This is a really good Git tutorial: <https://www.atlassian.com/git/tutorials> (Start with Setting Up A Repository)

**What is Git**

Git is written by Linus Torvalds (The person who invented Linux), and was released in April 2005.

Git means unpleasant and stupid person. Linus named it after himself…

Git works on Linux, MacOS, and Windows.

Git is widely used and is currently the most popular version control system (based on repositories in August 2019, <https://www.openhub.net/repositories/compare>)

Machine generated alternative text:
Compare Repositories 
Subversion: 324,633 (25%) 
CVS: 
Subversion: 25% 
Mercurial: I % 
Cit: 

Git is a distributed version control system (DVCS);Therefore, every machine that clones the repository contains the whole history of the

Machine generated alternative text:


A big misconception is that Git is the same as GitHub. This is NOT TRUE.

Git is the DVSC, and GitHub or GitLabs or BitBucket is the public server repository.

This will mostly cover the command line in Git

**Configuring Your Git**

Git allows you customize your repository and how you want things to look and what applications to open when certain commands are used

Add --global --local --system … to the following commands to indicate what user level the configuration is for.

The following is to let Git know who you are when you do any changes

**Git config user.name <name>**

**Git config user.email <email>**

The following is some configurations to use beyond compare with difftool and mergetool

**Git config diff.tool bc4**

**Git config merge.tool bc4**

**Git config difftool.bc4.path "C:/program files/beyond compare 4/bcomp.exe"**

**Git config mergetool.bc4.path "C:/program files/beyond compare 4/bcomp.exe"**

**Git config --add difftool.prompt false**

**Git config --add mergetool.prompt false**

The following is to set the default editor for git

**Git config core.editor notepad++**

This will just open the configuration settings in your default editor for git

**Git config --edit**

The following will add aliases **so** you do not have to type out the full command.

A common one is git config aliases.unstage "reset --"

**Git config Aliases.<PreferedName> <ActualCommand>**

To copy a repository onto your local machine and start working on it, you need to use:

**Git Clone <repository-url>**

The <repository-url> can be a Secure HyperText Transfer Protocol (HTTPS) [eg. <https://github.com/csoltenborn/GoogleTestAdapter.git> ] or Secure Shell (SSH) [eg. git@github.com:csoltenborn/GoogleTestAdapter.git]. If you use HTTPS, Git will ask you for credentials everytime you try to push to your remote repository. If you use SSH, you will need to set up a SSH key on your device and account, and Git won't ask for your credentials everytime you try to push.

When you first clone a repository, Git will automatically create a remote called "origin" for you with the URL from the clone command. Git will create a remote branch on your local machine called "origin/master" to represent the branch on the "origin", and Git will setup a branch named "master" that will be "upstream" to "origin/master".

Machine generated alternative text:
Øb743 
øb743 
git.Our-company.com 
a6b4c 
master 
f4265 
git clone janedoe@git.ourcompany. com: project. git 
a6b4c 
My Computer 
origin/master 
f4265 
master 
Remote branch 
Local branch 

**Git Remotes**

Git uses Remotes to collaborate between repositories

Remotes are technically just addresses (URLs) to repositories.

It makes life easier to type a simple remote name instead of the url all the time

You can view all remote repositories using:

**Git remote -v**

You can add new remote respositories

**Git Remote add <name> <url>**

You can remove remote respostories

**Git Remote remove <name>**

You can change the url of the remote repositories

**Git remote set-url <name> <newurl>**

You can rename the remote

**Git remote rename <oldname> <newname>**

Machine generated alternative text:
f4265 
øb743 
git.ourcompany.com 
31b8e 
o rxgxn 
19ea3 
git.teaml.ourconpany.com 
f4265 
ma S ter 
31b8e 
git remote add teanone git://git.teanl.ourconpany.com 
My Computer 
origin,'master 
a6b4c 
31b8e 
a38de 
f4265 
19ea3 
893cf 
master 

The Commit History Tree is the actual version control; the Commit History Tree is where you can see the progress of the repository throughout time.

**Git Branches & Git Commits**

A commit is a snapshot of the files along with a message describing what the commit does. They are like tfs checkins. Commits are saved in the Commit History Tree as a SHA-1 hashes so there is a very low chance of commits having the same identifier.

Branches are pointers and they point to commits. These commits indicate where the branches are in Git history.

Branches allow you to work on different tasks at the same time and not interfere with other feature developments.

Machine generated alternative text:
Master 

To list local branches uses

**Git branch**

To list all branches (remote branches too)

**Git branch -a**

To create a new branch use

**Git branch <branchname>**

To change branches

**Git checkout <branchname>**

To create a new branch and switch to that branch

**Git checkout -b <branchname>**

To delete a branch on your local repository use,

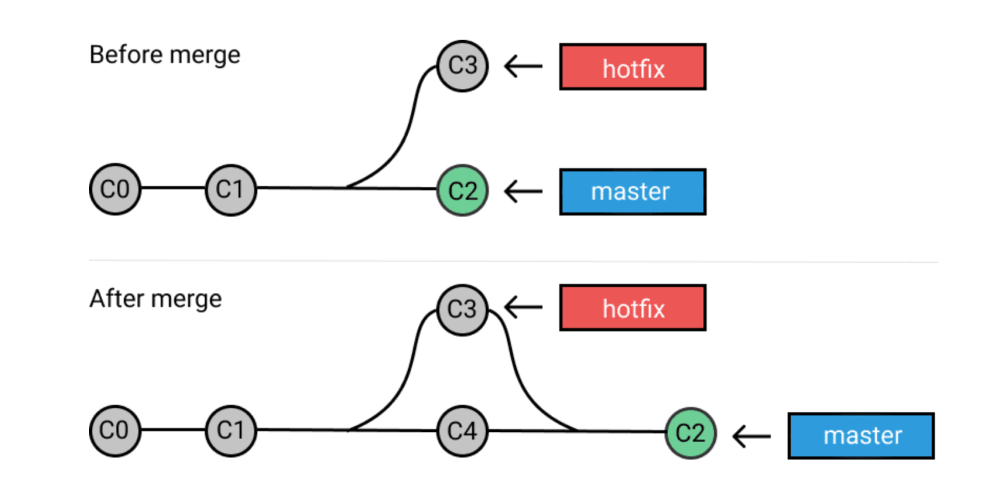
**Git branch -d <branchname>**

But the branch will still exist on other repositories if they have been pushed to, to delete a branch from a remote repository use,

**Git push --delete <remote> <branchname>**

We can also merge branches together. Checkout to branch you want to merge into and use,

**Git Merge <branchname>**



If the branches can't be fast forwarded, this will create a new merge commit and you might have to resolve any conflict. After you resolve any conflicts, Git will prompt you for a merge message for the commit.

Use the following command to bring up a tool to help with the merging (Beyond Compare 4)

**Git MergeTool**

To list all the commits in the commit tree

**Git log**

--graph (for visual graph in the command line)

Machine generated alternative text:
D: NdataXGitTest»git log 
commit 8749574d1gggad12866212522c613bd86sac9113 (HEAD 
master) 
uthor : 
Date : 
Thomas Vy (thomas . vy@geoslope.com) 
Thu sep S 2819 -860 
This is a commit message 
commit 8bc7c413caa1172c2a35682f725688fsa6c8b518 
uthor : 
Date : 
Thomas Vy (thomas . vy@geoslope.com) 
Thu Aug 29 2819 -860 
hello 

**HEAD**

HEAD is just a reference to that last commit on the current branch. It indicates where you are in the commit trees.

The branch actually points to the commit and HEAD just points to the branch

Machine generated alternative text:


Git checkout will change where the HEAD pointer is pointing to.

HEAD and commit hashes can combined with ~ or ^ to refer to a commit relative to that commit

~1 - go back one commit

^1 - go back to the first parent

Machine generated alternative text:
AAA3A2 
AN2A2 
c 
H 
AA2 
DA2 
AAIAI 

Compares files or commits in the command line

**Git diff <file1> <file2>**

**Git diff <commit1>..<commit2> -- <filename>**

**Git diff** (Compares the files in the working directory to the ones in HEAD)

--cached (Compares the staging area to the HEAD)

Brings up the difftool for comparisons (Beyond Compare 4)

**Git difftool <file1> <file2>**

**We only need the first 5 letters of the hash when using them in the command line**

**Git Fetch <remote> <branch>**

This will update your remote/branch on your local repo

Machine generated alternative text:
Øb743 
øb743 
git.Our-company.com 
a6b4c 
master 
f4265 
git clone janedoe@git.ourcompany. com: project. git 
a6b4c 
My Computer 
origin/master 
f4265 
master 
Remote branch 
Local branch 

Machine generated alternative text:
øb743 
øb743 
a6b4c 
a6b4c 
git.ourcompany.com 
f4265 
My Computer 
origin/master 
f4265 
31b8e 
a38de 
master 
Someone else pushes 
893cf 
master 

Machine generated alternative text:
Øb743 
Øb743 
a6b4c 
a6b4c 
git.ourconpany.com 
f4265 
git fetch origin 
My Computer 
f4265 
31b8e 
31b8e 
a38de 
master 
19øa3 
origin/master 
lgøa3 
893cf 
master 

A shortcut for git fetch and git merge, since it is so common, is

**Git Pull <remote> <branch>**

-r (rebase; creates a linear history line)

To make your changes public on a repo use,

**Git Push <remote> <branch>**

Note: If the repos have different commits (rewritten history and git can't fast forward), Git will reject your push. If you are 100% you want to overwrite what is on the repo, use --force or -f

All these command can be written in short form like the following (as long as you have set the upstream; -u <origin> <branch> to set the upstream),

Git fetch

Git pull

Git push

**Git's Three Tree (Area) System**

<https://www.atlassian.com/git/tutorials/undoing-changes/git-reset>

Git's workflow is typically a 3 step process:

1. Edit your files
2. Stage your files
3. Commit your files

Git has a Three Tree System. They are named the following:

**The Working Directory Area**

The working directory tree is essentially your local filesystem. It immediately gets updated to reflect your filesystem.

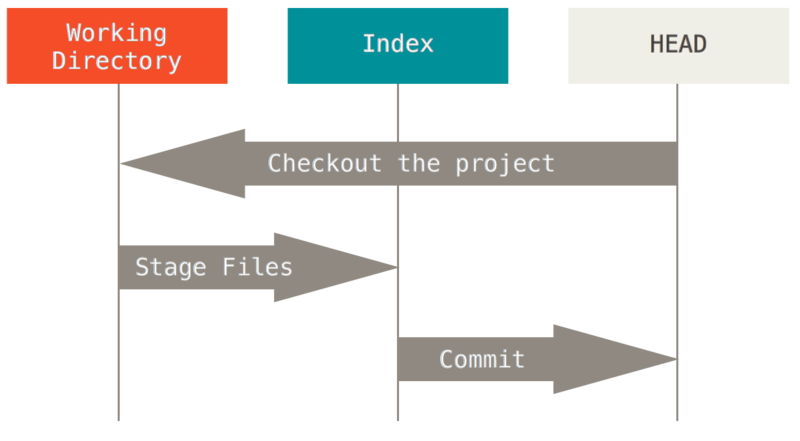
**Staging Index Area**

The staging index area is where files go after you promote them with "git add". Does not immediately reflect the filesystem.

This is how you group highly focused files together and get them ready to add to the Commit History Tree.

**Commit History (HEAD) Tree**

The Commit History Tree from before



To stage files (Add them to the staging area), use

**Git add -- <filenames>**

Addition arguments:

-u (updates the tracked files in the staging area)

-A (Puts all changed files in the staging area)

-n (Preview what will actually get put into the staging area)

Unstages files from the staging area

**Git reset -- <filenames>**

More about git reset in "Undo in Git"

Shortcut for removing a files and adding it to the staging area

**Git rm <filenames>**

Creates a commit in the Git Commit History Tree

**Git commit** (The default text editor will prompt up asking for a message)

-m "message"(Does not bring up the text editor

Machine generated alternative text:
Working Tree 
Staging Area 
(Index) 
History 
Commit B 
Commit A 
.git directory 

Machine generated alternative text:
Staging Area 
Working Tree 
(Index) 
History 
Commit B 
Commit A 
.git directory 

Machine generated alternative text:
Working Tree 
Staging Area 
(Index) 
History 
Commit B 
Commit A 
.git directory 

Machine generated alternative text:
Working Tree 
Staging Area 
(Index) 
History 
Commit C 
Commit B 
Commit A 
.git directory 

**More Info about git files**

All files in git are writable (You do not need to checkout a file to edit it). Git will automatically detect that there are changes within files.

**Tracked vs Untracked**

When a file is first introduced into a git repository (When a file is first added in the respository, the file is an "untracked-file", meaning that there has been no history for this file in git before)

A file becomes a "tracked-file" once the file has been promoted to the staging index for the first time.

**Staging Index Area**

Files in the staging index area are screenshots of the code at the point where the user typed in "git add". If you make any changes after staging a file, the changes after the staging are only in the working directory

**Uncommitted**

Not in the Commit History Tree yet

**Remove Changes in Working Directory**

**Git checkout**

If you want to experiment with changes in the past, (Similar to "Get this Version" in tfs) , you use git checkout <commit hash>, this will make HEAD "detached" (detached means the HEAD pointer will point to a commit instead of pointing to a branch name)

Machine generated alternative text:


Any commits made while in "detached" mode will not be saved unless you save the commit to a branch or stash it.

If you don't want to save any of your commits and you want to get out of "detached" mode, you can simply use

git checkout <branchname>

If you want to save your commits, you will have to create a new branch with the commits and merge it into the desired branch later, for this command use:

Git branch <newbranchname> <commit hash>

Another thing we can do with git checkout is discard unstaged (tracking) changes in our work directory so the next command will copy the staging index area to the working directory,

Git checkout -- <file name>

We can also specify a commit we want to recover a single file from; the next command will copy the committed file to the staging index and the working directory

Git checkout <commit hash> -- <filename>

**Git Clean**

Git Clean will clear any "untracked" work in your working directory (You can the specify the files in Git Clean). Git Clean is very destructive so Git Cleans requires a --force or a -f to delete files.

Some useful arguments

-n to preview the files that will be deleted

-d to delete directories

-i to interactively delete items

**Undo Commits**

**Git Revert <commit hash>**

Safest undo (preserves the commit history of git). It is similar to a rollback in tfs. Where it does an undo of the changes in a commit and then creates a new commit with the changes.

**Git Reset <commit hash> (Only do on local commits)**

--hard [makes it match up with the commit even the directory]

--mixed(default)[match up with the commit, the directory is in unstaged area]

--soft[match up with the commit, the directory is in staging area)

Machine generated alternative text:


**Saving In-Progress Work**

Git stash is like the tfs version of shelvesets but stashes are only local

Use **git stash save <stash message>**

By default, git stash save will only stash tracking files (files that have been previously added to the respository; both staged and unstaged); git stash save will not apply to ignored files and untracked files.

To include the untracked files, use the option --include-untracked

Machine generated alternative text:
HEAD 

Machine generated alternative text:
HEAD 

Machine generated alternative text:


Machine generated alternative text:
HEAD 

To view stashed files, use the command

Git stash list

To view git stash differences,

Git diff stash@{0}^1 stash@{0} - compares to the last commit before the stash

Git stash show stash@{0} - compares to HEAD (last commit on your branch)

**MISC.**

**Git Garbage Collection**

Git does not immediately deletes things. Git actually uses a Garbage Collection System that usually runs every 30 days (it's like C#'s garbage collector).

**Git Init**

Will Create a brand new git repository

**Git commit --amend**

Adds files in the staging area to last commit and brings up the default text editor so you can change the commit message

**Git Reflog**

Retrieves references logs so you can retrieve some commits (eg. Accidentally removing commits, retrieving detached HEAD commits)

**Git Blame**

Git blame is like tfs version of annotation.

**Git Rebase**

Sometimes a nice linear path is what we want, since rebasing allows git to have a nice linear history, but this rewrites

**Tags**

Tags are used to indicate importance about a certain commit. Usually tags are used to show a release point (eg. <https://github.com/google/googletest/tags>)

**Some Neat Tools for Git**

GitK or GitG - Graphics to view the git history

Git Kraken - Fancy Git UI .

**Git Cheat Sheet**

<https://github.github.com/training-kit/downloads/github-git-cheat-sheet.pdf>