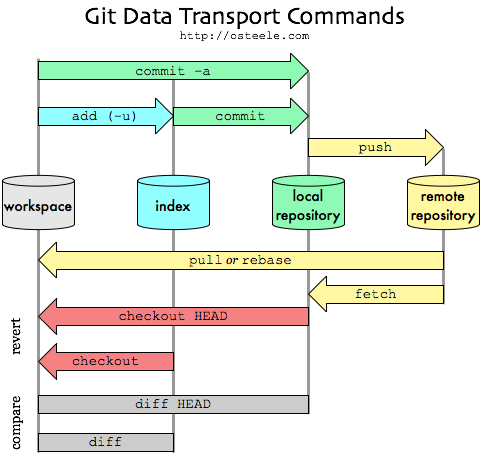
# Git Workflow



Where:

* **Workspace**: The directory tree of (source) files that you see and edit on your **local PC**.
* **Index** (aka *staging area* or *directory cache*): a single binary **file** (/.git/index), which lists all files in the current branch, their names, SHA1 checksums, and time stamps.
* **Local repository**: A **hidden directory** (.git) including an objects directory containing all versions of every file in the repo (local branches and copies of remote branches) as a compressed "blob" file.
* **Remote repository**: A repository hosted on the **Internet** or elsewhere. To be able to collaborate on any Git project, you need to know how to create and manage your remote repositories.

Note: Remote repositories can be on your local machine. It is entirely possible that you can be working with a “remote” repository that is, in fact, on the same host you are. The word “remote” does not necessarily imply that the repository is somewhere else on the Internet; only that it is elsewhere.

* **HEAD**: A reference to the **last commit in the currently checkout branch**. When you checkout to a branch, the HEAD points to the latest commit in this branch. However, in the same branch, when you checkout to a commit which is not the latest one, the HEAD point to this commit. So, note that the HEAD doesn’t mean the latest.

**NOTE**

**The way Git works on Linux and Windows is different** due to the incompatibility among different OSs. If you find weird things when using git status (and other related commands), please check which OS the repository (you’re using) is set to work with and ensure to clone this repository in the correct OS.

More details [here](https://stackoverflow.com/questions/54410355/git-status-between-windows-and-linux-does-not-agree) and [here](https://github.com/microsoft/WSL/issues/184).

# How to Clone an Existing Repository

## Cloning

Assume you just created a repo on Github, or someone just gave you a URL to a remote repo, you can clone it to your local PC using:

# cd to the directory which will be your local repo

$ git clone https://github.com/triho1110/master

**Note**: With the above Github link, we're cloning with HTTPS. We can also clone with SSH. To know the differences between these two techniques, check [here](https://help.github.com/en/github/using-git/which-remote-url-should-i-use).

To check info of the cloned repo, use:

$ git remote -v

origin https://github.com/triho1110/master (fetch)

origin https://github.com/triho1110/master (push)

It lists the name and URL of all remote repositories you’ve worked on. In all cases, you will see a repo called "origin" — that is the default name Git gives to the remote you cloned from.

## Tips

### How to clone big repositories

<http://gci2017fossasia.blogspot.com/2017/11/how-to-clone-biggest-repositories.html>

<https://stackoverflow.com/questions/34389446/how-do-i-download-a-large-git-repository>

<https://community.atlassian.com/t5/Bitbucket-questions/clone-depth-does-what-Why-do-I-care-about-this-setting/qaq-p/496247>

# How to Create Your Own Private Git Server

Prerequisites:

* Your server machine uses Linux OS.
* You already installed Git on both local and server machines.
* You already installed openssh-server on the server machine, and openssh-client on the local machine (if Linux) or PuTTY (if Windows). ?????
* You know how to connect to the server from your local machine via SSH protocol (because you have two machines to work on – local and server). If you don't, check this [guide](https://www.howtogeek.com/311287/how-to-connect-to-an-ssh-server-from-windows-macos-or-linux/).

Git supports several communication protocols (more details [here](https://medium.com/datadriveninvestor/git-good-git-server-protocols-for-beginners-d9a02b314bbe)). In this guide, we're using[**SSH protocol**](https://www.ssh.com/ssh/protocol).

## Steps to Create a New Private Git Server on Linux

**1. Set up user access**

Check section "[How to Manage Users Access to Git Server](#_How_to_Manage)".

**2. Initialize Git remote repo on the server machine**

Create a new folder for your project:

server-user@server: $ mkdir -p /home/git/project-1.git

Change to this folder and initialize a bare repo (the repo without a working directory):

server-user@server: $ cd /home/git/project-1.git

server-user@server: $ git init --bare

**Note**: Because of being a [bare repo](https://mijingo.com/blog/what-is-a-bare-git-repository), we cannot pull or push anything while being on it. This is absolutely normal so nothing to worry about.

**3. Initialize Git repo on the local machine**

Create a Git repo on the local machine:

local-user@local: $ mkdir -p /home/git/project

Change to this directory and initiate an empty repo:

local-user@local: $ cd /home/git/project

local-user@local: $ git init

**4. Commit and push changes to the server from the local machine**

Create a readme file (to be simple, let's make it empty) and add it to the repo.

local-user@local: $ touch readme.txt

local-user@local: $ git add .

Commit it:

local-user@local: $ git commit -m "Add empty Readme.txt"

**Note**: You have to create a file to be able to commit to Git. Otherwise, error "*error: src refspec master does not match any*".

Now you need to push your local changes to the server so that the work can be accessible over the Internet and you can collaborate with other team members.

First assign the repo URL to the origin:

local-user@local: $ git remote add origin server-user@server-IP-or-hostname:/home/git/project-1.git

// Or

local-user@local: $ git remote add origin ssh://server-user@server-IP-or-hostname/home/git/project-1.git

Then push changes to the remote server:

local-user@local: $ git push origin master

**5. Clone the repo from the server (by other team member)**

If there are other team members who want to work with the project, they can now clone the repo on the server to their local machine:

member-user@local: $ git clone server-user@server-IP-or-hostname:/home/git/project-1.git

// Or

member-user@local: $ git clone ssh://server-user@server-IP-or-hostname/home/git/project-1.git

Now this member can pull or push changes to the remote server.

## How to Manage User Access to Git Server

If you want some repositories to be read-only for certain users and read/write for others, you need to manage access permissions for each user.

Suppose several teams in your company use the same server machine. You're the one who manage that server and you don't want other team to have the access to the private repo another team is working on. There are a few ways you can manage repo access:

### Way #1: Set up an account for each team

This way is straightforward but can be cumbersome. You may not want to run adduser (or the alternative useradd) and have to set passwords for every new team.

### Way #2: Set up SSH access for each team

Create a single user account (e.g., named git) on the server machine, ask every user who is to have access to send you an SSH public key, and add that key to the ~/.ssh/authorized\_keys file of the git account.

First, create a git user account and a .ssh directory for that user:

$ sudo adduser git

$ su git # Switch to the 'git' user

$ cd # Go to the home dir of the 'git' user

$ mkdir .ssh && chmod 700 .ssh # Only owner can read, write and execute

$ touch .ssh/authorized\_keys

$ chmod 600 .ssh/authorized\_keys # Only owner can read and write

Next, add some developer SSH public keys to the authorized\_keys file (if you don't know how to generate SSH private and public keys, check [here for Windows](https://www.ssh.com/ssh/putty/windows/puttygen) or [here for Linux](https://www.ssh.com/ssh/keygen)). Let’s assume you have some trusted public keys and have saved them to temporary files. Again, the public keys look something like this:

$ cat /tmp/id\_rsa.john.pub

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCB007n/ww+ouN4gSLKssMxXnBOvf9LGt4L

ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4k

Yjh6541NYsnEAZuXz0jTTyAUfrtU3Z5E003C4oxOj6H0rfIF1kKI9MAQLMdpGW1GYEIgS9Ez

Sdfd8AcCIicTDWbqLAcU4UpkaX8KyGlLwsNuuGztobF8m72ALC/nLF6JLtPofwFBlgc+myiv

O7TCUSBdLQlgMVOFq1I2uPWQOkOWQAHukEOmfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPq

dAv8JggJICUvax2T9va5 gsg-keypair

You just append them to the authorized\_keys file in the git user's .ssh directory:

$ cat /tmp/id\_rsa.john.pub >> ~/.ssh/authorized\_keys

$ cat /tmp/id\_rsa.josie.pub >> ~/.ssh/authorized\_keys

$ cat /tmp/id\_rsa.jessica.pub >> ~/.ssh/authorized\_keys

Now all these users (john, josie, jessica) have read/write access to the Git server.

**[Optional] Restrict the git user to use *git shell* only (instead of other types of shell - *bash shell*, *dash shell*, etc.)**

You should note that john, josie or jessica can also log into the server and get a bash shell or dash shell as the git user. You can easily restrict the git user account to only Git-related activities with a shell tool called [git-shell](https://git-scm.com/docs/git-shell) that comes with Git. If you set this as the git user account’s login shell, then that account can’t have normal shell (bash, dash, etc.) access to your server.

**Note**: This is not about security, but it's about clarifying user management (which purpose the git user account is used for and NOT used for).

The idea is to specify git-shell instead of bash or dash for that account’s login shell. To do so, you must first add the full pathname of the git-shell command to /etc/shells if it’s not already there:

$ cat /etc/shells # see if git-shell is already in there. If not...

$ which git-shell # make sure git-shell is installed on your system.

$ sudo -e /etc/shells # and add the path to git-shell from last command

Now you can change the shell for the git user:

# Standing on the root user and run:

$ sudo chsh git -s $(which git-shell)

Now, the git user can still use the SSH connection to push and pull Git repositories but can’t shell onto the machine. If you try, you’ll see a login rejection like this:

$ ssh git@server

fatal: Interactive git shell is not enabled.

hint: ~/git-shell-commands should exist and have read and execute access.

Connection to server closed.

At this point, users are still able to use SSH port forwarding to access any host the git server is able to reach. If you want to prevent that, you can edit the authorized\_keys file and prepend the following options to each key you’d like to restrict:

no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty

The result should look like this:

$ cat ~/.ssh/authorized\_keys

no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty ssh-rsa

AAAAB3NzaC1yc2EAAAADAQABAAABAQCB007n/ww+ouN4gSLKssMxXnBOvf9LGt4LojG6rs6h

PB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4kYjh6541N

YsnEAZuXz0jTTyAUfrtU3Z5E003C4oxOj6H0rfIF1kKI9MAQLMdpGW1GYEIgS9EzSdfd8AcC

IicTDWbqLAcU4UpkaX8KyGlLwsNuuGztobF8m72ALC/nLF6JLtPofwFBlgc+myivO7TCUSBd

LQlgMVOFq1I2uPWQOkOWQAHukEOmfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPqdAv8JggJ

ICUvax2T9va5 gsg-keypair

Now Git network commands will still work just fine, but the users won’t be able to get a shell.

**Tip**:

If you want to change the shell back to other shells, such as *bash shell*, run:

# Standing on the root user and run:

$ sudo chsh git -s $(which bash)

# The Pull Process

**1.** List branches:

You can either:

$ git branch // List local branches only

Or

$ git branch –r // List remote branches only

Or

$ git branch –a // List all local and remote branches

**2.** Switch to another branch and restore its working directory tree on your local to match the version in the index. HEAD will point to this branch:

$ git checkout <branch-name>

**Notes:**

* Local changes to untracked files in the working tree will be kept.
* Local changes to tracked files, which belong to the old branch, in the working tree will be prompted to be committed (Git will throw a message “*Please commit your changes or stash them before you switch branches*”).
* Unchanged files, which belong to the old branch, will be removed from local after being switch.

**3.** Automatically fetch and then merge the remote branch into your current branch:

# You can stand on any branch

$ git [pull](#_Pull) <remote-repo-name> <branch-name>

**Note**: The git pull is shorthand for git [fetch](#_Fetch) followed by git merge FETCH\_HEAD.

# The Commit & Push Process

**WARNING**: Before committing anything, make sure you’re checked out to the right branch and [pull](#_More_About_Git_1) all latest changes from this branch.

**1.** Check the status of your current working directory tree. This command shows all file names (with their paths) which were added, removed and modified on your local compared to your last commit on the remote.

# checkout to the right branch

$ git status [options]

**Commonly used options:**

* -uno: do not show untracked files (files which haven’t never been committed)

**2.** Show changes between HEAD and working directory. This is to make sure you don’t commit unexpected changes:

$ [git diff](#_Diff) HEAD

**3.** Include in changes will be committed to the **index**. Changes, which are not included, will be ignored during committing:

$ git add <path-to-files-or-folders>

**Notes**:

* After adding, you’d better check again to make sure you won’t miss any file or commit wrong files using $ git status -uno.
* If you include a wrong file and want to exclude it, use:

$ [git reset](#_Reset) HEAD <file>

Or

$ git reset -- <file>

* If you want to include multiple files at once, use:

$ git add . // Add all files in the current directory

Or

$ git add ./\*.<file-extension>﻿ // Add all files having the same extension

* If you want to see diff before deciding whether to add your files to commit, use:

$ git add –p <file> // or –patch : interactive menu

**4.** Create a new commit containing the current contents of the added files (by git add) to the local repo, with a log message describing the changes. The new commit is a direct child of HEAD, usually the tip of the current branch, and the branch is updated to point to it.

$ git [commit](#_More_About_Commit) -m "descriptive-message"

**Notes:**

* Committing is the process of copying changes on the working directory tree to the **local** repo, not the remote repo.
* If you make a commit and then find a mistake immediately after that, you can recover from it (undo commit) with:

$ git [reset](#_Reset_1) --soft HEAD~1

**5.** Pushing what you’ve committed to the remote repo.

$ git push <remote> <branch>

**6.** Open Git GUI:

$ gitk

**Tip**: [Commit only part of a file](https://stackoverflow.com/a/1085202)

# More About Git Pull

## Fetch vs Pull

**Fetch**

"git fetch" **only downloads new changes** from a remote repository - but it **doesn't integrate** any of these changes into your working files. It is great for getting a fresh view on all things happened in your remote repository.

Due to it's "harmless" nature, you can rest assured: fetch will never manipulate, destroy, or screw up anything.

Commands:

* Fetch all branches from the repo:

$ git fetch <remote-repo-name> // mostly ‘origin’

* Only fetch the specified branch.

$ git fetch <remote-branch-name> <branch-name>

**Pull**

"git pull" is used with a different goal: updating your current HEAD branch with the latest changes from the remote repository. This means that pull **not only downloads new data but also directly integrates it** into your current working copy files.

Commands:

$ git pull <remote-repo-name> <branch-name>

Eg1: Suppose you’re at branch *featureA*, you want to fetch the latest changes from *featureA* (in remote *origin*) then merge them to *featureA*, run:

$ git pull origin featureA

Eg2: Suppose you’re in branch *featureA*, you want to fetch the latest changes from *featureB* (in remote *origin*) then merge them to *featureA*, run:

$ git pull origin featureB

# For the use case of this command, check [here](#_Keep_Feature_Branches).

Note: If NO argument(s) is specified, Git will choose default remote repo and branch defined in *gitconfig* file.

Eg3: Suppose you already ran ‘git branch --set-upstream master origin/master’, meaning the following info was added to your *gitconfig* file:

[branch "master"]

remote = origin

merge = refs/heads/master

Now $ git pull is exactly same as $ git pull origin master

**Fetch vs Pull – When?**

Since "git pull" tries to merge remote changes with your local ones, a so-called "merge conflict" can occur. In this case, you have to [resolve merge conflicts](#_How_to_Resolve) manually. If you don’t want to do that, use "git fetch" instead.

It's highly recommended to start a "git pull" only with a clean working copy. This means that you should not have any uncommitted local changes before you pull. But if you have, save your local changes temporarily using "[git stash](#_Stash)".

## Keep Feature Branches Up To Date with Parent Branch

Suppose you’re working on a feature branch called *feature* which is based off a parent branch called *master*. As time goes by, your feature branch can become significantly out of sync with the latest master branch. This may mean that merging your work when you’ve finished will become very tricky. You can avoid this by frequently running:

$ git pull *origin* *master*

or

$ git pull --rebase *origin* *master*

Both commands will fetch all new commits from the *master* and then add all of them into *feature.*

Their differences are:

* git pull = git fetch + git merge
* git pull --rebase = git fetch + git rebase

So suppose you have two commits in local branch:

D---E feature

/

A---B---C---F master

After "git pull", will be:

D--------E

/ \

A---B---C---F----G master+feature

After "git pull --rebase", there will be no merge point G. Note that D and E become different commits:

A---B---C---F---D'---E' master+feature

# More About Git Commit

## Change the last commit

You committed something, but then you realize you that you forgot to add another file? Or you want to extend your message? It is not necessary to revert the commit every time.

**If you want to add file to the last commit, run:**

$ git add <new\_files>

$ git commit --amend --no-edit

The --no-edit command means you don’t want to change the commit message.

**If you want to change commit message for the last commit, run:**

$ git commit --amend -m "commit message"

What’s happening: git commit --amend will update and replace the most recent commit with a new commit that combines any staged changes with the contents of the previous commit. With nothing currently staged, this just rewrites the previous commit message.

**Note**: Amending the last commit rewrites history. So when you push the commit to the remote repo, you MOST LIKELY need to run the following command. Or else, you will get rejected error: *failed to push some refs to …*

$ git push origin <branch\_name> --force-with-lease

Must read, more about forced push: <https://estl.tech/a-gentler-force-push-on-git-force-with-lease-fb15701218df>

**Warning**

* Do not use --amend if you have already pushed the commit to the remote repo and someone pulled it. This could, or better - probably will, cause problems.

## Change commit authors and emails

Globally:

$ git config --global user.name "your-name"

$ git config --global user.email "your-email"

Per repository:

$ git config user.name "your-name"

$ git config user.email "your-email"

Per commit:

Check [this section](#_Change_commit's_authors).

# More About Git Push

## --force vs --force-with-lease

git push --force is destructive because it unconditionally overwrites the remote repository with whatever you have locally. It’s strongly **discouraged** as it can destroy other commits already pushed to a shared repository. One of the most common causes of force pushes is when we're forced to rebase a branch.

git push --force-with-lease is a **safer** option as it doesn’t overwrite any work on the remote branch if more commits were added to the remote branch (by another team-member or what have you). It ensures you do not overwrite someone else's work by force pushing.

For example, we have a project with a feature branch that both Alice and Bob are working on. Alice initially completes her part and pushes this up to the repository. Bob also finishes his work, but before pushing it up, he notices some changes had been merged into master. Wanting to keep a clean tree, he performs a rebase against the master branch. Of course, when he goes to push this rebased branch, it will be rejected. However, not realizing that Alice has already pushed her work, he performs a push --force. Unfortunately, this erases all record of Alice's changes in the central repository.

What --force-with-lease does is refuse to update a branch unless it is the state that we expect; e.g. nobody has updated the branch upstream. The word “lease” assume you took the lease on the ref when you fetched to decide what the rebased history should be, and you can push back only if the lease has not been broken.

More about these two options: <https://stackoverflow.com/questions/52823692/git-push-force-with-lease-vs-force>

# Reset

## Undo commits on private branches

When used with HEAD~<number>, this command resets the HEAD to a specified commit (changes on the remote repo will be reset too). This usage of reset is a simple way to **undo commits that haven’t been shared with anyone else**.

# Checkout to the right branch

$ git reset [options] HEAD~<number>

Tip: Can replace HEAD~<number> by <commit-hash> to checkout commit/file to a specific revision. Similar apply to OTHER git commands.

|  |  |
| --- | --- |
| Before | After reset HEAD~2 |
| Resetting the hotfix branch to HEAD-2 | Resetting the hotfix branch to HEAD-2 |

**Commonly used options:**

* $ git reset --hard HEAD~<number>: The last <*number*> commits were bad, so you want to not only **undo** these commits but also **delete** all local changes to tracked files on your local. Notice the [differences between HEAD~ and HEAD^](https://stackoverflow.com/a/45916285).
* $ git reset --soft HEAD~<number>: Same as above, but this time you only want to **undo** the last <*number*> commits while keeping all changes on your local. This command will take you back to the stage where you already ran git add, but have not run git commit yet.

Reset a specific file (**unstage this file, not remove its changes on local**):

$ git reset HEAD~<number> -- <file>

# Note: The –-soft and --hard flag have NO effect on the file-level version of git reset

## Reset local changes

When used with HEAD, this command removes all uncommitted local changes in your working directory tree to match the HEAD.

# checkout to the right branch

$ git reset --hard HEAD

# Checkout

## Inspect old commits

By moving HEAD to a specific commit, this command **updates local changes** in the working directory tree to match the specific commit.

Checkout a commit:

$ git checkout HEAD~<number>

|  |  |
| --- | --- |
| Before | After checkout HEAD~2 |
| Move the HEAD ref pointer to a specified commit | Sequence of commits on the master branch |

Checkout a specific file (**remove its changes on local, not unstage it**):

$ git checkout HEAD~<number> -- <file>

# Note: Unlike the commit-level version, this does not move the HEAD ref

**Notes**:

When you want to go back to the latest revision (right before your checkout), run git checkout <current branch>. Using git checkout HEAD has NO meaning in this case, because the HEAD is now pointing to the current revision.

## Switch between branches

When passed with a branch name, it lets you switch between branches. So, local changes in the working directory tree will be applied to the branch scope, not just the commit scope.

$ git checkout <branch>

## Notes

* In all cases, the checkout command cannot undo a commit (like reset or revert command). It simply updates changes on your local.
* Because there is no branch reference to the current HEAD, the checkout command puts you in a **detached HEAD state**. This can be dangerous if you start adding new commits because there will be no way to get back to them after you switch to another branch. For this reason, you should always create a new branch before adding commits to a detached HEAD.

# Revert

## Undo commits on public branches

This command undoes commits by creating a new commit. This is a safe way to undo changes, as it has no chance of rewriting the commit history (like git reset). But like reset –-hard and checkout, revert does update local changes in your working directory tree.

$ git revert HEAD~<number>

|  |  |
| --- | --- |
| Before | After revert HEAD~2 |
|  |  |

**Note**: This command can only be run at a commit-level scope and has NO file-level scope.

## Undo a revert

<https://stackoverflow.com/questions/8728093/how-do-i-un-revert-a-reverted-git-commit>

# Remove Tracked Files From Git

## Stop Tracking Files

Scenario: You accidentally added *app.log* to the repo; now every time you run the application, git reports there are unstaged changes in *app.log*.

You put \*.log in the .gitignore file, but it’s still there. That’s because while .gitignore prevents Git from tracking changes to files or even noticing the existence of files it’s never tracked before, **once a file has been added and committed, Git will continue noticing changes in that file**.

So to tell git to to “undo” tracking changes in this file, run:

$ git rm --cached app.log

# Branch Management

## Create New Branches

Follow the steps below to create a new local and remote Git branch:

**1.** The new branch will be part of the master branch. So your master needs to be up to date:

$ git pull

**2.** Create the new branch LOCALLY on your local machine and switch to it:

$ git checkout -b <branch-name>

**3.** The remote branch is automatically created when you push it to the remote server:

$ git push <remote-name> <branch-name>

Where <remote-name> is typically origin, the name which git gives to the remote you cloned from. Your teammate would then just pull that branch, and it will be automatically created locally.

**4.** When you want to commit changes to your new branch after working locally on it for a while, add -u parameter to set upstream.

$ git push -u <remote-name> <branch-name>

**Note**: You can skip step 3 if you have no changes to commit at this time.

## Rename Branches

Follow the steps below to rename a local and remote Git branch:

**1**. First, switch to the local branch you want to rename:

$ git checkout <old\_name>

**2**. Rename the local branch:

$ git branch -m <new\_name>

**3**. If you’ve already pushed the <old\_name> branch to the remote repository, delete the <old\_name> remote branch:

$ git push origin --delete <old\_name>

**4**. Finally, push the <new\_name> local branch and reset the upstream branch:

$ git push origin -u <new\_name>

## Remove Branches

<https://linuxize.com/post/how-to-delete-local-and-remote-git-branch/>

<https://stackoverflow.com/a/2003515>

# Stash

**1.** Save your local changes away before resetting the working directory tree to match the HEAD commit.

# Checkout to the right branch

$ git stash save -m <“descriptive message”>:

* -u: stash untracked files in addition to tracked files.

This command is most commonly used when:

* Switching between branches without having to committing local changes. So, we can avoid warning “*Please commit your changes or stash them before you switch branches*.”
* Reset working directory tree TEMPORARILY.

**2.** List all stashes you haven’t made:

$ git stash list

**3.** Show all stashes with a summary of the stash diffs:

$ git stash show [stash@{<n>}]

**4.** Restore local changes on the working directory tree, then delete the stash index from the stash list:

$ git stash pop [stash@{<n>}]

* Git names stashes as *stash@{0}* which is the most recently created stash, *stash@{1}* which is the one before it, etc. If no *[stash@{<n>}]* is specified on the command above, *stash@{0}* is assumed.

**5.** Restore local changes on the working directory tree WITHOUT DELETING the stash index from the stash list:

$ git stash apply [stash@{<n>}]

**Tip**: If you want to stash changes but *don’t want to lose local changes on your working directory*, combine git stash save and git stash apply.

**6.** Remove all stash entries, meaning all local changes will be deleted:

$ git stash clear

**7.** Remove one stash entry at a time. If no *<stash@{<n>}>* is specified, stash@{0} is assumed:

$ git stash drop <stash@{<n>}>

**Notes:**

* Both git stash and git reset delete local changes, but git stash backups these changes on your local before deleting them. Also note that git stash doesn’t commit anything; it just saves.
* Both git stash and git reset only restore local changes at a specific [stash@{<n>}]. So, we cannot restore all <n>-time changes if we use <n> stashes; only one of them. In most cases, the most recently stash has, by itself, all the changes we want to restore.

# Logs

Show commit logs:

# Checkout to the right branch

$ git log [options] <commit>

**Commonly used options:**

* $ git log -<number>: Only show the last <*number*> commits

# Clean

Remove untracked files from the working tree:

$ git clean –xdf

# x: both ignore and non-ignore files

# d: directories

# f: force

**Other commonly used options:**

-e: Exclude files to clean

# Deletes all untracked files except those with the .txt extension

$ git clean -xdfe \*.txt

-i: Show what would be done and clean files interactively. You will choose one of these actions:

\*\*\* Commands \*\*\*

1: clean 2: filter by pattern 3: select by numbers

4: ask each 5: quit 6: help

# Ignore

Not every file created or updated in your code should be committed to the repo. We can prevent Git from staging these files by ignoring them.

**Note**:

There is no explicit git ignore command. Instead, we need to edit the *.gitignore* file by hand when we have new files that you wish to ignore.

**Example:**

1. To ignore files with extension *\*.vcxproj.user*, add the following line into *.git/info/exclude*:

\*.vcxproj.user

2. To ignore directories named *Debug*, add the following line into *.git/info/exclude*:

Debug/

More details:

<https://www.atlassian.com/git/tutorials/saving-changes/gitignore>

[Differences between .git/info/exclude and .gitignore](https://stackoverflow.com/a/1753078)

# Diff

1. Show changes between commits:

$ git diff <SHA1> <SHA2>

**Note**: <SHA1> <SHA2> means that taking SHA2 and comparing it to SHA1. Not the opposite! (important to know Added or Deleted files between commits).

2. Show changes between commits for a single file:

$ git diff <SHA1> <SHA2> -- <file>

3. Show changes between the working tree and the commit at revision <SHA>:

$ git diff <SHA>

**Note**: <SHA> can be the HEAD. In this case, the command shows changes between the working tree and the HEAD.

4. Show changes between the index and the commit at revision <SHA>:

$ git diff --cache <SHA>

**Note**: This command is useful when you have new file (haven't committed before, but added to the index). We need it because the default git diff only shows changes among committed files.

5. Show only changed file names:

$ git diff --name-status <SHA1> <SHA2>

**Tip**: To use **VS Code** as default editor for git difftool and git mergetool, add following content into *Git/config*:

[diff]

tool = vscode

[difftool "vscode"]

cmd = code --wait --diff $LOCAL $REMOTE

[merge]

tool = vscode

[mergetool "vscode"]

cmd = code --wait $MERGED

# Patch File

## What Is A Patch File?

A patch is a file showing changes (in a diff format) made in a repository.

## What Are Patches Used For?

#1: Patch files are generally used when someone (from outside your team) has read-only access but had a good code change available. He then creates a patch file and sends it to you. You apply it and push it to your Git repo.

#2: You have good changes, but you don't want to commit and push them to your repo for any reason. In this case, you can make a patch file and later on, apply it somewhere else (another repo or send it to somebody).

*You might not know!*

While Git does not use patches internally, one design goal for Git is to make it easy to exchange patches (because many projects work that way, e.g. Linux and Git itself). So Git has special commands for handling patches (git diff shows changes as patches by default, git apply lets you apply a patch, etc.).

## How to Create Patches?

<https://nithinbekal.com/posts/git-patch/>

## How to Apply Patches?

**Way #1: git am**

# Checkout to the correct branch where you want to apply the patch

$ git am <patch-file>

**Note**: When git am fails to apply a patch, use git am --3way to try a 3-way merge. This lets you manually edit the conflicted sections before running git add for the conflicted files. After resolving conflicts, use git am --resolved to move to the next patch.

**Example:**

$ git am rspec-changes.patch

Applying: Add rspec to gemfile

Applying: Add notes file

$ git log --oneline

ac9caff Add notes file

f784b22 Add rspec to gemfile

8619310 ...older commits...

**Way #2: git apply**

<https://nithinbekal.com/posts/git-patch/>

<https://jayeshkawli.ghost.io/applying-git-diff/> (error: patch failed - patch does not apply)

**git apply vs git am**

git apply takes a patch (e.g. the output of git diff or git format-patch) and applies it to the working directory. The changes remain unstaged, so you can use git status or git diff to check changes made in the patch you have applied.

git am takes a mailbox of commits formatted as an email message (e.g. the output of git format-patch) and applies it (or a series of patches) to the current branch. It's like the "opposite" of git format-patch. Because of creating a commit, with git am you won't see anything when using git status, but you can use git log to check if your patch was applied.

git am uses git apply behind the scenes, but does more work before (reading a Maildir or mbox, and parsing email messages) and after (creating commits).

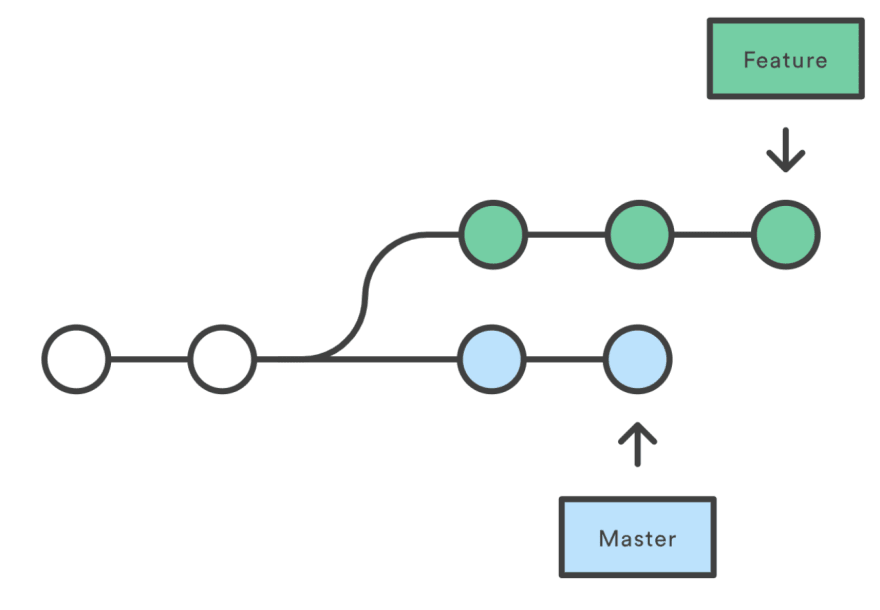
**Way #3: patch**

<https://www.drupal.org/node/1129120>

# Merge vs Rebase

Both git merge and git rebase offer the same service: incorporating commits from one branch into another. The key distinction lies in how this result is achieved.

To understand these two commands, consider the example below:



|  |  |
| --- | --- |
| **Merge** | **Rebase** |
| git checkout **master**  git merge **feature** | git checkout **feature**  git rebase **master** |
| git merge | Rebase |
| Create a new commit. | Doesn’t create a new commit. |
| Change commit history. | Doesn’t change commit history. |
|  | At a high level, rebasing can be understood as “moving the base of a branch onto a different position”.  At a lower level, what rebase actually does is replaying feature branch commits one by one (chronologically) from a new starting point.  This is ideal for our situation, since all we are currently trying to do is keep our feature branch up-to-date with any new commits from master. This is definitely not a meaningful event we want to preserve in our project’s history. |

**When to use git merge:**

If used too liberally, merge commits can clutter up your Git logs, and make it harder to understand the flow of your project’s history. To avoid this pitfall, try to use merges sparingly. Avoid branching and merging when only making minor changes.

The one-line summary: **Merge when you make major changes or want a set of commits to be clearly grouped together in history**. This way, the merge commits act as milestones that others can use to figure out when these major changes were incorporated into the project.

**When to use git rebase:**

As discussed, rebasing changes the commit history. The one-line summary: should not be apply on the public branches (eg: master) where other people are working on. This makes git confuse that your master branch diverge from other’s master branch.

# Rebase

## Combine multiple commits on the same branch into one

For example, if you want to combine the two last commits on your *feature* branch, run:

$ git checkout feature

$ git rebase -i HEAD~2 // Can replace [HEAD~<number>] by [commit-hash]

Running this will open a script file named *git-rebase-todo* in your editor (the one you registered when installing git, e.g., Notepad++, VS Code, etc.)

**Note**: The TODO list of this file is in the reverse order as compared with the output of git log. For example, if your git log results to:

$ git log -2 --pretty=oneline

e0018bb 2nd commit

6985236 1st commit

Then, your *git-rebase-todo* will be:

pick 6985236 1st commit

pick e0018bb 2nd commit

By modifying this file and saving it before closing it, you apply what you want to rebase. For example, if you want to squash the *2nd commit* into the *1st commit*, then change *git-rebase-todo* into:

pick 6985236 1st commit

squash e0018bb 2nd commit

**Note**: Changing 1st commit’s pick to squash will result in an error: *cannot 'squash' without a previous commit*.

After saving and closing your editor, you'll open another script file whose contents are:

# This is a combination of 2 commits.

# The first commit's message is:

1st commit

# This is the 2nd commit message:

2nd commit

You only need to save and close this file without modifying anything.

Finally, don't forget to push your changes using git push –-force-with-lease. Your rebase is done!

## Merge commits on two branches

## Change commit's authors and emails

<https://stackoverflow.com/a/3042512>

## Other use cases

In *git-rebase-todo* file, there are other helpful commands for you to choose, including:

* reword: To change commit message
* drop: To remove commit message

# How to Resolve Merge Conflicts

<https://stackoverflow.com/questions/3065650/whats-the-simplest-way-to-get-a-list-of-conflicted-files>

<https://help.github.com/en/articles/resolving-a-merge-conflict-using-the-command-line>

<https://www.git-tower.com/learn/git/faq/solve-merge-conflicts>

## How a merge conflict occurs

"Merging" is the act of **integrating another branch into your current working branch**. You're taking changes from another context and combine them with your current working files.

Git makes merging extremely easy: in most cases, Git will figure out how to integrate new changes, using its **Auto-Merge** function.

However, there're situations where you might have to step in and tell Git what to do. Most notably, this is when **changing the same file**. Even in this case, Git will most likely be able to figure it out on its own. But if two people changed the **same lines** in that same file (or delete a file you modified, or add a file with the same name), Git simply cannot know what is correct. Git will then mark the file as having a conflict - which you'll have to solve before you can continue your work.

## How to solve a merge conflict

The easiest way is using:

$ [git mergetool](#_Diff_1)

After merging, simply quit the tool to give Git a hint that you're done with your file. Behind the scenes, this will tell Git to execute a "git add" command on the (now formerly) conflicted file. This marks the conflict as solved.

Finally, after solving all conflicts, a merge conflict situation needs to be concluded by a regular commit.

## How to undo a merge

You can return to the state before you started the merge at any time, run:

$ git merge --abort

In case you've made a mistake while resolving a conflict and realize this only after completing the merge, you can still easily undo it: just roll back to the commit before the merge happened with "git reset --hard " and start over again.

# How to Create an Archive of Changed Files

**1**. Open Git Bash in the repo’s **root directory**.

Else, might get error “*fatal: pathspec '<dir>' did not match any files*” at step 3.

**2**. List all changed files names with their status (*modified/added/deleted*) between two commits:

$ git diff --name-status <SHA1> <SHA2>

**3**. Create an archive to store these changed files:

$ git archive –o changes.zip HEAD $(git diff --name-only <SHA1> <SHA2> --diff-filter=d)

# Where "changes.zip" is the archive output

# HEAD adds the commit's latest update to the archive file

# --diff-filter=d ignore deleted files