Working with Gerrit

CORD uses a code review system called Gerrit to manage all code submissions. Gerrit acts as a gatekeeper to the underlying git repository. Code is submitted to Gerrit, goes through automatic verification to check that it compiles and passes tests, and then is available for others to review in the Gerrit UI. Changes must be approved by others in the community before they can be merged into the git repository and become part of the codebase.

Before submitting changes to Gerrit, first make sure you've read through the documentation on Getting the Source Code which shows you how to clone the code over SSH, set up SSH keys in Gerrit and install the git-review tool.

Because Gerrit considers each git commit that is submitted as a separate change, it forces the user to think a little differently about how code is packed into commits before submission. This page describes the best practices for submitting changes with git and Gerrit.

Introduction to Gerrit

Terminology

Change

In Gerrit, each git commit that is submitted is a *change*. A change should be a self-contained unit of work that can be submitted into the codebase. It should be complete enough that it makes sense for reviewers to understand and comment on. The change should build properly and all tests should pass – in most repositories this will be verified by an automatic Jenkins job.

Patch set

A change can have multiple *patchsets*, which are different versions of that change. As a change goes through the review process, the developer may need to make updates and submit new versions of the change.

Change-Id

Gerrit uses a *Change-Id* to determine which git commits belong to the same change. The Change-Id is automatically inserted in the commit message by the git review tool. Different Change-Ids are treated as different changes in Gerrit, with a different set of comments and review scores for each one. If you are submitting a new patchset to an existing review, take care to ensure the Change-Id is the same.

Rebase

The Gerrit workflow makes heavy use of a git feature called a *rebase,* so it is useful to learn what this is and how to use it. A rebase allow you to take a set of commits and change the parent of them to a new parent commit. This has the effect of placing the commits at a different point in the tree of commits. For example, if you have a feature commit in a branch and you want to update that branch to pull in the lastest changes from master, you can rebase your feature branch on an up-to-date master branch. This will take the commits in the feature branch and replay them on top of the commits from the latest master branch.

Review score

Once a change is submitted to Gerrit, others in the community will review it. The following table shows the available review responses:

Response Score	Meaning
-2	There is something fundamentally blocking about this check-in (e.g. style, design). It cannot be merged as is.
-1	Some issues require addressing. Please update and submit patchset (amend commit) for re-review.
0	A comment, with no opinion on whether or not to accept the change.
+1	Change looks good, but someone else must approve.
+2	Change looks good, approved.

Note that review scores are not cumulative and don't cancel one another out. For example, if one reviewer gives -1 and another review gives +2, this doesn't mean the change has a cumulative score of +1. The -1 review will stay there until the reviewer who gave it changes their score to something else. This means if someone see a reason not to merge a change right away, that issue must be addressed to their satisfaction before the change can proceed to be merged.

Typically, if a module owner +2's a review, he/she will also submit (merge) the change also.

Verification

Most repositories have an automatic Jenkins job that will verify that the change is sane. Mostly this involves checking that it compiles and that the unit tests pass (but different repositories can have different things that are verified). If the change verifies, Jenkins will report back to Gerrit with a Verified +1 score. If not, then the change will get a Verified -1 score and the author will have to fix the issues before this change can be submitted.

Feature branches

Generally the easiest way to work with Gerrit is to develop each feature, change or bug fix in its own branch. Branches can be given sensible names and make it easy to keep track of your work if you are working on multiple things at once (which is likely even if you work serially, because you'll probably submit one patch for review then continue working on the next task while you wait for the first patch to be reviewed and merged). Working in branches also has the advantage that your master is always clean and reflects the latest master you pulled, and you never have to deal with merging commits together.

Creating a feature and submitting to Gerrit

Start with a clean, up-to-date master branch

```
$ repo checkout master
$ repo sync
```

Given that you should never need to commit on master, a pull on the master branch should always be able to simply fast-forward the local branch to be up to date with the remote master, and should never result in a merge.

Create a local feature branch in which to develop your feature

```
$ repo start feature project
```

This creates a new branch named 'feature' based off the HEAD commit of the previous branch that we were in on the project you specify. You should pick a name for the branch that describes the change you are making. The command git branch can be used to list all of the branches to check that the name isn't already used. If a branch with the desired name is no longer needed, it may be deleted with git branch -d <branchName> and the above commands repeated to re-create a new branch.

You can now make changes to the code.

Commit the changes:

```
$ git add <path_to_file>
$ git commit
```

'git commit' will open your default editor where you can write a commit message. Save the message and exit the editor to finish creating the commit.

Now if you look at your git log, you'll see the chain of commits from master followed by your one recent commit.

Send the commit to Gerrit for review

```
$ repo upload
```

Now if you browse to the Gerrit UI, you'll see that your change has been created with its first patchset.

Amending Submissions

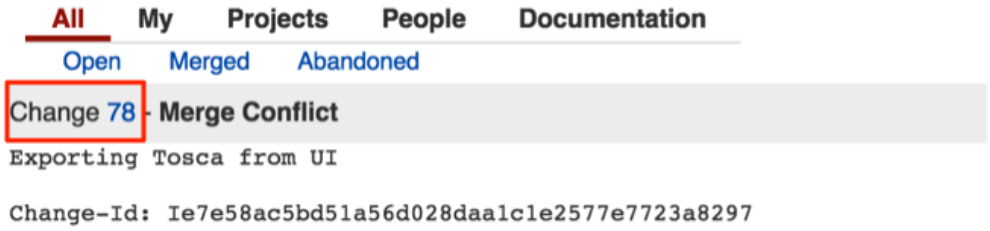
Sometimes the first patchset of a change will be submitted as-is, but often reviewers will have comments and will request changes be made. In this case, you'll want to make your changes and submit another patchset to the original change.

Switch to a local branch containing the original commit

EITHER: If you followed the previous procedure, you should still have a local branch containing your original submission. If so, you can simply check that out:

```
$ git checkout feature
```

OR: Otherwise if you no longer have your feature branch locally, you can download it from Gerrit. On the Gerrit UI you'll see that your change as a number associated with it.




```
$ repo download project 78/1
```

This will download the change from Gerrit into a local branch, and switch you to that branch. You can specify the patch set you want to download for that specific review.

Now that you are on a branch with the previous commit, you can make changes to the code.

Amend the previous commit

If you were to simply make another commit with your new changes in git now, your two commits will be two different reviews in Gerrit. This means people won't be able to see the history of the change and it will be hard to track comments across the two reviews. Instead, what you want to do is *amend* your previous commit with your new changes. This actually creates an new commit in git which is a combination of your original commit and your new changes, and because they are all packaged in one commit with the same Change-Id, Gerrit will know this is a second patchset of the original change.

```
$ git add <path_to_file>
$ git commit --amend
```

You may change the commit message if you want to, but make sure the leave the Change-Id the same.

Now if you run `git log` you'll still see your one commit on top of changes from master, but it will be amended with your new changes.

Submit the new commit to Gerrit

This is done in the same way as before:

```
$ repo upload .
```

Now if you look at the Gerrit UI, you'll see that your change has a second patchset.

Bringing in the latest changes

Sometimes you'll be working on a change and want to update your feature branch to bring in the latest changes that have been submitted since you created your feature branch. Sometimes the code in master will change in such a way that your changes can't be merged cleanly, so you need to manually resolve the differences. Or, you may simply need to use new APIs or features.

Pull the latest changes into your local master branch

First make sure that any changes you've made are checked in to your feature branch, either by amending a previous commit or making a new commit if you haven't yet made one.

Then you can switch to your master branch and pull the latest changes:

```
$ repo checkout master
$ repo sync
```

Rebase your feature branch on the latest master

Now you can switch back to your feature branch and rebase it on the master branch which now contains the latest changes:

```
$ git checkout feature
$ git rebase -i master
```

The rebase command will open an editor, which will show you a list of the commits that are in your feature branch but not in master. You can save and exit the editor to perform the rebase. If your changes cannot be rebased cleanly, git will inform you and you will have to fix the merge conflicts. You can use `git status` to see which files are in merge conflict. Once the conflicts are resolved, use `git add` to add the files and then `git rebase --continue` to finish the rebase.

Working on unsubmitted code

When you submit a change for review, it can take some time for the code to be submitted into master, because it has to be reviewed and comments have to be addressed. While the review process is happening it is still possible to continue working on a new feature that builds on the original submission. In this case we end up with dependencies between commits in Gerrit - the first commit has to be submitted before the second is able to be.

You can even use this technique to base a change off an unsubmitted change that has been made by someone else.

Create a feature branch for the new feature

You need to start from a branch that contains the commit you want to be based off of. You can either use the local feature branch if you have it or check out the code from Gerrit (see Amending Submissions for how to download changes from Gerrit). Assuming you already have a local feature branch:

```
$ git checkout feature1
$ repo start feature2
```

This creates a new branch called 'feature2' which is based off of feature1. You can then do work and commit the changes as usual.

Submit the new change to Gerrit

If the first change has not been submitted in the meantime, you can use the same git review command to submit to Gerrit.

```
$ repo upload
You are about to submit multiple commits. This is expected if you are
submitting a commit that is dependent on one or more in-review
commits. Otherwise you should consider squashing your changes into one
commit before submitting.

The outstanding commits are:

956fdfc (HEAD -> feature2) change2
4d6e325 change 1

Do you really want to submit the above commits?
Type 'yes' to confirm, other to cancel:
```

This time you'll get a warning message about submitting two changes at once. As long as the first change is still under review and hasn't yet been merged, this is OK and you can type 'yes' to confirm the submission. In this case you'll see the two changes in the Gerrit UI, and there will be a dependency between them.

However, if the first change did get submitted in the meantime, then you can't submit these two changes again because one of them is already closed (you'll get an error if you try to do this). In this case we need to pull down the latest changes in the master branch and rebase our second feature on master.

```
$ repo sync
$ git rebase -i master
```

In the interactive editor, you'll see two commits listed:

```
pick 85ee562 change1
pick d41f8e3 change2

...
```

Because our first change is already present in the master branch, we don't want to include it in the rebase. We can simply delete the first line of editor (`pick 85ee562 change1`) to drop that change and rebase only the second change. Be sure that the commit is actually included in the master when you drop a commit from a rebase. Otherwise the commit will not be attached to a branch anymore and will get lost.

Again if there are merge conflicts, they need to be manually addressed and then the rebase can be finished with `git rebase --continue`.

Now we are ready to submit just the second change to Gerrit using git review:

```
$ repo upload
```

It is possible to use this technique to create more than two dependent changes, but beware that when the chain of dependencies becomes longer it gets much harder to review and work with. Therefore, only make changes dependent on one another if you need to; if the changes aren't dependent, it's best to base them off the master branch each time to prevent long dependency chains.

Draft changes

Gerrit allows you to submit draft change, which are not public and are only visible to people who you select as reviewers on the UI. This feature can be useful for work in progress that you want to share with other people. To submit a draft change, use the following command once you've committed to your local branch:

```
$ repo upload -d
```