Git Essentials

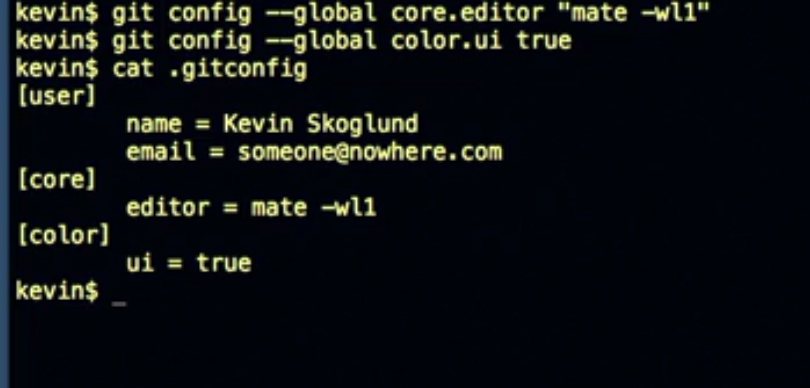
Configuration can be as follows:

1. System level using: git config --system
2. User level using: git config --global
3. Project level using no explicit flag: git config

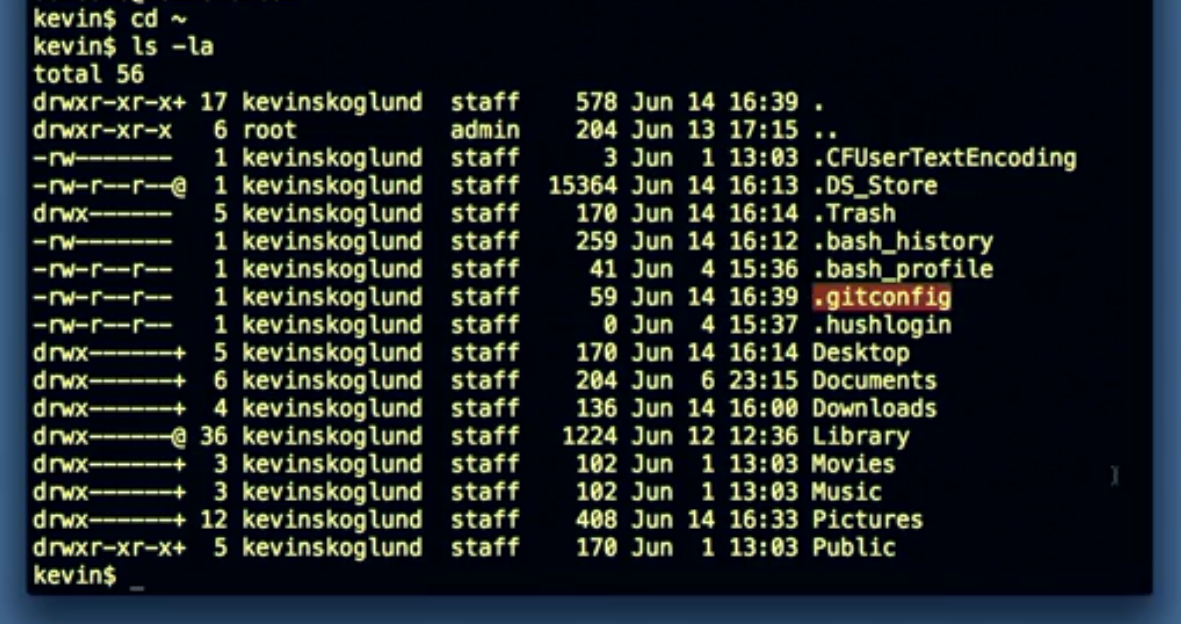
Example:



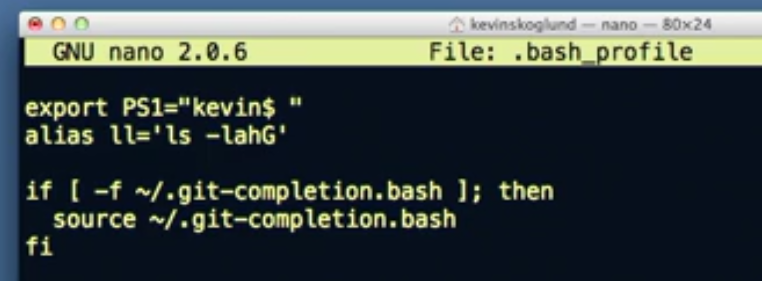
Set default text editor and colors:



All these configs are saved to a .gitconfig file in our HOME directory. We can also see its contents by using cat .gitconfig



Enable auto complete in git on Mac & Unix: First we download git completion script from a url by using curl. Then we rename it to .git-completion.bash so that it’s a hidden file. Then we edit .bash\_profile and add below code:



Overview

* git clone <url> (fetches all branches) or git clone -b <branch name> --single-branch <url> (Fetches only single branch)
* git remote show origin (checks correct branch has pull and push configured correctly)
* git remote -v (Shows git url for push and fetch in remote repo)
* git add . (puts everything into staging area)
* git commit -m "msge here" (sends everything to repository)
* git log (shows the list of commits we have made, we can use -n or --oneline etc)
* git status (shows local vs repository changes)
* git commit -amend -m "new msge here"
* git revert <last commit hash>

Branches Related:

* git branch (shows all branch including the active branch)
* git branch -r (shows remote repo branches only)
* git branch -a (shows local repo + remote repo branches both)
* git branch <branch name> (creates a new branch but we still point to old branch only)

or

git branch <branch name> <origin/branchname> (creates a new branch taking mentioned branch name as its root instead of taking default master as its root. New branch will have all the contents of the mentioned branch.)

* git checkout <branch name> (switches to specified branch)
* Git checkout -b <branch name> (creates a new branch from default master branch and switches us into it in single command)

or

Git checkout -b <branch name> <origin/master> (creates a new branch from origin/master branch and switches us into it in single command)

* git branch -m <existing branch name> <new branch name> (Rename a branch) or git branch --move <existing branch name> <new branch name> (Rename a branch)
* git branch --delete <branch name> (Deletes a branch) or git branch -d <branch name> (Deletes a branch)
* Delete a file:
* git rm <filename> and then git commit -m "msge here"
* Rename a file:
* git mv <existing file name> <new file name> and then git commit -m "msge here"

Merge Branches:

* Git merge <branch name> (You need to be in master branch to merge the <branch name> specified into master)
* In case of conflict, abort it (if required) using: git merge –abort
* In case of conflict, resolve manually and then, git add <filename> and then git commit

Git stash (Stashes are available across branches):

* Git stash save “msge here” (saves local changes into stash)
* Git stash list (shows us the stash list)
* Git stash show <stash number> (Shows a specified stash details) or Git stash show -p <stash number>
* Git stash apply <stash number> (pulls the stash into local directory leaving a copy in stash also)
* Git stash pop <stash number> (pulls the stash into local directory & removes it from stash)
* Git stash drop <stash number> (deletes the specified item from the stash)
* Git stash clear (Deletes everything from stash)

Remote:

* Git remote (shows us all remote configured)
* Git remote add <alias name like origin> <git url> (configures a remote with specified url)
* Git remote -v (shows push and fetch urls for remote repo)
* Git remote rm <alias name like origin> (removes the remote)
* Git push -u <alias like origin> <local branch name whose change you wanna push like master>:<remote repo branch name like master> (pushes local change to remote repo and origin/branch\_name gets auto updated as per diagram shown in last section below)
* Git fetch <alias like origin>

Note: Always do: git fetch <alias like origin> before starting local changes. Then do: git log –oneline and also: Git log –oneline <origin/master> to know that we are in sync with remote repo. If not, then we need to merge as below.

* Git merge origin/master (You need to be on master branch if you wish to merge origin/master to master)

OR

* Instead of a fetch and then merge, we can do git pull (fetch + merge)
* Git push <alias like origin> <local branch\_name>:<remote repo branch name> (creates a repo in remote server with contents of your local branch that we would have created using: git branch <branch name>)
* Git push <alias like origin> :<remote branch name> (Deletes specified branch from remote repository) or git push <alias like origin> --delete <remote branch name>

Move a file:

* git mv <filename> <new folder/filename> and then git commit -m "msge here"

Reset:

* git reset --soft/mixed/hard <hash> (Brings HEAD to this hash commit. Soft brings changes after this hash to staging while mixed brings it to Local area.)

First Commit

* Add a new file:

Create a directory say, My\_First\_Project.

Go inside it and open git bash. Type: **git init** (now git tracks changes inside this project folder).

Add a new file say, first\_file.

Now, **git add .** (adds everything from current directory to git staging)

Now, **git commit -m “Initial First commit”** (adds to repository)

*Instead of i) git add . and ii) git commit -m “Msge”,*

*We can do it in one line also: git commit -am “Msge” (it sends all modifications to repository. Doesn’t work if we created new files or deleted files.)*

* See all commits with Git log or detailed commit with Git diff:

We can see this commit using: **git log** or **git log -n <number of commits>** or **git log –since=2021-06-15** or **git log –until=2021-06-15** or **git log –author=”Kush”** or **git log –grep=”<commit msge keyword>”**(shows all git commit logs we have done)

**Git log --oneline or git log –format=oneline** (gives all commits with their commit msges in single line)

**Git log –oneline –graph –all --decorate**

**Git log <start hash>..<end hash>** (Gives all commits in between these 2 hash commits)

**Git log <start hash>..<end hash> <filename>** (Gives all commits in which specified file was changed)

Git log -p <start hash>..<end hash> <filename> (gives patches to this file in range of mentioned commit)

**Git status** (tells us about status of repository, staging area and local)

**Git diff or git diff <filename>** (shows us the difference between repository and local after a change done locally)

**Git diff --staged** (shows us the difference between repository and staging area)

**Git diff --color-words <filename>**

**Git diff <hash>**

**Git diff -w <hash>** (-w is for ignoring spaces in commits)

**Git diff <hash1>..<hash2>**

**Git diff <hash1>..<hash2> <filename>**

**Git diff <branch1>..<branch2>**

**Git show <hash>** (shows details about commit files)

* Ammend your last commit

Suppose, you created a file and added some content.

Now, you do: **git add .** and then **git commit -m “msge here”**

But then, we realize we want to do more change in the same file. So, we can edit our last commit as well.

Make more changes in the file in file system and then use: **git commit --amend -m “<last commit msge or we can give a new msge also>”**

* Revert your last commit

Create a file and do: **git add .** and then **git commit -m “Msge here”**

Do: **git log** (shows the commit with its hash)

Now, to revert this commit, use: **git revert <hash>** (Directly performs a new commit to the repository with a revert of the last commit. We can change the commit msge here for the revert)

* List all branches, Switch among different branches, Rename branches, Delete branches of your project

**Git branch** (shows all branches in our project. Also shows the active branch)

**Git branch <BRANCH NAME>** (creates a new branch. HEAD will still point to previous branch in which we were. We can check that with: **git branch**)

**Git checkout <branch name>** (Switches to specified branch. We can confirm with: **git branch**)

OR

**Git checkout -b <branch name>** (creates a new branch and also switches us into it in single command)

Example:

Lets say we are in master branch right now. We have a single file with some content already in repo also.

Now, do: **git branch new\_feature** (creates a new branch with all master branch contents in it i.e the single file. but we are still in master branch)

Do: **git checkout new\_feature** (Now, we are in new\_feature branch)

Do some changes in the single file we have. Then: **git add .**

**Git commit -m “Edited the file content”** (Notice that commit is done in new\_feature branch only.)

**Git log** (Notice that commit is done in new\_feature branch only.)

Now, switch back to master: **git checkout master** (switches us back to master branch of project.)

Now, do: **git log** (Notice that commit not available here and is done in new\_feature branch only.)

If we switch back to new\_feature and do: **git branch –merged** (it shows that everything in master is there in new\_feature by returning list :”Master”, “New-feature” which means that master is there as branch, then, new\_feature has everything that master has in it plus it might have additional changes.)

To rename a branch, we can do:

**Git branch –move <existing branch name> <new branch name>**

Or

**Git branch -m <existing branch name> <new branch name>**

To delete a branch, we can do:

**Git branch –delete <branch name>** or **git branch -d <branch name>**

* Checkout file/folder

Lets say you are in master branch. May be you changed a file/folder by mistake and want to retrieve repository version of it. Basically its like discarding your local changes and getting repo)

**Git checkout -- <file or folder name>**

* Checkout a branch

**Git checkout (checks out master branch by default)**

**Git checkout <branch name>**

Note: Whenever we checkout a branch, make sure we don’t have a local change in our working directory else git wont allow and we either have to discard it or stash it.

* Checkout a specific version

Do: **git log** (shows our commit list)

Choose any commit out of the list and note down its hash.

Now, do: **git checkout <hash> -- <filename>** (Checks out the file from that version of commit and also directly add this into our staging area)

* Delete an existing file

Now, lets delete a file, say first\_file.

Now, do: **git rm .** or **git rm <filename>** (its just like git add . but for delete we use git rm . Also, it sends the change to staging)

Now, do **git commit -m “Deleting file”** (adds change to repository)

OR

If we don’t wanna delete file manually from file system. Then, don’t do it. Just directly use:

**Git rm <filename>** (Remember, it deletes the file from file system and also adds the change to staging)

Now, do **git commit -m “Deleting file”** (adds change to repository)

* Rename a file

Now, just rename the file in file system.

Then, **git rm <previous name of the file>**

**Git add <new name of the file>**

**Git commit -m “Rename the file”**

OR

Don’t manually rename the file in the file system.

Just do: **git mv <previous name of file> <new name of the file>** (Remember, it renames the file and also adds it to staging)

**Git commit -m “rename the file”**

* Move a file

Create a new folder. Now, move the file into this folder.

Then, **git add <new folder>**

**Git rm <filename>**

**Git commit -m “Move to new folder”**

OR

**Git mv <filename> <new folder/filename>** (it moves the file to new folder and also adds change to staging)

**Git commit -m “Move to new folder”**

* Remove new changes in working directory in one go

Suppose, you do: **git checkout** (it gives you say, 1.txt and 2.txt files from repo)

Now, you add 2 new files: dummy1.txt and dummy2.txt

Also modify content of 1.txt

Do: **git status** (shows that 1.txt was modified and dummy1 and dummy2 are newly added.)

Do: **git clean -n** (shows what all will be cleaned from local directory. Its just a test and doesn’t delete anything)

Do: **git clean -f** (deletes dummy1 and dummy2 as they are newly created files. Leaves 1.txt as it is.)

* Remove files from staging area (Suppose you modified 2 files and added them to staging but you want only one file in staging and one in local only for committing it in later time)

Create 2 files. Then, do: **git add .**

Now, both files are in staging. Now, do: **git status** (It will show a command to bring back the file to local.)

* Reset to previous versions (soft, mixed, hard)

**Git reset –soft <hash of a previous commit>** (Brings the HEAD pointer to that commit and any further commits will be done right after this one. It also brings any other changes after the hash commit to our staging area)

Ex:

1. Add dummy.txt and do: **git add .** and then, **git commit -m “Added dummy.txt file”**
2. Delete dummy.txt and do: **git rm dummy.txt** and then, **git commit -m “Deleted dummy.txt”**
3. Add 1.txt and 2.txt and do: **git add .** and then, **git commit -m “Added 2 files”**
4. Now, Make changes locally in 1.txt file. do: **git add .** and then, **git commit -m “Updated 1.txt”**
5. Now, Delete 2.txt and do: **do: git rm 2.txt** and then, **git commit -m “Deleted 2.txt”**

Now, do: **git logs** (Save these logs). At this point, we have made 5 commits (5 hashes are there) and we have only 1.txt file with us. Lets take the hash of (2), then do:

**git reset –soft <hash>**

This brings the HEAD pointer to this commit and all changes after (2) are brought to our staging. i.e. staging will have now 1.txt in it.

**Git reset –mixed <hash of a previous commit>** (Brings the HEAD pointer to that commit and any further commits will be done right after this one. It also brings any other changes after the hash commit to our local area)

Ex:

1. Add dummy.txt and do: **git add .** and then, **git commit -m “Added dummy.txt file”**
2. Delete dummy.txt and do: **git rm dummy.txt** and then, **git commit -m “Deleted dummy.txt”**
3. Add 1.txt and 2.txt and do: **git add .** and then, **git commit -m “Added 2 files”**
4. Now, Make changes locally in 1.txt file. do: **git add .** and then, **git commit -m “Updated 1.txt”**
5. Now, Delete 2.txt and do: **do: git rm 2.txt** and then, **git commit -m “Deleted 2.txt”**

Now, do: **git logs** (Save these logs). At this point, we have made 5 commits (5 hashes are there) and we have only 1.txt file with us. Lets take the hash of (2), then do:

**git reset --mixed <hash>**

This brings the HEAD pointer to this commit and all changes after (2) are brought to our local. i.e. working tree will have now 1.txt in it. Staging wont have anything.

**Git reset –hard <hash of a previous commit>** (Brings the HEAD pointer to that commit and any further commits will be done right after this one. It doesnt bring any other changes after the hash commit to our local or staging area)

Ex:

1. Add dummy.txt and do: **git add .** and then, **git commit -m “Added dummy.txt file”**
2. Delete dummy.txt and do: **git rm dummy.txt** and then, **git commit -m “Deleted dummy.txt”**
3. Add 1.txt and 2.txt and do: **git add .** and then, **git commit -m “Added 2 files”**
4. Now, Make changes locally in 1.txt file. do: **git add .** and then, **git commit -m “Updated 1.txt”**
5. Now, Delete 2.txt and do: **do: git rm 2.txt** and then, **git commit -m “Deleted 2.txt”**

Now, do: **git logs** (Save these logs). At this point, we have made 5 commits (5 hashes are there) and we have only 1.txt file with us. Lets take the hash of (2), then do:

**git reset --hard <hash>**

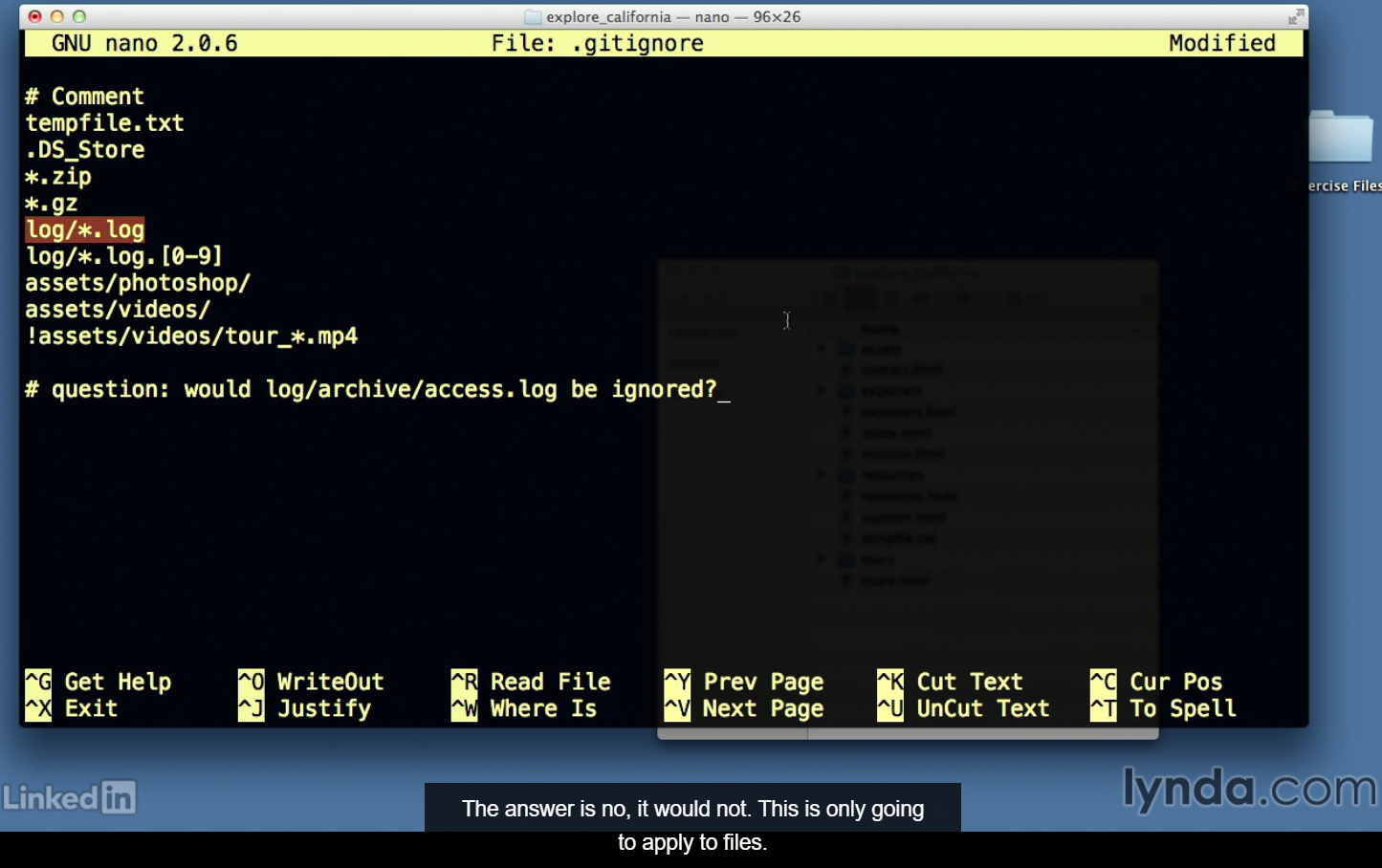
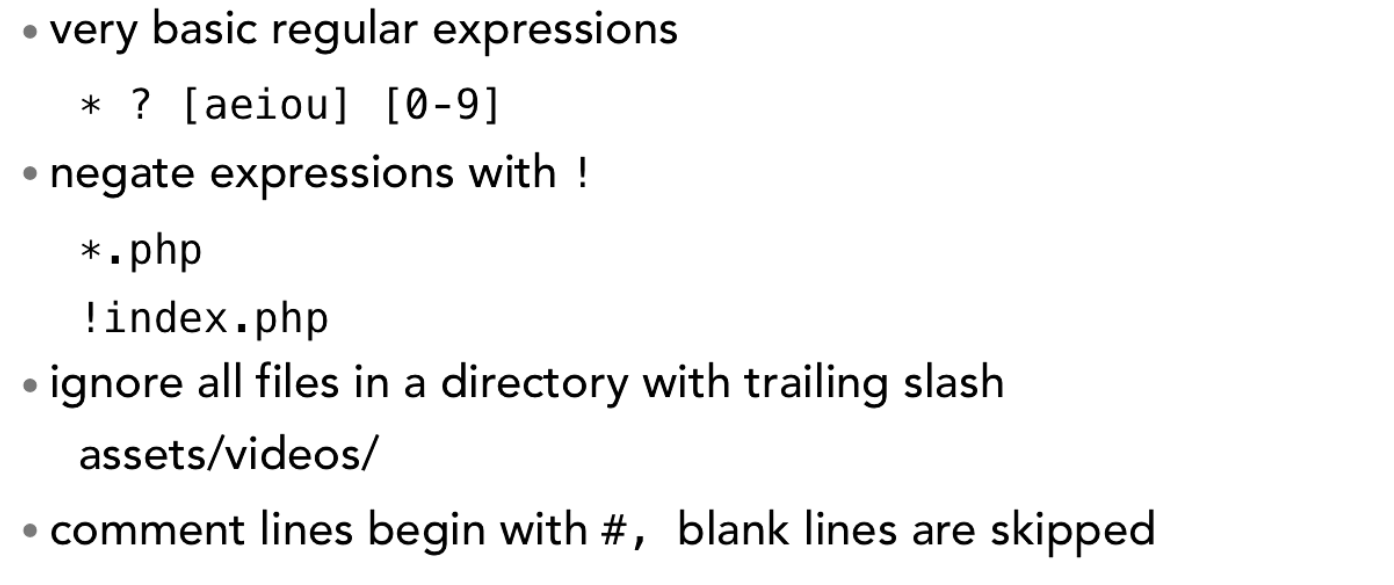
This brings the HEAD pointer to this commit and all changes after (2) are with git only as objects and are not brought to our Local or Staging.

We can always bring back the HEAD pointer to its original position by git reset –(soft/hard/mixed) <latest hash>

* Project wise .gitignore files

Add a new file in project directory with name “.gitignore”

Now, we can tell git to ignore bunch of files by mentioning it in this file. We need to commit this file as well to repo.



* .gitignore\_global file

Create a .gitignore\_global file. Add whatever you wish to ignore globally. Then do:

**Git config --global core.excludesfile <path to file like ~/.gitignore\_global>**

* Track empty Directory using .gitkeep

By default, if you create a new directory in your project and do “**git status**”. It wont track that directory. So, ideally, we also need to add either a .gitignore file or .gitkeep file in it. Then, “**git status**” tracks it. We can keep these .gitignore or .gitkeep files empty or put a comment.

* Tree in git

**Git ls-tree HEAD** (Head points to last commit, hence it shows all pushed files in last commit.)

**Git ls-tree master** (gives all files where HEAD points to in master branch)

**Git ls-tree master^ OR Git ls-tree master~1** (gives all files where last-1 commit points to in master branch)

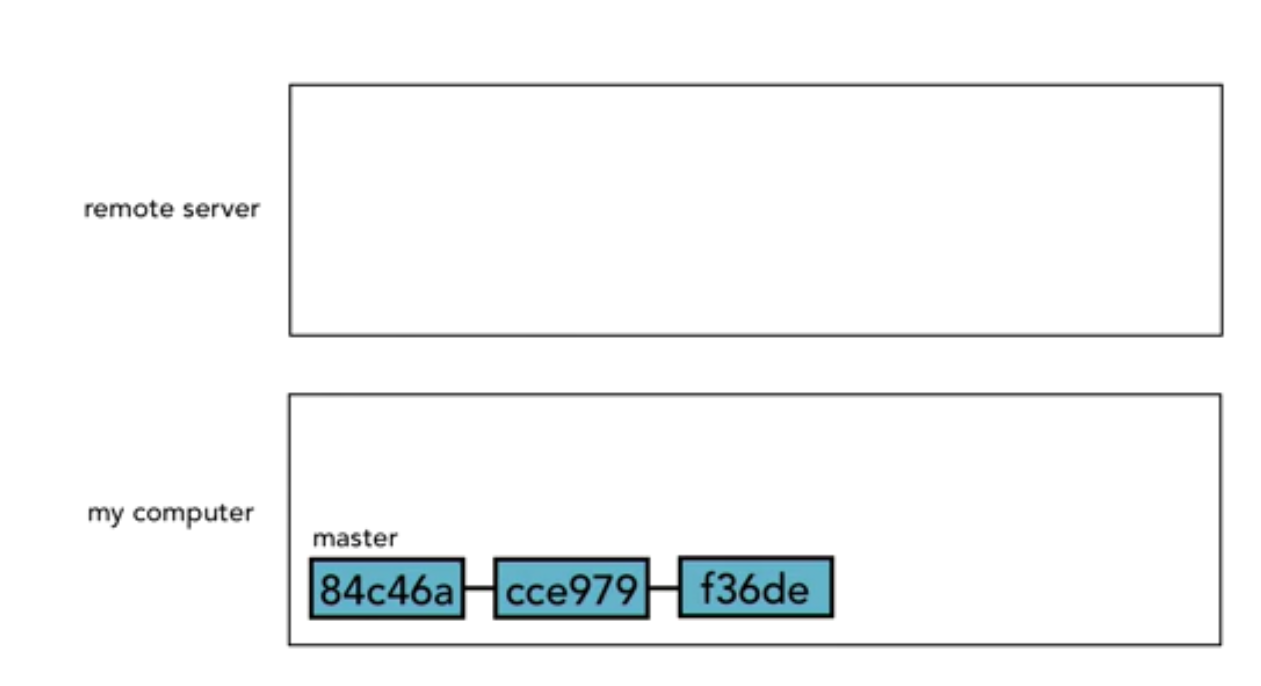
**Git ls-tree master^^ <file or folder name>/ OR Git ls-tree master~2 <file or folder name>/**

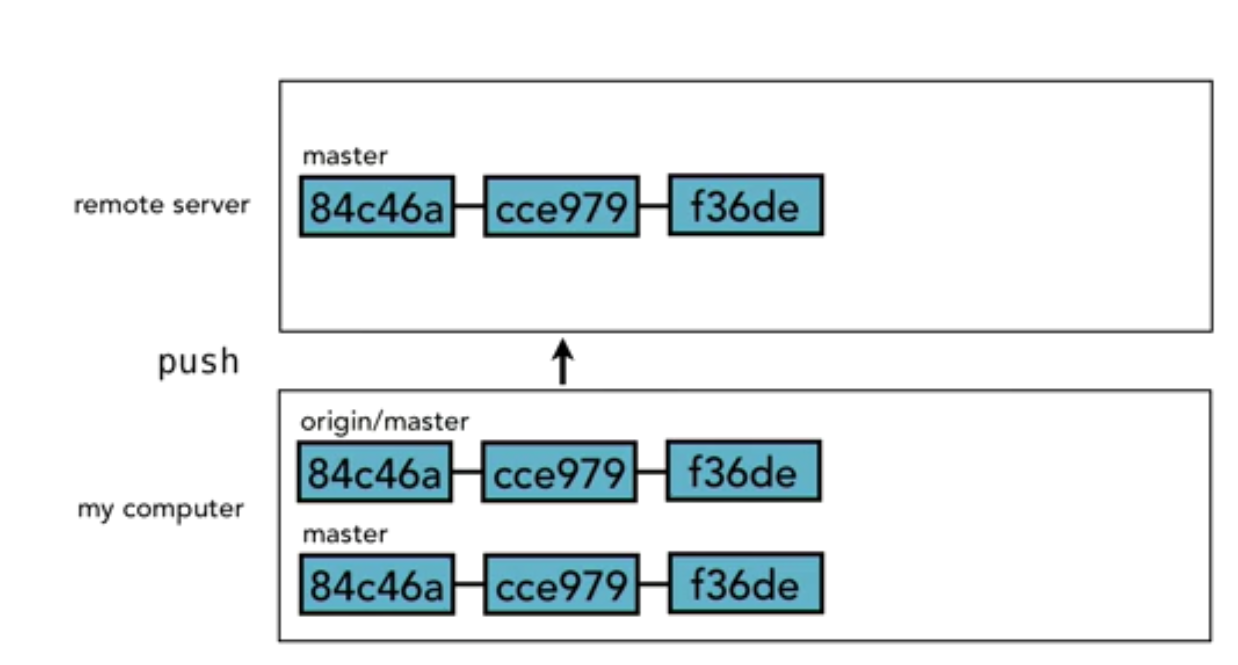
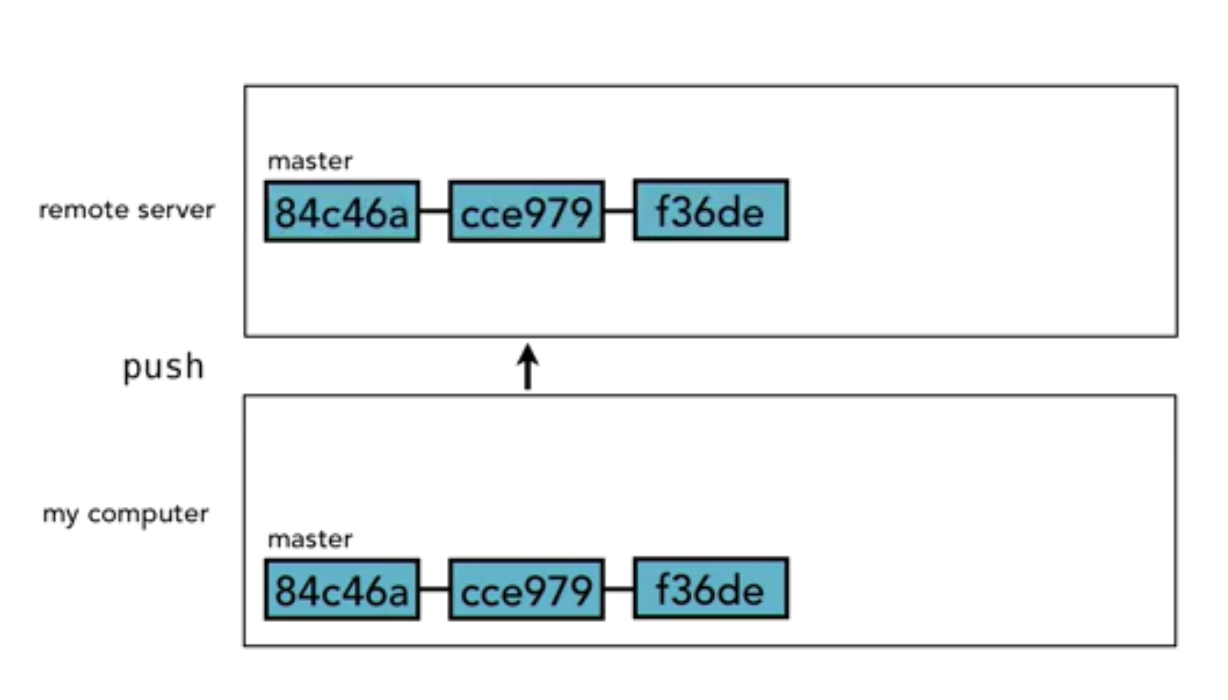
Note:

1. HEAD always points to the last/latest commit in the repository of a checked out branch. So, we can even do:  **git log HEAD** (gives us the last commit details)

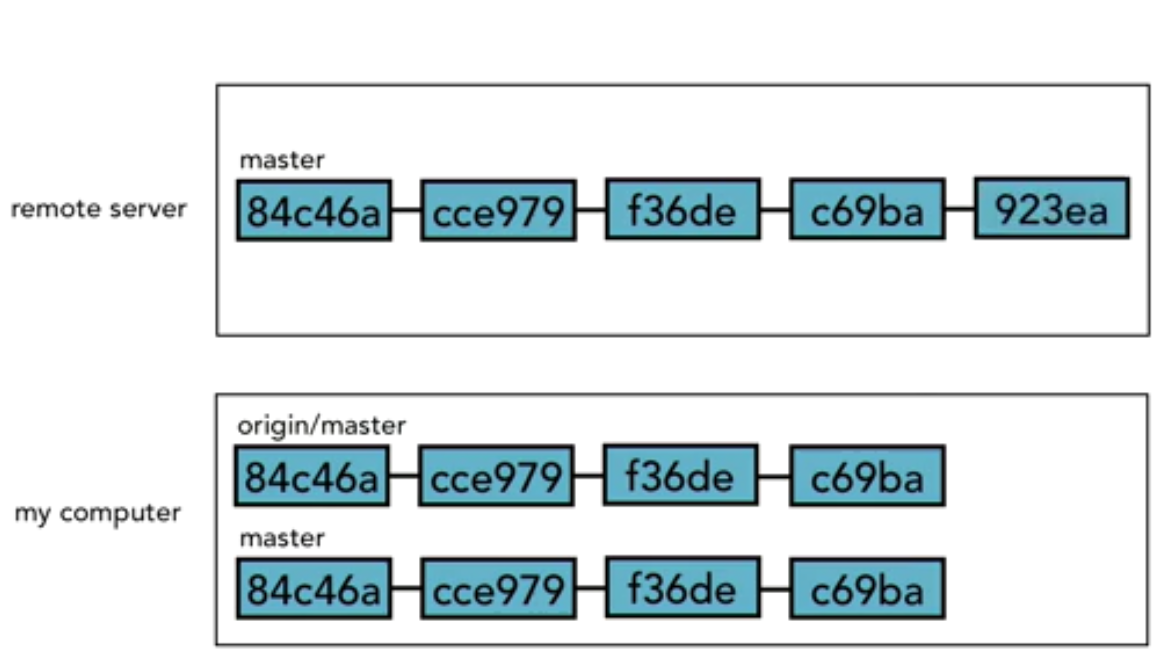
**Remote Server**

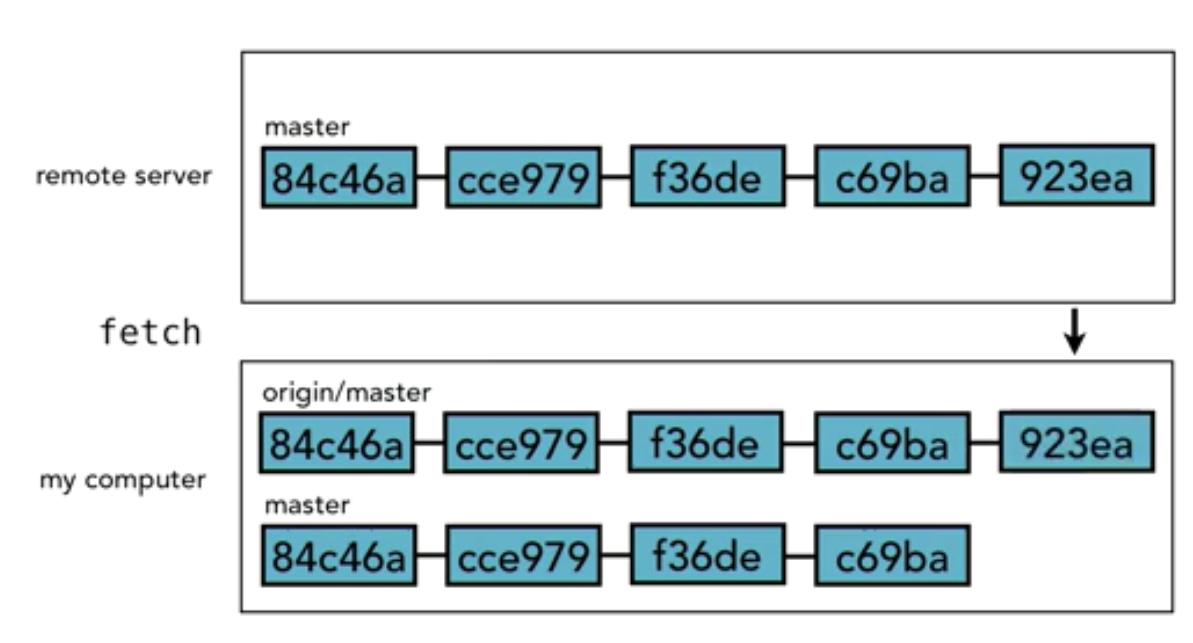
**PUSH**

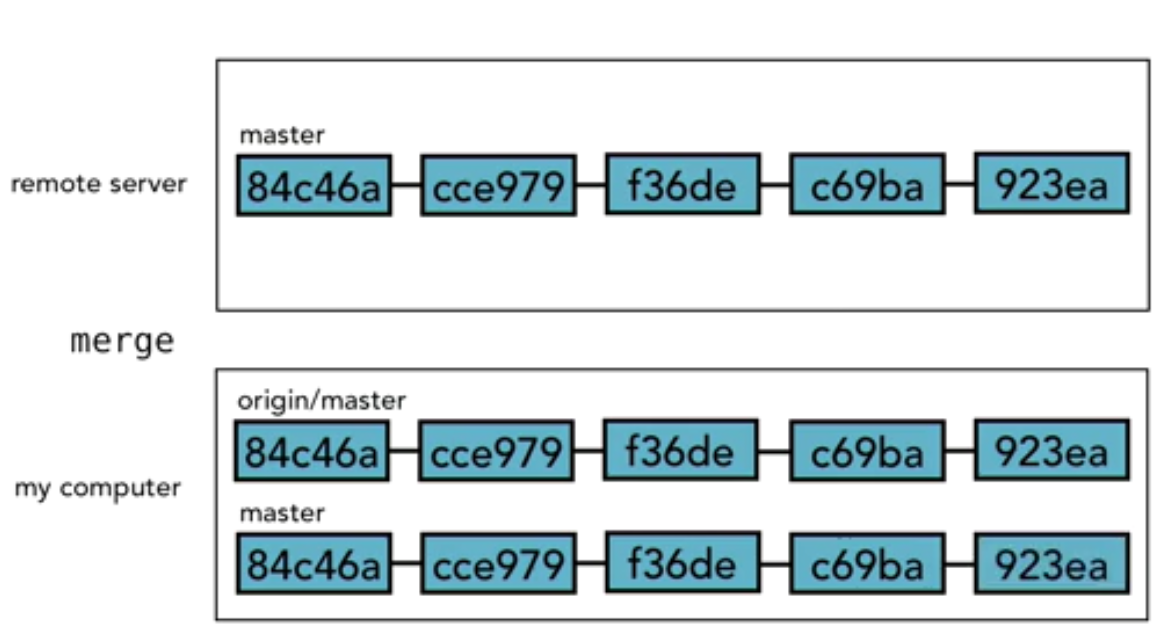
****



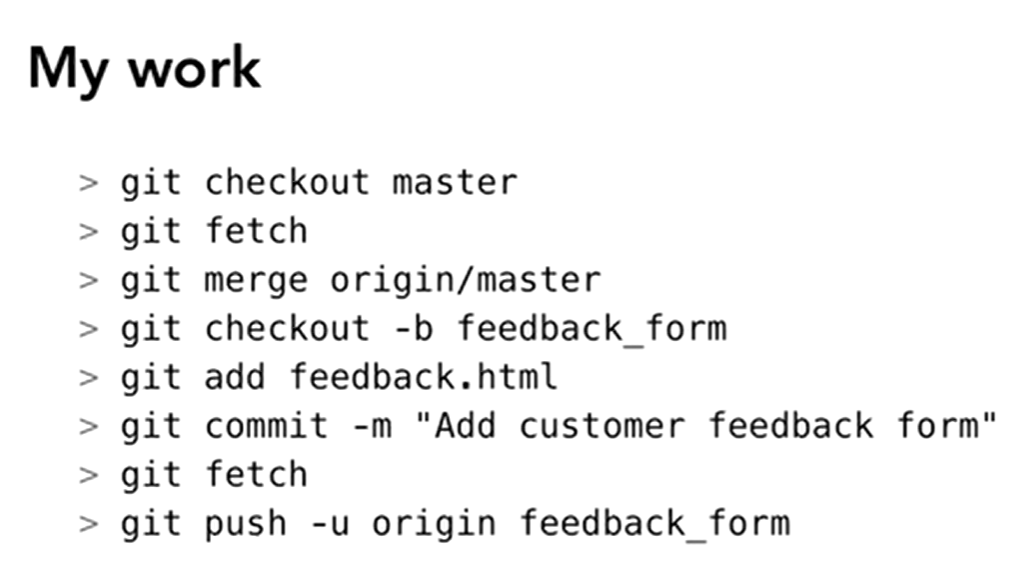
**FETCH**

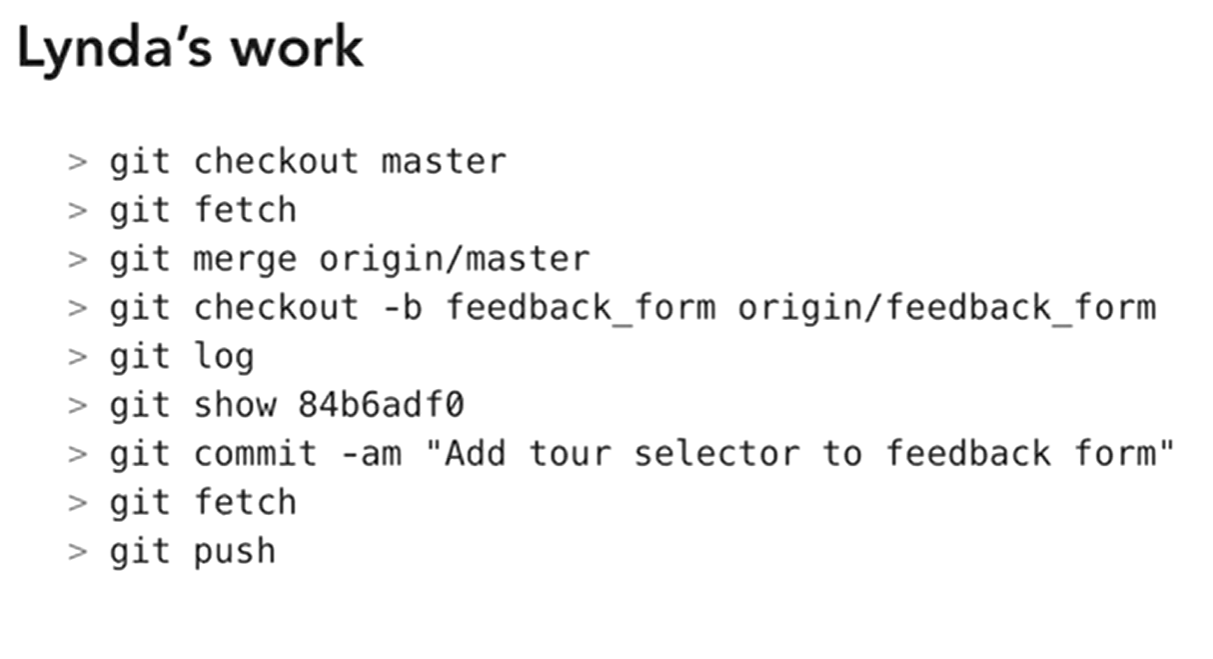


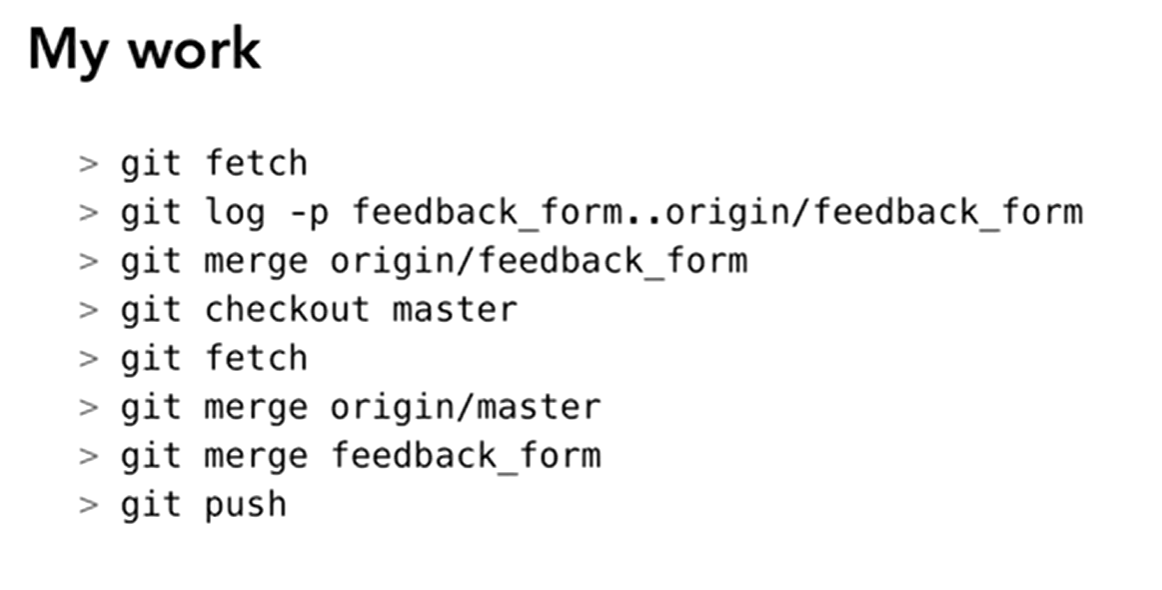




**Sample Work Flow**

****

****

****

# Update a Fork with Original repo updates

Git clone <forked repo url>

Git branch -a (note that we are in origin’s master right now)

Git remote -v (shows that forked repo is added with remote as Origin)

Git remote add upstream <original parent repo url>

Git remote -v (now it shows that forked repo is added with remote as origin but original as upstream)

Git fetch upstream (fetches original repo)

Git merge upstream/master master (merges upstream/master branch with our origin master branch)

Git status (shows that our forked repo origin/master is ahead in commits because it now has updates from upstream parent repo also)

Git push origin master:master (updates the forked repo with the local repo changes which eventually makes it even with parent repo)

git remote add upstream <url>

git pull --rebase upstream main