
GIT
Lesson 02 - Introduction to GIT
Capgemini

Lesson Objectives

To understand …

- Git Basics
- Getting a Git Repository
- Recording Changes to the Repository
- Viewing the Commit History
- Undoing Things

GIT Basics : What is GIT

- Git is a distributed revision control and source code management (SCM) system with an emphasis on speed, data integrity and support for distributed, non-linear workflows.
- As with most other distributed revision control systems, and unlike most client–server systems, every Git working directory is a full-fledged repository with complete history and full version-tracking capabilities, independent of network access or a central server.
- Like the Linux kernel, Git is free software distributed under the terms of the GNU General Public License version 2.

GIT Basics : How does GIT handle data

- Git deals with data like a series of snapshots of a miniature file system.
- With Git, every time you commit, or save the state of your project, Git takes a picture of the files at that moment and stores a reference to that snapshot. T
- To be efficient, if files have not changed, Git doesn't store the file again, just a link to the previous identical file it has already stored.
- Git deals with data like a stream of snapshots.
- Everything in Git is check-summed before it is stored and is then referred to by that checksum. So it's impossible to change the contents of any file or directory without Git knowing about it, this functionality is built into Git at the lowest levels and you can't lose information in transit or get file corruption without Git being able to detect it.

GIT Basics : How does GIT handle data

- Git uses SHA-1 hash mechanism for check-summ, it 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

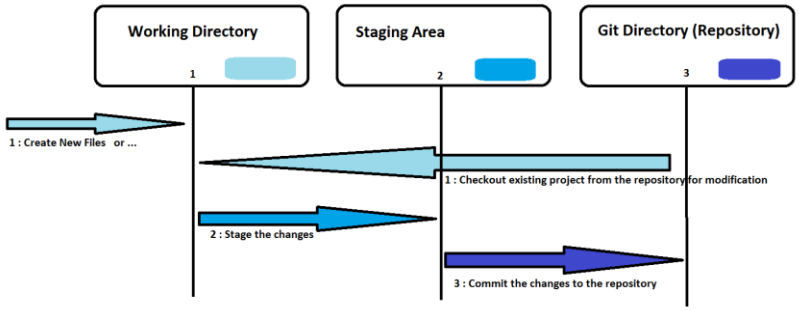
- Git stores everything in its database not by file name but by the hash value of its contents
- All actions in Git, only *add* data to the Git database.
- More in-depth look at how Git stores its data and how you can recover data that seems lost will b

GIT Basics : GIT Operations

- Most of the Git Operations are local in nature, since no information is needed from another computer on ther network.
- The entire history of Git project is on local disk, so most local operations seem almost instantaneous.
- The **Git directory**, the directory identified by you as a repository to store your data files, is where Git stores the metadata and object database for your project.
- This is the most important part of Git, and it copied when you *clone* a repository from another computer to yours.
- 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.

GIT Basics : GIT Operations

• The **staging area** is where cache where files to be committed, have their information stored about what will go into itsr next commit. Its technical name in Git parlance is the "index", but the phrase "staging area" works just as well.



GIT Basics : GIT Operations

• In Git, any file can be in once to the three states : *committed*, *modified*, and *staged.*

1. Modified means that you have changed the file but have not committed it to your database yet. Or this can be new files created in the working tree/direcory.
2. Staged means that you have marked a modified file in its current version to go into your next commit snapshot.
3. Committed means that the data is safely stored in your local database

• So, The basic Git workflow goes something like this:

- You modify files in your working tree.
- You selectively stage just those changes you want to be part of your next commit, which **adds *only* those changes to the 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.

GIT Basics : GIT Operations

- 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**.
- If it was changed since it was checked out but has not been staged, it is **modified**.
- More in-depth look at these states and how you can either take advantage of them or skip the staged part entirely, is out of scope of this course, however the references are shared for further learning.

Getting a GIT repository: Installing GIT

- Install Git either as a package or via another installer, or download the source code and compile it yourself. Download and install as per your OS. Reference links are provided.
- Git has a tool called git config that lets you get and set configuration variables that control all aspects of how Git looks and operates. These variables can be stored in three different places:
- /etc/gitconfig file: Contains values applied to 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. (Because this is a system configuration file, you would need administrative or superuser privilege to make changes to it.)
- ~/.gitconfig or ~/.config/git/config file: Values specific personally to you, the user. You can make Git read and write to this file specifically by passing the --global option, and this affects all of the repositories you work with on your system.
- config file in the Git directory (that is, .git/config) of whatever repository you're currently using: Specific to that single repository. You can force Git to read from and write to this file with the --local option, but that is in fact the default. (Unsurprisingly, you need to be located somewhere in a Git repository for this option to work properly.)

Getting a GIT repository: Installing & Configuring GIT

- Each level overrides values in the previous level, so values in .git/config trump those in /etc/gitconfig.
- On Windows systems, Git looks for the .gitconfig file in the $HOME directory (C:\Users\$USER for most people). It also still looks for /etc/gitconfig, it's relative to the Sys root, that is wherever you decide to install Git on your Windows system when you run the installer.
- After installing Git, set your user name and email address, since every Git commit uses this information. If you want to override this with a different name or email address for specific projects, you can run the command without the --global option when you're in

```
$ git config --global user.name "<you name>"
$ git config --global user.email <your email id>
```

Getting a GIT repository: Configuring GIT

- To check configuration settings, use git config --list command to list all the settings Git can find at that point
- Git reads the same key from different files (/etc/gitconfig and ~/.gitconfig, for example). and, Git uses the last value for each unique key it sees.
- Check what Git uses as a specific key's value is, by typing git config <key>
- $ git config user.name
- Query Git as to the origin for values it displays, and it will show configuration file had the final say in setting the values, use git config --show-origin rerere.autoUpdate command for this.
- To get help while using Git, use git help <verb> for example : git help config

Getting a GIT repository: Get a repository

- Repository is a collection of refs together with an object database containing all objects which are reachable from the refs, possibly accompanied by meta data from one or more porcelains. A repository can share an object database with other repositories via alternates mechanism.
- Get a Git repository in one of two ways, and end up with a Git repository on your local machine.

1. A local directory (either already existing or new to be crested) that is currently not under version control, is turned into a Git repository or

2. *Clone* an existing Git repository from elsewhere

Getting a GIT repository: Get a repository

- Local directory turned into a Git repository
  - at default location : ***git init***
  - at particular location : ***git init /testGIT/gitrepo1***
  - bare repository at default location : ***git init –bare***
    - –More on 'bare repositories' later …

- Clone an existing Git repository from elsewhere
  - Following command That creates a directory named libgit2, initializes a .git directory inside it, pulls down all the data for that repository, and checks out a working copy of the latest version. : ***git clone https://github.com/libgit2/libgit2***
  - Git has a number of different transfer protocols you can use apart from https://protocol, like git:// or user@server:path/to/repo.git, which uses the SSH transfer protocol. More in-depth look at transfer protocols, is out of scope of this course, however the references are shared for further learning.

Recording changes to GIT repository

• After setting up a Git repository typically, we start making changes and committing snapshots of those changes into the repository each time the project reaches a state we want to record.

• Files in working directory can be in one of two states: ***tracked*** or ***untracked***.

• ***Tracked*** files are files that were in the last snapshot; they can be unmodified, modified, or staged. In short, tracked files are files that Git knows about i.e files in the staging area.

• ***Untracked*** files are everything else — any files in working directory that are not in last snapshot and are not in your staging area.

• When we first clone a repository, all of your files will be tracked and unmodified because Git just checked them out and haven't been edited.

• After editing the files, Git sees them as modified, because we've changed them since last commit. As we work, we selectively stage these modified files and then commit all those staged changes, and the cycle repeats.

Recording changes to GIT repository

• After setting up a Git repository typically, we start making changes and committing snapshots of those changes into the repository each time the project reaches a state we want to record.

• Files in working directory can be in one of two states: ***tracked*** or ***untracked***.

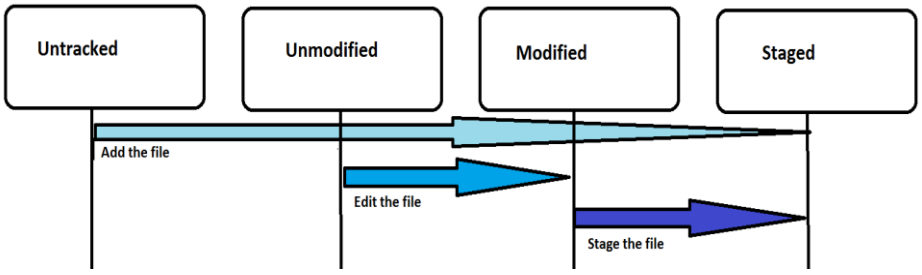
• ***Tracked*** files are files that were in the last snapshot; they can be unmodified, modified, or staged. In short, tracked files are files that Git knows about i.e files in the staging area.

• ***Untracked*** files are everything else — any files in working directory that are not in last snapshot and are not in your staging area.

• When we first clone a repository, all of your files will be tracked and unmodified because Git just checked them out and haven't been edited.

• After editing the files, Git sees them as modified, because we've changed them since last commit. As we work, we selectively stage these modified files and then commit all those staged changes, and the cycle repeats.

Recording changes to GIT repository



- Determine which files are in which state using the ***git status*** command.
- Using the command ***git status -s*** or ***git status –short*** get simplified status of files
- To ignore files that you don't want Git to automatically like log files or system files create a file listing patterns to match them named .gitignore. Here is an example .gitignore file:

```
$ cat .gitignore
*.[oa]
*~
```

Viewing commit history

• Most basic and powerful tool to view commit history is the ***git log*** command.

git log -p -2

• Using options is -p or --patch, which shows the difference (the *patch* output) introduced in each commit. You can also limit the number of log entries displayed, such as using -2 to show only the last two entries.

git log –stat

▪ To see abbreviated stats for each commit, we can use the --stat option

git log --pretty=oneline

▪ The option ***--pretty*** changes the log output to formats other than the default.
The ***oneline*** option prints each commit on a single line. The ***short, full, and fulleroptions*** show the output in roughly the same format but with less or more information, respectively

Undoing Things

• This is one of the few areas in Git where you may lose some work if you do it wrong.

• One of the common undo takes place when commit is done too early and possibly we forget to add some files, or we mess up our commit message.

• If you want to redo that commit, make the additional changes you forgot, stage them, and commit again using the ***--amend*** option: ***$ git commit –amend***

• The above command takes your staging area and uses it for the commit. If you've made no changes since the 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.

• The same commit-message editor fires up, but it already contains the message of your previous commit. You can edit the message the same as always, but it overwrites your previous commit.

Undoing Things

• As an example, if you commit and then realize you forgot to stage the changes in a file you wanted to add to this commit, you can do something like this:

$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit --amend

• Here you end up with a single commit, the second commit replaces the results of the first.

• It's important to understand that when you're amending your last commit, you're not so much fixing it as *replacing it entirely with a new, improved commit that* pushes the old commit out of the way and puts the new commit in its place. Effectively, it's as if the previous commit never happened, and it won't show up in your repository history. The obvious value to amending commits is to make minor improvements to your last commit, without cluttering your repository history with commit messages of the form, "Oops, forgot to add a file" or "Darn, fixing a typo in last commit".

Unstaging a Staged File

- Here we determine the state of staging area and how to undo changes to staging area.
- Assuming we want to commit two changed files as two separate changes, but we accidentally did it as ***git add ** and staged the files. How can we unstage one of the two? The ***git status*** command reminds you of the status.

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

    renamed:    README.md -> README
    modified:   CONTRIBUTING.md
```

Unstaging a Staged File

- Now the CONTRIBUTING.md file is modified but once again unstaged. Use it with causion.

```
$ git reset HEAD CONTRIBUTING.md
Unstaged changes after reset:
M   CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README

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:   CONTRIBUTING.md
```

Lesson1 demo1 : Working with GIT

- # switch to home cd ~/ # create a directory and switch into it
- mkdir ~/repo01
- cd repo01 # create a new directory mkdir datafiles
- # Initialize the Git repository# for the current directory
- git init
- # switch to your new repository
- cd ~/repo01 # create another directory # and create a few files
- mkdir datafiles
- touch test01
- touch test02
- touch test03
- touch datafiles/data.txt
- # Put a little text into the first file
- ls >test01
- # add all files to the index of the # Git repository
- git add .

Lesson1 demo1 : Working with GIT continued

- # switch to home cd ~/ # create a directory and switch into it
- # commit your file to the local repository
- git commit -m "Initial commit"
- # show the Git log for the change
- git log
- # Create a file and commit it
- touch nonsense2.txt git add . && git commit -m "more nonsense"
- # remove the file via
- Git git rm nonsense2.txt
- # commit the removal
- git commit -m "Removes nonsense2.txt file"
- # Create a file and put it under version control
- touch nonsense.txt git add . && git commit -m "a new file has been created"
- # Remove the file
- rm nonsense.txt
- # Try standard way of committing -> will NOT work
- git add . && git commit -m "a new file has been created"

Lesson1 demo1 : Working with GIT continued

- # switch to home cd ~/ # create a directory and switch into it
- # commit the remove with the -a flag
- git commit -a -m "File nonsense.txt is now removed"
- # alternatively you could add deleted files to the staging index via
- # git add -A .
- # git commit -m "File nonsense.txt is now removed"
- # create a file and add to index
- touch unwantedstaged.txt
- git add unwantedstaged.txt
- # remove it from the index
- git reset unwantedstaged.txt
- # to cleanup, delete it
- rm unwantedstaged.txt
- # assume you have something to commit
- git commit -m "message with a tpyo here"
- git commit --amend -m "More changes - now correct"

Summary

In this lesson, you have learnt

- Git Basics
- Getting a Git Repository
- Recording Changes to the Repository
- Viewing the Commit History
- Undoing Things

Add the notes here.