1 – Getting Started

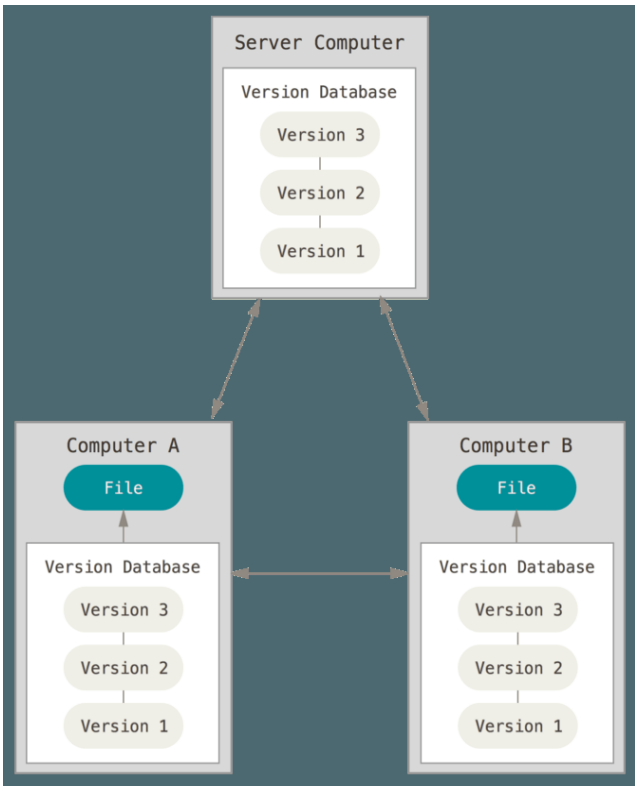
1. Introduction

➔ Different versions of the project is controlled using 'Version Control System(VCS)'. There are many type of VCS but most used one was Centralized VCS(CVCS).

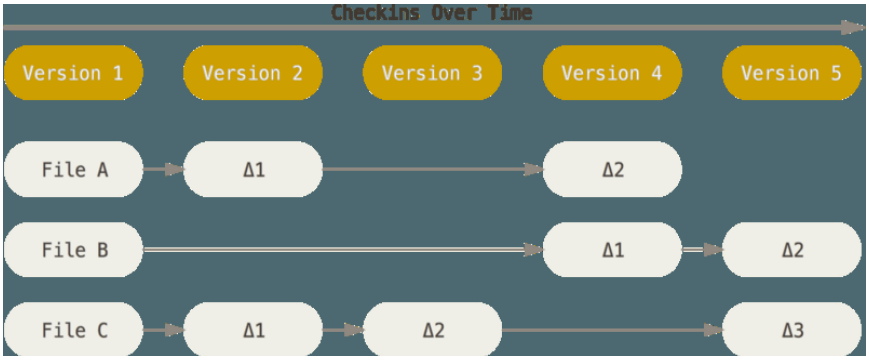
➔ The disadvantage of CVCS is that if central server/database goes down or corrupted then full project will be lost. The only remaining will be latest snapshot of the project on local machine of any contributor. Hence we have Distributed VCS(DVCS) now like Git, Mercurial, Bazaar or Darcs etc.

➔ How DVCS works is that when someone copies a project then it not only copies the latest snapshot but the whole history of the project along with every version of every single file.

➔ Thus if any server or database dies then any of the client repository could be copied on the central server to restore it. Every **clone** is actually full backup of all the data as shown below



➔ Apart from that CVCS stores data as what commonly described as delta-based version control. That is these systems set of files and changes made to each file over time as shown below

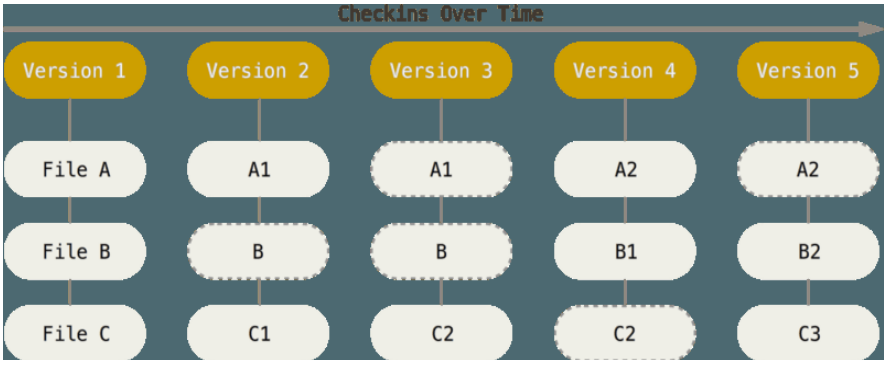


Storing data as changes to a base version of each file.

➔ On the other hand when one saves state of his project on git then git takes picture of what all of your files

looks like at that moment and stores the reference to that snapshot.

➔ To be more efficient, if files have not changed, git doesn't stores the file again, just a link to the previous identical file it has already stored.



Storing data as snapshots of the project over time.

➔ Also most of the operations of gits are local. Like CVCS one dont need a network to work with a particular project, because entire history of the project is on your local system. Hence speed is faster and most of the operations are instantaneous.

➔ Even when there is no network one can commit it locally and upload it later when network is available.

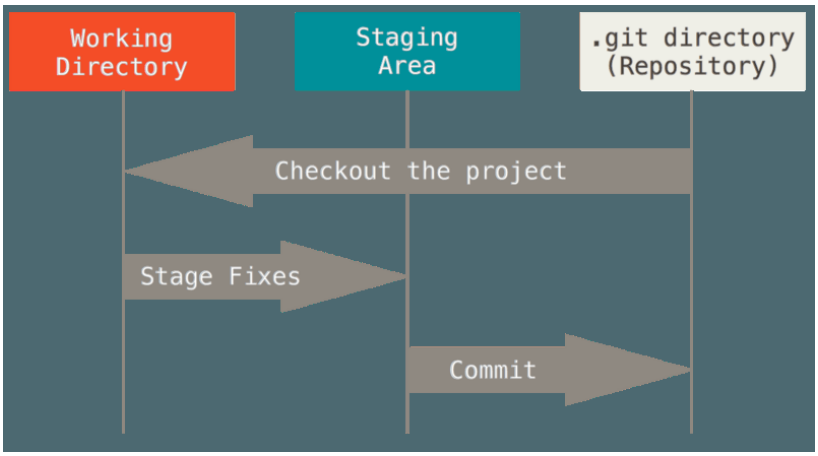
➔ Everything in Git is checksumed before it is stored. Git uses SHA-1 hash for checksumming. Its a 40-character hexadecimal characters(0-9 and a-f) and calculated based on contents of file or directory structure in git. It look like this

24b9da6552252987aa493b52f8696cd6d3b00373

One will see it all over the place while using git because it uses them so much. Infact Git stores everything in database not by file name but by hash values.

2. Git Project

➔ Git project has the three main section. **Git directory**, **Working directory** and **Staging area(Index)** as shown below.



➔ Git directory is where git stores the metadata and object database for your project. This is what is copied when one **clone** a repository from another computer.

➔ Working tree/directory is a single checkout of one version of the project. These files are pulled out of the compressed database in git directory and placed on disk for you to use or modify.

➔ Staging area is a file, generally contained in the git directory that stores information about what goes in your next commit. Its technical name in git parlance is '**Index**'.

➔ The basic git workflow goes like this

1. One will modify files in working tree.
2. Then he selectively stage just those changes he want to be part of your next commit, which adds *only* those changes to the staging area.
3. One do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

➔ If a particular version of a file is in the Git directory, it's considered committed. If it has been modified and was added to the staging area, it is staged. And if it was changed since it was checked out but hasnot been staged, it is modified.

➔ Command line is the only place where one can use all the git functionality. Most of the GUIs based git apps implements only subset of the git functionality for simplicity.

2. Installing Git

2.1 git config

➔ When git is installed, first thing one needs to do is customize git environment.

➔ This one needs to do only once and they will stick around between upgrades.

➔ One can also change them anything from command again.

➔ For this git comes with a tool called 'git config', which will be used to set configuration variables which controls whole aspect of how git looks and operates.

➔ These variables can be stored in three different places.

1. **/etc/gitconfig file :**

➔ Contains values applied to every user on the system and all their repositories.

➔ If one pass the option `--system` to git config, it reads and writes specifically from this file.

➔ This is system configuration file and one needs administrative or superpower priviledge to make changes to it.

➔ If one is using windows XP or newer then the system level config file is present at C:/ProgramData/Git/config, which can only be changed with `git config -f <file>` as an admin.

2. **~/.gitconfig or ~/.config/git/config file :**

➔ Values specific personally to you, the user. One can make git read or write to this file by passing `--global` option.

3. **config file :**

➔ Present in the git directory(.git/config) of whatever repository one is currently using, specific to that directory.

➔ Each level overrides the values in the previous level. So values in .git/config trumps over /etc/gitconfig.

Your Identity

➔ The first thing one should do when git has been install is to set ones username and email.

➔ This is important because every Git commit uses this information.

```
$ git config --global user.name "Jayant Joshi"
$ git config --global user.email jayantjoshi@example.com
```

[Note : One can unset name using

```
$ git config --global --unset core.name
```

and reset using

```
$git config --global core.name “Jayant Joshi”
```

]

➔ Many GUI tools will help you do this when you first run them.

Your Editor

➔ One can configure text editor of his choice to type in messages. If not then git uses systems default editor.

$ git config –global core.ditor emacs

(Note : emacs is a name of an editor).

➔ If someone wants to use very famous notnepad++ as an editor then he should provide full path to the its executable file as shown below

```
$ git config --global core.editor "'C:/Program Files/Notepad++/notepad++.exe' -multiInst -nosession"
```

[Note : To unset the editor one should use `$ git config–global --unset core.editor`]

Checking your setting

➔ To check ones configuration settings we have command

```
$ git config --list
```

which brings out

```
$ git config --list
core.symlinks=false
core.autocrlf=true
core.fscache=true
color.diff=auto
color.status=auto
color.branch=auto
color.interactive=true
help.format=html
rebase.autosquash=true
http.sslcainfo=C:/Program Files/Git/mingw64/ssl/certs/ca-bundle.crt
http.sslbackend=openssl
diff.astextplain.textconv=astextplain
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
credential.helper=manager
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
user.name=Jayant
user.email=jayantjoshi0209@gmail.com
core.name=Jayant Joshi
core.repositoryformatversion=0
core.filemode=false
core.bare=false
core.logallrefupdates=true
core.symlinks=false
core.ignorecase=true
remote.origin.url=https://github.com/jayantjoshi0209/CloneTesting
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
branch.master.remote=origin
```

➔ One can also check with single key as

```
$ git config core.name
Jayant Joshi
```

➔ One can also see above repeating key/value pairs in the listing. This happens because git reads configuration

records from different places(like local and system etc). If one want to see the origin of particular key/value pair and git will show which config file have a final say in setting that value.

```
$ git config --show-origin core.name
file:C:/Users/Jayant/.gitconfig Jayant Joshi
$ git config --show-origin filter.lfs.clean
file:C:/Users/Jayant/.gitconfig git-lfs clean -- %f
```

2. Getting Help

➔ To open the comprehensive manual page(manpage) help for any command we have two ways

1. $ git help <verb>
2. $ man git-<verb>

This will open comprehensive manual page from the docs present locally. In my case it opens

```
file:///C:/Program%20Files/Git/mingw64/share/doc/git-doc/git-config.html
```

➔ If someone dont want full blow manpage then he can have quick refresher on the git command.

```
$ git config -h
usage: git config [<options>]

Config file location
    --global              use global config file
    --system              use system config file
    --local               use repository config file
    -f, --file <file>     use given config file
    --blob <blob-id>      read config from given blob object

Action
    --get                 get value: name [value-regex]
    --get-all             get all values: key [value-regex]
    --get-regexp          get values for regexp: name-regex [value-regex]
    --get-urlmatch        get value specific for the URL: section[.var] URL
    --replace-all         replace all matching variables: name value [value_rege
x]
    --add                 add a new variable: name value
    --unset               remove a variable: name [value-regex]
    --unset-all           remove all matches: name [value-regex]
    --rename-section      rename section: old-name new-name
    --remove-section      remove a section: name
    -l, --list            list all
    -e, --edit            open an editor
    --get-color           find the color configured: slot [default]
    --get-colorbool       find the color setting: slot [stdout-is-tty]

Type
    --bool                value is "true" or "false"
    --int                 value is decimal number
    --bool-or-int         value is --bool or --int
    --path                value is a path (file or directory name)

Other
    -z, --null            terminate values with NUL byte
    --name-only           show variable names only
    --includes            respect include directives on lookup
    --show-origin         show origin of config (file, standard input, blob, com
mand line)
```