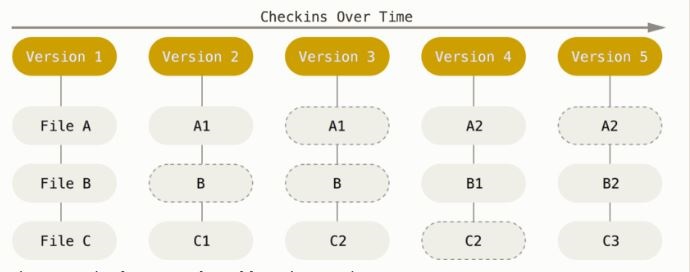
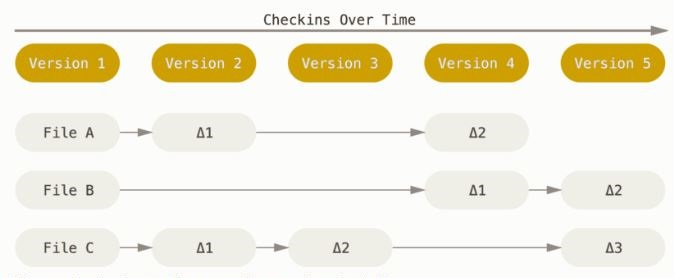
History

* In 2002, the Linux kernel project began using a proprietary DVCS called BitKeeper.
* In 2005, the relationship between the community that developed the Linux kernel and the commercial company that developed BitKeeper broke down.
* Linus Torvalds, the creator of Linux to develop their own tool based on some of the lessons they learned while using BitKeeper
* Some of the goals of the new system were as follows:
  + Speed
  + Simple design
  + Strong support for non-linear development (thousands of parallel branches)
  + Fully distributed
  + Able to handle large projects like the Linux kernel efficiently (speed and data size)
* Birth in 2005

GIT Concept

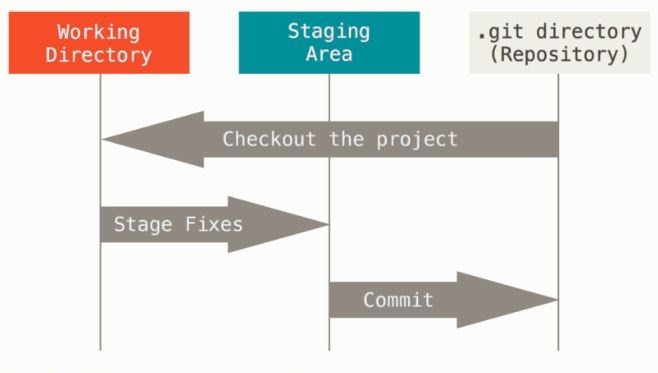
* The major difference between Git and any other VCS is the way Git thinks about its data.
* Git thinks of its data more like a set of snapshots of a miniature filesystem. (***IMAGE: git\_file\_store***)
* 
* Other vcs stores the file in each commit, if there is a change or not (***IMAGE: OtherCVS\_file\_store***)
* 
* Git check for the changes, if there no change then it will point to the previous identical one
* Nearly every operation in git is local
* Git Has Integrity
  + It’s impossible to change the contents of any file or directory without Git knowing about it
  + Everything in Git is check-summed before it is stored and is then referred to by that checksum.
  + The mechanism that Git uses for this checksumming is called a **SHA-1 hash**
* Git stores everything in its database not by file name but by the hash value of its contents.
* Git Generally Only Adds Data
* When you do actions in Git, nearly all of them only add data to the Git database. It is hard to get the system to do anything that is not undoable or to make it erase data in any way.

## 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*.

* The Three States of git( **Most important**)

Git has three main states that your files can reside in:

* + **committed** - the data is safely stored in your local database
  + **Modified** - you have changed the file but have not committed it to your database yet
  + **Staged** - you have marked a modified file in its current version to go into your next commit snapshot
* The three main sections of a Git project
* 
* **The Git directory** - 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.
* **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.
* **The staging area** - is a file, generally contained in your Git directory, that stores information about what will go into your next commit. It’s sometimes referred to as the “index”, but it’s also common to refer to it as the staging area.
* The basic Git workflow goes something like this:
  + You modify files in your working tree.
  + You stage the files, adding snapshots of them to your staging area.
  + You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

## Customize your Git environment

* **git config**

***/etc/gitconfig file***: Contains values for every user on the system and all their repositories. If you pass the option --system to git config, it reads and writes from this file specifically. ( In Windows C:\Documents and Settings\All Users\Application Data\Git\config on Windows XP, and in C:\ProgramData\Git\config on Windows Vista and newer.)

***~/.gitconfig or ~/.config/git/config file*** (In windows under user): Specific to your user. You can make Git read and write to this file specifically by passing the --global option.

***config file in the Git directory*** (that is, .git/config) of whatever repository you’re currently using: Specific to that single repository.

* **For your identity**

***git config --global user.name "Athul"***

***git config --global user.email athuldevadas@gmail.com***

* + Git commit uses this information.
  + if we want to override this info just run the command without --global from the project directory(git repo)
* ***Editor*** - can configure the default text editor that will be used when Git needs you to type in a message

***git config --global core.editor <name>***

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

* ***git config --list*** - command to list all the settings Git can find at that point.

You may see keys more than once, because Git reads the same key from different files (/etc/gitconfig and ~/.gitconfig, for example). In this case, Git uses the last value for each unique key it sees.

You can also check what Git thinks a specific key’s value is by typing ***git config <key>***

Eg: git config user.name

* ***Git Aliases : git config --global alias.ci commit :*** now we can use git ci in place of git commit
* **For getting help**:

***git help <verb>***

***git <verb> --help***

***man git-<verb>***

Eg: git help config

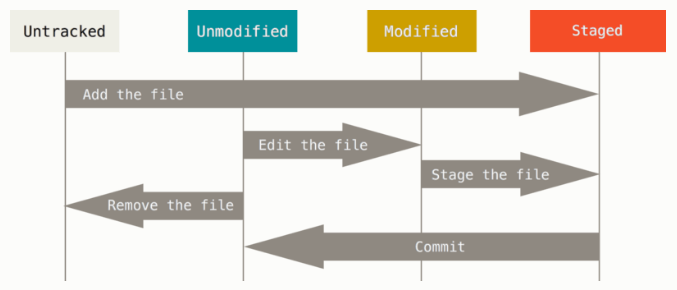
# Basic Commands

* ***git init –*** Create an empty Git repository or reinitialize an existing one
* ***git add -*** Add file contents to the index/staging area
* ***git commit -*** Record changes to the repository
  + ***git commit --amend:*** This command takes your staging area and uses it for the commit. If you’ve made no changes since your last commit (for instance, you run this command immediately after your previous commit), then your snapshot will look exactly the same, and all you’ll change is your commit message
  + You end up with a single commit – the second commit replaces the results of the first
* ***git status -*** used to obtain a summary of which files have changes that are staged for the next commit
* ***git rm -*** Remove files from the index, or from the working tree and the index
  + ***git rm --cached <file>*** remove file from staging area/index , so it will come in working tree( it will become untracked) (if we are checking the diff, then it will show that the file is deleted from index/staging area)
  + ***git rm --f <file>*** remove the file from both index/staging area and working tree
* ***git reset -*** Reset current HEAD to the specified state
  + ***git reset HEAD <file>*** (if we check the diff, it will not have any clue that any file was added and then removed(reset) it will move to the last commit)
  + ***git reset --soft HEAD^*** (remove the last commit and move the head to previous commit and the file will be there in staging area with whatever changes we made)
    - For Eg: Made mistake in commit message then we can use this option
  + ***git reset –hard HEAD***^(completely move to previous commit, which means the changes that we made will also be deleted)
    - For Eg: if you accidently commit the changes to master in place of a new branch then we can use this option. Here first create a new branch then apply this command. Then move to new branch
  + HEAD,HEAD^,HEAD~2,HEAD~3…. Different commit from last
* ***git mv -*** Move or rename a file, a directory, or a symlink
  + - ***git mv source destination*** : here it rename source(file,directory,symlink) to destination.
    - ***git mv source destination\_dir*** : destination\_dir should be present and it should be a directory, will move the source to dir
    - in both case the changes will be already there in staging area, just need to commit
* ***git clone*** - Clone a repository into a new directory
  + git clone https://github.com/libgit2/libgit2 mylibgit
  + here it will create a new directory named mylibgit and clone everything inside it, if it is not mentioned then everything will be under libgit2 directory
* ***git diff***: to see the changes made
  + ***git diff*** : will check the change b/w working directory with the current staging area, will show the difference.
  + ***git diff --staged***: will show the difference b/w the current staged one with the last commit
  + ***git diff --cached*** : what you have staged so far.
  + (cached and staged are synonyms).

## Unmodifying a Modified File

* git checkout -- <file>: if the file is in working directory/ untracked stage, then if we want to go back to how it was in previous commit, just need to checkout the file. This will remove the changes that we made.

# Git Repo changes:



## Commit History:

***git log*** : with no argument will show all the commit to that repo. Latest first

***git log –p -2*** : which shows the difference introduced in each commit. You can also use -2, which limits the output to only the last two entries

***git log –stat*** : abbreviated stats for each commit (change details). Prints below each commit entry a list of modified files, how many files were changed, and how many lines in those files were added and removed. It also puts a summary of the information at the end

--pretty:  changes the log output to formats other than the default, Prebuild formats are also there.

Eg: ***git log --pretty=oneline :*** prints each commit on a single line, which is useful if you’re looking at a lot of commits.

Many other option are also there.

## Working with Remotes

* ***git remote:*** will show the remote short name( if clone it will atleast contain origin)
  + ***git remote –v*** : will show the url and shortname
  + we can add more remote by : ***git remote add <shortname> <url>***
  + we can rename remote : ***git remote rename <oldname> <newname>***
  + we can remove remote: ***git remote remove <shortname> or git remote rm <shortname>***
  + ***git remote show***: will show more details about the git repo. his command shows which branch is automatically pushed to when you run git push while on certain branches. It also shows you which remote branches on the server you don’t yet have, which remote branches you have that have been removed from the server, and multiple local branches that are able to merge automatically with their remote-tracking branch when you run git pull.
* ***git fetch***
  + ***git fetch <shortname>*** : will download the file from remote to local. But it will not merge it.
* ***git pull***
  + ***git pull <shortname>*** : will download and try to merge the latest changes from remote

## Tagging

* ***git tag :*** show available tags
  + ***git push*** command doesn’t transfer tags to remote servers. You will have to explicitly push tags to a shared server after you have created them
  + ***git push origin [tagname].***
  + ***git push origin –tags*** to push all tags
  + Now these tags are available for others who pull or fetch or clone.
* ***git tag –l “1.7\*”*** : show tag starting with 1.7
* Git uses two main types of tags: **lightweight** and **annotated**.
  + Annotated: 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)
  + ***git tag -a <tagname> -m “tagmsg”*** : will tag with the commit at that point.
  + We can also tag on the past commit by providing the commit checksum at the end of the command.
* Lightweight: but if you want a temporary tag or for some reason don’t want to keep the other information, lightweight tags are available too.
  + ***git tag <tagname>***
* ***git show <tagname>*** : used to see tag info
* we cant really checkout a tag. If we want we can create a new branch with specific tag
  + ***git checkout -b [branchname] [tagname]:***