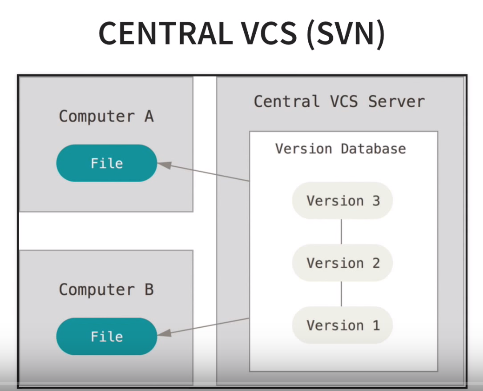
SVN to Git

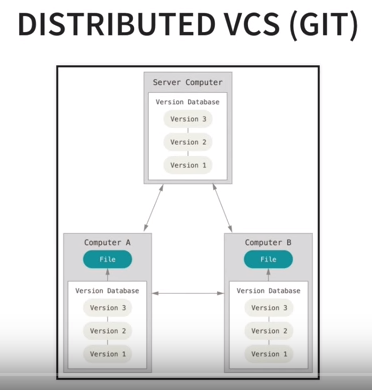
What is the difference between SVN (Central VCS) and Git (Distributed VCS)?

Svn is central in one place or repository so if there is no access to this network, there is a problem, and if you got this repository corrupted, then you have to have a backup somewhere.

****

Git, everybody has a local repository, and your local repository has all the information that your remote repository has based on the time that you sink those both together. If you don’t have access to that remote repository, you still can view each single detail of change done on this repository since it is created.

Everybody has a backup in case you can’t access the server repository, or in case something happened.



Note: Git is much faster than any other CVS system, because git does not need to go out the server to get the history and display it for you. Most operations in Git need only local files and resources to operate – generally no information is needed from another computer on your network.

If you want to see the changes introduced between the current version of a file and the file a month ago, Git can look up the file a month ago and do a local difference calculation, instead of having to either ask a remote server to do it or pull an older version of the file from the remote server to do it locally.

This also means that there is very little you can’t do if you’re offline or off VPN. If you get on an airplane or a train and want to do a little work, you can commit happily (to your *local* copy, remember?) until you get to a network connection to upload. If you go home and can’t get your VPN client working properly, you can still work. **In many other systems, doing so is either impossible or painful**. In Perforce, for example, you can’t do much when you aren’t connected to the server; and in Subversion and CVS, you can edit files, but you can’t commit changes to your database (because your database is offline). This may not seem like a huge deal, but you may be surprised what a big difference it can make.

**Git Has Integrity**

Everything in Git is check-summed before it is stored and is then referred to by that checksum. This means it’s impossible to change the contents of any file or directory without Git knowing about it. This functionality is built into Git at the lowest levels and is integral to its philosophy. You can’t lose information in transit or get file corruption without Git being able to detect it.

The mechanism that Git uses for this checksumming is called a SHA-1 hash. This is a 40-character string composed of hexadecimal characters (0–9 and a–f) and calculated based on the contents of a file or directory structure in Git. A SHA-1 hash looks something like this:

24b9da6552252987aa493b52f8696cd6d3b00373

You will see these hash values all over the place in Git because it uses them so much. In fact, Git stores everything in its database not by file name but by the hash value of its contents.

Commands:

1. To get the version of git

Git --version

1. Configure username and email:

Git config –global user.name “Mohammad Abu Shaira”

Git config –global user.email [mohdabushaera@gmail.com](mailto:mohdabushaera@gmail.com)

To retrieve the configuration:

Git config –list

Git config user.name

Since Git might read the same configuration variable value from more than one file, it’s possible that you have an unexpected value for one of these values and you don’t know why. In cases like that, you can query Git as to the origin for that value, and it will tell you which configuration file had the final say in setting that value:

$ git config --show-origin rerere.autoUpdate

file:/home/johndoe/.gitconfig false

1. Any Comman:

Git help <verb>

Or

Git <verb> --help

Like git help config or git config --help

Two ways to start tracking with git:

1. You have an existing project on your local machine

Command: git init

From the directory you want to track. Then the .git file will be created, it contains all tracking info.

If you want to remove the tracking from a directory

Rm –rf .git

Before Commit:

Git status

You will see red files, these files are untracked files.

Some files should be added to the ignore list like .project file, because there files contains personal preferences, and we do not want these file to override other developers preferences.

touch .gitignore

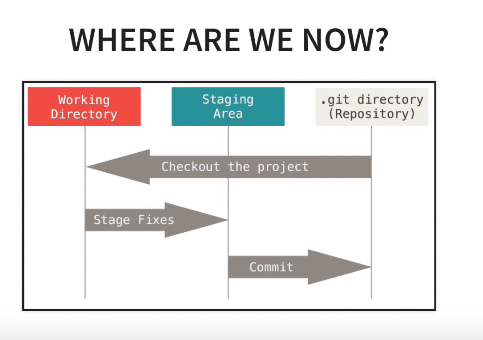
copy and paste the below inside .gitignore file

.jws

.data

.adf

\*.jpr



3 states:

**Working directory:** untracked and modified files would be in the tracking directory, we can a list of those when we run git status.

**Staging area:** what we organize what we want to be committed to our repository. The reason of staging area is that we can do multiple commits, as we can choose the files we want to commit if we have done a lot of work, then we can choose what we want to commit.

From the book:

* Committed means that the data is safely stored in your local database.
* Modified means that you have changed the file but have not committed it to your database yet.
* Staged means that you have marked a modified file in its current version to go into your next commit snapshot.

1. The Git directory is where Git stores the metadata and object database for your project. This is the most important part of Git, and it is what is copied when you clone a repository from another computer.
2. The working tree is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.
3. The staging area is a file, generally contained in your Git directory, that stores information about what will go into your next commit. Its technical name in Git parlance is the “index”, but the phrase “staging area” works just as well.

The Command Line:

Note: the command line is the only place you can run all Git commands.

* To add all files that are untracked to the staging area:

Git add –A

Git status

You can add a specific file like

Git add .gitignore

If you want to commit you can run a commit command to commit to your repository.

To remove files from the staging area, you can use

Git reset

Or if you want to remove a specific file

Git reset .file-name or folder-name

To Commit

Git commit –m “initial commit”

To see the log:

Git log

You will see the commit you have just made, you will see the hash of that commit, and it is unique, you will also see the author, email, and the message of that commit.

1. There is an existing project remotely that you want start developing in

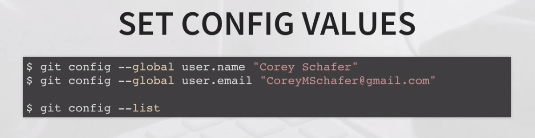
Suppose that your company has a repository, and you want to clone that repository and start working on.

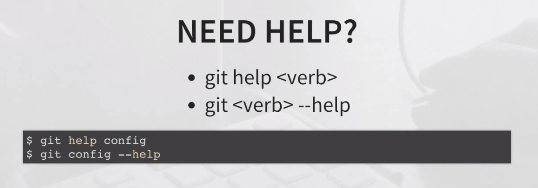
Git clone <url> <where to clone>

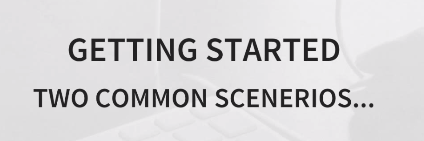
If you put . in the where to close, it just means the current directory.

**Commands:**

****

****

****

****

**Scenario #1:**

****

**The .git directory will be created and it contains all information to track your project.**

**To remove the initialization just type rm –rf .git**

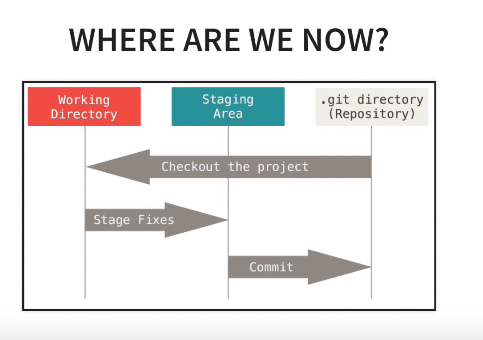
**Then the .git file will be removed and no longer tracking this directory with git.**

****

****

**Command: touch .gitignore then run git status**

**You will see that files or wildcards in the gitignore file do not appear as files to be committed.**



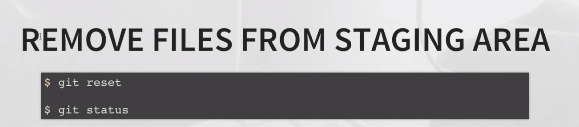
**To see the files that need to be committed in the working directory use git status.**

**To add the files to the staging area use git add**

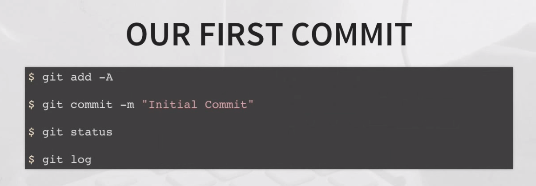
**Git add –A -> to add all untracked files from the working directory to the staging area.**

**Git add .gitignore -> to add a specific file from the working directory to the staging area.**

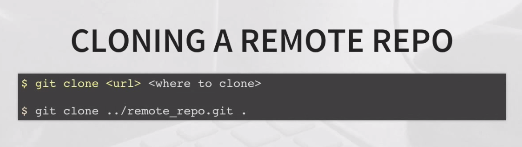
**Then, run git status -> to make sure that all untracked (selected) files are added to the staging area.**

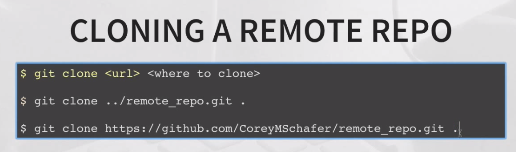
****

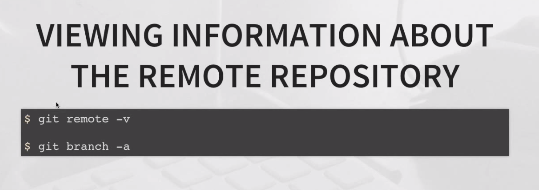
**To do this for a specific file use git reset calc.py (file-name)**

****

**Scenario #2:**

****

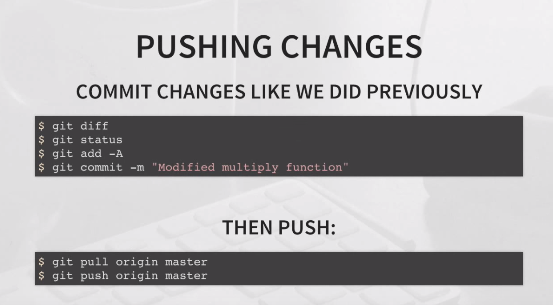
****

****

**Git remote –v -> lists the information of the repository**

**Git branch –a -> lists all branches in our repository, not only locally but remotely as well.**

**Now what you should do when you make a change to the cloned code.**

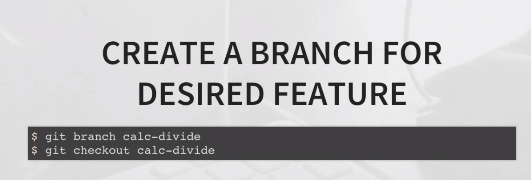
****

**Origin is the name of our remote repository.**

**Master is the name of the branch we want to push to.**

****

**When you want to add a feature for example, you would normally create branch, and then start working in that branch. Let’s suppose that you want to work on a divide function in your java file.**

****

**You can use the command git branch to list all branches you have. The one that has the star, means this is the active one or the one you are working on.**

**Use git checkout branch-name to make the branch you want the active one.**

**After you check out the branch you want, you can start working on the changes you want in the code (divide fn).**

**Then**

**git status**

**git add –A**

**git commit –m “Divide Function”**

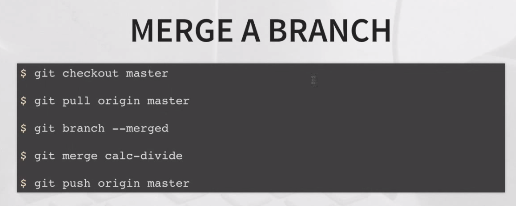
**By doing this, we have committed the change on our branch locally, and this has no effect on the master branch, and has no effect on our remote repository.**

****

**Origin is the name of our remote repository.**

**Calc-divide is the name of the branch that we want to push.**

**Now if all the unit test is OK, then you need to merge the branch to the master branch.**

****

**Command1: to switch to the master.**

**Command2: pull the changes that may other developers have done.**

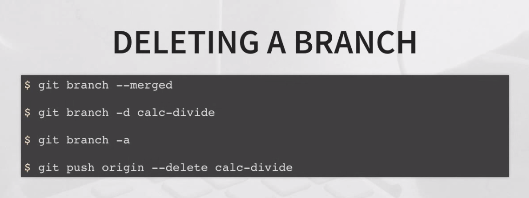
**Command3: this command tells us the branches that we’ve merged in so far. Note that till now the calc-divide branch is still not merged.**

**Command4: to merge the calc\_divide branch with the master branch.**

**Command5: to push the changes on the master to the remote repository.**

**Note: now you can run command3 again, if the calc-divide branch appeared in the result, then the merge is done successfully.**

**Now after pushing up the feature (branch) to the remote repository, then, the feature is there and you are ready to delete the branch.**

****

**Command1: to push the changes on the master to the remote repository.**

**Command2: to delete the branch locally.**

**Command3: will show you the branches you have, locally and remotely, and you will notice that the calc-divide branch is deleted locally, but it exists remotely.**

**Command4: to delete the calc-divide branch from the remote repository.**

**Note:** if you want to get a copy of an existing Git repository — for example, a project you’d like to contribute to — the command you need is git clone. If you’re familiar with other VCS systems such as Subversion, you’ll notice that the command is "clone" and not "checkout". This is an important distinction — instead of getting just a working copy, Git receives a full copy of nearly all data that the server has. Every version of every file for the history of the project is pulled down by default when you run git clone.

**Note:** You clone a repository with git clone <url>. For example, if you want to clone the Git linkable library called libgit2, you can do so like this:

$ git clone https://github.com/libgit2/libgit2

That creates a directory named libgit2, initializes a .git directory inside it, pulls down all the data for that repository, and checks out a working copy of the latest version. If you go into the new libgit2directory that was just created, you’ll see the project files in there, ready to be worked on or used.

If you want to clone the repository into a directory named something other than libgit2, you can specify that as the next command-line option:

$ git clone https://github.com/libgit2/libgit2 mylibgit

That command does the same thing as the previous one, but the target directory is called mylibgit.

**Note:** Remember that each file in your working directory can be in one of two states: *tracked* or *untracked*. Tracked files are files that were in the last snapshot; they can be unmodified, modified, or staged. In short, tracked files are files that Git knows about.

Untracked files are everything else — any files in your working directory that were not in your last snapshot and are not in your staging area. When you first clone a repository, all of your files will be tracked and unmodified because Git just checked them out and you haven’t edited anything.

Untracked basically means that Git sees a file you didn’t have in the previous snapshot (commit)

Note: git add is a multipurpose command — you use it to begin tracking new files, to stage files, and to do other things like marking merge-conflicted files as resolved. It may be helpful to think of it more as “add precisely this content to the next commit” rather than “add this file to the project”.



Note: For now, that branch is always “master”, which is the default;

### Note: Short Status

While the git status output is pretty comprehensive, it’s also quite wordy. Git also has a short status flag so you can see your changes in a more compact way. If you run git status -s or git status --short you get a far more simplified output from the command:

$ git status -s

M README

MM Rakefile

A lib/git.rb

M lib/simplegit.rb

?? LICENSE.txt

New files that aren’t tracked have a ?? next to them, new files that have been added to the staging area have an A, modified files have an M and so on. There are two columns to the output - the left-hand column indicates the status of the staging area and the right-hand column indicates the status of the working tree. So for example in that output, the README file is modified in the working directory but not yet staged, while the lib/simplegit.rb file is modified and staged. The Rakefile was modified, staged and then modified again, so there are changes to it that are both staged and unstaged.

**Note: Ignoring Files**

You will edit the .gitignore file, and follow the below rules to set your ignore expressions:

* 1. An asterisk (\*) matches zero or more characters
  2. [abc] matches any character inside the brackets (in this case a, b, or c) ex. \*.[oa] to ignore any files ending in “.o” or “.a”
  3. a question mark (?) matches a single character
  4. brackets enclosing characters separated by a hyphen ([0-9]) matches any character between them (in this case 0 through 9)
  5. Ex. \*~ The second line tells Git to ignore all files whose names end with a tilde (~), which is used by many text editors such as Emacs to mark temporary files.
  6. Blank lines or lines starting with # are ignored. So if there is a blank line or a line starting with a # in the gitignore file, it will be ignored.
  7. Standard glob patterns work, and will be applied recursively throughout the entire working tree.
  8. You can start patterns with a forward slash (/) to avoid recursivity.
  9. You can end patterns with a forward slash (/) to specify a directory.
  10. You can negate a pattern by starting it with an exclamation point (!).
  11. You can also use two asterisks to match nested directories; a/\*\*/z would match a/z, a/b/z, a/b/c/z, and so on.

Examples:

# ignore all .a files

\*.a

# but do track lib.a, even though you're ignoring .a files above

!lib.a

# only ignore the TODO file in the current directory, not subdir/TODO

/TODO

# ignore all files in the build/ directory

build/

# ignore doc/notes.txt, but not doc/server/arch.txt

doc/\*.txt

# ignore all .pdf files in the doc/ directory and any of its subdirectories

doc/\*\*/\*.pdf

In the simple case, a repository might have a single .gitignore file in its root directory, which applies recursively to the entire repository. However, it is also possible to have additional .gitignore files in subdirectories. The rules in these nested .gitignore files apply only to the files under the directory where they are located. (The Linux kernel source repository has 206 .gitignore files.)

**Note: git diff use it to answer these two questions:**

What have you changed but not yet staged? And what have you staged that you are about to commit? Although git status answers those questions very generally by listing the file names, git diff shows you the exact lines added and removed — the patch, as it were.

That command compares what is in your working directory with what is in your staging area. The result tells you the changes you’ve made that you haven’t yet staged.

If you want to see what you’ve staged that will go into your next commit, you can use git diff --staged. This command compares your staged changes to your last commit. --staged and --cached are synonyms)

**Note:** Undoing Things:

1. If you want to amend your last commit because the message is wrong or because you missed a file not staged you can use the command git commit –amend

Note that using this command will replace your previous commit as it never happened.

1. Unstaging a staged file

Git reset head file-name

1. Let’s suppose you changed a file on your working directory and you want to discard these changes

Git checkout - - file-name

**Note:** Working with Remotes:

1. Showing your Remotes:

Git remote

Git remote –v

$ git clone https://github.com/schacon/ticgit

Cloning into 'ticgit'...

remote: Reusing existing pack: 1857, done.

remote: Total 1857 (delta 0), reused 0 (delta 0)

Receiving objects: 100% (1857/1857), 374.35 KiB | 268.00 KiB/s, done.

Resolving deltas: 100% (772/772), done.

Checking connectivity... done.

$ cd ticgit

$ git remote

Origin

$ git remote -v

origin https://github.com/schacon/ticgit (fetch)

origin https://github.com/schacon/ticgit (push)

If you’ve cloned your repository, you should at least see origin

You can also specify -v, which shows you the URLs that Git has stored for the shortname to be used when reading and writing to that remote:

1. Adding Remote Repositories

git remote add <shortname> <url>

Ex. git remote add moha https://github.com/mohammad/ticgit

$ git remote

origin

$ git remote add pb https://github.com/paulboone/ticgit

$ git remote -v

origin https://github.com/schacon/ticgit (fetch)

origin https://github.com/schacon/ticgit (push)

pb https://github.com/paulboone/ticgit (fetch)

pb https://github.com/paulboone/ticgit (push)

If you clone a repository, the command automatically adds that remote repository under the name “origin”.

If you clone a repository, the command automatically adds that remote repository under the name “origin”. So, git fetch origin fetches any new work that has been pushed to that server since you cloned (or last fetched from) it. It’s important to note that the git fetch command only downloads the data to your local repository — it doesn’t automatically merge it with any of your work or modify what you’re currently working on. You have to merge it manually into your work when you’re ready.

you can use the git pull command to automatically fetch and then merge that remote branch into your current branch. the git clone command automatically sets up your local master branch to track the remote master branch (or whatever the default branch is called) on the server you cloned from. Running git pull generally fetches data from the server you originally cloned from and automatically tries to merge it into the code you’re currently working on.

1. Pushing to Your Remotes

git push <remote> <branch>

If you want to push your master branch to your origin server (again, cloning generally sets up both of those names for you automatically), then you can run this to push any commits you’ve done back up to the server:

$ git push origin master

1. Inspecting a Remote

git remote show <remote>

$ git remote show origin

\* remote origin

Fetch URL: https://github.com/schacon/ticgit

Push URL: https://github.com/schacon/ticgit

HEAD branch: master

Remote branches:

master tracked

dev-branch tracked

Local branch configured for 'git pull':

master merges with remote master

Local ref configured for 'git push':

master pushes to master (up to date)

1. Renaming and Removing Remotes

git remote rename to change a remote’s shortname

$ git remote rename pb moha

$ git remote

origin

moha

If you want to remove a remote for some reason

$ git remote remove moha

$ git remote

origin

**Note: Tags: are specific releases important in the history.**

1. How to get a list of all tags

Git tag

1. How to filter the long list of tags, like you are interested in the 1.8.5 series

Git tag –l “v1.8.5\*”

-l or –list acts the same

1. How to create a tag, two types light weighted, and annotated:
   1. Annotated: git tag –a v1.4 –m ‘my tag no 1.4’

Git show v1.4

* 1. Lightweight: git tag v1.4lw

Git show v1.4lw

What is the difference?

* If you run git show on the tag, you don’t see the extra tag information. The command just shows the commit:
* A lightweight tag is very much like a branch that doesn’t change — it’s just a pointer to a specific commit.
* Annotated tags, however, are stored as full objects in the Git database. They’re checksummed; contain the tagger name, email, and date; have a tagging message; and can be signed and verified with GNU Privacy Guard (GPG).
* It’s generally recommended that you create annotated tags so you can have all this information; but if you want a temporary tag or for some reason don’t want to keep the other information, lightweight tags are available too.

1. You can tag the commit later on if you forget to tag it.

git log --pretty=oneline

git tag -a v1.2 9fceb02

1. **Sharing Tags : transferring the tags to the remote server**

Git push origin <tagname>

If you have a lot of tags that you want to push up at once, you can also the –tags

Ex. Git push origin –tags

This will transfer all of your tags to the remote server that are not already there.

1. S
2. S
3. s

**Git Branching**

Branching means you diverge from the main line of development and continue to do work without messing with that main line.

Some people refer to Git’s branching model as its “killer feature,” why? The way Git branches is incredibly lightweight, and switching back and forth between branches generally just as fast.

The default branch in git is the “master” branch, it is created when you run the “git init” command.

Let’s suppose that you have a project of three files, and you did an initial commit. The below is what happen:



Your Git repository now contains five objects: one blob for the contents of each of your three files, one tree that lists the contents of the directory and specifies which file names are stored as which blobs, and one commit with the pointer to that root tree and all the commit metadata.

If you make some changes and commit again, the next commit stores a pointer to the commit that came immediately before it.



A branch in Git is simply a lightweight movable pointer to one of these commits.

**Creating a New Branch**

**Note: Git VS Other CVS**

It’s worth pointing out that Git determines the best common ancestor to use for its merge base; this is different than older tools like CVS or Subversion (before version 1.5), where the developer doing the merge had to figure out the best merge base for themselves. This makes merging a heck of a lot easier in Git than in these other systems.

**Questions:**

1. **How I get a specific version later on?**
2. **In Distributed CVS like git, any client repository can be copied back up to the server to restore it, how we can do this?**