ClearVision

Presents

GIT 101

A USEFUL GUIDE TO THE BASICS

Including A GIT CHEAT SHEET

CLEARVISION-CM.COM/GIT

GIT 101

The mission

Many of the world's largest Dev teams have adopted Git and it's not hard to see why.

- It can handle small and large projects easily.
- It has a tiny footprint.
- It outclasses other version control tools.
- It's simple to learn.

Take that last bullet with a pinch of salt...

Git 101 is designed to give you the basics of Git in 15-minutes with the accompaniment of a cheat sheet.

How to create a new Git repository, make changes to it, and keep track of those changes.

GIT COMMANDS

Git commands to type in your terminal are highlighted in **bold**, e.g. `git status`.

OS COMMANDS

OS commands are also entered on the command line and are highlighted in red, e.g. `cd <directory>`.

Assumptions

- Git is installed.
- You are using the command line terminal or Git Bash for Windows as your Git client.

WHAT THIS WHITE PAPER WON'T TELL YOU

Whilst this white paper will help you get started, we recommend a *Git training course* to get to grips with all the features.

CREATING A NEW REPOSITORY

LET'S GO

CREATING A NEW REPOSITORY

When history is added to a Git repository, it's important for audit purposes that Git knows the author of the changes.

The first thing to do is to tell Git who you are. Git needs this information regularly (every time you create history in fact). By configuring these settings at the start, a lot of time is saved later on.

`git config --global user.name` "myname"

The author is configured as a username and an email address.

`git config --global user.email` "myemail"

The global argument applies "myname" to all local Git repositories you use (not by others who share the system).

In your users 'home' directory, type the following commands:

`mkdir git_repos` – **If you do not already have a location for your local Git repositories.**

`cd git_repos` – **To move to the newly created directory.**

`git innit newrepo`

This creates a directory called 'newrepo' and initialises it as a Git repository. Inside the new directory is a hidden directory named '.git'.

WARNING

Until you are comfortable with Git, *do not* modify the contents of the directory, as this is where Git keeps track of things (that's why it's hidden!). In most cases, Git will manage the contents of the directory for you.

MAKING CHANGES

Let's create some content within 'newrepo':

`cd ~/git_repos/newrepo` – **Change directory to 'newrepo' if you're not already there.**

`echo red > red.txt` – **Create a file called 'red.txt' containing the word 'red'.**

`git status`

The status command shows the status of our working directory and staging area in comparison to the contents of the repository. At this point, we can see Git is aware of a new file 'red.txt', however, it is not being tracked as of yet.

`git add red.txt`

The add command takes content from our Working Directory (in this case 'red.txt') and adds it to the Staging Area. This has the side effect of telling Git to 'track' changes to the file.

`git status`

The status has now changed and you can see that Git is tracking 'red.txt' (along with a handy hint to untrack it).

`git commit -m 'added file red.txt'`

We have 'committed' our new file from the Staging Area to our Local Repository database. The '-m' option allows us to specify a commit message to identify what it is for. If it is omitted, the default editor will appear for the message to be entered. All Git commits must include data on both the author (configured earlier) and a commit message - it cannot be blank.

`git status`

Our Working Directory is now clean. That doesn't mean it is empty, it means that our Local Repository and our Working Directory are in step.

Now, let's modify 'red'txt' and create another new file:

`echo red >> red.txt` – **Append a new line containing 'red' at the end of file 'red.txt'.**

`echo blue > blue.txt` – **Create the new file 'blue.txt' containing the text 'blue'.**

We now have two files in our Working Directory; it's time to ask Git to track the new file.

`git add blue.txt`

Git is now tracking 'blue.txt', and the changes have been added to the staging area.

`git add red.txt`

Although Git is already tracking 'red.txt', we must add our changes into the staging area to plan for the next commit before we create new history.

`git status`

Git shows two changes ready for 'commit', one new and one modified file.

`git commit -m 'Modifying red and adding blue'`

Git has now committed our changes from the Staging Area to our Local Repository.

VIEWING HISTORY

LET'S SEE

VIEWING HISTORY

`git log -5`

Git is now showing us our commit history (to the last 5 commits). We can see we have created two commit points.

Each commit has a 40-character hash which uniquely identifies who did it, when it happened, and why.

This hash is important as there are no sequential revision numbers in Git, so it provides our only unique identification method.

SHARING CODE WITH OTHERS

It's likely when you start working, that you won't create your own repository but copy a project that exists elsewhere through the command "`git clone`".
In this scenario, Git maintains a link back to the original repository named 'origin'.
As we started from scratch, we don't have this link and must create it.

`git remote add origin 'URL/to/repository'`

Define the location of 'origin' as another repository, reachable via a URL.

`git push -u origin --all`

Push the contents created to the remote repository configured as '*origin*' - note, this assumes we are using the default master branch, or another branch that exists in the remote repository.

You're good to go!

Use the accompanying cheat sheet to remind yourself of the use cases for each of these commands, and a few more you might find useful as well!

Alternatively, if you are interested in understanding this guide in greater detail, Clearvision has a range of training courses. Contact us at, sales@clearvision-cm.com.

OTHER RESOURCES

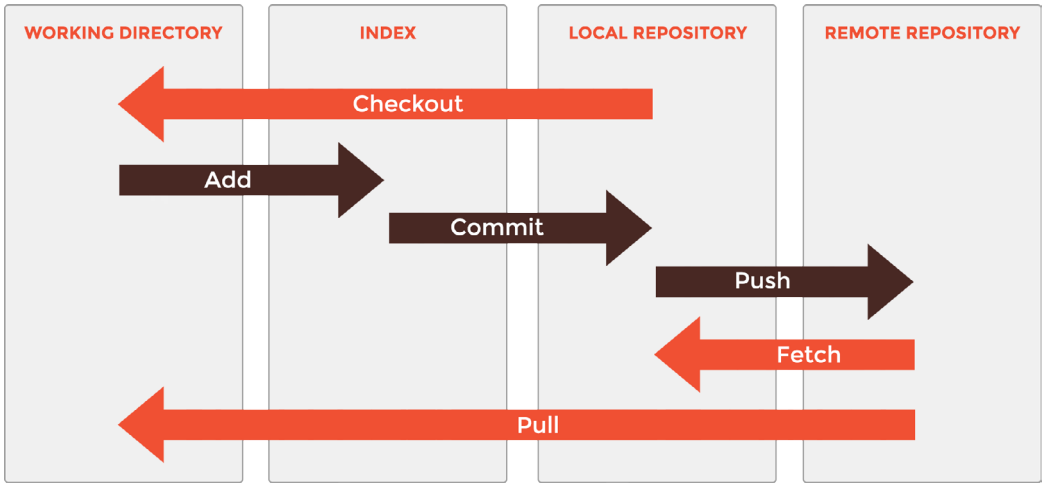
As an open-source platform, Git has almost infinite support available online. We've selected some of the best:

- ***Git training courses:*** (Basics, Advanced, and Admin) – Comprehensive courses delivered virtually or in a classroom.
- ***Git Consultants*** - Configuration and implementation support, plus, health checks and bespoke assistance.
- ***Git-SCM*** – A great resource for downloads and relevant documentation.

Plus:

git CHEAT SHEET

INFRASTRUCTURE



REFERENCES

`HEAD`
Current working position as shown by the contents of your working directory.

`master`
Default branch.

`origin`
Default upstream repository.

`origin/master`
Remote branch tracking the location of the master branch in the origin repository.

`HEAD^` or `HEAD~n`
An ancestor of the current head where ^ = 1 or n = any number above 1.

CREATE

`git innit <path>`
Turns the directory at <path> into a Git repository, even if not empty. If <path> does not exist, this command will create it.

`git clone <../existing/repo/path> <../new/repo/path>`
Copies the repository at <../existing/repo/path> to <../new/ repo/path> and records <../existing/repo/path> as the origin.

CHANGE

`git add < ile>`
Stages a change to be included in the next commit.

`git add * or git add .`
Stages all changes.

`git add -u .`
Stages all local modifications from the current directory - does not work on new (untracked) files.

`git add -u :/`
Stages all local modifications from the repository root, does not work on new (untracked) files.

`git add -A`
Stages all local modifications.

`git commit`
Converts staged changes into binary objects and puts a new commit at the top of the current branch.

`git commit -m <message>`
Converts staged changes into binary objects and puts a new commit at the top of current branch with comment <message>.

`git commit -a`
Stages all local modifications and commits (does not work on new files).

`git commit < ile(s)>`
Creates a new commit ignoring all changes other than <file(s)>.

`git rm < ile>`
Deletes <file> and stages the change.

`git mv < ile> <new ile>`
Move or rename <file> to <newfile>.

`git stash save/apply`
Save or re-apply local modifications to / from a stash.

FIXING ERRORS

`git commit --amend`
Creates a new commit in place of the current HEAD (correct the staging area first).

`git reset < ile>`
Unstage <file>.

`git reset --hard`
Resets all modifications back to the HEAD, this cannot be undone unless you Stash first.

`git revert <SHA-1>`
Revert the delta of <SHA-1> creating a new commit at the top of the current branch.

`git checkout <SHA-1> < ile>`
Recover <file> from specific commit <SHA-1>.

LOG

`git log (-n)`
Shows the history of the current branch to the (nth) ancestor.

`git log -p < ile>`
Shows the history of <file>.

`git show <SHA-1>`
Shows details of the object with the id <SHA-1>.

`git show <SHA-1>:< ile>`
Shows details of <file> referenced by commit <SHA-1>.

`git blame < ile>`
Shows details of who changed <file> and when on a line by line basis.

DIFF

`git status`
Shows any changes in the working directory and/or index.

`git diff`
Shows changes to unstaged files.

`git diff --cached`
Shows changes to staged files.

`git diff <SHA-1>`
Shows changes since <SHA-1>, this can also be HEAD or <branch>.

`git diff <SHA-1><SHA-1>`
Compare two commits.

BRANCHING & MERGING

`git checkout <SHA-1>`
Switch to the <SHA-1> or replace with a <branch>.

`git merge <branch>`
Merge <branch> into current branch.

`git branch <branch>`
Create branch named <branch> based on the HEAD.

`git checkout -b <branch>`
Create branch <branch> based on <SHA-1> and switch to it.

`git branch -d`
Delete branch <branch>.

UPDATES

`git fetch <remote>`
Recover remote changes to remote branches <remote>.

`git pull <remote>`
Recover remote changes to remote branches from <remote> and merge to local versions.

`git push <remote> <branch>`
Send local changes up to remote server <remote> <branch>.

Get in touch today!

See how Clearvision's training, migration, and support services can help you!

SPEAK TO A GIT EXPERT TODAY

Clearvision