Contributor Guidelines

freely inspired and copied pasted from the various URLs given here

What is Gerrit ?

The Linux Foundation proposes a LF Gerrit Guide at this URL: https://docs.releng.linuxfoundation.org/en/latest/gerrit.html Please look at it.

You can find more details on Gerrit by itself at https://gerrit-review.googlesource.com/Documentation/intro-quick.html

Gerrit is a web-based code review tool built on top of the Git version control system. Gerrit makes code review easier by providing a lightweight framework for reviewing commits before they are accepted by the codebase i.e. project committers.

If you are not familiar with Git, please look at the Git appendix at the end of this Howto.

Gerrit uses massively git hooks. Git hooks are commands and scripts to add automatically specific treatments in git basic commands. **The main hook treatment provided by Gerrit is to add a "Change-Id" tag to all commit messages. This "Change-Id" allows Gerrit to link together different versions of the same change being reviewed.**

If you want to look at the hooks contents, you can retrieve them manually on the gerrit/git server with those commands .

```
$ scp -p -P 29418 gerrithost:hooks/commit-msg RecipeBook/.git/hooks/
$ chmod u+x .git/hooks/commit-msg
```

Most of the time, retrieving hooks manually is useless since they are automatically invoked by git commands. With Gerrit, it is also much easier to use git-review that invokes those hooks. **Git-review is a tool that helps submitting git branches to Gerrit for review, this means it can supersede the usual git fetch, git pull and git push commands.** The git-review package is available on most GNU/Linux distribution.

In Gerrit, several ongoing-review branches or "changes" can coexist without conflicts until they are merged to a mainstream branch (usually the master). This way, several paths for the project can be explored simultaneously even if they implement a same feature. However, if a change is getting too old, it may become impossible to merge because the mainstream branch has evolved too much meanwhile. In that case, you have to rebase the change on the mainstream branch and resolve potential conflicts before merging it.

Upload a Change

Uploading a change to Gerrit is done by pushing a git commit to the Gerrit origin server. The commit must be pushed to a ref in the refs/for/ namespace which defines the target branch: refs/for/< target-branch >. The magic refs/for/ prefix allows Gerrit to differentiate commits that are pushed for review from commits that are pushed directly into the repository, bypassing code review (this is usually a bad idea). For the target branch it is sufficient to specify the short name, e.g. master, but you can also specify the fully qualified branch name, e.g. refs/heads/master.

Push for Code Review

```
$ git commit
$ git push origin HEAD:refs/for/master

// this is the same as:
$ git commit
$ git push origin HEAD:refs/for/refs/heads/master

// or with git-review
$ git commit
$ git-review -T master
// or simply
$ git-review -T  // provided you used the default master branch with git-clone when cloning the repository
```

Push with bypassing Code Review => Beware this is usually a bad idea !

```
$ git commit
$ git push origin HEAD:master

// this is the same as: $ git commit $ git push origin HEAD:refs/heads/master
```

Upload a new Patch Set

If there is feedback from code review and a change should be improved a new patch set with the reworked code should be uploaded.

This is done by amending the commit of the last patch set. If needed this commit can be fetched from Gerrit by using the fetch command from the download commands in the change screen (right top corner).

It is important that the commit message contains the Change-Id of the change that should be updated as a footer (last paragraph). Normally the commit message already contains the correct Change-Id and the Change-Id is preserved when the commit is amended.

Push Patch Set

```
// fetch and checkout the change
// (checkout command can be copied from change screen, right top corner in download)
$ git fetch https://gerrithost/myProject refs/changes/74/67374/2 && git checkout FETCH_HEAD

//or with git-review

$ git-review -d 67374   // provided 2 is the latest Patch Set, otherwise if there is a Patch Set 3,
$ git-review -d 67374,2 //specifying a Patch-Set number only works with newer git-review versions

// rework the change
$ git add < path-of-reworked-file >
[ ... ]

// amend commit
$ git commit --amend

// push patch set
$ git push origin HEAD:refs/for/master
// or with git-review
$ git-review -T master
// or simply "$ git-review -T" provided you used the default master branch with git-clone when cloning the
repository
```

NOTE Never amend a commit that is already part of a central branch.

Pushing a new patch set will trigger an email notification to the reviewers.

The option -T is used to avoid adding a topic to the change. If no topic is specified, git-review will add the change number as a topic in Gerrit we interface.

Submitting simultaneously several changes for review : a simple example

Sometimes, it can be interesting to push simultaneously several interdependent changes for review. Here is a simple example that modifies an existing change and proposes a new change on top of it.

```
// fetch and checkout the change
// (checkout command can be copied from change screen, right top corner in download)
$ git clone https://gerrithost/myProject
$ git-review -d 67374

// rework the change 67374
$ git add < path-of-reworked-file >
[ ... ]

// amend commit
$ git commit --amend
```

```
// add a new change
$ git add < path-of-worked-file >
[ ... ]
// add a new commit
$ git commit
```

Repeat the operation as much time as necessary

```
$ git-review -T
```

git-review usually displays a warning and ask confirmation when doing this. If the changes are accepted, the Gerrit web interface will display information on changes submitted together when looking at one of them. The option -y avoids this message.

NOTE: When cascading more changes, the first call "git-review" may fail because of the absence of a change-id in the git commit message logs. Retry "git-review" in that case or try to run git hook manually to modify the git log history (not so easy). If you do not have gerrit git hooks pre-installed, this only works for the absence o Change-id in the last commit. Use interactive rebase and reword ("git-rebase -i HEAD~N") in that case.
Sometimes, it can be useful to rework dependencies or "squash" several ongoing-review changes. In that case use git interactive rebase https://backlog.com/git-tutorial/stepup/stepup7_5.html

Opendaylight and common Best Practices

All details on Opendaylight best practices are available at this URL https://wiki-archive.opendaylight.org/view/BestPractices .

Implicit rules

The first rule is that the author or at least the owner(=uploader) of the change is responsible for the code posted on the Gerrit server. This means that the author or the owner has to be responsive and must answer all questions or make all the adaptations asked by the committers in Gerrit comments. Committers are in charge of making the mainstream branch clean and conform to the project rules before merging it in the mainstream branch. Other reviews from non-committers are also welcome even if they do not meet a mandatory aspect. It may sound a little awkward but **many developers consider a "-1" review as good news as a "+1" review . Both mean someone has looked at their code and posted useful comments, potentially reusable elsewhere.** There can be several interpretations of what to do in some case and Gerrit comments can be very useful to clarify points in case of disagreements. If possible, the change uploader/owner must be the code author so that the review is more interactive and responsive.

The second rule is to keep the code posted reviewable. The change should not bring regression nor new compilation errors and warnings. It is a good idea to look at the Gerrit interface editor once your code has been posted for review. Most common errors are colored in red. Those errors pollutes the review process, not least because they generate many warnings during the compilation process. Posting a quick fix for those most common issues in a new Patch Set will ease the reviewers and committers work. If you are not confident of what you have done, you can test your change in Gerrit by using the draft mode before choosing to delete it or making it public.

Huge amounts of code are also generally difficult to review. Gerrit changes dashboard has a size indicator on the right. Bigger changes are colored in red and small ones in green. If the indicator is red, you should consider to split your change into several ones.

Coding Guidelines and common issues

More details at https://wiki-archive.opendaylight.org/view/BestPractices/Coding_Guidelines

Commit message

More details at https://wiki.openstack.org/wiki/GitCommitMessages

The commit message should reflect the feature or improvements posted in the change. The message must give a good idea of what have been done. **It must be understood by anybody with a sufficient knowledge on the topic, not necessarily someone taking part to the project.** External references are welcome to point out to more details, but they should not be a substitute to a correct description.

Here is a summary of Git commit message structure (https://wiki.openstack.org/wiki/GitCommitMessages#Summary_of_Git_commit_message_structure)

- Provide a brief description of the change in the first line.
- Insert a single blank line after the first line.
- Provide a detailed description of the change in the following lines, breaking paragraphs where needed.
- The first line should be limited to 50 characters and should not end with a period.
- Subsequent lines should be wrapped at 72 characters. There are some exceptions to this rule: for example, URL should not be wrapped. Vim (the default $EDITOR on most distros) can wrap automatically lines for you. In most cases you just need to copy the example vimrc file (which can be found somewhere like /usr/share/vim/vim74/vimrc_example.vim) to ~/.vimrc if you don't have one already. After editing a paragraph, you can re-wrap it by pressing escape, ensuring the cursor is within the paragraph and typing gqip. Put the 'Change-id', 'Closes-Bug #NNNNN' and 'blueprint NNNNNNNNNNN' lines at the very end.

NOTE: It is common practice across many open source projects using Git to include a one or several "Signed-off-by" tags (generated by 'git commit -s'). If the change has several authors, you are encouraged to use the 'Co-authored-by' tag. Bug issues can be now pointed in the commit message using the JIRA tag.

Files formatting

Files posted for review should use the UNIX/linux file format. This means that their line terminator is "\n" aka LF or LineFeed. **Other format such as MSDOS (with "\r\n" aka CRLF aka Carriage Return Line Feed terminators) should be avoided.** Encoding formats commonly accepted are Unicode and ASCII.

You can use the "file" linux command to check the actual status of your files.

```
$ file *
 transportpce-common-service-path-types.yang: UTF-8 Unicode text
 transportpce-pathDescription.yang: UTF-8 Unicode text
 transportpce-pce.yang: UTF-8 Unicode text, with CRLF line terminators
 transportpce-renderer.yang: UTF-8 Unicode text, with CRLF line terminators
 transportpce-routing-constraints.yang: UTF-8 Unicode text
 transportpce-servicehandler.yang: UTF-8 Unicode text, with CRLF line terminators
```

and combine it with find and xargs + grep to detect MSDOS file

```
$ find . | xargs file | grep CRLF
 ./networkmodel/src/main/java/org/opendaylight/transportpce/networkmodel/service/NetworkModelServiceImpl.java UTF-
8 Unicode text, with CRLF line terminators
 [..]
```

then create a script with sed to remove the "\r" special character and convert them in the UNIX format.

```
$ for file in `find . | xargs file | grep CRLF | cut -d':' -f1`; do file $file;cat $file | sed -e 's/\r$//' > /tmp
/pivot; mv /tmp/pivot $file; file $file;done
```

More easily, the vim editor can convert MSDOS file to UNIX format with ':set ff=unix' If you are on windows, do not use notepad since it only uses the MSDOS format. Consider using textpad++ or another advanced editor.

The ODL Java style guide limits the Java files line length to 120 characters and 72 or 80 chars for javadoc. It prohibits also the use of tabs. They should be replaced with 4 whitespaces. Below is a shell script to automate the operation inside a folder.

```
$ for file in * ; do cat $file | sed 's/\t/ /g' >/tmp/$file.pivot; mv /tmp/$file.pivot $file; done
```

Trailing blanks should be avoided too. Below is a shell script to remove trailing whitespaces inside a folder.

```
$ for file in * ; do cat $file | sed 's/ *$//' > /tmp/$file.pivot; mv /tmp/$file.pivot $file;done
```

Useless empty lines must also be avoided.

License issues

The EPL license or a compatible license should be present on all projects code file in the header. The maven java checkstyle plugin will check the presence of this license.

https://wiki-archive.opendaylight.org/view/BestPractices/Coding_Guidelines#General_Code_headers

License issues are considered particularly sensible by the opensource communities.

TransportPCE Gerrit guidelines and practical advices

clone the transportPCE repo and get a local copy of the code

To access opendaylight repo, you need a Linux Foundation ID to log in - See https://identity.linuxfoundation.org/

Once that done, you need to generate your SSH keys pair and publish your public on your ODL gerrit account as described in the links below: https://git.opendaylight.org/gerrit/Documentation/user-upload.html#ssh https://www.tutorialspoint.com/gerrit/gerrit_add_ssh_key_to_gerrit_account.htm

To clone the current master branch of transportPCE

```
 $ git clone ssh://< login >@git.opendaylight.org:29418/transportpce
```

To clone another mainstream branch

```
 $ git clone -b < branch_name > ssh://< login >@git.opendaylight.org:29418/transportpce
```

for example nitrogen

```
 $ git clone -b nitrogen ssh://<login>@git.opendaylight.org:29418/transportpce
```

Once that done, a particular change (i.e. a specific branch used for staging reviews before they are merged to the master branch or another mainstream branch) can be retrieved by using git-review

```
$ cd transportpce/
$ git-review -d < change_number >
```

This will create another local repo along the master branch copy. You can use git branch to verify it.

prepare a change

Once you have a local copy of the repo, you can make your modifications. Please follow the best practices given above and in the opendaylight wiki https://wiki-archive.opendaylight.org/view/BestPractices/Coding_Guidelines. Remember to check what you have done. **Be particularly careful to the license headers, the trailing blanks, the empty lines and do not use the MSDOS file format but the UNIX file format. Try also to remove compilation warnings.** If you are using an IDE , it can be a good idea to use an editor profile that implements those rule such as the Eclipse profile in this link. Since ODL compilation process is particularly verbose, consider using compilation logs file or piped commands such as:

```
$ time mvn clean install -DskipTests 2>&1 |tee ../mvn.log.0 |grep 'WARN\|ERROR'
```

If you propose an update for an existing feature with functional tests already available, it is a good idea to run those functional tests (e.g. for the pormtapping "$ cd tests/ && tox -e portmapping") and see what happens.

Once ready, use git status to check the staging files. $ git status If you want to commit all your changes, you can skip the next 2 steps and use directly "$ git commit" with the option "-a"

If not, add the right files to your commit.

```
$ git add [ ... ]
```

You might want to remove some files from the remote repo in this commit. In that case, you should use "git rm" instead of only "rm". The same way, use "git mv" if you want to rename or move a file in the remote repo too. It is a good idea to check again your git status before committing.

```
$ git status
```

Then commit and add a commit message. Using "-s" to sign-off your commit is usually a good idea

```
$ git commit -s
```

Please abide by the commit messages rules given above and at the URL https://wiki.openstack.org/wiki/GitCommitMessages ***Be careful with the title length (50 char), the empty line after the title, and the body length (72 char).*** *If your commit includes work from other contributors, do not hesitate to use the "co-authored-by" tag. If you are not the author of the changes, you can upload it although but you should use the option "--author=" with "git-commit".*

At that step, you can still rework your modifications and include more files with "git add". In that case, use "git commit --amend" after. This command also allows you to rework your commit message.

Upload your change

Once ready, you can use git-review to upload it in the remote repo for review. Simply:

```
$ git-review
```

```
If you want to upload it on another mainstream branch for review, you can add the branch name at the end.
```

```
$ git-review <branch_name>
```

for example nitrogen

```
$ git-review nitrogen
```

```
If the command succeed, you new change should be available for review in the gerrit web interface.
```

Check your change on Gerrit

Each file added, modified, moved, renamed or deleted will be listed in the Gerrit page displaying your change. If you click on a file name, the differences with the previous version of the file will be displayed. Main common errors such as trailing blanks usually appears in red. Please check every file and list those common errors. Then (or in parallel) you can go to the next steps and correct them quickly. This is a good idea to do it before asking other people to review your change in Gerrit.

Modify your change by creating a new patch set

If you have no more a local copy of your change, you can use "git clone" and "git-review -d" to retrieve it (just as described in the first section " clone the transportPCE repo and get a local copy of the code ").

Then start working on it just as you will do for a new commit with "git add/rm/mv etc...". Once ready, instead of simply doing "$ git commit -s" do instead

```
$ git commit --amend
```

You can edit the previous version of the commit message if needed, but do not change the tag "Change-id" added by git hooks. Otherwise, you will create a new change. Then upload it as usual.

```
$ git-review or $ git-review [branch_name]
```

Thanks to the change-id in the commit message, Gerrit will detect that the change was already there and that you want to create a new Patch Set to amend it. The new Patch Set should now appear in the Gerrit web interface.

Working with drafts

Sometimes, you want to test something and you do not want to share your work publicly. In that case, you can use git-review the the option **"--draft"** or **"-D"**. Drafts can be used for a new patch set or for a new change. They allow you to later "delete" you work from the Gerrit server instead of closing the associated change. Deleted changes or PS can no more be retrieved from Gerrit, contrary to closed changes. You may also change your mind and "publish" your work to make it publicly available. If so, it will no more be removable from Gerrit, just as classical changes. Meanwhile, you can add reviewers to review your draft in the Gerrit web interface. Others contributors will not be able to see your draft in Gerrit until you add them as reviewers in the interface. If you want to launch remote compilation tests with Jenkins, you can add jenkins-releng as a reviewer in your draft.

** Note you can also upload as drafts new Patch Sets in someone-else's change (i.e. a change of which you are not the owner). In that case, you will not be able to delete or publish the draft Patch Set, only the owner of the change can do it. **

Upload several changes

This can be done by cascading several commits without using git-review between them.

```
$ git add [...]

$ git commit -s

[ ... ]

$ git add [...]

$ git commit -s

$ git-review
```

If an error occur and depending on the git-review version you used, you may need to launch "git-review" twice to generate all the needed change-ids. In any case, a warning will be displayed by git-review saying that it will upload several changes and it will ask for confirmation.

Once that done, you should find in the Gerrit web interface all the changes corresponding to the commits you added . This will also create some kind of dependencies between all those change reviews in Gerrit. Additional information (Submitted together, Related Changes) will be displayed on their Gerrit pages.

Modify several changes

While they have not been merged in the remote repo, it is still possible to rework the changes you've posted simultaneously. If you have no more a local copy of them, just retrieve the latest change in you git history from the Gerrit remote repo. Check the history with

```
 $ git log
```

It should display all the commits posted.

"git commit --amend" only allows to rework the last commit. You must use another method to rework the previous commits.

The easiest way to do that is to use interactive rebase 2 syntaxes can be used:

```
$ git rebase -i < commit >
```

where <commit> is the commit hash reference used by "git log"

or

```
$git rebase -i HEAD~< number of commits >
e.g. "$git rebase -i HEAD~5" to rework the five previous commit
```

you should now see commits short descriptions in a text editor (usually vim) It should look like this.

```
pick 239da71 Renderer and OLM update
pick f85398e Bugs correction in Portmapping
pick 6cb0144 Minor checkstyle corrections
pick e51e0b9 Network topology and inventory init
pick f245366 Bugs correction in NetworkModelService

# Rebase afe9fcf..f245366 onto afe9fcf
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

The editor allows you to proceed to several actions on the git history: - remove a commit from the history: just delete its line - rework dependencies: swap line orders - meld several commits into one: replace "pick" by "squash" or "fixup" - rework only a specific commit message: replace "pick" by "reword" - rework a specific commit: replace "pick" by "edit" then "git add/rm/mv ..." , "git commit --amend", "git rebase --continue"

Beware you may have to deal with potential conflicts when doing this.

Alternate methods exist to do this. For example, you can use non-interactive git rebase , i.e without the option "-i". But you must keep a copy of the original "git log" history. Once the copy made, use

```
$ git checkout < commit_hash >
```

where < commit_hash > is the hash of a previous commit, let' say N commits before the last one. Do your modifications:

```
$ git add/rm/mv [...]
$ git commit --amend
```

A new commit hash (< newhash >) will be generated. Keep it.

```
$ git checkout < commit_hash-1 >
```

where < commit_hash-1 > is the hash of the previous commit, N-1 commits before the last one. If you look at "git log", the history has not changed and the old hash is still there. you need to rebase to apply the modifications made in the previous commit.

```
$ git log
$ git rebase < newhash >
$ git log
```

Conflicts may appear but should be solveable. Proceed the same way with the N-2 previous commits up to the last commit. Then upload

```
$ git-review
```

Cherry-picks / backports

Cherry-pick consist in importing the content of a specific change (or commit) from another (review) branch into your own local branch.

The basic git cherry-pick method is described at this URL https://backlog.com/git-tutorial/stepup/stepup7_4.html

The principle remains the same with Gerrit but you have to deal with the Gerrit branch review system. You can use the "git cherry-pick" classical command. In that case, you'd better to copy/paste it from the right-top corner of the change review page. The easiest option is to use git-review with the option "-x" instead.

```
 $ git-review -x < change_number >"
```

You can use also "-X" to keep a trace of the cherry-pick operation in the git log. The "-N" option prepare the cherry-pick but the commit message is not imported.

Several cherry-picks can be cascaded this way.

Once the change appears in your local branch, you can upload it to the gerrit remote repo as usual with git-review.

Cherry-pick can also be used to backport changes between several mainstream branches of the gerrit remote repo. The procedure is the same. Here is an example.

```
 $ git clone -b nitrogen ssh://< login >@git.opendaylight.org:29418/transportpce $ git-review -x 66657 //Change
66657 is on the ATT-Sandbox branch and not the nitrogen branch $ git-review [--draft] [nitrogen]
```

Working with several remote git repositories

Git allows to work with several remote repositories simultaneously (https://backlog.com/git-tutorial/reference/remote.html#sec2) The cherry-pick operation can be used to import/export new features or bug corrections between several remote repositories. **When you work simultaneously on a projet and its fork (for example a private corporate repo), you may have to deal with 2 different remote repositories. In that case, it is a good practice to sync the 2 repo regularly in order to benefit from new features/bug corrections and also to avoid bringing regressions when you share the work you have done on a repo to the other one.** Cherry-pick works but is not the best option in that case. The administrator of the forked repo may consider using autosyncing.

Resolving conflicts

Conflict resolution in Gerrit is not different from Git. For more details, you can look at the Git tutorial at those URLs: https://backlog.com/git-tutorial/intro/intro5_1.html https://backlog.com/git-tutorial/intro/intro5_2.html https://backlog.com/git-tutorial/intro/intro6_1.html https://backlog.com/git-tutorial/intro/intro6_2.html https://backlog.com/git-tutorial/stepup/stepup2_7.html

Conflict can occur during Git merges, pushes or rebases. For example, if two or more members make changes on the same part of a file in a remote and a local branch that are being merged, Git will not be able to automatically merge them and you will get a merge conflict. When this happens, conflicting files will be listed in the resulting messages as in the example below.

```
$ git merge issue3
Auto-merging myfile.txt
CONFLICT (content): Merge conflict in myfile.txt
Automatic merge failed; fix conflicts and then commit the result.
```

And Git will add some standard conflict-resolution markers to those conflicting files. The markers act as an indicator to help us figure out sections in the content of the conflicting file that needs to be manually resolved.

Example of a conflict occurrence

```
Git commands reminder
add: Register a change in an index
<<<<<<< HEAD
clone: Clone a repository into a new directory
commit: Save the status of an index
=======
clone: Clone a repo into a new folder
pull: Obtain the content of a remote repository
>>>>>>> issue3
fetch: Download objects and refs from another repository
```

Each conflicting section in the file is delimited by lines alike "<<<<<<< HEAD" and ">>>>>>> issue3" . When merging remote code into your local branch, everything above " ======== " is your local content, and everything below it comes from the remote branch. Before going further, we need to resolve the conflicting parts and removes those markers as shown in the example below.

```
Git commands reminder
add: Register a change in an index
clone: Clone a repository into a new directory
commit: Save the status of an index
pull: Obtain the content of a remote repository
fetch: Download objects and refs from another repository
```

Once we are done with resolving the conflict, you can commit the change (git commit -m) , or pursue a rebase if you were in a rebasing process.

Git appendix

Git basics

Git very basics are available on the on-line tutorial below. https://backlog.com/git-tutorial/intro/intro1_1.html Please look at it if you are beginning with git.

It is important to understand the concept of local and remote repositories in git. Most of your work is done locally and once ready it is usually shared remotely on a git server. Contrary to older control systems such as CVS and SVN, several local and remote repositories can be used simultaneously with git. This allows to incorporate in your work the contributions from 2 or more remote independent repositories, and optionally to synchronize those remote repositories.

The other important notion to understand is the concept of branch. https://backlog.com/git-tutorial/stepup/stepup1_1.html

In a collaborative environment, it is common for several developers to share and work on the same source code. Some developers will be fixing bugs while others would be implementing new features. Therefore, there has got to be a manageable way to maintain different versions of the same code base. This is where the branch function comes to the rescue. Branch allows each developer to branch out from the original code base and isolate their work from others. Another good thing about branch is that it helps Git to easily merge the versions later on.

Although, merges (and rebases) can be tricky operations. Git can solve automatically many conflicts when merging 2 branches, but not all of them. In the case a conflict can not be resolved automatically, you have to solve it manually according to the procedures in the URLs below. https://backlog.com/git-tutorial/stepup/stepup2_7.html https://backlog.com/git-tutorial/intro/intro5_2.html

Git practical advices

What to remind and useful practical advices are listed on the URL https://gist.github.com/blackfalcon/8428401. Here are some of them.

Basic branching

When working with a centralized git workflow (e.g. gerrit or github) the concepts are simple, master represented the official history and is always deployable. With each new scope of work, aka feature, the developer is to create a new branch. Gerrit usually automatically creates a branch for each review before it is merged. Gerrit branch names for review are generated according gerrit rules. If you use another workflow that does not specify a branch naming policy, make sure to use descriptive names for your branches.

Before you create a branch, be sure you have all the upstream changes from the origin/master branch (or the other remote branch you are forking from). The following command is good to know to list out the branches you have locally as well designate which branch you are currently on.

```
$ git branch
```

The checked out branch will have a * before the name. If the return designates anything other then master then switch to master

```
$ git checkout master
```

Once on master and ready to pull updates, I use the following:

```
$ git pull origin master
```

The git pull command combines two other commands, git fetch and git merge. When doing a fetch the resulting commits are stored as remote branch allowing you to review the changes before merging. Merging on the other hand can involve additional steps and flags in the command, but more on that later. For now, I'll stick with git pull.

Now that you are all up to date with the remote repo, you'll create a branch. For efficiency, you will use the following:

```
$ git checkout -b my-new-feature-branch
```

This command will create a new branch from master as well checkout out that new branch at the same time. Doing a git branch will list out the branches in my local repo and place a * before the branch that is checked out.

```
  master
* my-new-feature-branch
```

Do you have to be on master to branch from master?

No. There is a command that that allows me to create a new branch from any other branch while having checked out yet another branch.

```
$ git checkout -b transaction-fail-message master
```

In that example, say I was in branch github-oauth and I needed to create a new branch and then checkout the new branch? By adding master at the end of that command, Git will create a new branch from master and then move me (checkout) to that new branch.

This is a nice command, but make sure you understand what you are doing before you do this. Creating bad branches can cause a real headache when trying to merge back into master.

Branch management

As I am working on my new feature branch, it is a good idea to commit often. This allows me to move forward without fear that if something goes wrong, or you have to back out for some reason, I don't lose too much work. Think of committing like that save button habit you have so well programed into you.

Each commit also tells a little bit about what I just worked on. That's important when other devs on the team are reviewing my code. It's better to have more commits with messages that explain the step versus one large commit that glosses over important details.

Commit your code

As I am creating changes in my project, these are all unseated updates. With each commit there most likely will be additions, and there will also be deletions from time to time. To get a baring of the updates I have made, lets get the status.

```
$ git status
```

This command will give you a list of all the updated, added and deleted files.

To add files, I can add them individually or I can add all at once. From the root of the project I can use:

```
$ git add .
```

In order to remove deleted files from the version control, I can again either remove individually or from the root address them all like so:

```
$ git add -u
```

I'm lazy, I don't want to think, so the following command I make heavy use of to address all additions and deletions.

```
$ git add --all
```

All the preceding commands will stage the updates for commitment. If I run a git status at this point, I will see my updates presented differently, typically under the heading of Changes to be committed. At this point, the changes are only staged and not yet committed to the branch. To commit, do the following:

```
$ git commit -m "a commit message in the present tense"
```

It is considered best to illustrate your comment in the tense that this will do something to the code. It didn't do something in the past and it won't do something in the future. The commit is doing something now.

A bad example would be:

```
 $ git commit -m "fixed bug with login feature"
```

A good example would be:

```
 $ git commit -m "update app config to address login bug"
```

Comments are cheap. For more on how to write expressive commit messages, read 5 Useful Tips For A Better Commit Message.

Push your branch

When working with feature branches on a team, it is typically not appropriate to merge your own code into master. Although this is up to your team as how to manage these expectations, but the norm is to make use of pull requests on github or change reviews on Gerrit. Pull requests require that you push your branch to the remote repo.

To push the new feature branch to the remote repo, simply do the following:

```
 $ git push origin my-new-feature-branch
```

As far as Git is concerned, there is no real difference between master and a feature branch. So, all the same Git features apply.

My branch was rejected?

This is a special case when working on a team and the branch I am are pushing is out of sync with the remote. To address this, it's simple, pull the latest changes:

```
 $ git pull origin my-new-feature-branch
```

This will fetch and merge any changes on the remote repo into my local branch with the changes, thus now allowing you to push.

Working on remote feature branches

When I am are creating the feature branch, this is all pretty simple. But when I need to work on a co-workers branch, there are a few additional steps that I follow.

Tracking remote branches

My local .git/ directory will of course manage all my local branches, but my local repo is not always aware of any remote branches. To see what knowledge my local branch has of the remote branch index, adding the -r flag to git branch will return a list.

```
 $ git branch -r
```

To keep my local repo 100% in sync with deleted remote branches, I make use of this command:

```
 $ git fetch -p
```

The -p or --prune flag, after fetching, will remove any remote-tracking branches which no longer exist.

Switching to a new remote feature branch

Doing a git pull or git fetch will update my local repo's index of remote branches. As long as co-workers have pushed their branch, my local repo will have knowledge of that feature branch. By doing a git branch you will see a list of my local branches. By doing a git branch -r I will see a list of remote branches. There is a good chance that the new feature branch is not in my list of local branches.

The process of making this remote branch a local branch to work on is easy, simply checkout the branch.

```
 $ git checkout new-remote-feature-branch
```

This command will pull it's knowledge of the remote branch and create a local instance for me to work on.