

What is git

Git is most commonly used **distributed version control system** that helps you:

- a. To make record of versions so you can revert back when needed
- b. Makes collaboration easy

Version Control System (VCS)

Suppose you are working on a project. Your daily tasks would be:

- a. Create things (write code)
- b. Save things (save the code that you created)
- c. Edit things (code is not working, do the changes)
- d. Save the things again

When saving things again and again, we often need answers to the questions like

- when I did the last change
- why I did the changes
- what were those changes etc.

For people like me, who even can't tell what I had in breakfast, it becomes really hard to remember all the changes that I did, when I did and why I did. Now, think of the scenario when an entire team is working on a project, adding, editing and saving things.... how difficult it becomes to blame who did that

Ques: Should I use git if I am the only person working on the project?

Ans: Once you call it a project, it is worth using VCS. I would love to see a pharmacist ask: "Should I store medications in an organized way or just throw them all in a drawer? Is it worth the effort?"

This is where VCS comes into picture. VCS tracks things for you so that you can track the changes. Since VCS has the track, it can help us to revert those changes or merge those changes (which are features of VCS)

Ques: Why use VCS instead of maintaining the copy of folder and saving it with timestamp?

Ans: a. This is because if your project is of 200 GB and you made changes in 5 of the files, you will have to copy entire 200 GB project just for those 5 file changes

b. After a month, you may forget the name of those 5 files in which changes were made

c. Suppose you just copied those changed files and renamed them with timestamp to solve problem a and b, undoing a change in those 5 files will be hard to remember. That's why, we use VCS

Ques: What does **distributed means here?**

Ans: Apart from having remote repo located in a server, git provides a local repo which can be stored on the local system of people working on a same task

Ques: If git is distributed VCS, then do we have any other type of VCS too?

Ans: Yes, VCS are of two types:

- a. Distributed VCS
- b. Centralized VCS

Ques: Is there any other VCS tool apart from git that is available in market?

Ans: There are other VCS tools too like SVN, Perforce, ClearCase etc

Ques: What is the difference between SVN and git?

Ans: The major difference between SVN and git is its nature. SVN is centralized VCS while git is distributed VCS

Ques: What is difference between git, github and gitlab?

Ans: Although they may look like same, but git, github and gitLab are different

Git is a VCS tool which is used to track changes. Github and GitLab are service providers that manage your git repos so that people can access your code from anywhere. They offer all the functionality of VCS while adding their own features

Git has three states:

- a. Working tree
- b. Staging area
- c. Repository (HEAD)

Working tree is where the actual files and folders are stored on the local machine. Any change that is made in working tree is considered 'untracked' until the changes are made to track by running 'git add' command. Staging area is an intermediate area where we can prepare changes for our next commit. Any changes made in the file in staging area will move the file to modified area. Modified files can be moved back to staged area by running "git add filename" command. Files that are committed from staging area are moved to the committed area. Repo is where all commits are stored. HEAD is the pointer to the latest commit in that branch

Although we can directly move the changes from working tree to repo without using staging area, but this is not considered as a good practice

Using git

In order to use git, make sure you have git installed on your system by running

```
$ git --version
```

```
ak.katiha@GSG1PM-GI0210:~$ git --version  
git version 2.25.1
```

Now, that you have git installed on your system, let's see how we can configure the git environment. Git comes with a tool called `git config` that is used to set configuration variables. These variables can be stored at 3 different places:

- a. Local: It is config file in the git directory (`.git/config`) of repo that you are currently using
- b. Global: It will be located at `$home/.gitconfig` location and contains values specific w.r.t the user

Passing `--global` will read and write from this file

- c. System: It will be located `[path]/etc/gitconfig` and contains values applied to every user on the system and all their repo

Passing `--system` will read and write from this file

Each level overrides values present in previous level. So local changes will override global changes will override system changes

To see all your settings and from where they are coming from, use

```
$ git config -l --show-origin
```

```
ak.katiha@GSG1PM-GI0210:~$ git config --list --show-origin  
file:/home/ak.katiha/.gitconfig user.name=gem-akatiha  
file:/home/ak.katiha/.gitconfig user.email=akshay.katiha@geminisolutions.com
```

These config files can be used to make custom config messages, aliases and for other useful changes that we need to configure our git with

Our Identity

We can set our username and email address using command given below. It is important because every git commit uses this information

```
$ git config --global user.name "<username>"
```

```
$ git config --global user.email "<useremail>"
```

```
ak.katiha@GSG1PM-GI0210:~$ git config --global user.name "gem-akatiha"
ak.katiha@GSG1PM-GI0210:~$ git config --global user.name "akshay.katiha@geminisolutions.com"
```

Default Editor

Another useful command that we should configure is the default editor for git. Here, notepad++ is configured to be used as default editor for git

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

Note: To start the notepad++, use command `$ start notepad++ <fileName>`

Default branch name

By default, git will always create a master branch when we create a new repo with git init command

```
akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop
$ git config --list --show-origin
file:C:/Program Files/Git/etc/gitconfig diff.astextplain.textconv=astextplain
file:C:/Program Files/Git/etc/gitconfig filter.lfs.clean=git-lfs clean -- %f
file:C:/Program Files/Git/etc/gitconfig filter.lfs.smudge=git-lfs smudge -- %f
file:C:/Program Files/Git/etc/gitconfig filter.lfs.process=git-lfs filter-process
file:C:/Program Files/Git/etc/gitconfig filter.lfs.required=true
file:C:/Program Files/Git/etc/gitconfig http.sslbackend=openssl
file:C:/Program Files/Git/etc/gitconfig http.sslcainfo=C:/Program Files/Git/mingw64/etc/ssl/certs/ca-bundle.crt
file:C:/Program Files/Git/etc/gitconfig core.autocrlf=true
file:C:/Program Files/Git/etc/gitconfig core.fscache=true
file:C:/Program Files/Git/etc/gitconfig core.symlinks=false
file:C:/Program Files/Git/etc/gitconfig pull.rebase=false
file:C:/Program Files/Git/etc/gitconfig credential.helper=manager
file:C:/Program Files/Git/etc/gitconfig credential.https://dev.azure.com.usehttppath=true
file:C:/Program Files/Git/etc/gitconfig init.defaultbranch=master
```

We can overwrite this setting by using command `$ git config --global init.defaultBranch <branchName>`

How to get config values?

- a. If you want to check your configuration settings, you can use command:

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

```
ak.katiha@GSG1PM-GI0210:~$ git config --list
user.name=gem-akatiha
user.email=akshay.katiha@geminisolutions.com
```

- b. You can also use command `$ git config <key>` to get specific value

```
akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop
$ git config user.email
akshaykatiha@gmail.com
```

If you are confused about the value, you can always use the command `$ git config --show-origin` to get the resource of the value

```
akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop
$ git config --show-origin user.name
file:C:/Users/akshay.katiha/.gitconfig Akshay Katiha
```

Getting a git repo

We can obtain a git repo in two ways:

- a. Making a local directory that is currently under VCS into git repo
- b. Cloning an existing git repo from elsewhere

A. Initializing a repo in an existing directory

If a directory is not under version control and you want to start controlling it with git, then go to that project's directory and type the command given below:

```
$ git init
```

This will create a new subdirectory named `.git` that contains all necessary files. This command also creates a new main branch

```
ak.katiha@GSG1PM-GI0210:~/Desktop$ cd gitInternal/  
ak.katiha@GSG1PM-GI0210:~/Desktop/gitInternal$ git init  
Initialized empty Git repository in /home/ak.katiha/Desktop/gitInternal/.git/  
ak.katiha@GSG1PM-GI0210:~/Desktop/gitInternal$ ls -a  
.  
..  
.git
```

B. Cloning an existing repo:

If you want to get a copy of an existing git repo, then use the command below

```
$ git clone <repo-link>
```

```
ak.katiha@GSG1PM-GI0210:~/Desktop$ git clone https://github.com/gem-akatiha/gitInternal.git  
Cloning into 'gitInternal'...  
remote: Enumerating objects: 3, done.  
remote: Counting objects: 100% (3/3), done.  
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0  
Unpacking objects: 100% (3/3), 596 bytes | 596.00 KiB/s, done.
```

This will create a directory named `gitInternal`, initialize a `.git` subdirectory inside it and pull down all the data available in that repo and checks out a working copy of the latest version

Ques: Is it possible to change the name of directory getting created during clone to something else?

Ans: Yes, you can change the name of the cloned repo by passing the name of the directory in which you want to clone the repo using command given below

```
$ git clone <repo-that-needs-to-be-cloned> <directoryName>
```

```
ak.katiha@GSG1PM-GI0210:~/Desktop$ git clone https://github.com/gem-akatiha/gitInternal.git newGitInternal
Cloning into 'newGitInternal'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 596 bytes | 596.00 KiB/s, done.
```

Note:

- a. If directoryName not already exists, then git will create an empty directory and then clone the required repo
- b. Git will throw error if the directoryName already contains a .git file

```
ak.katiha@GSG1PM-GI0210:~/Desktop$ git clone https://github.com/gem-akatiha/gitInternal.git newGitInternal
fatal: destination path 'newGitInternal' already exists and is not an empty directory.
```

Ques: Is there a way to get the URL of the repo that you used to clone the repo?

Ans: Yes, by running the command `$ git config --get remote.origin.url`. Note: This command will return none if the repo was created using git init command

```
akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/athena_ui (master)
$ git config --get remote.origin.url
https://github.com/gem-salotinagpal/athena_ui.git
```

There are two major protocols that are used to clone a repo. One is HTTPS and another one is SSH.

Recording changes to the repo

Once a git repo is created, the next thing that you want to do is start making changes into the repo each time the project reaches a state you want to record. In git, each file in repo can be in one of the three states:

- a. Tracked
- b. Untracked
- c. Ignored

Untracked basically means that git see a file that is not available in the last snapshot (commit)

To determine which file are in which state, use the command `$ git status`

```
ak.katiha@GSG1PM-GI0210:~/Desktop/gitInternal$ git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
```

Clean working directory means none of your tracked files are modified and there is no untracked file. This command also informs you the branch that you are working on

Let's suppose now you have added a few files here. You can see your untracked files by running the `$ git status`

```
ak.katiha@GSG1PM-GI0210:~/Desktop/gitInternal$ git status
On branch main
Your branch is up to date with 'origin/main'.

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

Note: If your repo contains an empty directory, git status will not track that directory. This is because git is a content tracker and empty directories are not content

```
ak.katiha@GSG1PM-GI0210:~/Desktop/gitInternal$ ls
newFile  README.md
ak.katiha@GSG1PM-GI0210:~/Desktop/gitInternal$ mkdir newFolder
ak.katiha@GSG1PM-GI0210:~/Desktop/gitInternal$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   newFile
```

Note: If you don't want to get all the details, command `$ git status --short` or `$ git status -s` can be useful. It uses flags to show status of the files inside the repo. Short status flags are:

?? - untracked files

M – Modified files

A- Files added to stage

D – Deleted files

There are two columns to the output — the lefthand column indicates the status of the staging area and the right-hand column indicates the status of the working tree

Tracking new files

To begin tracking the files, run the command `$ git add <filename>`

To add more than one file at the same time, use `$ git add - - all` or `$ git add .`

Note: `$ git add -A` is shorthand command for `git add - - all`

```
ak.katiha@GSG1PM-GI0210:~/Desktop/gitInternal$ git add newFile
ak.katiha@GSG1PM-GI0210:~/Desktop/gitInternal$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   newFile
```

At this stage, we can do two things:

- a. Rollback the staged changes
- b. Modify the staged changes
- c. Commit the staged changes

If you commit at this point, the version of file at the time you ran the command `git add` will be added in the subsequential historical snapshot.

If you want to roll back the changes, simply run command `$ git restore - -staged <fileName>`

```
ak.katiha@GSG1PM-GI0210:~/Desktop/gitInternal$ git restore --stage newFile
ak.katiha@GSG1PM-GI0210:~/Desktop/gitInternal$ git status
On branch main
Your branch is up to date with 'origin/main'.

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


Staging Modified Files

When a staged file is modified before it is committed, it has to be staged again in order to commit

```
ak.katiha@GSG1PM-GI0210:~/Desktop/gitInternal$ vi newFile
ak.katiha@GSG1PM-GI0210:~/Desktop/gitInternal$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   newFile

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

Here you can see that a strange thing has happened. The modified file is available under both staging area (Changes to be committed) and working directory (**Changes not staged for commit**) heading. This happens because git stages a file exactly as it is when you run the git add command. Even after modifying the file, if we run the git commit at this point, the version of file as it was when we last ran the git add command will go into commit, not the version of the file as it looks after modification

Ques: Is there any difference between `git add .` and `git add -A` command?

(Read more about this topic from [here](#))

Ans: Yes, there is a difference between both the commands. The main difference between `git add .` and `git add -A` is that `git add .` command will not stage changes, if there are in other subdirectories/higher-level directories but running `git add -A` will stage all the changes inside the git repo. **git add -A** ensures that the staging area fully represents the current state of the repository. Let's understand this with an example.

Suppose there are two directories inside our git repo named beta and dev and a untracked file3 in parent repo. Each directory has 2 files something like this

gitPractice/

```
├─ beta/
|   ├── file_beta1
|   └─ file_beta2
├─ dev/
|   ├── file_dev1
|   └─ file_dev2
└─ file3
```

```

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice (master)
$ ls -R
.:
beta/  dev/  file1  file2  file3

./beta:
file_beta1  file_beta2

./dev:
file_dev1  file_dev2

```

Let's go inside beta branch and run git status command. It will show that there are untracked files in current directory (./), in directory dev (../dev/) and file3 itself (../file3)

```

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice (master)
$ cd beta

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice/beta (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    ./
    ../dev/
    ../file3

```

Use command git status -u to get additional information about the untracked files. **git status -u** (or **git status --untracked-files**) provides additional control over how untracked files are displayed.

```

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice/beta (master)
$ git status -u
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    file_beta1
    file_beta2
    ../dev/file_dev1
    ../dev/file_dev2
    ../file3

```

Let's do git add . and see what happens. This is going to stage file_beta1 and file_beta2, keeping rest files (file_dev1, file_dev2 and file3 as it is)

```

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice/beta (master)
$ git add .

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice/beta (master)
$ git status -u
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   file_beta1
    new file:   file_beta2

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    ../dev/file_dev1
    ../dev/file_dev2
    ../file3

```

Now, let's add two more file in beta branch file_beta3 and file_beta4, go to dev branch and run git add --all command and see what happens

```

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice/beta (master)
$ touch file_beta3 file_beta4

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice/beta (master)
$ ls
file_beta1 file_beta2 file_beta3 file_beta4

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice/beta (master)
$ cd .. && cd dev

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice/dev (master)
$ ls
file_dev1 file_dev2

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice/dev (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   ../beta/file_beta1
    new file:   ../beta/file_beta2

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    ../beta/file_beta3
    ../beta/file_beta4
    ../file3

```

```

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice/dev (master)
$ git add --all

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice/dev (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   ../beta/file_beta1
    new file:   ../beta/file_beta2
    new file:   ../beta/file_beta3
    new file:   ../beta/file_beta4
    new file:   file_dev1
    new file:   file_dev2
    new file:   ../file3

```

You will see that all files are staged when we run `git add --all`. This is the difference between `git add .` and `git add --all` command

Ques: What is difference between command `git add .` and `git add *` ?

Ans: The only difference between `git add *` and `git add .` is that when you will do `git add *`, it will not add files starting with dot in the current directory but will add files starting with dot in subdirectories recursively while `git add .` will add all modified files (not deleted) including files starting with dot in current directory as well as subdirectories recursively.

For ex: We have `.file3` and `file4` in `subDir` and have `.file1` and `file2` in `subDir/internalSubDir`

gitPractice/

|— subDir/

| |— .file3, file4

| |— internalSubDir

| |— .file1, file2

|—.file, file0

Let's do **git add *** from subDir and see what happens. All files which are in **subDirectory** and in **internalSubDir** will be staged except **.file3** since it lies in the current directory. **.file** and **file0** will not be staged because **git add .** and **git add *** has no effect on higher level directories

```
akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice (master)
$ cd subDir/

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice/subDir (master)
$ git status -u
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
  ../.file
  ../file0
  .file3
  file4
  internalSubDir/.file1
  internalSubDir/file2

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

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice/subDir (master)
$ git add *

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice/subDir (master)
$ git status -u
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
  new file:   file4
  new file:   internalSubDir/.file1
  new file:   internalSubDir/file2

Untracked files:
  (use "git add <file>..." to include in what will be committed)
  ../.file
  ../file0
  .file3
```

Let's add **.file4** and **file5** in **subDir**, **.file6** and **file7** in **subDir/internalSubDir**, run **git add .** and see what happens. All files in **subDir** and **subDir/internalSubDir** were staged.

```
akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice/subDir (master)
$ touch .file4 file5 internalSubDir/.file6 internalSubDir/file7

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice/subDir (master)
$ git status -u
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
  new file:   file4
  new file:   internalSubDir/.file1
  new file:   internalSubDir/file2

Untracked files:
  (use "git add <file>..." to include in what will be committed)
  ../.file
  ../file0
  .file3
  .file4
  file5
  internalSubDir/.file6
  internalSubDir/file7
```

```

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice/subDir (master)
$ git add .

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice/subDir (master)
$ git status -u
On branch master

No commits yet

changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   .file3
    new file:   .file4
    new file:   file4
    new file:   file5
    new file:   internalSubDir/.file1
    new file:   internalSubDir/.file6
    new file:   internalSubDir/file2
    new file:   internalSubDir/file7

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    ../.file
    ../file0

```

This is the difference between **git add .** and **git add ***. Read more about them [here](#)

Ques: What does `git add -u` command does?

Ans: `git add -u` or `git add --update` command is used to stage all the modified, deleted or renamed files that are already been tracked by git. Note: This command will not stage any untracked file.

For ex: There were a few files in the repo which were already being tracked by git

```

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   .gitignore
    new file:   project
    new file:   temp/project/file1

```

Now, we added a new file named file1 and modified file project that was already being tracked by git. When running `git add -u` command, the file project was added to the staging area, but file1 still remain untracked by git

```

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice (master)
$ touch file1

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice (master)
$ echo "editing project file" > project

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice (master)
$ git status -u
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   .gitignore
    new file:   project
    new file:   temp/project/file1

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

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    file1

```

```

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice (master)
$ git add -u
warning: in the working copy of 'project', LF will be replaced by CRLF the

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice (master)
$ git status -u
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   .gitignore
    new file:   project
    new file:   temp/project/file1

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    file1

```

Ignoring files (.gitignore file)

More about gitignore can be found [here](#) and [here](#)

There will be few files and directories that we do not want to be part of git commit. It can be log files, temp files etc. In order to ignore those files or directories, create a file named **.gitignore** in the root repo and add the name of the files or directories that you want to ignore. Let's create a project.log file that we do not want to track

```

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice (master)
$ start notepad++ project.log

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    file1
    file2
    project.log

```

In order to ignore the project.log file from being tracked, create .gitignore file and add project.log file to it. Run git status again. You will see that project.log file will no longer be part of git status output

```
akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice (master)
$ echo "project.log" > .gitignore

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore
        file1
        file2
```

Ques: Why am I seeing .gitignore file as untracked file if the purpose of this file is to ignore other files and directories?

Ans: This is because the **.gitignore** file, although specifying which files and directories should be ignored by Git, is itself tracked by Git like any other file in your repository.

So, using .gitignore is a way to tell git which **untracked file should remain untracked** and never get committed. What does untracked file should remain untracked means?

This means that if a file was added to the git commit, git will keep tracking the file even if you add that file to gitignore. The **.gitignore** file does not affect files that are already staged or been committed. It only affects untracked files, preventing Git from tracking them in the future. For ex: we have file1 and file2 above. Let's stage the files first and then add them to gitignore file

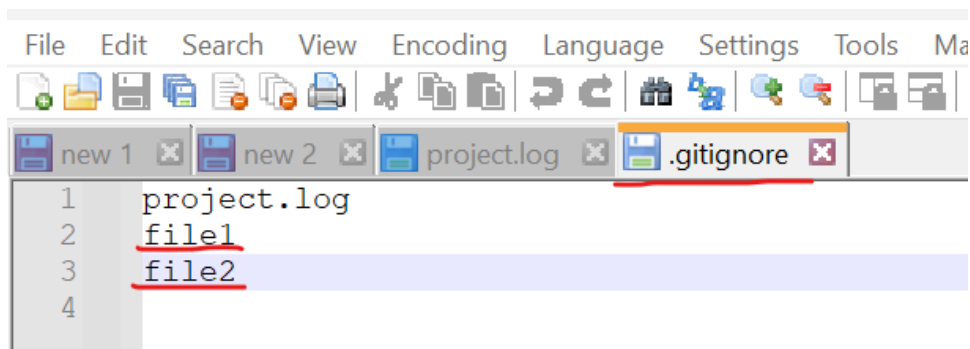
```
akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice (master)
$ git add file1 file2

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   file1
        new file:   file2

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



When you run git status again, you will see that git status did not ignore file1 and file2

```
akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   file1
        new file:   file2

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

Hence, when a file is staged or committed, adding them to gitignore will have no effect. So, it is better to avoid adding files in gitignore after committing or staging them.

Ques: When working tree is clean, we can't check files being tracked by git. One option is to list all files in the repo and check for the file names that are not added in gitignore file. Is there any other way to check name of all files being tracked by git?

Ans: Run command `$ git ls-files`. Other useful commands are given in screenshot added below.

- `git ls-files --error-unmatch <fileName>` command returns filename if it is being tracked and returns if the file is not being tracked
- `git ls-files` will list all the files being tracked by git
- `git ls-files <fileName>` will return filename if it is being tracked by git, and will return nothing if file is not getting tracked


```

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice (master)
$ git ls-files --error-unmatch file1
file1

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice (master)
$ git ls-files --error-unmatch abc
error: pathspec 'abc' did not match any file(s) known to git
Did you forget to 'git add'?

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice (master)
$ git ls-files
file1
file2

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice (master)
$ git ls-files project.log

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice (master)
$ git ls-files file1
file1

```

Ques: What if a file was committed or staged earlier, but now I want to ignore that file?
Ans: If such thing happens, you have to remove those files from commit or stage area by running commands like `git rm --cached <fileName>` or `git rebase` or `git reset HEAD` etc depending on the situation, add file names to `.gitignore` file again.

Ques: Should I have multiple `.gitignore` files or a single `.gitignore` file?

Ans: It completely depends on your repo structure and requirements. However, it is also a common practice to have multiple `gitignore` files.

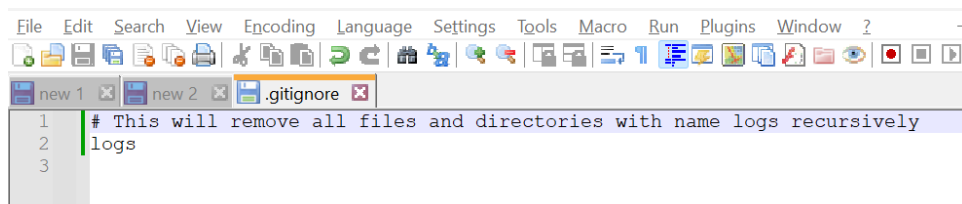
Ques: Where should I put `.gitignore` file in repo?

Ans: If you have only one `gitignore` file, it is better to keep the file at the root of the repo, which applies recursively to the entire repo. The rules in nested `.gitignore` files apply only to the files under the directory where they are located.

Rules for ignoring files in `.gitignore`

1. Blank lines and lines starting with `#` are ignored
2. `<fname>` will ignore all files and directories with name `fname`

For example: In the repo `gitPractice`, we have a directory named **logs** and a file named **logs** in directory `temp`. Before adding **logs** in `.gitignore` file, `git status` shows both file and directory but as soon as `logs` is entered in `.gitignore` file, both directory and file are untracked by git



```

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice (master)
$ ls -R
.:
logs/ temp/

./logs:
file1

./temp:
logs

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice (master)
$ git status -u
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    logs/file1
    temp/logs

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

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice (master)
$ echo "logs" > .gitignore

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice (master)
$ git status -u
On branch master

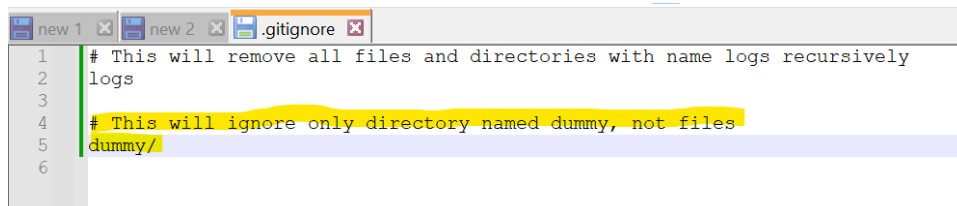
No commits yet

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

```

3. Appending a slash **<name>/** will ignore directories only, not files.

For ex: In repo gitPractice, we have one file name **dummy** and one directory as **temp/dummy** which contains file project.log. When **dummy/** is added to .gitignore file, the subdirectory dummy is ignored by the git, but not the file.



```

1 # This will remove all files and directories with name logs recursively
2 logs
3
4 # This will ignore only directory named dummy, not files
5 dummy/
6

```

```

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice (master)
$ ls -R
.:
dummy logs/ temp/

./logs:
file1

./temp:
dummy/ local/ logs

./temp/dummy:
project.log

./temp/local:
logs/

./temp/local/logs:
file1

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice (master)
$ git status -u
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore
        dummy
        temp/dummy/project.log

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

akshay.katiha@GEMGN-210247 MINGW64 ~/Desktop/gitPractice (master)
$ git status -u
On branch master

No commits yet

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

```

4. A leading ****** followed by slash means match in all directories. ****/name** will match any file or directory

Committing your changes

Once your staging area is setup, you can commit your changes. Any file that you have created or modified after you run `git add` won't go into this commit. Git considers each commit as a save point

To commit your changes, run `$ git commit -m "<commit message>"`

```

ak.katiha@GSG1PM-GI0210:~/Desktop/gitInternal$ git commit
Aborting commit due to empty commit message.
ak.katiha@GSG1PM-GI0210:~/Desktop/gitInternal$ git commit -m "add file"
[main acab5e3] add file
 2 files changed, 3 insertions(+), 1 deletion(-)
 create mode 100644 secondFile
ak.katiha@GSG1PM-GI0210:~/Desktop/gitInternal$ git status
On branch main
Your branch is ahead of 'origin/main' by 2 commits.
  (use "git push" to publish your local commits)

```

Ques: Can we commit changes directly without moving a file to staged area?

Ans: Yes, we can commit changes directly by skipping the staging area by using command

```
$ git commit -a -m "<commit-message>"
```

Although skipping staging environment is possible, but is not generally recommended

Commit History

We can see the history of commits that we have done on a repo by using the command

```
$ git log
```

```
ak.katiha@GSG1PM-GI0210:~/Desktop/gitInternal$ git log
commit acab5e3ef885ad936c0fda0dc0214cf2ae036cca (HEAD -> main)
Author: gem-akatiha <akshay.katiha@geminisolutions.com>
Date: Mon Aug 29 00:25:51 2022 +0530

    add file

commit 0abed3877fb907d49e1636338ce321d49f82af72
Author: gem-akatiha <akshay.katiha@geminisolutions.com>
Date: Mon Aug 29 00:16:42 2022 +0530

    first commit

commit c2350bc4b373062a119ae1923cc969ff143ce8ac (origin/main, origin/HEAD)
Author: gem-akatiha <104412645@gem-akatiha@users.noreply.github.com>
Date: Thu Aug 25 00:45:25 2022 +0530

    Initial commit
```

Branching in Nutshell

(More about git branching can be found [here](#))

Branching in simple terms is that you can diverge from the main line and continue your work, without messing that main line and when that work is done, simply merge that branch

Creating branch

We can create a new branch by using the command `$ git branch <branch-name>`

To see the available branches, simply run command `$ git branch`

```
ak.katiha@GSG1PM-GI0210:~/Desktop/gitInternal$ git branch beta
ak.katiha@GSG1PM-GI0210:~/Desktop/gitInternal$ git branch
  beta
* main
```

* main specifies that we are currently using main branch.

Moving in between branches

In order to switch between branches, using command `$ git checkout <branchname>`

```
ak.katiha@GSG1PM-GI0210:~/Desktop/gitInternal$ git checkout beta
Switched to branch 'beta'
ak.katiha@GSG1PM-GI0210:~/Desktop/gitInternal$ git branch
* beta
main
```

How branches separate the changes

Let's see through an example, what happens when you make changes in one branch

On beta branch, let's do some modifications as given below

```
ak.katiha@GSG1PM-GI0210:~/Desktop/gitInternal$ vi newFile
ak.katiha@GSG1PM-GI0210:~/Desktop/gitInternal$ ls > betaFile
ak.katiha@GSG1PM-GI0210:~/Desktop/gitInternal$ git status
On branch beta
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   newFile

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

I have changed the content of newFile and added another file named betaFile. Committing the new changes in beta branch

```
ak.katiha@GSG1PM-GI0210:~/Desktop/gitInternal$ git add -A
ak.katiha@GSG1PM-GI0210:~/Desktop/gitInternal$ git status
On branch beta
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   betaFile
        modified:   newFile

ak.katiha@GSG1PM-GI0210:~/Desktop/gitInternal$ git commit -m "beta changes"
[beta 885ec21] beta changes
2 files changed, 5 insertions(+), 1 deletion(-)
create mode 100644 betaFile
```

Let's list all the content that we have after commit in beta branch

```
ak.katiha@GSG1PM-GI0210:~/Desktop/gitInternal$ git status
On branch beta
nothing to commit, working tree clean
ak.katiha@GSG1PM-GI0210:~/Desktop/gitInternal$ ls
betaFile  newFile  README.md
```

So, we have betaFile available in repo that we just created. Let's change to main branch and see what would have happened there using command `git checkout main`

```
ak.katiha@GSG1PM-GI0210:~/Desktop/gitInternal$ git checkout main
Switched to branch 'main'
Your branch is ahead of 'origin/main' by 3 commits.
  (use "git push" to publish your local commits)
ak.katiha@GSG1PM-GI0210:~/Desktop/gitInternal$ ls
newFile  README.md
```

On changing back to main branch and listing the content, we see that betaFile is not available in repo. Also, the changes made to newFile were reverted. This is how working on different branches keeps the changes separate

Note: If you do not have a branch, you can simply create and checkout to that branch using command `$ git checkout -b <branchName>`

In short, `git checkout -b <branchName>` is combination of `git branch <branchName> && git checkout <branchName>`

Merging branches

We have done all the new changes in betaFile and now want to add them in the main branch. To do this, let's first move to main branch and run command

```
$ git merge <name-of-branch-having-updated-code>
```

```
ak.katiha@GSG1PM-GI0210:~/Desktop/gitInternal$ git merge beta
Updating b11583b..d8cc1c3
Fast-forward
 betafile | 3 +++
 1 file changed, 3 insertions(+)
 create mode 100644 betafile
```

Since beta branch was created directly from main branch content and no other changes were made to main branch, git see this as a continuation of master. That's why, you are seeing the **Fast-forward** in the output

Once changes are merged and if we do not want to use branch further, we can delete the branch using the command

```
$ git branch -d <branchName>
```

Note: We can also rename the branch by taking checkout of that branch and running command `$ git branch -m <newName>`

```

ak.katiha@GSG1PM-GI0210:~/Desktop/gitInternal$ git checkout beta
Switched to branch 'beta'
ak.katiha@GSG1PM-GI0210:~/Desktop/gitInternal$ git branch -m beta2.0
ak.katiha@GSG1PM-GI0210:~/Desktop/gitInternal$ git branch
* beta2.0
  main

```

This is the basic info that you would need before start using git. Other docs that may help you to understand git are:

1. [Git FAQ](#)
2. [Git Cheat sheet](#)
3. [Git tutorial for beginners](#)

Questions

1. I am getting warning: in the working copy of 'a', LF will be replaced by CRLF the next time Git touches it. Why
2. Suppose I have a repo which contains a .git folder. I copied that .git folder and pasted it to another directory that is not under VCS. What will happen? Also, what will happen if I swap two directories .git folders
3. Can git and github credentials be different?

Ans: Yes, they can be different

4. What is the difference between command git add . and git add - - all
5. .gitignore file should be tracked?
6. Can we rename .gitignore file
7. What is the use of .git/info/exclude file? How it is different from .gitignore file
8. What is .gitignore_global file? When should one use it?
9. Ques: What is .gitignore_global file?
- 10.

Useful Linux commands

1. How to make nested folders

use command **mkdir -p dir1/dir2** to make nested directories

2. Recursively list all the files available in each directory and subdirectory

use command **ls -R**. Note: The command is case sensitive