- Git is a software that helps you keep a track of changes you make to your work.
- It is a Version Control System/ Source Code Management Tool
- Distributed Version Control – Each user maintains their own repository.
- Changes are stored as patches
- No single repository but many working files.
- Faster/no network access required/ there is no master file- so no single point of failure.
- Use – Can be used for tracking any text file/ collaboration

## Git config

Go to Git bash

**git config –global** (to change user config)

**git config** (to change config for a project)

**git config** –system (to change system configuration)

## Get help in git

**git help log** - helps you with the command log

## Initialize

**git init** – initializing a project on git >> we are asking git to start tracking a project

1. Decide where you are keeping your project.
2. Navigate to that folder from Git
   Eg Ames Housing
- cd C:/Projects/
- cd MSBA
- cd Git_practice
3. Once you are in the folder, initialize git and from now all changes to this file will be tracked

```
$ git init
Initialized empty Git repository in C:/LAGASA_2018/MSBA/1Git_AMES/.git/
```

## Where does git store all these files?

1. Pwd: look at your current directory
2. Ls: To look at the contents of the folder
3. Ls -la: To look at all files including dot files (i.e. git files)

```
k$ ls -la
 total 11472
 drwxr-xr-x 1 Priyanka_Peace 197121        0 Mar 18 00:08  ./
 drwxr-xr-x 1 Priyanka_Peace 197121        0 Mar 18 00:10  ../
 drwxr-xr-x 1 Priyanka_Peace 197121        0 Mar 18 00:08  .git/
 -rw-r--r-- 1 Priyanka_Peace 197121 11694038 Feb 21 19:23 'Ames Iowa House Price
 Prediction (1).docx'
```

***.git is the directory that got created when we initialized git and now it will store all the changes we make.***

1. Ls -la .git >>> takes you to files inside git
2. Do not make any changes here.
3. The config file here is the project specific configuration. To change the project config you don't really need to come here.

You could just use **git config** to do that

**Make your first commit**

1. **Make the change >>> eg.** Created a new blank folder
2. Add it to the project folder that git is tracking

3. **Add the change >>>** In git bash type
   **git add .** (. = this directory. We are already in our project directory and this command asks git to add all changes made in this folder).
   If you want to add the changes to a specific file in that folder
   **Git add file (eg. git add file.py)**

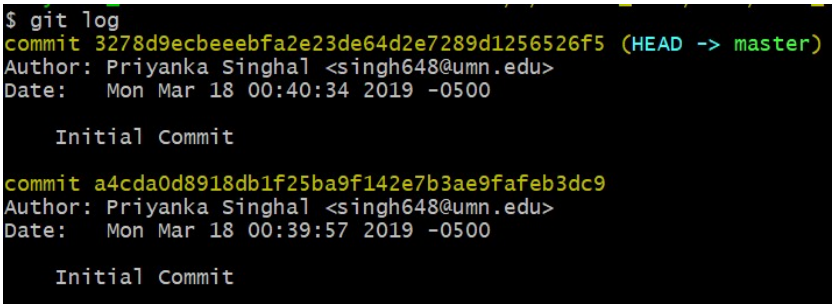
4. **Commit the change >>>** Now we commit the change
   **git commit -m "added a blank text editor"**

**How to write a commit message**

1. Title of change – in < 50 characters
2. 1 blank line
3. A description of the commit (<72 characters)
4. Present tense ---eg "This fixes a bug"

*Use a text editor for multiline commit messages*

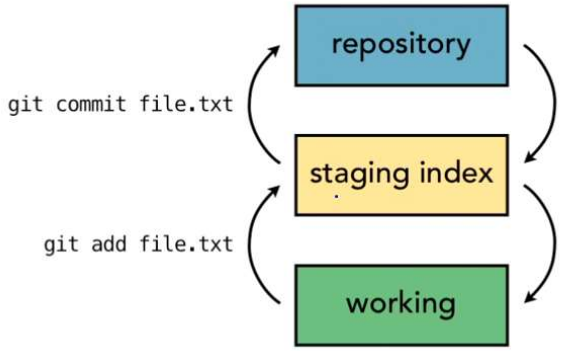
**Viewing Commit log**




Yellow line has the commit id. Everyone is unique

- Use **git help log** to see all commits
- **Git log -n 5** >>> Shows the most recent five commits
- **Git log --since= 2019-03-18 >>>** commits after a date
- **Git log --until= 2019-03-18 >>>** commits up until a date
- **Git log --author="Priyanka">>>** commits by a certain author
- **Git log –grep="code" >>>>** This will help you look at any code changes. Thus, writing commit messages clearly helps in searching and tracking logs.

**Three tree architecture**




1. **Working**: Copies where we have made all our change, saved but not yet committed
2. **Staging index**: We stage things for the commit
3. **Commit**: Permanently added

Eg. We might make 10 possible changes but not commit all. We want to go ahead with five of those changes.
Those 5 are added and they move to the staging index.
These changes are then committed.

**Refer to a commit**

- For every data, git generates a checksum (Checksums are algorithms which convert data files to a number. If the checksum of two files is different, git knows that there has been a change)
  This takes care of **data integrity**.
- Git used SHA-1 algorithm to create checksums
- SHA value >> These are 40-character unique commit ids related to every commit we make.

**HEAD**

- Points to a specific commit
- Points to the last state of the repository
- The next commit would be made on this state of the repository – i.e. the place the work has been last left off.
- Tip of the current branch in the repository

**Adding files**

- **Git status >>** Tells you the difference between the 3 trees working directory/ staging index/ repository

Master is the default branch as we have nothing to commit.
This means that there is no difference between **Working Directory** and **Repository**
Cumsum is same.

- I just created two text files.

```
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        second_file.txt
        third_file.txt

nothing added to commit but untracked files present (use "git add" to track)
```

I added two files but since they haven't been added their changes are not being tracked.

- To track changes we must add them as
  **Git add . (***if we want to add everything that is there in our directory***)**

**Git add first_file – Copy.txt** *(to add a specific file)*

- Remember this add takes it to the staging index
- We add only second file to see how git status tracks the branches

  **Git add second_file.txt**
- To see the status of our branches

  **Git status**

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   second_file.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        third_file.txt
```

- Our second file is part of the changes to be committed and our third file is not being tracked yet
- Commit the second file with

  **Git commit -m "Adding second file"**
- Now git status will just show that third file is not being tracked.

**Editing files**

- I modify first_file.txt in my local which reflects in **git status**

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   first_file.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

- I have not committed this change yet.
- Add the changes in first_file **git add first_file.txt**
- Make changes to second file as well.

**Git status**

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   first_file.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   second_file.txt
```

- The changes in first file have been committed and are in the staging directory
- The changes in the second file are in the working directory
- Now you can add this change as well and commit

  **Git add second_file.txt**

  **Git commit -m "Made changes to the first and second text files"**

>>>> Three essential steps

1. **Git status** – to see if anything has changed
2. **Git add** – to add these changes to the staging index
3. **Git commit -m "comment" –** To commit these changes to the repository

**Viewing changes before commit**

- **Git diff** >>> compares diff between working directory and repo
- **Git diff first_file.txt**

```
$ git diff
diff --git a/first_file.txt b/first_file.txt
index b830df0..da74f98 100644
--- a/first_file.txt
+++ b/first_file.txt
@@ -1 +1 @@
-This is the first file added to the project.
+This is the first file of the project.
diff --git a/second_file.txt b/second_file.txt
index bc86162..84478d8 100644
--- a/second_file.txt
+++ b/second_file.txt
@@ -1 +1,2 @@
-This is the second file added to this project
+This is the second file of the project.
+This is the most important one.
```

- Red lines are the old text, green lines is the new text. The blue lines refer to the line numbers.
- If we move the file to the staging index but haven't yet committed, to see changes in file, use

**Git diff – staged >>>** compares diff between staging index and repo

**Delete files**

**WAY 1**

- Manually remove the file.
- Then we add it to staging using git **rm File_to_delete1.txt**
- G**it commit "delete file1"**

**WAY 2**

- **Git rm File_to_delete2.txt** >> directly removes the file and moves it to staging area
- **Git commit -m "delete file2**"

- In way 2, the file is not available in recycle bin. But its latest saved version can be retrieved from your repo.

**Rename files**

**WAY 1**

- Manually change it from the file status
- Git sees this as old file deleted and a new file added
- **Git add new files**
- **Git rm old file**
- **Git commit -m ""**

**WAY 2**

- **Git mv first_file.txt first_file_new_name.txt**
- This takes care of renaming and moving the file to staging index
- **Git commit -m ""**

**To move a file to a folder**

**Git mv third_file.txt moving_file/third_file.txt**
**Git commit -m ""**

**Example: Explore California – website**

**Getting started**

1. **Git init –** Initialize – ask git to start tracking this folder
2. **Git add . –** Add all files in the folder to staging index
3. **Git commit -m "initial commit"-** move these files to your repository. Now any changes that happen will be tracked against the files here until you make another commit.

**Committing changes**

**Eg1** You change the 24hour support number on the site

- **Git status –** shows you the files changed
- **Git add . –** Adds these changes to the staging index
- **Git commit -m "changed 24-hour contact number from XXXX to XXXY"**

**Eg2** Change an html file name

- **Git mv tours/tour_details/backpack.html tours/tour_details/backpack_cal.html**
- **Git add .**
- **Git commit -m ""**

**Undo changes**

- **Working directory –** you made a change by mistake
- **Git diff –** Tells you the changes you made by comparing your changed file with the latest it has in the repository
- **git checkout file1 –** git replaces your file1 in working directory with the file1 in repo. Note: if the file1 name exists twice, git wouldn't know exactly which one to checkout and there might be an error.
- **Git checkout –file1 –** this tells git to be in the same branch as you are and to checkout file1 avoiding errors. (-- means same branch as I am in git right now)

**Unstage**

- We have made a change and moved it to the staging index. But we want to move it back to working directory.
- You may want to recheck it or assemble similar changes and make a collective commit.

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage

        modified:   resources.html
```

- Note the comment in brackets in the last white line gives a hint here.
- **Git reset HEAD file1 –** It resets the HEAD in the staging index to the file1 in repository.
- Note: the HEAD of the working directory points to the changed file1 as we haven't undone that change.

**Undoing Commits**

- Git refers to commits with a hash
- Only the most recent commit can be changed. The one where the HEAD points to now.
- You could add your new change -> move it to the staging index with ***git add***
- When committing use **git commit –amend -m "message"**
- This would update the latest commit to incorporate your new change.
- You could also use it to change the commit message.
- Remember a new SHA gets generated even for this change.

**<u>Undoing older commits</u>**

- You need to make a change to undo the older commit
- You cannot hide a mistake from Git 😊

**WAY 1**

- You could manually undo the change and commit it

**WAY 2 - REVERT**

- **Git log -** copy the SHA for the commit you want to reverse
- **Git revert commit SHA**

**WAY 3 - REST**

- You guide the GIT head to a point
- **--soft –** changes the pointer in repo
- **-- mixed –** changes staging index but not working directory
- **-- hard –** changes all branches

***PS:*** *Before resetting, please copy the latest HEAD SHA from git log*

**Git reset –soft SHA** *(where you want your head to point now)*

Git diff –staged will still show you your changes in staging index

*No message generated*

**Git reset –mixed SHA**

Git diff – you still have your changes but in working directory. You will have to add them back to index and then commit

```
$ git reset --mixed 61be5e48f
Unstaged changes after reset:
M       contact.html
```

**Git reset –hard SHA**

You lose your most recent changes

```
$ git reset --hard 61be5e48f
HEAD is now at 61be5e4 aa
```

PS: if you have copied your latest SHA before resetting you can use it to go back to that HEAD. GIT does not remove it. It tracks all changes.

**Remove untracked files**

- There might be some files in the folder that git is not tracking.
- You created them but haven't added them to be tracked and they appear under untracked files.
- You can remove them from git using
- **Git clean -n: -** It's a check. It tells you which files the command will remove if you used the command clean

```
$ git clean -n
Would remove junk.txt
```

- **Git clean -f –** Removes all untracked files.

```
$ git clean -f
Removing junk.txt
```

**Ignore files**

- There may be files that you don't want to track but are important otherwise. Eg log file. It will keep changing but you don't want to commit changes in it.
- Create a temp.txt file in your project folder
- Create a .gitignore file in any text editor with a file name *.txt there. This means we are asking it to ignore all text files.
- Or write the file name you want git to ignore eg. temp.txt in the .gitignore file .
- If you want to ignore all txt files but one names project.txt, type in the .gitignore file **!project.txt**
- Save this file with name .gitignore. Save as type – all files and place it in your project folder tracked by git
- Then add gitignore file
- **Git add .gitignore**
- **Git commit -m "Git ignore file"**
- Refer to link https://help.github.com/en/articles/ignoring-files which suggests which files should be commonly ignored.
- https://github.com/github/gitignore/blob/master/Python.gitignore files you can ignore when working on a Python project

**Ignore Globally** >>> ignore these files not just for this project repository but for all repositories

- These are part of git config
- **Git config - - global core.excludesfile ~/.gitignore_global**
- The file named .gitignore_global will contain the listing of all files you want to ignore.
- You could name it anything in the form .gitignore* and place it anywhere (preferably the location of your global git config file exists)

**Ignoring tracked files**

- If you created the rule for ignoring a file now but it was already being tracked by git, you need to first untrack it.

- A file was being tracked. You don't want to track it now.
- Add that files name to your file **.gitignore**

**WAY 1**

- Remove the file **git rm filename**

**WAY 2**

- But I want to keep the file but not track it anymore
- **Git rm -- cached filename**
- It wouldn't be tracked anymore
- **Git commit -m "file removed from staging index"**
- The file will be ignored, and no future changes are tracked.

**Track empty files**

- Git doesn't track empty directories. Git tracks files not directories.
- So, add an empty text file to it as a trick to a directory you plan to track. Common name convention here is gitkeep
- **Touch pathofemptydirectory/.gitkeep>>>** creates an empty file named .gitkeep