Week -1 , Day -3

1. Why Do We Need a Version Control System?

Version control systems **allow multiple developers, designers, and team members to work together on the same project**.

It helps them work smarter and faster! A version control system is critical to ensure everyone has access to the latest code and modifications are tracked.

Version control **allows developers to store the history of changes and who made them, enabling them to revert or look back to previous versions**

Version control software is **used to track revisions, solve integration conflicts in code, and manage different artifacts involved in software projects**.

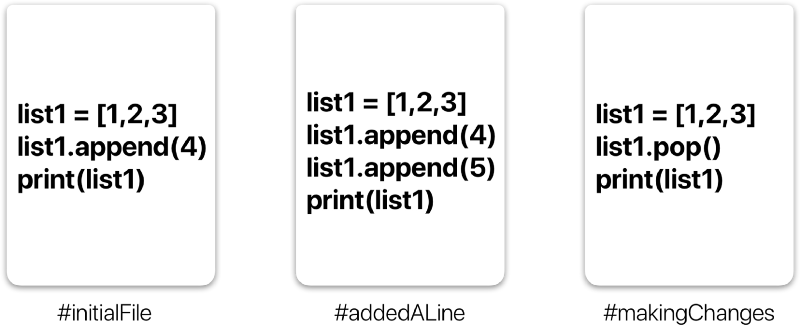
Here are a few of the most popular types of VCS:

* [Helix Core](https://www.perforce.com/products/helix-core) (Perforce)
* [Git](https://www.perforce.com/resources/vcs/git-best-practices)
* [SVN](https://www.perforce.com/blog/vcs/what-svn)
* [ClearCase](https://www.perforce.com/blog/vcs/what-is-clearcase-basics)
* [Mercurial](https://www.perforce.com/blog/vcs/git-vs-mercurial-how-are-they-different)
* [TFS](https://www.perforce.com/blog/vcs/what-team-foundation-server)

1. **What is Git?**

Git is a version-control system for tracking changes in computer files and coordinating work on those files among multiple people. Git is a ***Distributed Version Control System***. So Git does not necessarily rely on a central server to store all the versions of a project’s files. Instead, every user “clones” a copy of a repository (a collection of files) and has the ***full*** history of the project on their own hard drive. This clone has *all* of the metadata of the original while the original itself is stored on a self-hosted server or a third party hosting service like GitHub.

Git helps you ***keep track of the changes*** you make to your code. It is basically the history tab for your code editor(With no incognito mode ?). If at any point while coding you hit a fatal error and don’t know what’s causing it you can always revert back to the stable state. So it is very helpful for debugging. Or you can simply see what changes you made to your code over time.

A simple example of version history of a file.

In the example above, all three cards represent different versions of the same file. We can select which version of the file we want to use at any point of time. So I can jump to and fro to any version of the file in the git time continuum.

Git also helps you ***synchronise code*** between multiple people. So imagine you and your friend are collaborating on a project. You both are working on the same project files. Now Git takes those changes you and your friend made independently and merges them to a single “**Master**” repository. So by using Git you can ensure you both are working on the most recent version of the repository. So you don’t have to worry about mailing your files to each other and working with a ridiculous number of copies of the original file. And collaborating long distance becomes as easy as HTML ?.

**Git Workflow:**

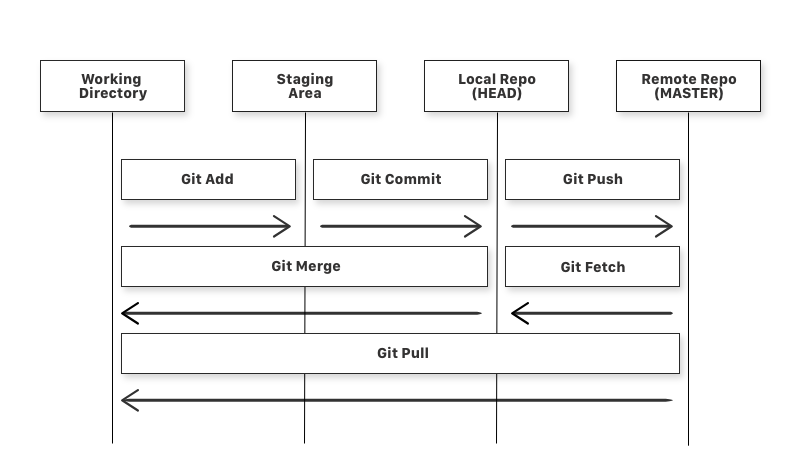
Before we start working with Git commands, it is necessary that you understand what it represents.

**What is a Repository ?**

A **repository** a.k.a. **repo** is nothing but a collection of source code.

**There are four fundamental elements in the Git Workflow.**

**Working Directory**, **Staging Area**, **Local Repository** and **Remote Repository**.

Diagram of a simple Git Workflow

**If you consider a file in your Working Directory, it can be in three possible states.**

1. **It can be staged.** Which means the files with the updated changes are marked to be committed to the local repository but not yet committed.
2. **It can be modified**. Which means the files with the updated changes are not yet stored in the local repository.
3. **It can be committed**. Which means that the changes you made to your file are safely stored in the local repository.

* git add is a command used to add a file that is in the working directory to the staging area.
* git commit is a command used to add all files that are staged to the local repository.
* git push is a command used to add all committed files in the local repository to the remote repository. So in the remote repository, all files and changes will be visible to anyone with access to the remote repository.
* git fetch is a command used to get files from the remote repository to the local repository but not into the working directory.
* git merge is a command used to get the files from the local repository into the working directory.
* git pull is command used to get files from the remote repository directly into the working directory. It is equivalent to a git fetch and a git merge .

LIFE CYCLE OF GIT

General workflow is as follows −

* You clone the Git repository as a working copy.
* You modify the working copy by adding/editing files.
* If necessary, you also update the working copy by taking other developer's changes.
* You review the changes before commit.
* You commit changes. If everything is fine, then you push the changes to the repository.
* After committing, if you realize something is wrong, then you correct the last commit and push the changes to the repository.

Shown below is the pictorial representation of the work-flow.



1. basic local Git operations

▪ creating a repository,

## **Create a Bare Repository**

Let us initialize a new repository by using **init** command followed by **--bare** option. It initializes the repository without a working directory. By convention, the bare repository must be named as **.git**.

[gituser@CentOS ~]$ pwd

/home/gituser

[gituser@CentOS ~]$ mkdir project.git

[gituser@CentOS ~]$ cd project.git/

[gituser@CentOS project.git]$ ls

▪ cloning a repository,

he Clone operation creates an instance of the remote repository.

Jerry creates a new directory in his home directory and performs the clone operation.

[jerry@CentOS ~]$ mkdir jerry\_repo

[jerry@CentOS ~]$ cd jerry\_repo/

[jerry@CentOS jerry\_repo]$ git clone gituser@git.server.com:project.git

The above command will produce the following result.

Initialized empty Git repository in /home/jerry/jerry\_repo/project/.git/

remote: Counting objects: 3, done.

Receiving objects: 100% (3/3), 241 bytes, done.

remote: Total 3 (delta 0), reused 0 (delta 0)

Jerry changes the directory to new local repository and lists its directory contents.

[jerry@CentOS jerry\_repo]$ cd project/

[jerry@CentOS jerry\_repo]$ ls

README

▪ making and recording changes

Jerry clones the repository and decides to implement basic string operations. So he creates string.c file.

He compiled and tested his code and everything is working fine. Now, he can safely add these changes to the repository.

Git add operation adds file to the staging area.

[jerry@CentOS project]$ git status -s

?? string

?? string.c

[jerry@CentOS project]$ git add string.c

▪ staging and committing changes,

To commit the changes, he used the git commit command followed by –m option.

[jerry@CentOS project]$ git commit -m 'Implemented my\_strlen function'

After commit to view log details, he runs the git log command. It will display the information of all the commits with their commit ID, commit author, commit date and **SHA-1** hash of commit.

[jerry@CentOS project]$ git log

▪ viewing the history of all the changes

▪ undoing changes