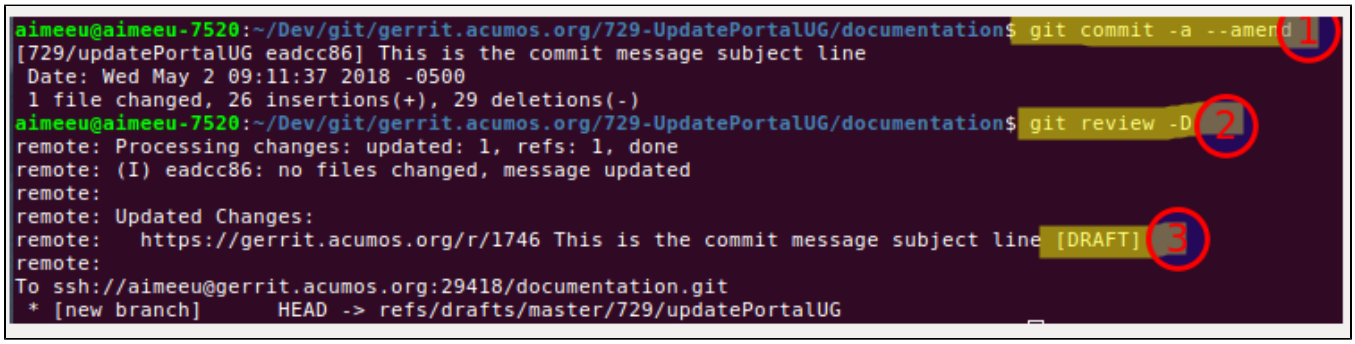


(1) git commit -a --amend

(2) git review to submit your patch

(3) notice there is no "[DRAFT]" tag in the output

Reverting to Draft



(1) git commit -a --amend

(2) git review -D to submit your patch as a Draft

(3) notice the "[DRAFT]" tag in the output