Version Control - GIT

Avinash Patel

Agenda

1. Introduction to Version Control
2. Introduction to Git
3. Git – Basic Commands
4. Branching & Merging

Table of Content

- Distinguish Centralized Version control and distributed version control system
- Describe Working directory, staging, local and remote repository
- Explain below listed commands
  - status and log
  - push, pull and clone commands
  - checkout
- Perform Branching & Merging and manage conflicts while merging
- Perform rewrite, Rollback & gitignore
- Exercises
  - Create local and Remote Repository
  - Create and track source file(s) Locally - Access versions by ref/tag
  - Push local repository to remote repository
  - Clone source copy from remote repository
  - Demonstrate versions from local / remote repository
  - Access the content by history / branch / using tags

Introduction

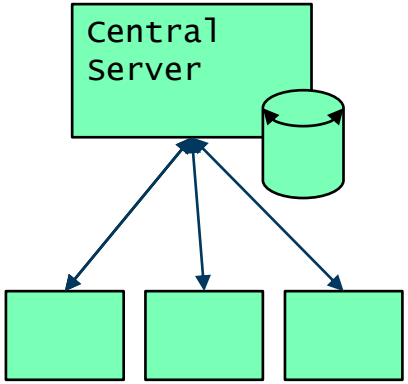
- In SDLC process, non-linear workflows and distributed framework are imminent.
- It is common that a piece of code is being accessed and possibly edited by a geographically dispersed team.
- Maintaining Data Integrity is very crucial when many team members (Developer/Tester) work on same files.
- Revision control is an efficient way to address the problem of sharing files.
  - It is also called as Version Control and Source Control.
  - Each of these revisions is typically identified by Time Stamps

What is version control ?

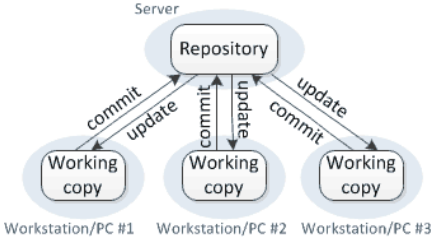
- Version control is a system that records changes to a file or set of files over time. It helps to recall or **recover specific version** as and when required
  - i.e. a system which allows the management of a code base

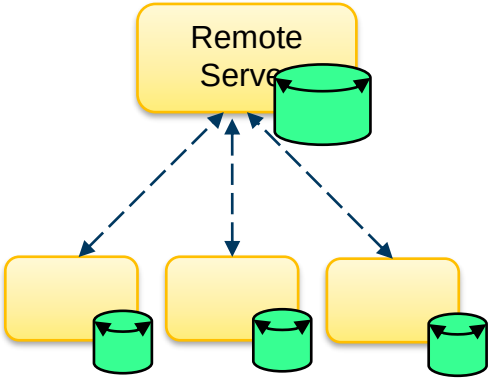
- Version control enables to roll back to the previous state of a file or files or a entire project.
  - It allows multiple versions to exist simultaneously
  - It compare changes over time
  - It checks the last modification
  - It easily tracks and recover with very little overhead

Types of Version Control (VC)

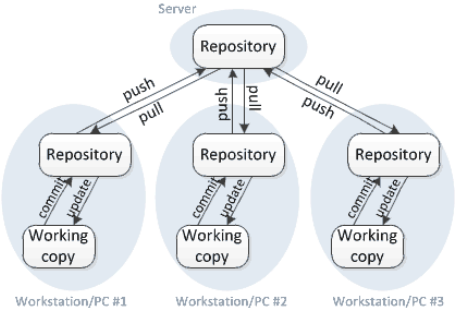


Centralized version control





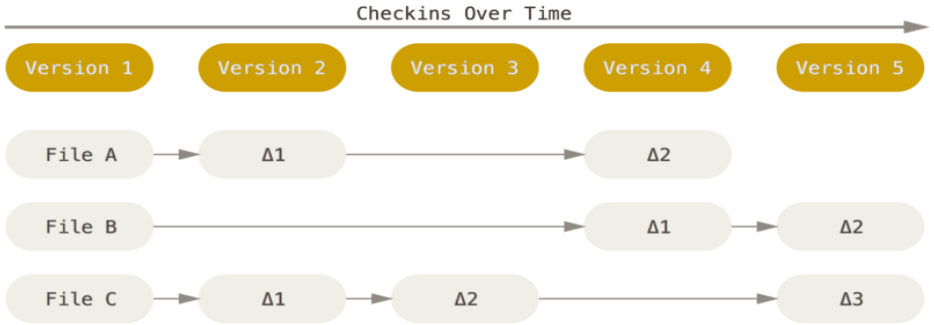
Distributed version control



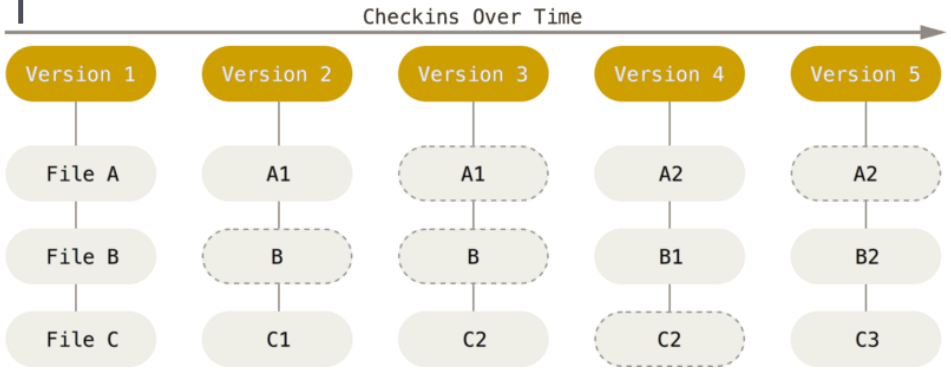
Introduction to GIT

Introduction

- Most Version Control systems



- GIT



Features of GIT

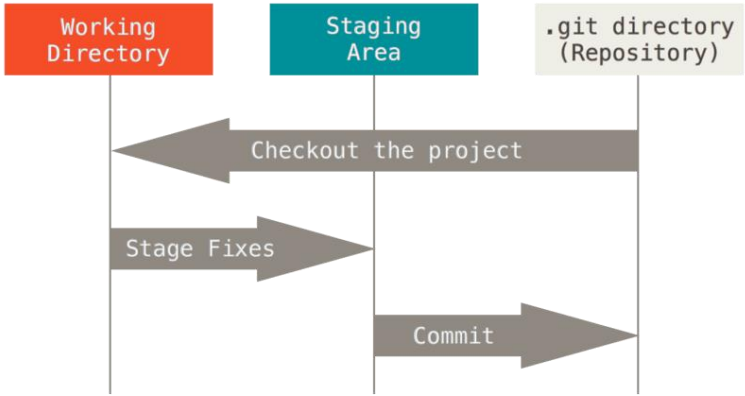
- Almost Every Operation Is Local:
  - Most operations in Git need only local files and resources to operate. No other information is needed from another computer on a network

- Git Has Integrity
  - Everything in Git is check-summed before it is stored and is referred to, by that checksum.
  - It’s impossible to change the contents of any file or directory without Git knowing about it

- Git Generally Only Adds Data
  - When actions are done in Git in the form of commands, nearly all of them only add data to a Git repository

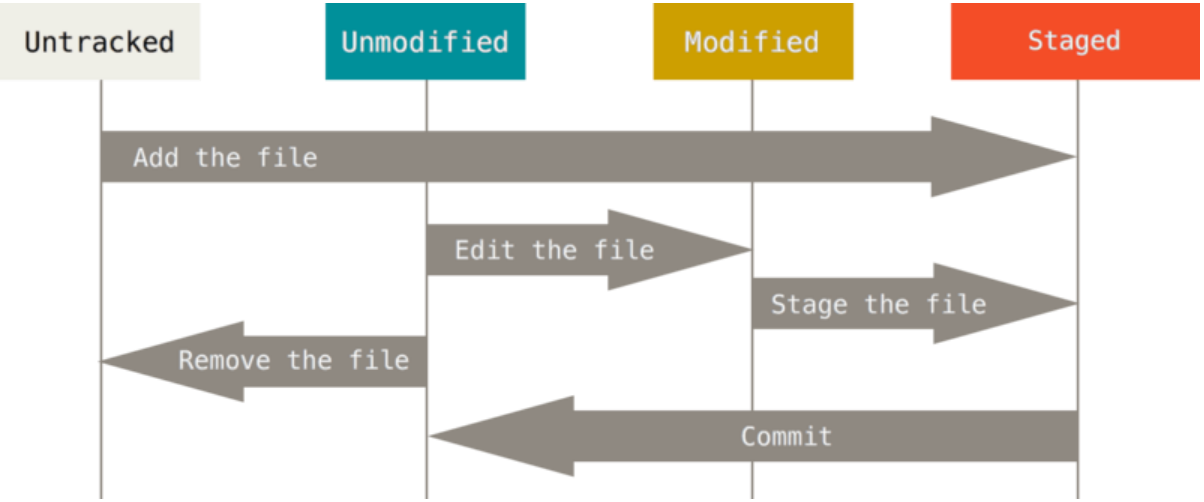
GIT Basics- Three states

- Three stages are
  - **Committed**: The data is safely stored in local database.
  - **Modified**: It implies that the file is changed and yet to be committed into database.
  - **Staged**: It means that modified file is marked in its current version to go into next commit snapshot.



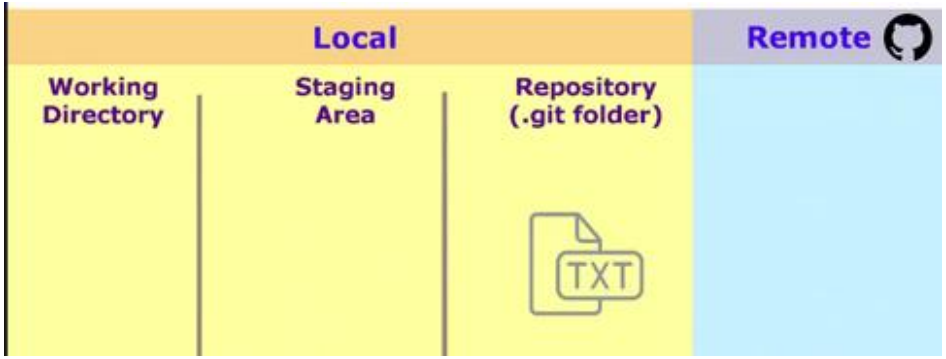
- Three main sections of a GIT project:
  - Working directory
  - Staging area
  - GIT directory

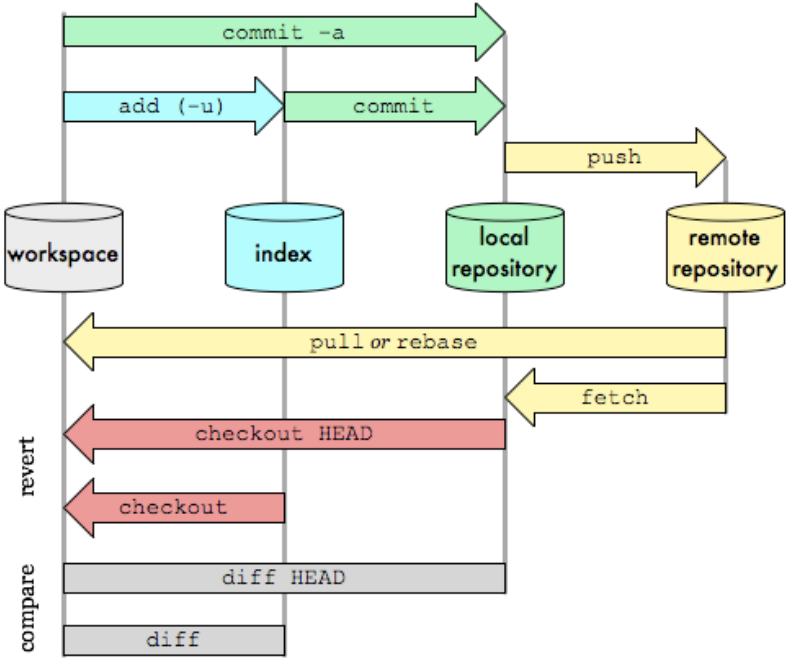
The lifecycle of the status of files



Git workflow

- Data can be placed in central repository (remote) from local git repository





Getting Started

- To download git for windows

  https://git-scm.com/download/win

- Signup and/or Create your own remote (public/private) repository
  http://topgear-training-gitlab.wipro.com

Basic Set UP / Getting Started with GIT

- Check GIT version
  - git --version

    git version 2.11.0.windows.1
- Create a distributed public repository at http://wosggitlab.wipro.com
  - Note: For example create “MyRepo.git” as a public repository
- Define user account's default identity
  - git config --global user.name “Any Valid User Name"
  - git config --global user.email your_email@whatever.com
  - git config --global core.editor “’C:/Program Files(x86)/Notepad++/notepad++.exe’ -multiInst -nosession"

Checking the Settings

- For checking individual values of the keys
  - git config user.name
- To view all settings use the below command
  - git config –list

Command Prompt

```
C:\Users\avitepa>git --version
git version 2.11.0.windows.1

C:\Users\avitepa>git config user.name "avinashpatelin"

C:\Users\avitepa>git config user.email "avinashpatelin@gmail.com"

C:\Users\avitepa>git config user.name
avinashpatelin

C:\Users\avitepa>git config user.email
avinashpatelin@gmail.com
```

Checking the Settings contd..

```
C:\Users\avitepa>git config --list
core.symlinks=false
core.autocrlf=true
core.fscache=true
color.diff=auto
color.status=auto
color.branch=auto
color.interactive=true
help.format=html
http.sslcainfo=C:/Program Files/Git/mingw64/ssl/certs/ca-bundle.crt
diff.astextplain.textconv=astextplain
rebase.autosquash=true
credential.helper=manager
difftool.usebuiltin=true
gui.recentrepo=E:/TestGitClone
gui.recentrepo=F:/myGIT
gui.recentrepo=F:/GITdemoFolder
gui.recentrepo=D:/testGit
user.email=avinashpatelin@gmail.com
user.name=avinashpatelin
core.editor='C:/Program Files/Notepad++/notepad++.exe'-multiInst -nosession
core.repositoryformatversion=0
core.filemode=false
core.bare=false
core.logallrefupdates=true
core.symlinks=false
core.ignorecase=true
gui.wmstate=zoomed
gui.geometry=893x435+130+130 370 341
```

Basic Commands

Working with Git Repository

- Option 1: Place an existing project or directory into Git (Remote repository)

- Option 2: Have a cloned copy of an existing Git (Remote repository)

Option1: Create local project

- Create a folder ‘localRepo’ and initialize as local repository
  - **F:\AvinashGIT\localRepo>git init**

```
Initialized empty Gitrepository in F:/AvinashGIT/localRepo/.git/
```

Note: It creates a new hidden subfolder named .git. This folder is used to contain all of your necessary repository files. At this point, nothing in that project is tracked.

- Create a file “Welcome.java” within ‘localRepo’ folder

```
F:\AvinashGIT\localRepo>dir
 Volume in drive F is New Volume
 Volume Serial Number is 98C3-7CFF

 Directory of F:\AvinashGIT\localRepo

02/19/2017  11:03 PM    <DIR>          .
02/19/2017  11:03 PM    <DIR>          ..
02/19/2017  11:03 PM               115 Welcome.java
               1 File(s)            115 bytes
               2 Dir(s)  90,844,487,680 bytes free

F:\AvinashGIT\localRepo>
```

Option1: Steps to Commit to Local Repo.

```
F:\AvinashGIT\localRepo>git status -s
?? Welcome.java

F:\AvinashGIT\localRepo>git add.
git: 'add.' is not a git command. See 'git --help'.

Did you mean this?
        add

F:\AvinashGIT\localRepo>git add
Nothing specified, nothing added.
Maybe you wanted to say 'git add .'?

F:\AvinashGIT\localRepo>git add .

F:\AvinashGIT\localRepo>git status -s
A  Welcome.java

F:\AvinashGIT\localRepo>git commit -m "first commit"
[master (root-commit) 85cbafd] first commit
 1 file changed, 6 insertions(+)
 create mode 100644 Welcome.java

F:\AvinashGIT\localRepo>git log
commit 85cbafd232468ce786e04787b74e87cb5cfb08a
Author: avinashpatelin <avinashpatelin@gmail.com>
Date:   Sun Feb 19 23:25:24 2017 +0530

    first commit
```

1.Status after init

2. add a specific file or everything

3. Correct One:
add .

4. Status after add

5. Commit the file

6. Check the log

Option1: push local copy to remote repository (gitlab)

- Step1: git config remote.origin.url <url>

  Example: git config remote.origin.url http://topgear-training-gitlab.wipro.com /AVITEPA/demo.git

  Note: This project should have been created in gitlab already

- Step2: Push files from local to remote repository
  - **F:\AvinashGIT\localRepo>git push -u origin master**

Option1: Verify Central repository

- @ http://topgear-training-gitlab.wipro.com/AVITEPA/demo

Option 2

- Get a copy of an existing Git repository from remote repository
  - **F:\git clone** http://topgear-training-gitlab.wipro.com **/AVITEPA/demo.git**
- Use `git log` and observe the previous version track details

Command Prompt

```
E:\test\demo>git log
commit d0866ce5163444382f328ef7f70f5b53ba817763
Author: Avinash Patel <avinash.patel@wipro.com>
Date:   Fri Jan 12 12:06:54 2018 +0530

    c3

commit 70773702104cdca7f104f61f3d62bdee21d31715
Author: Avinash Patel <avinash.patel@wipro.com>
Date:   Fri Jan 12 12:04:38 2018 +0530

    c2

commit 653228ca3540bb4c37b72efccd65ecd86c8303ff
Author: Avinash Patel <avinash.patel@wipro.com>
Date:   Fri Jan 12 12:04:07 2018 +0530

    c1

commit d4ae01fce40f515f290a5baf425f3b0321788c5a
Author: Avinash Patel <avinash.patel@wipro.com>
Date:   Fri Jan 12 12:02:58 2018 +0530

    initial

E:\test\demo>
```

Points to ponder

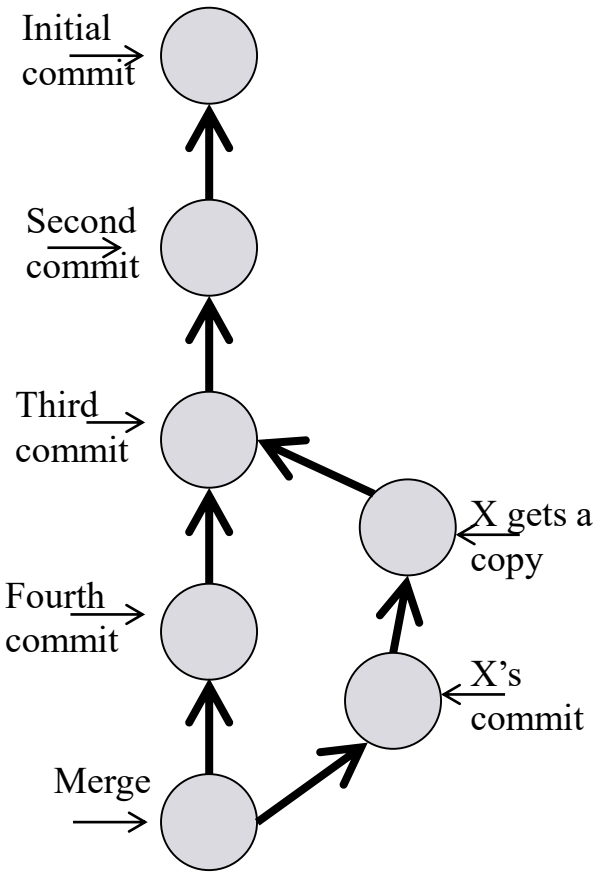
- *config* (*global & list*)
  - user.name <optional>
  - user.email <optional>
  - core.editor <optional>
  - remote.origin.url <optional>
- *init*
- add (. Or <filename>
- status (-s)
- *log*
- *commit* (-m <name>)
- Local repository to a remote repository - config to push
- *push* - Actual push
- *clone <url>*
- -- version

Branching & Merging

Branch

- A branch in Git is simply a movable pointer to one of the commits.

- In Git, the default branch name is ‘master’.
  - The master branch will point to the latest commit, as the commits are made
  - “master” branch is not a special branch When ‘git init’ command gets executed, It gets created by default.



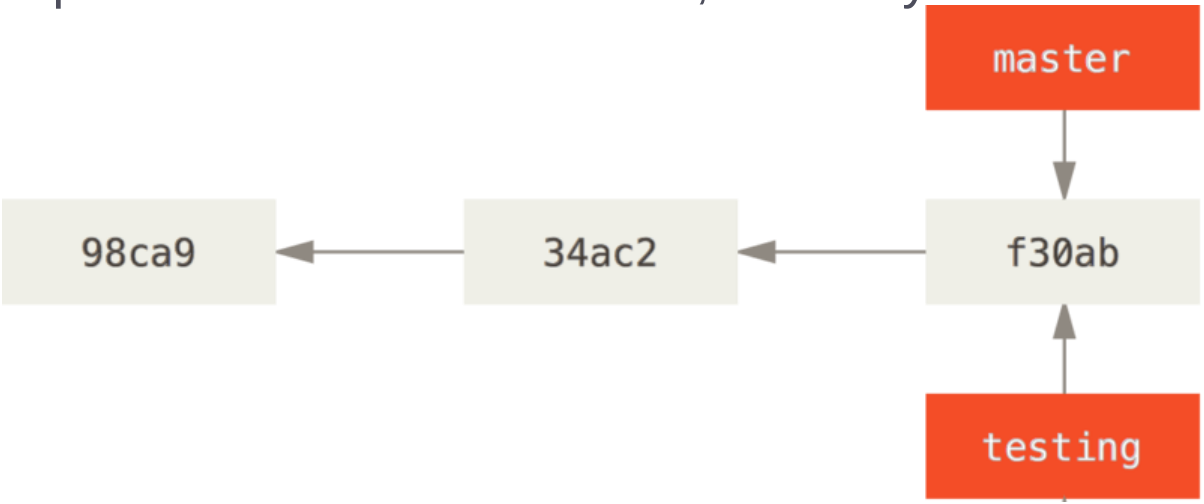
Branches

- Git doesn’t store data as a series of change sets or difference, but instead as a series of snapshots.
- When a commit is made, Git stores a commit object that contains a pointer to the snapshot of the content being staged.
- This object also contains the author’s name and email, the message that are typed, and pointers to the commit or commits that directly came before this commit:
- zero parents for the initial commit, one parent for a normal commit, and multiple parents for a commit that results from a merge of two or more branches.

Create a New Branch

- When a new branch is created, a pointer also gets created to move around.
- Use the command ‘git branch <branch name>’ to create a new branch

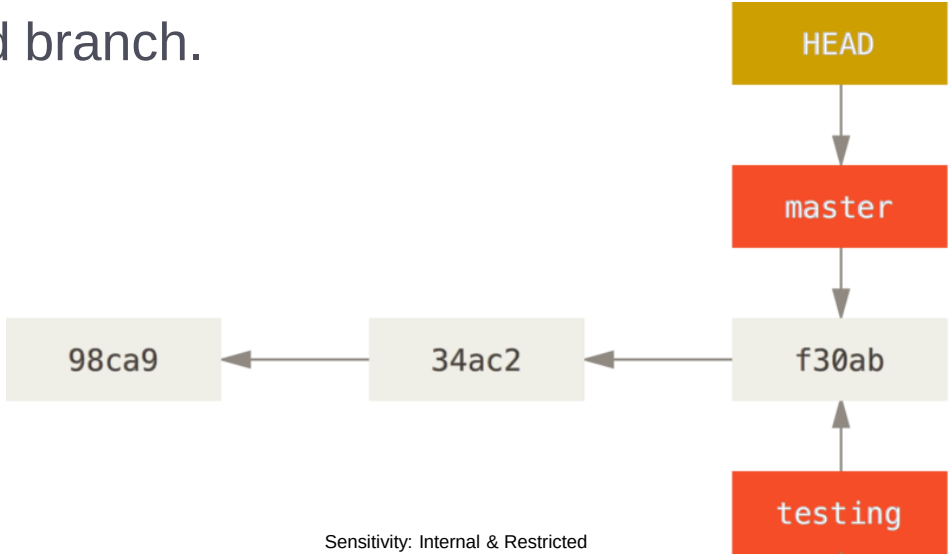
For example: ‘git branch testing’ creates a new branch called ‘testing’. It also creates a new pointer to the same commit, currently on.



HEAD pointing to a branch

- Git uses a special pointer called HEAD to know the current branch.
  - This pointer pointing to the local branch. In the diagram given below, it points to master

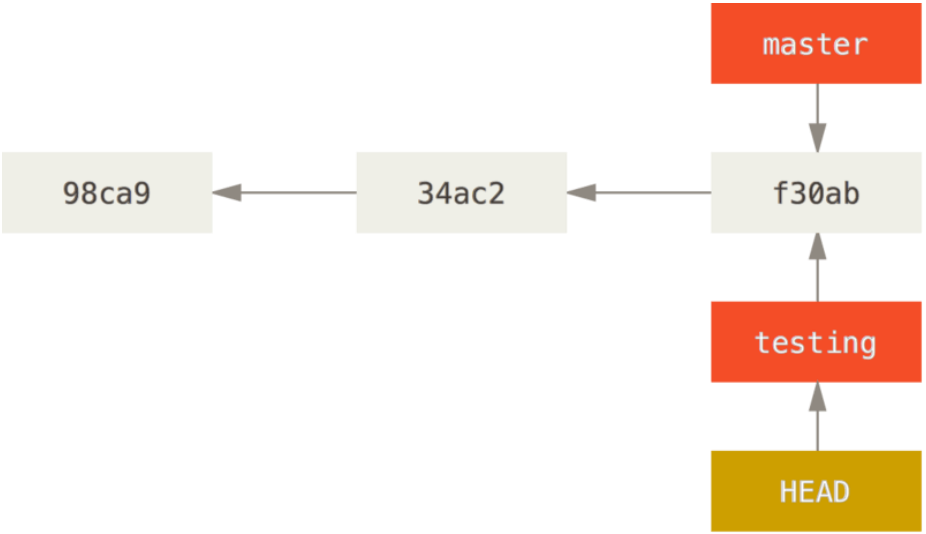
- ‘git branch’ command creates a new branch. However it doesn’t switch to the newly created branch.



Switch to a Branch

- To switch to an existing branch, execute the command:
  - git checkout <name of the branch>

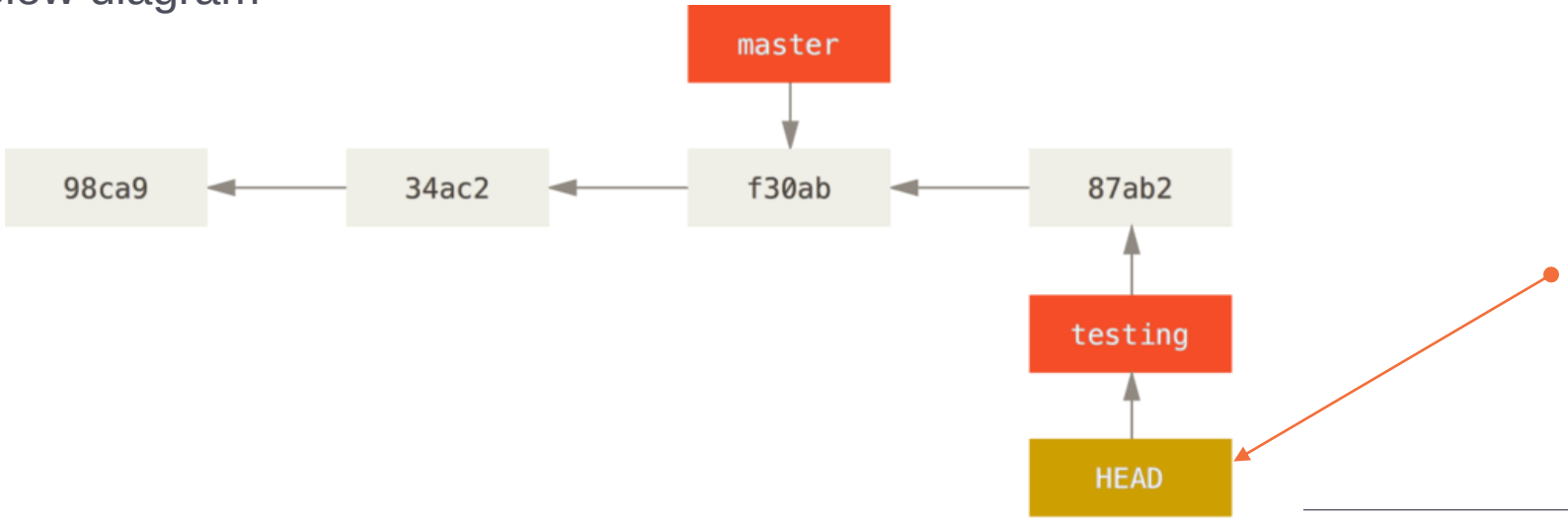
    For example : git checkout **testing**
  - This command makes the HEAD, to point to the **testing** branch.



Commit at the branch level

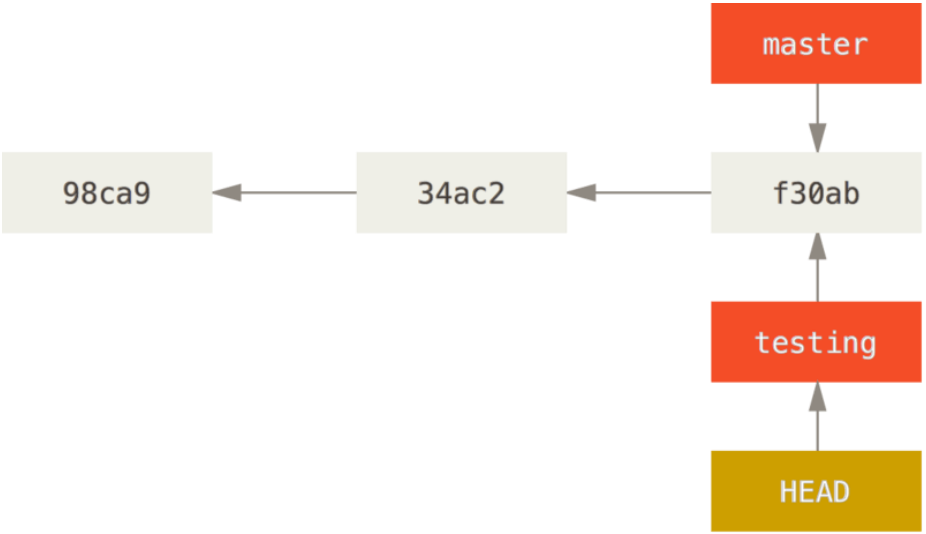
- Make few changes and commit at the branch level
  - git commit -a -m ‘few changes have been made‘

- As a result, ‘Testing’ branch pointer has moved forward along with HEAD. However master branch pointer still points to earlier commit as shown in the below diagram

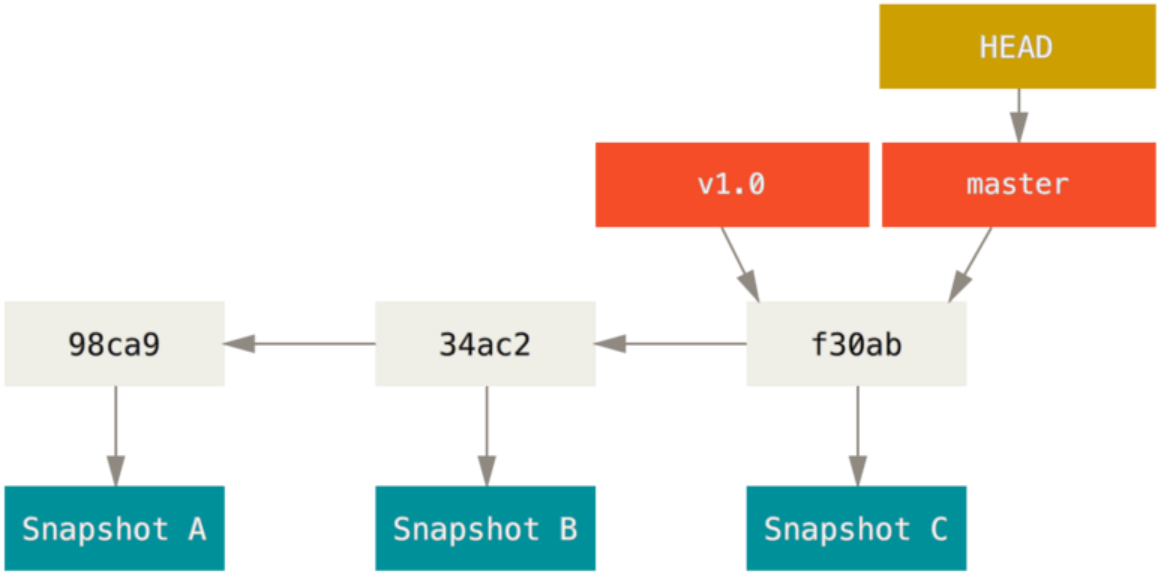


Switching Branches

- To switch to an existing branch, run the git checkout command.
- $ git checkout testing
- This moves HEAD to point to the testing branch.

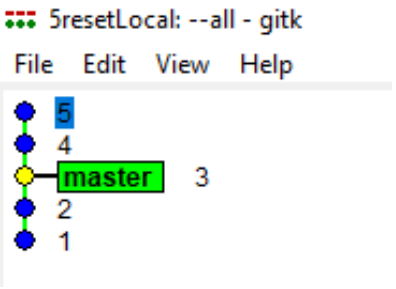


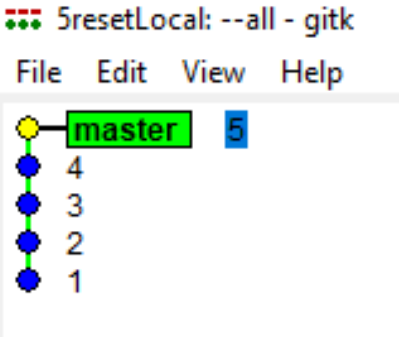
A branch and its commit history

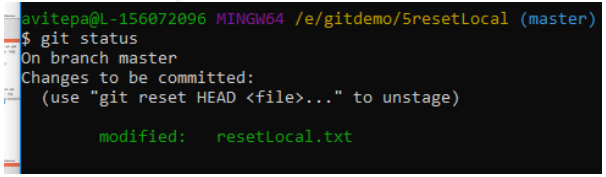


reset

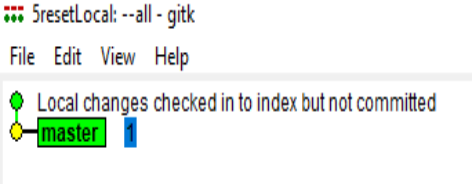
- git reset --hard <b374cb1>



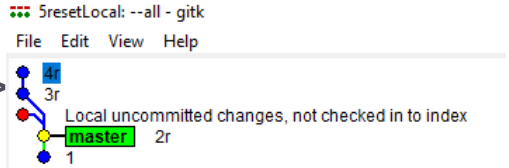


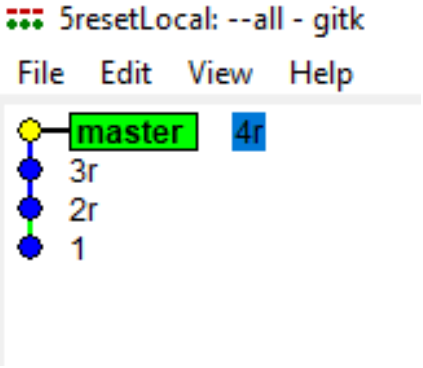


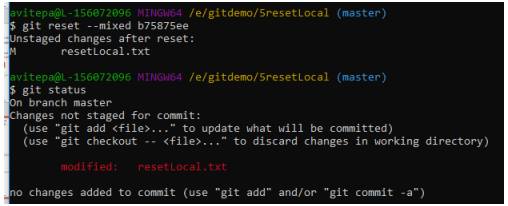
- git reset --soft <3480cbe>



- git reset --mixed <b75875ee>







Reset from remote repo

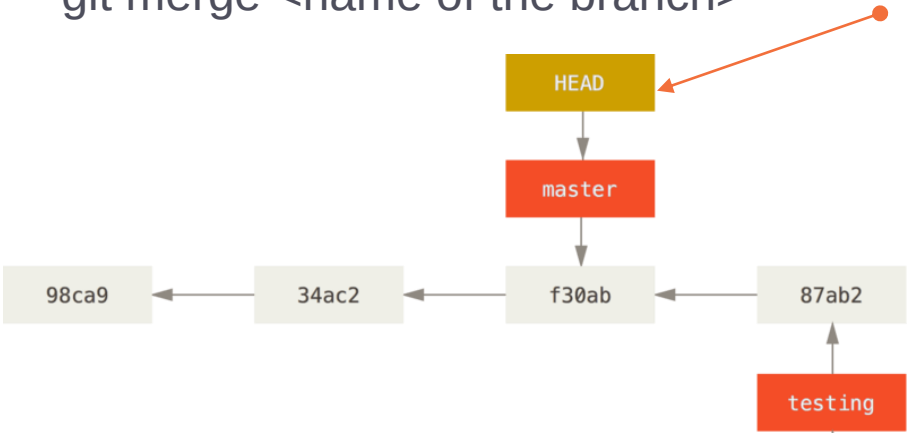
- git reset --hard <8021aebe>
- git push -f origin master

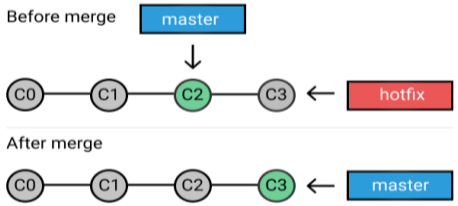
- git push -f origin <last_known_good_commit>:<branch_name>

- git revert <oldest_commit_hash>..<latest_commit_hash>

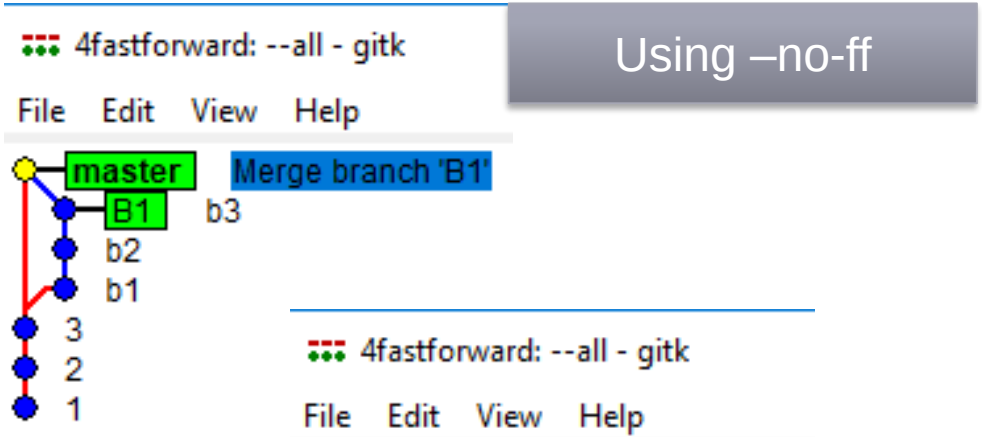
Merge

- Switch back to master branch by using the command:
  - git checkout master
  - This command makes the HEAD pointer to point to master branch again. Thereafter, any change made will reflect on master only
- Use the below command to merge branch (testing) with master
  - git merge <name of the branch>

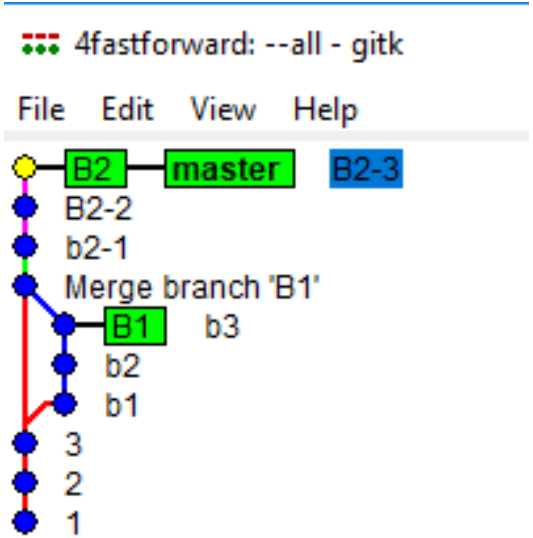


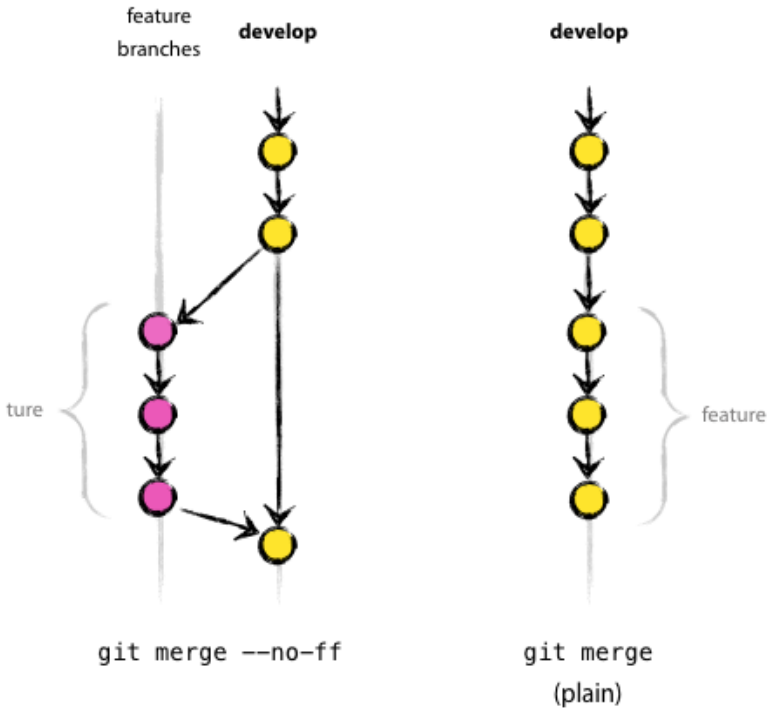


Merge

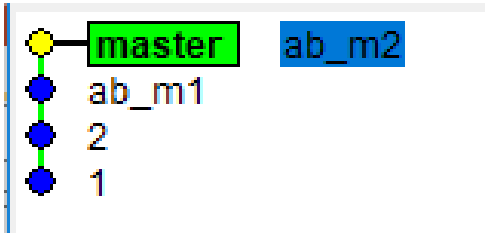


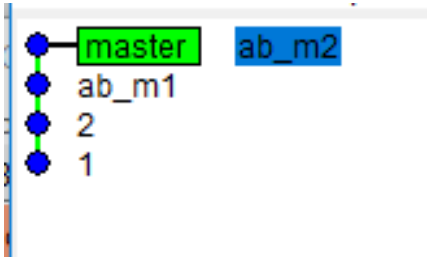
Using –no-ff

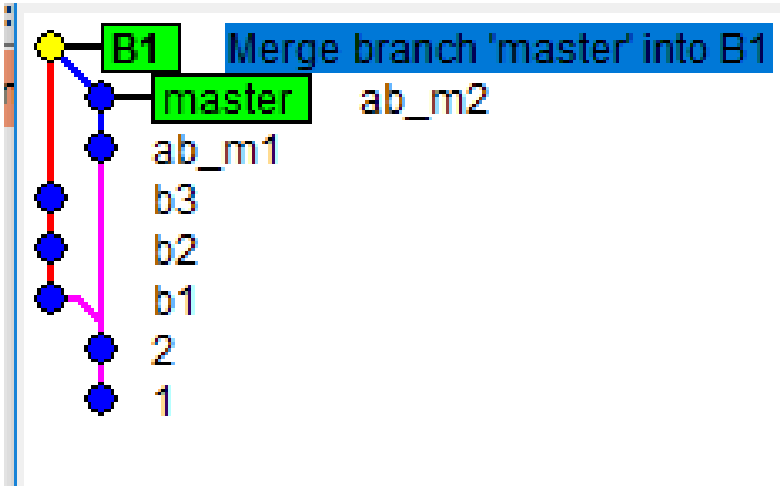


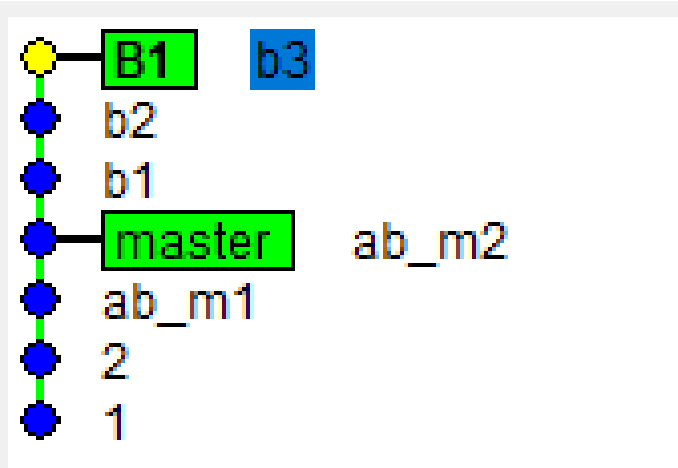


Merge | rebase







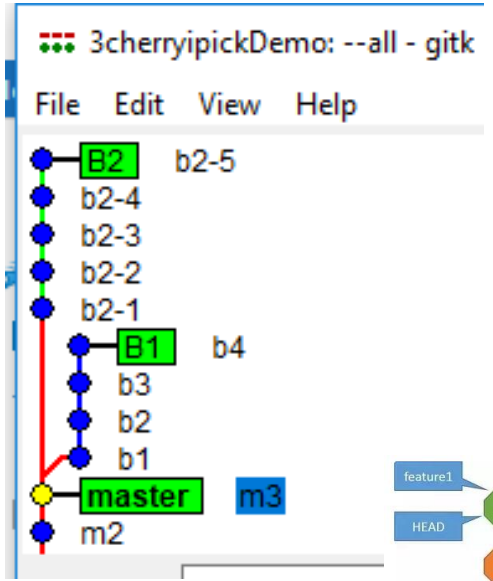


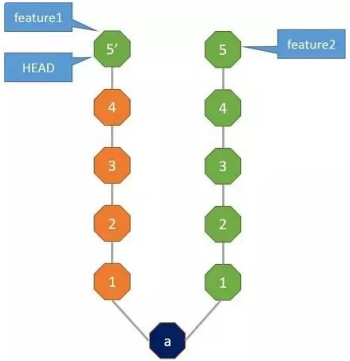
Comparison

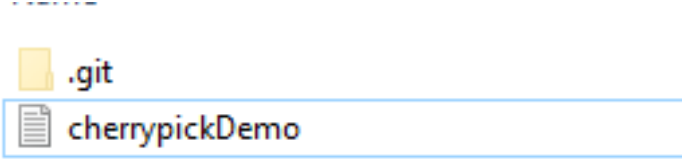
- git merge
  - Applies all unique commits from branch X into branch Y
  - Doesn’t rewrite commit history, just adds one new commit
  - When **“git pull”** performed, merge Is a default behavior
  - Use merge in cases where you want a set of commits to be clearly grouped together in history

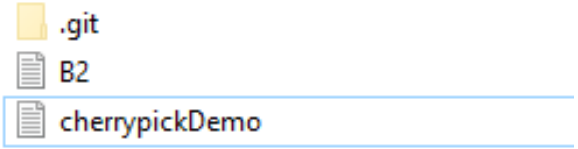
- git rebase
  - Gets all unique commits from both branches and applies them one by one
  - rewrites commit history but doesn’t create extra commit for merging
  - Used to make your commit history more clear and consistent (use it only before pushing to remote servers to keep your name and karma clean)
  - temporary local branches
  - use **git rebase -i** *(interactive)* for rewriting your local commit history into pretty one before pushing it on the remote server
  - Use rebase when you want to keep a linear commit history
  - DON’T use rebase on a public/shared branch
  - **rebase** is changing what is happening at the **bottom of the history**, thus rewriting it

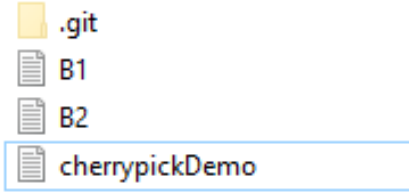
Cherry-pick











- Using cherry-pick <id>
  - How to get back file from reverted commit done on a master branch to a different branch (e.g. feature branch)

The Protocols

Git can use four major protocols to transfer data:

- Local
- HTTP
- Secure Shell (SSH)
- Git

Local Protocol

- The most basic is the Local protocol, in which the remote repository is in another directory on disk

- This is often used if everyone on team has access to a shared filesystem such as an NFS mount
- If everyone logs in to the same computer, it wouldn’t be ideal, because all code repository instances would reside on the same computer, making a catastrophic loss much more likely.

Local Protocol

- If there is a shared mounted filesystem, then clone, push to, and pull from a local file-based repository can be done
- To clone a repository like this or to add one as a remote to an existing project, use the path to the repository as the URL.
- Eg., to clone a local repository, run like this:

- git clone /opt/git/project.git

  Or

- git clone file:///opt/git/project.git

Local Protocol

- Git operates slightly differently if file:// at the beginning of the URL is explicitly specified.
- If just path is specified, Git tries to use hard links or directly copy the files it needs.
- If file://, is specified, it fires up the processes that it normally uses to transfer data over a network which is generally a lot less efficient method of transferring the data.

- To add a local repository to an existing Git project, run
- git remote add local_proj /opt/git/project.git

The HTTP Protocols

- Git can communicate over HTTP in two different modes. Prior to Git 1.6.6 there was only one way it could do this which was very simple and generally read-only.
- In version 1.6.6 a new, smarter protocol was introduced that involved Git being able to intelligently negotiate data transfer in a manner similar to how it does over SSH.

- Smart HTTP
- Dumb HTTP

Smart HTTP

- The “smart” HTTP protocol operates very similarly to the SSH or Git protocols but runs over standard HTTP/S ports and can use various HTTP authentication mechanisms
- it’s easier on the user since username/password basic authentication is used rather than having to set up SSH keys.

Dumb HTTP

- If the server does not respond with a Git HTTP smart service, the Git client will try to fall back to the simpler “dumb” HTTP protocol.
- The Dumb protocol expects the bare Git repository to be served like normal files from the web server.
- Feature of Dumb HTTP protocol is the simplicity of setting it up.
- It requires putting bare Git repository under HTTP document root and set up a specific post-update hook
- To allow read access to the repository over HTTP
- git clone --bare /path/to/git_project gitproject.git
- cd gitproject.git

Dumb HTTP

- The post-update hook that comes with Git by default runs the appropriate command (git update-server-info) to make HTTP fetching and cloning work properly.
- This command is run when something pushed to this repository; then, clone can be done:

- git clone https://example.com/gitproject.git

The SSH Protocol

- A common transport protocol for Git when self-hosting is over SSH. This is because SSH access to servers is already set up in most places. SSH is also an authenticated network protocol; and because it’s ubiquitous, it’s generally easy to set up and use.

- To clone a Git repository over SSH, specify ssh:// URL:

- git clone ssh://user@server/project.git

  Or use the shorter scp-like syntax for the SSH protocol:

- git clone user@server:project.git

The Git Protocol

- This is a special daemon that comes packaged with Git;
- It listens on a dedicated port (9418) that provides a service similar to the SSH protocol, but with absolutely no authentication.
- In order for a repository to be served over the Git protocol, create the git-daemon-export-ok file – the daemon won't serve a repository without that file in it – but other than that there is no security.
- Either the Git repository is available for everyone to clone or it isn't. This means that there is generally no pushing over this protocol.

Some tips

- Passing credentials with push command
  - git push https://username:password@myrepository.com/repo.git --all

Thank You