# version control

What is “version control”, and why should you care? Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later.

**Types of Version Control**

1. **LVCSs (Local Version Control Systems)**

**Tools**: RCS

**Drawbacks**: Developers can not collaborate with on other systems. single point of failure

1. **CVCSs (Centralized Version Control Systems)**

**Tools**: CVS,Subversion, and Perforce

**Advantages**: Everyone knows to a certain degree what everyone else on the project is doing.

Administrators have fine-grained control over who can do what.

it’s far easier to administer a CVCS than it is to deal with local databases on every client.

**Drawbacks**: Single point of failure

If the hard disk the central database is on becomes corrupted, and proper backups haven’t been kept, you lose absolutely everything – the entire history of the project except whatever single snapshots people happen to have on their local machines.

1. **DVCSs (Distributed Version Control Systems)**

**Tools**: Git, Mercurial, Bazaar or Darcs

**Advantages**: Clients don’t just check out the latest snapshot of the files: they fully mirror the repository. Thus if any server dies, and these systems were collaborating via it, any of the client repositories can be copied back up to the server to restore it. Every clone is really a full backup of all the data.

**Drawbacks**:

# GIT

**Configuring Git**

System Level Configuration

$ git config –system

Stored in /etc/gitconfig or C:\ProgramData\Git\config

User level Configuration

$ git config --global

Stored in ~/.gitconfig or C:\Users\Pavan\.gitconfig or ~/.config/git/config file: Specific to your user

Repository-Level Configuration

$git config

Stored in .git/config in each repo

Each level overrides values in the previous level, so values in .git/config

trump those in /etc/gitconfig

**Your Identity**

**$** git config --global user.name "John Doe"

**$** git config --global user.email [johndoe@example.com](mailto:johndoe@example.com)

**Your Editor**

you can configure the default text editor that will be used when Git needs you to type in a message. If not configured, Git uses your system’s default editor.

**$** git config --global core.editor emacs

Or in windows

**$** git config --global core.editor "'C:/Program Files (x86)/Notepad++/notepad++.exe' –multiInst

**Checking Your Settings**

**$** git config –global –list

**Getting Help**

If you ever need help while using Git, there are three ways to get the manual page (manpage) help for any of the Git commands:

**$** git help <verb>

**$** git <verb> --help

**$** man git-<verb>

EX: **$** git help config

Git helpfully tries to figure out what you meant, but it still refuses to do it. If you set help.autocorrect to 1, Git will actually run this command for you:

**$** git config –global help.autocorrect 1

Check BY this command.

$ git statsu

**Colors in Git**

Git automatically colors most of its output, but there’s a master switch if you don’t like this behavior. To turn off all Git’s colored terminal output, do this:

**$** git config --global color.ui false

The default setting is auto

**Formatting and Whitespace**

Formatting and whitespace issues are some of the more frustrating and subtle problems that many developers encounter when collaborating, especially cross-platform.

If you’re on a Windows machine, set it to true – this converts LF endings into CRLF when you check out code:

**$** git config --global core.autocrlf true

If you’re on a Linux or Mac system that uses LF line endings, then you don’t want Git to automatically convert them when you check out files; however, if a file with CRLF endings accidentally gets introduced, then you may want Git to fix it. You can tell Git to convert CRLF to LF on commit but not the other way around by setting core.autocrlf to input:

**$** git config --global core.autocrlf input

If you’re a Windows programmer doing a Windows-only project, then you can turn off this functionality, recording the carriage returns in the repository by setting the config value to false:

**$** git config --global core.autocrlf false

To check all settings.

**$** git config –global –list

Or

$ cat ~/.gitconfig

Change to repo directory and check

$ cat .git/config

$ git config user.name “Pavan”

$ git config --list

To remove

$ git config --unset user.name

$ git config –list

You can change with vi edit or also by doing

$ vi .git/config

## Working locally with Git

Creating Local Repository

Adding files

Commiting changes

Viewing History

Viewing a diff

Working copy, staging, and repository

Deleting files

Cleaning the working copy

Ignore files with .gitignore

Creating Local Repository , Adding files , Commiting changes

$ mkdir pavan

$ cd pavan

$ ls –la

$ git init

$ ls –la

$ echo “Hello, Git” > READEME.TXT

$ git status

$ git add README.TXT

$ git status

$ git commit –m “Added Readme.txt”

$ git log

TO update file

$ vim README.TXT

:wq

$ git status

$ git add –u (for All update files to be staged)

$ git status

$ git commit –m “Updated README.TXT”

**Viewing history and diffs**

**$ git log**

To check change between two commits.

$ git diff dd6819..a15ec6

Or

$ git diff HEAD~1..HEAD

OR

$ git diff HEAD~1..

$ touch file1.txt file2.txt

$ git status

$ git add file1.txt

or

$ git add –A (To add all files including untracked ones)

$ git status

$ git commit –m “Added cool new feature”

$ git log

$ git diff HEAD~1

**Staging changes as multiple commits**

**$ vim file1.txt**

Add some matter and save.

$ git status

$ git add file1.txt

$ git status

$ git commit –m “Fixed but\*1234”

Deleting and renaming files

$ rm file2.txt

$ git status

$ git add –u

$ git status

$ vim file3.txt

Add data and save.

$ git status

$ git add file3.txt

$ git status

$ mv file1.txt new\_file\_name.txt

$ git status

$ git add –A

$ git status

$ git commit –m “Reorganized the feature”

$ git status

$ git log

Undoing changes to the working copy

$ vim README.TXT

Modify data and save

$ git status

TO undo the changes to previous head

$ git checkout README.TXT

$ git status

$ cat README.txt

Previous details are there.

To reset the data in working copy to HEAD.

$ rm file1.txt README.TXT

$ git status

$ git reset –hard

$ git status

Undoing and redoing changes in the repository

$ git status

$ git log

$ git reset --soft HEAD~1

$ git status

$ git log

Cleaning the working copy

$ touch temp1.txt temp2.txt

$ git status

$ git clean

$ git clean –n (shows What it would do)

$ git clean –f

$ git status

Ignoring files with .gitignore

If you have temp files and logs that don’t need to be commited to git. You can use the following commands.

$ mkdir logs

$ touch logs/log.tx

$ git status

Add .gitignore file in root of repository.

$ vim .gitignore

Add path to ignore files inthat directory.

/logs/\*.txt

Or

/logs/\*.log

or

/logs

$ git status

$ git add .gitignore

$ git commit –m “Added .git ignore”

$ git status

$ git log

## Working remotely with GIT

Cloning a remote repository

Listing remote repositories

Fetching changes from a remote

Merging changes

Pulling from a remote

Pushing changes remotely

Working with tags

Cloning a Remote Repository

$ git clone https://github.com/jquery/jquery.git

3. Basic Repository Statistics

To check line count

$ git log –online | wc –l

$ git log –online –graph

Showing different merges and