GIT

Why Version Control ??

==========

Have you ever:

- Made a change to code, realised it was a mistake and wanted to revert back?
- Lost code and didn't have a backup of that code ?
- Had to maintain multiple versions of a product ?
- Wanted to see the difference between two (or more) versions of your code ?
- Wanted to prove that a particular change in code broke application or fixed a application ?
- Wanted to review the history of some code ?
- Wanted to submit a change to someone else's code ?
- Wanted to share your code, or let other people work on your code ?
- Wanted to see how much work is being done, and where, when and by whom ?
- Wanted to experiment with a new feature without interfering with working code ?

In these cases, and no doubt others, a version control system should make your life easier.

Key Points

========

- **Backup**
- **Collaboration**
- **Storing Versions**
- **Restoring Previous Versions**
- **Understanding What Happened**

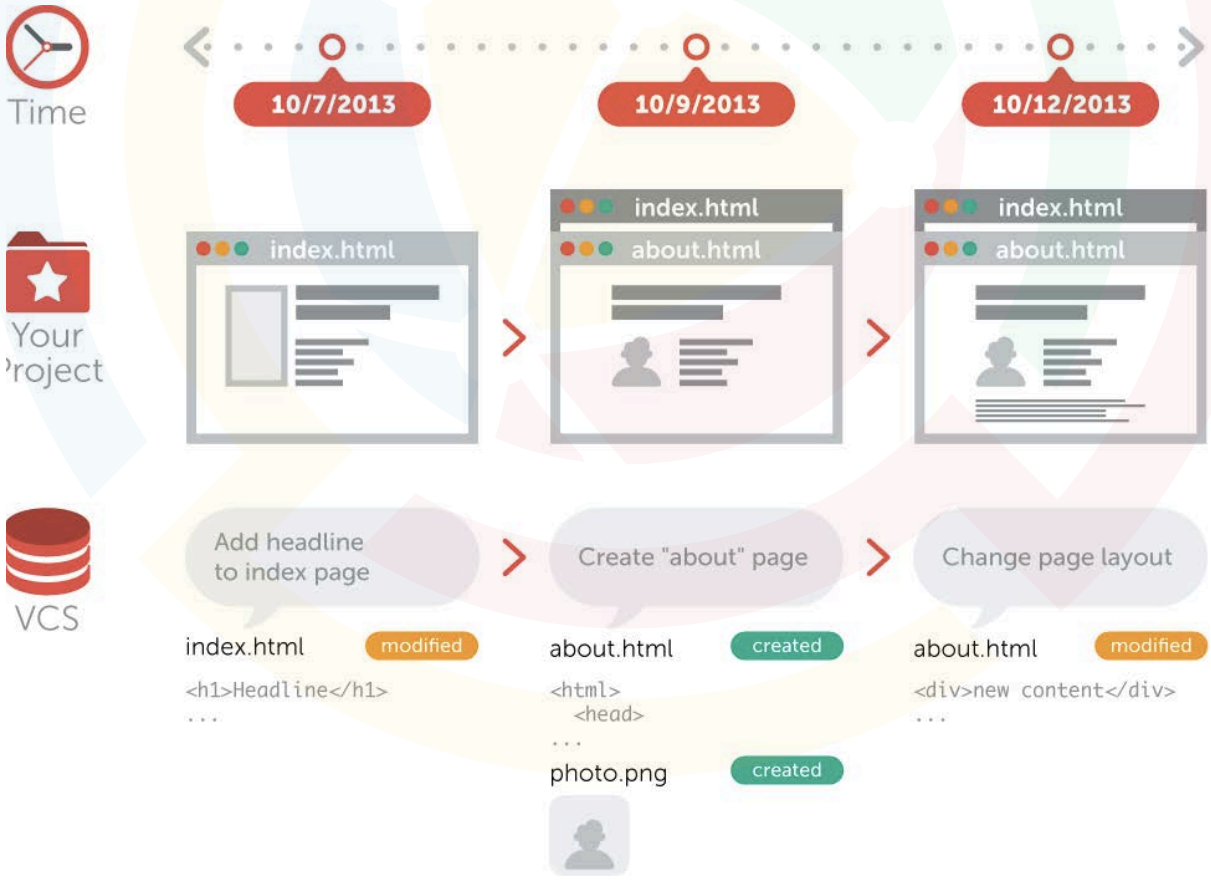
Version Control / Revision control / Source Control is is a software that helps software developers to work together and maintain a complete history of their work.

You can think of a version control system ("VCS") as a kind of "**database**".

It lets you save a snapshot of your complete project at any time you want. When you later take a look at an older snapshot ("version"), your VCS shows you exactly how it differed from the previous one.

A version control system **records the changes** you make to your project's files.

This is what version control is about. It's really as simple as it sounds.



Popular VCS

=========

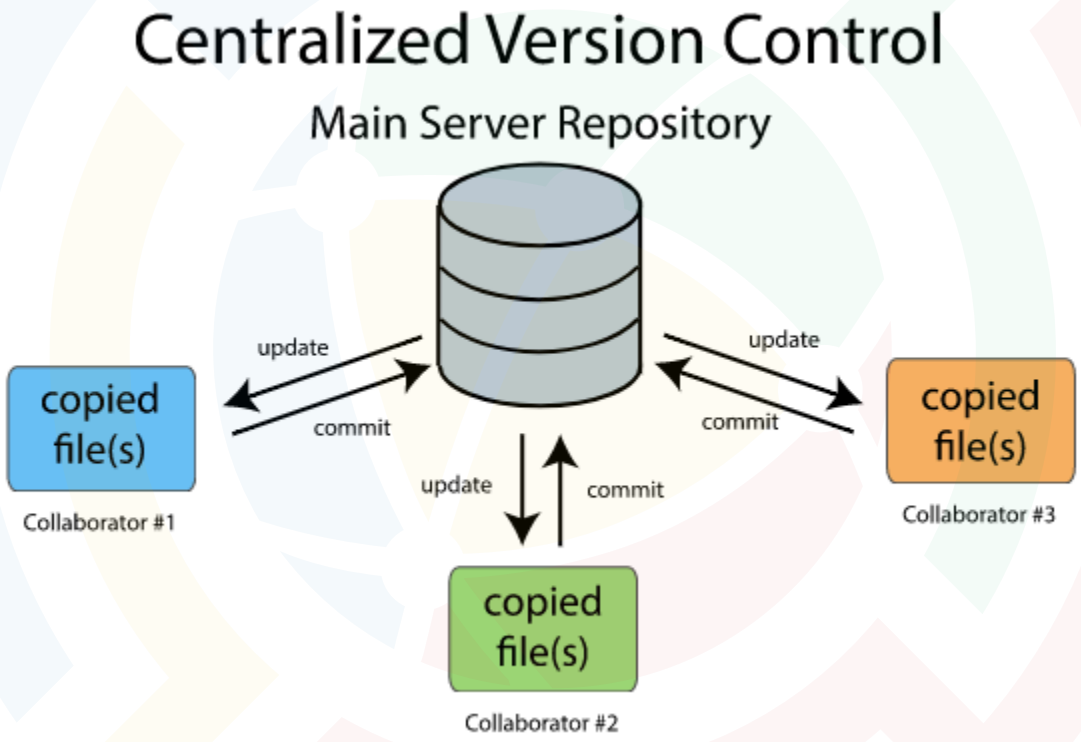
Types of VCS

- **Centralized version control system (CVCS)**
- **Ex: CVS, SVN**
- **Distributed version control system (DVCS)**
- **Ex: Git, Mercurial**

Centralized Version Control System (CVCS)

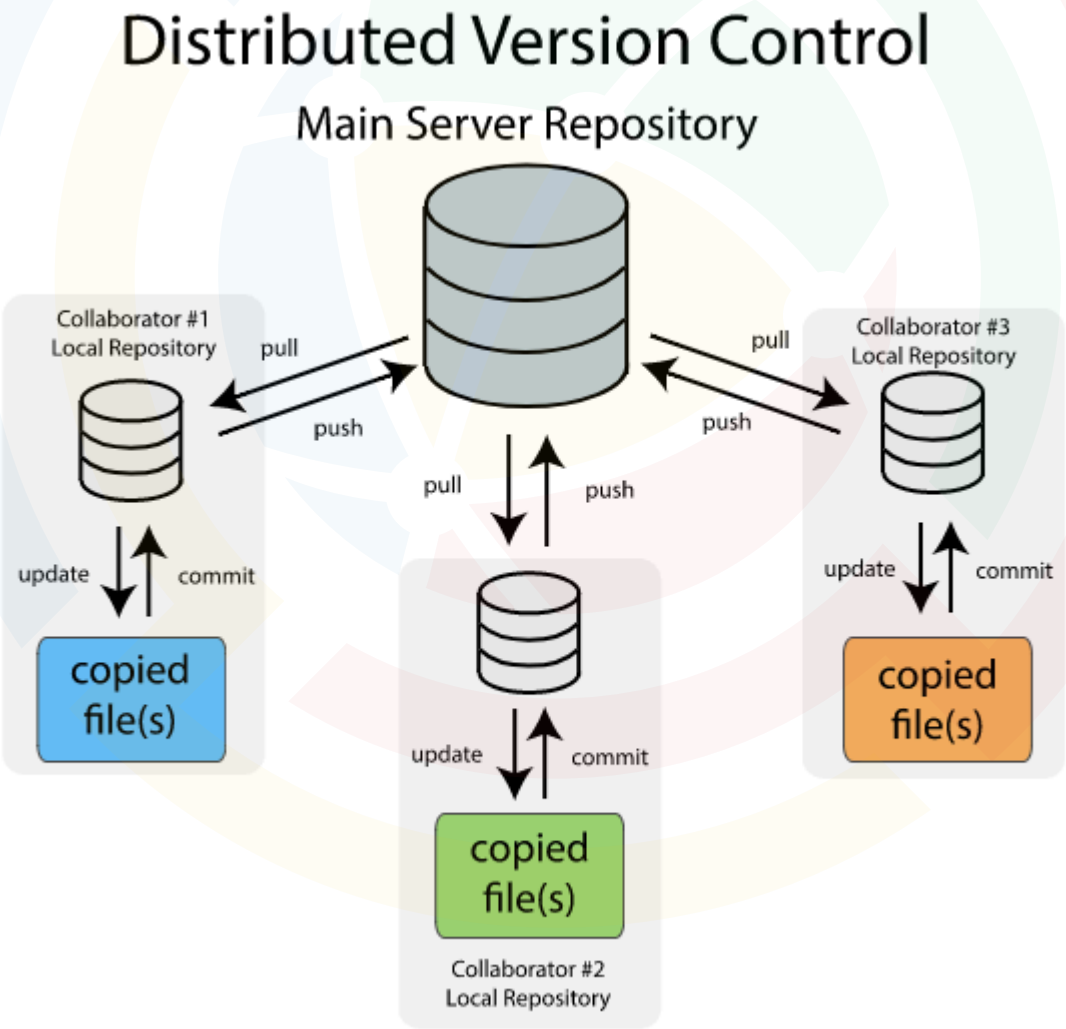
Uses a central server to store all files and enables team collaboration.

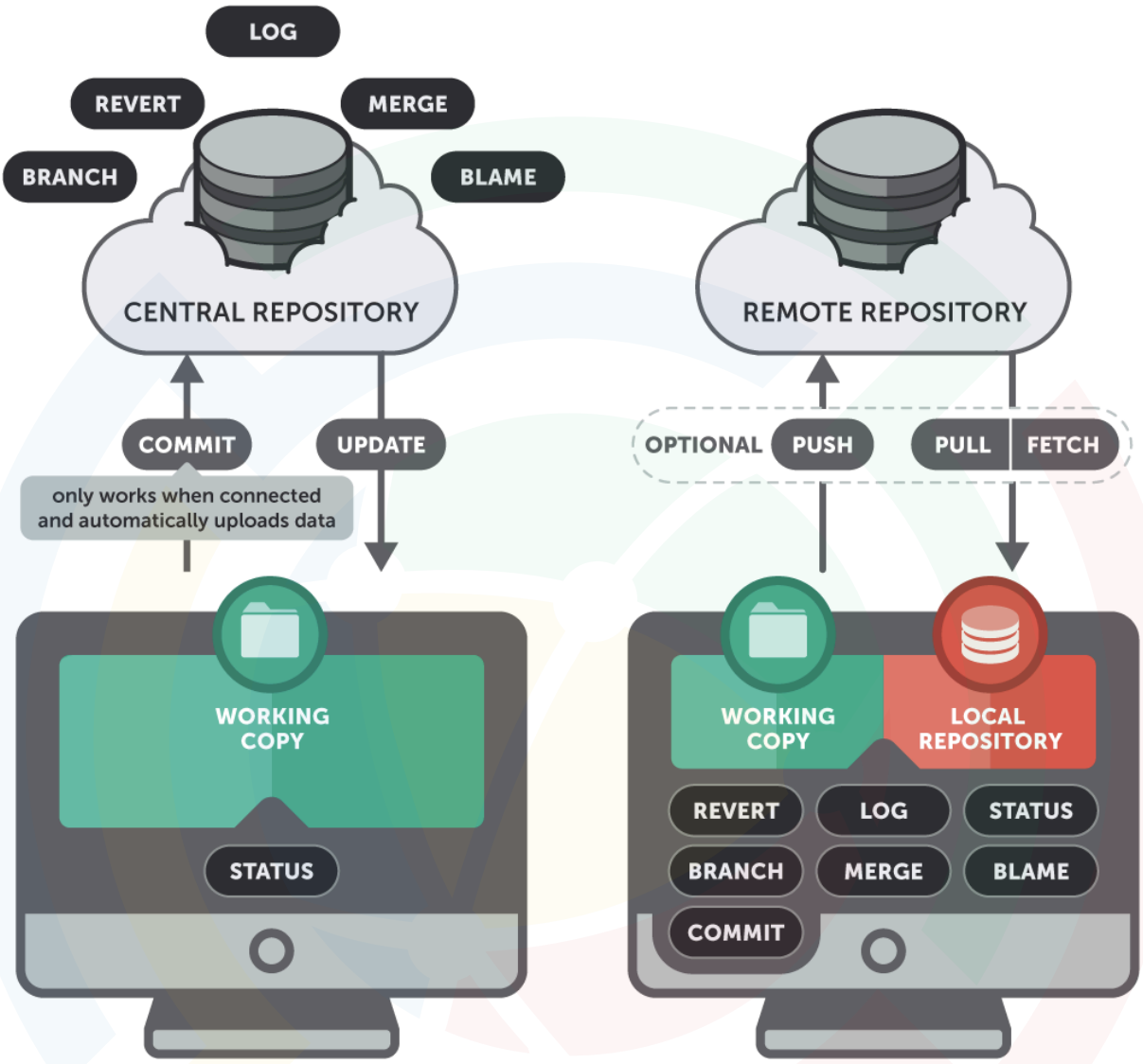
But the major drawback of CVCS is its **single point of failure,** i.e., failure of the central server. Unfortunately, if the central server goes down for an hour, then during that hour, no one can collaborate at all.

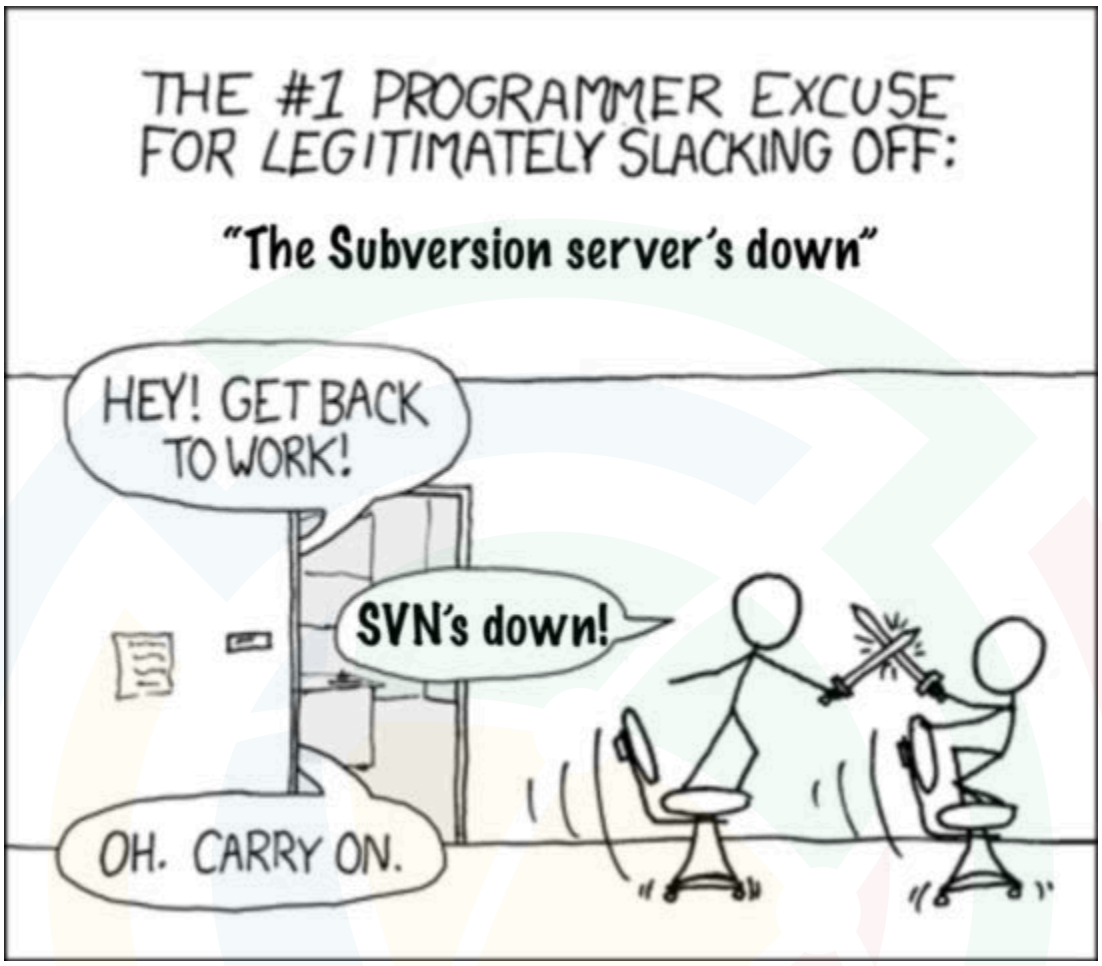


Distributed Version Control System (DVCS)

DVCS does not rely on the central server and that is why you can perform many operations when you are offline. You can commit changes, create branches, view logs, and perform other operations when you are offline. You require network connection only to publish your changes and take the latest changes.




SUBVERSION
GIT
LOG
REVERT
MERGE
BRANCH
BLAME
CENTRAL REPOSITORY
COMMIT
UPDATE
only works when connected
and automatically uploads data
WORKING
COPY
STATUS
REMOTE REPOSITORY
OPTIONAL
PUSH
PULL
FETCH
WORKING
COPY
LOCAL
REPOSITORY
REVERT
LOG
STATUS
BRANCH
MERGE
BLAME
COMMIT



How the Typical VCS works

==========

A typical VCS uses something called **Two tree architecture**, this is what a lot of other VCS use apart from git.

Usually, a VCS works by having two places to store things:

1. **Working Copy**
2. **Repository**

These are our two trees, we call them trees because they represent a file structure.

Working copy [CLIENT] is the place where you make your changes.

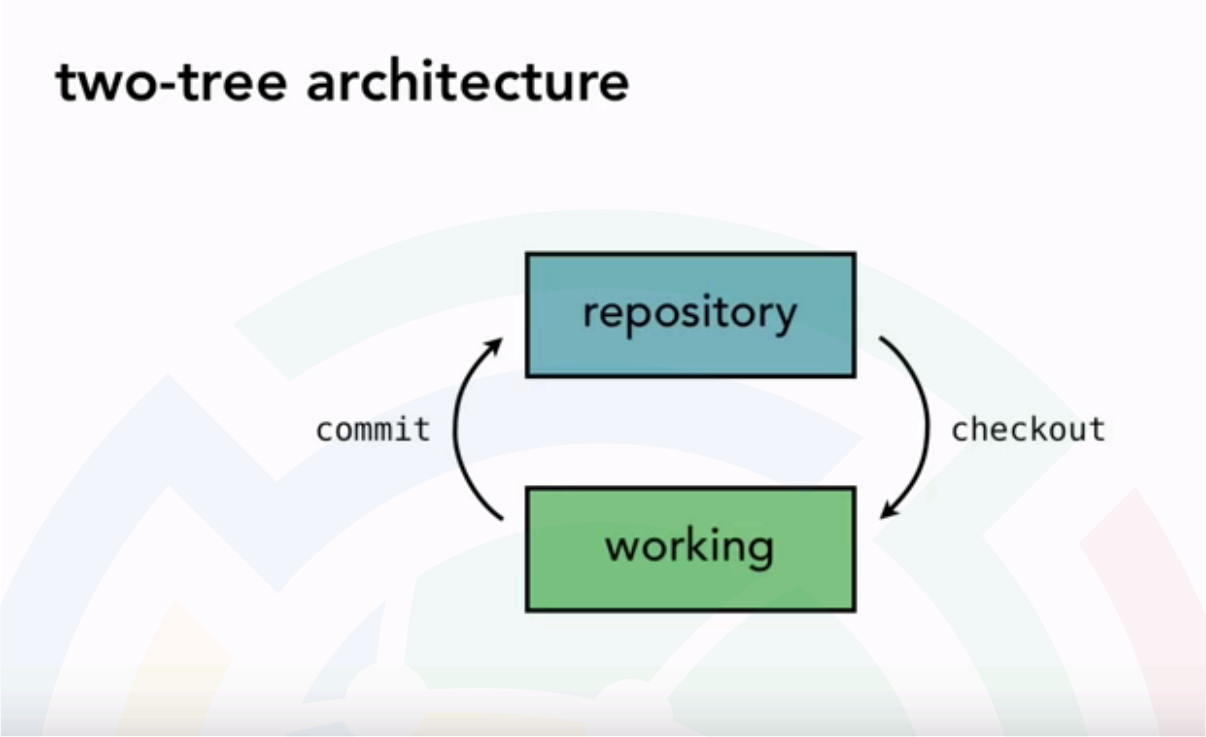
Whenever you edit something, it is saved in working copy and it is a physically stored in a disk.

Repository [SERVER] is the place where all the version of the files or commits, logs etc is stored. It is also saved in a disk and has its own set of files.

You cannot however change or get the files in a repository directly, in able to retrieve a specific file from there, you have to checkout

Checking-out is the process of getting files from repository to your working copy. This is because you can only edit files when it is on your working copy. When you are done editing the file, you will save it back to the repository by committing it, so that it can be used by other developers.

Committing is the process of putting back the files from working copy to repository.



Hence, this architecture is called **2 Tree Architecture.**

Because you have two tree in there **Working Copy** and **Repository**.

The famous VCS with this kind of architecture is Subversion or SVN.

How the Distributed DVCS works

==========

Unusually, a DVCS works by having three places to store things:

1. **Working Copy**
2. **Staging**
3. **Repository**

As Git uses Distributed version control system, So let's talk about Git which will give you an understanding of DVCS.

Git was initially designed and developed by Linus Torvalds in 2005 for Linux kernel development. Git is an Open Source tool.

History

========

For developing and maintaining Linux Kernel, Linus Torvalds used **BitKeeper** which is also one of the VCS, and is open source till 2004.

So instead of depending on other tools, they developed their own VCS.

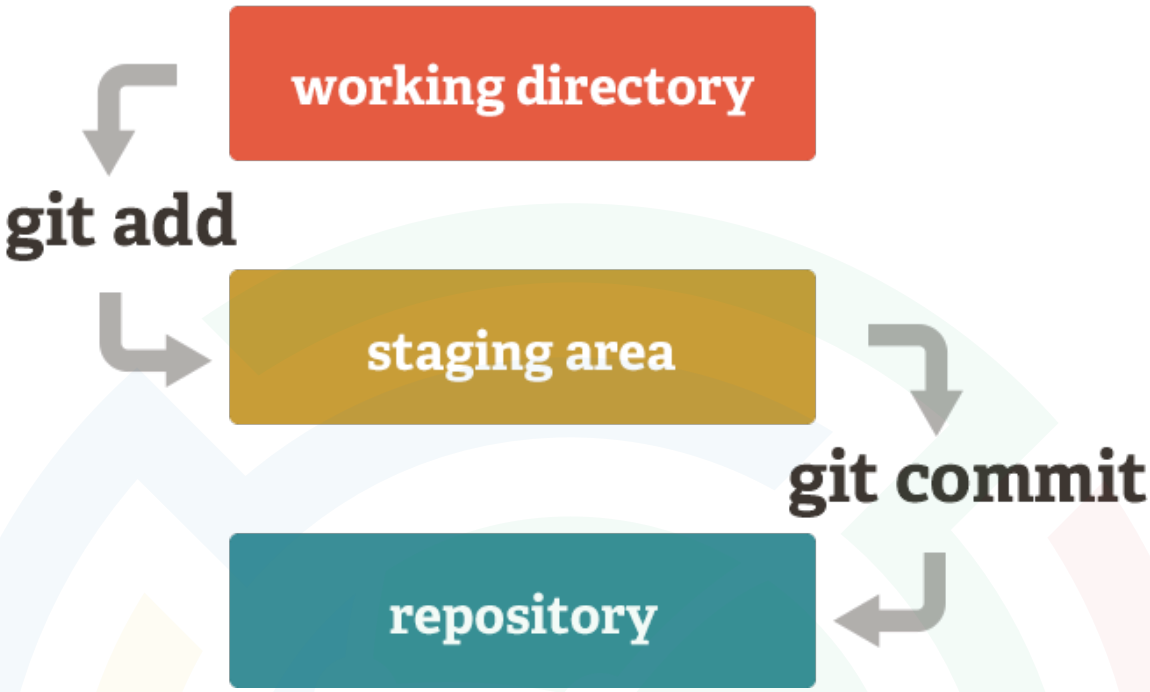
Just see the wiki of Git.

Git Architecture

==========

Git uses three tree architecture.

Well interestingly Git has the **Working Copy** and **Repository** as well but it has added an extra tree **Staging** in between:



As you can see above, there is a new tree called **Staging**.

What this is for ?

This is one of the fundamental difference of Git that sets it apart from other VCS, this **Staging tree** (usually termed as **Staging area**) is a place where you prepare all the things that you are going to commit.

In Git, you don't move things directly from your working copy to the repository, you have to stage them first, one of the main benefits of this is, **to break up your working changes into smaller, self-contained pieces.**

To stage a file is to prepare it for a commit.

Staging allows you finer control over exactly how you want to approach version control.

Advantages Of Git

==========

Git works on most of OS: Linux, Windows, Solaris and MAC.

Installing Git

==========

- Download Git {git website}

To check if git is available or not use:

```
# rpm -qa | grep git
# sudo yum install git
```

 # git --version

Setting the Configuration

==========

 # git config --global user.name "Ravi Krishna"

 # git config --global user.email "info@gmail.com"

 # git config --list

NOTE :: The above info is not the authentication information.

What is the need of git config

==========

When we setup git and before adding bunch of files, We need to fill up username & email and it's basically git way of creating an account.

Working with Git

==========

Getting a Git Repository

 # mkdir website

Initialising a repository into directory, to initialize git we use:
 # git init

The purpose of Git is to manage a project, or a set of files, as they change over time. Git stores this information in a data structure called a repository.

git init is only for when you create your own new repository from scratch.

It turns a directory into an empty git repository.

Let's have some configuration set:

git config --global user.name "Ravi"

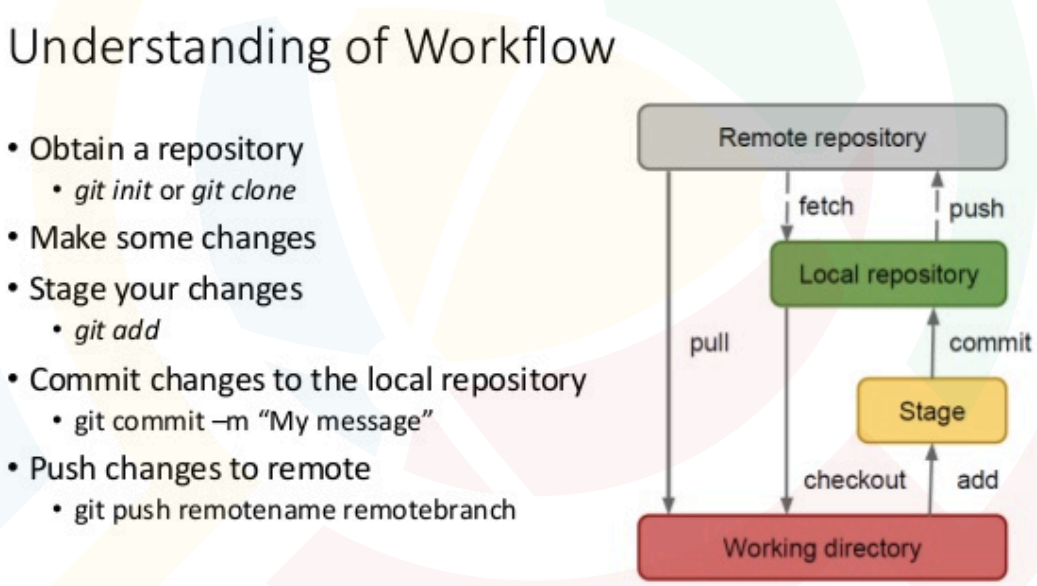
git config --global user.email "ravi@digital-lync.com"

Taking e-commerce sites as example.

Basic Git Workflow

==========

1. You modify files in working directory
2. You stage files, adding snapshots of them to your staging area.
3. You do a commit, which takes the files as they are in the staging area and stores that snapshot to your git repository.



mkdir website

git init

git status {Branches will talk later}

vi index.html

{put some tags <html><title><h1><body> just structure}

git status

git add index.html {staged the changes}

git commit -m “Message” {moves file from staging area to local repo}

git status

You can skip the staging area by # git commit -a -m “New Changes”

Commit History - How many Commits have happened ??

==========

To see what commits have been done so far we use a command:

 # git log

It gives commit history basically commit number, author info, date and commit message.

Want to see what happend at this commit, zoom in info we use:

 # git show <commit number>

Let’s understand this **commit number**
This is sha1 value randomly generated number which is 40 character hexadecimal number which will be unique.

Let’s change the title in index.html and go with

git add status commit

git log {gives the latest commit on top and old will get down}

Git diff

========

Let’s see, the diff command gives the difference b/w two commits.

 # git diff xxxxxx..xxxxxx

You can get diff b/w any sha's like sha1..sha20.

Now our log started increasing, like this the changes keep on adding file is one but there are different versions of this file.

git log --since YYYY-MM-DD

git log --author ravi

git log --grep HTML { commit message }

git log --oneline

Git Branching

==========

In a collaborative environment, it is common for several developers to share and work on the same source code.

Some developers will be fixing bugs while others would be implementing new features.

Therefore, there has got to be a manageable way to **maintain different versions** of the same code base.

This is where the branch function comes to the rescue. **Branch allows** each developer **to branch out from the original code base and isolate their work from others.** Another good thing about branch is that **it helps Git to easily merge** the versions later on.

It is a common practice to create a new branch for each task (eg. bug fixing, new features etc.)

Branching means you diverge from the main line(master-working copy of application) of development and continue to do work without messing with that main line.

Basically, you have your master branch and you don't want to mess anything up on that branch.

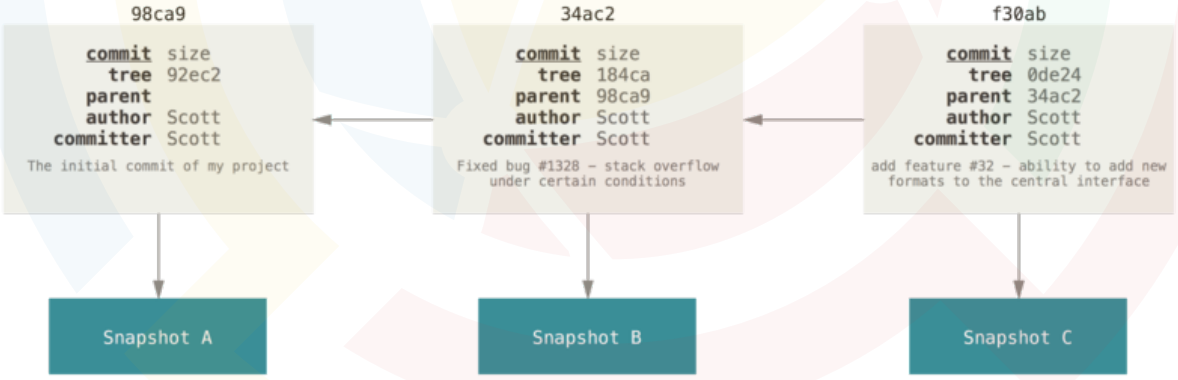
In many VCS tools, **branching** is an **expensive process**, often requiring you to create a new copy of your source code directory, which can take a long time for large projects.

Some people refer to **Git's branching model as its "killer feature"** and it certainly sets Git apart in the VCS community.

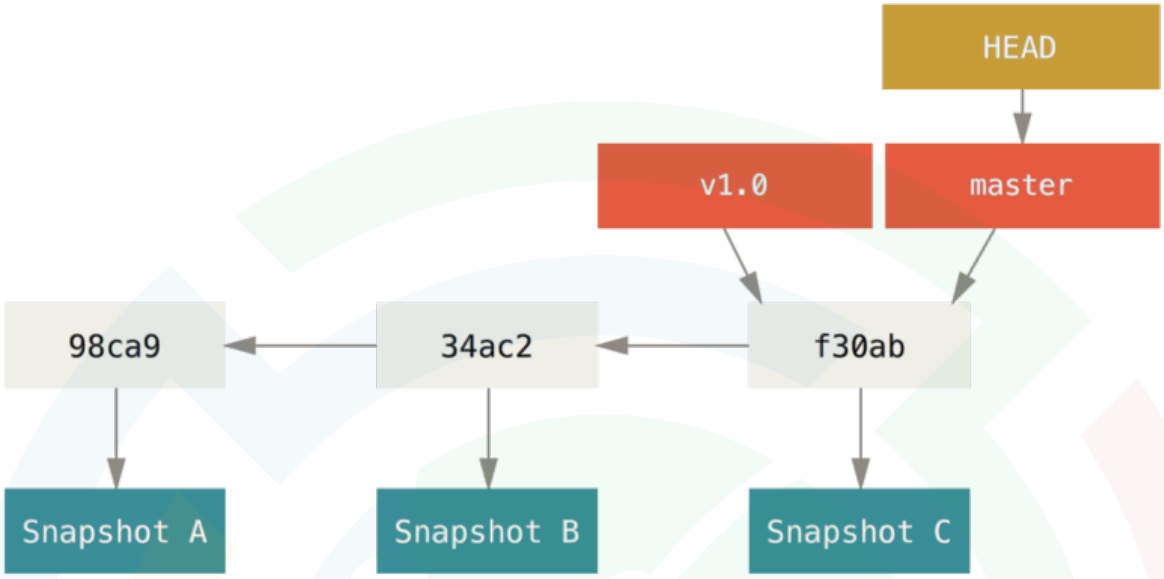
Why is it so special?

The way Git branches is incredibly **lightweight**, making branching operations nearly **instantaneous**, and switching back and forth between branches generally just as fast.

When we make a commits, this is how git stores them.



A branch in Git is simply a lightweight movable pointer to one of these commits. The default branch name in Git is master. As you start making commits, you're given a master branch that points to the last commit you made. Every time you commit, it moves forward automatically.

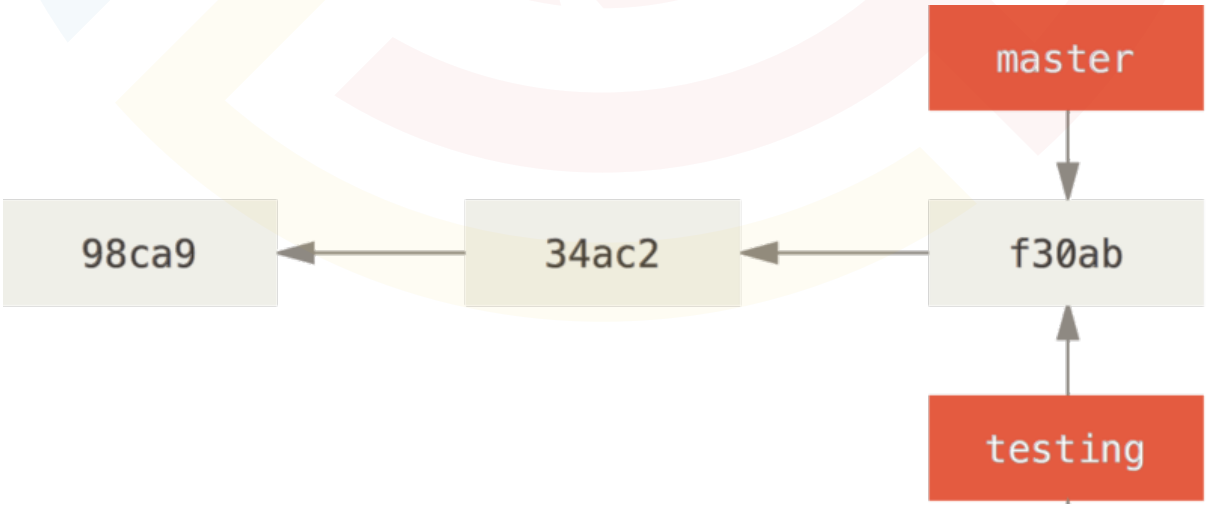


What happens if you create a new branch? Well, doing so creates a new pointer for you to move around.

Let's say you create a new branch called testing.

 # git branch testing

This creates a new pointer to the same commit you're currently on.



Two branches pointing into the same series of commits.

How does Git know what branch you're currently on?

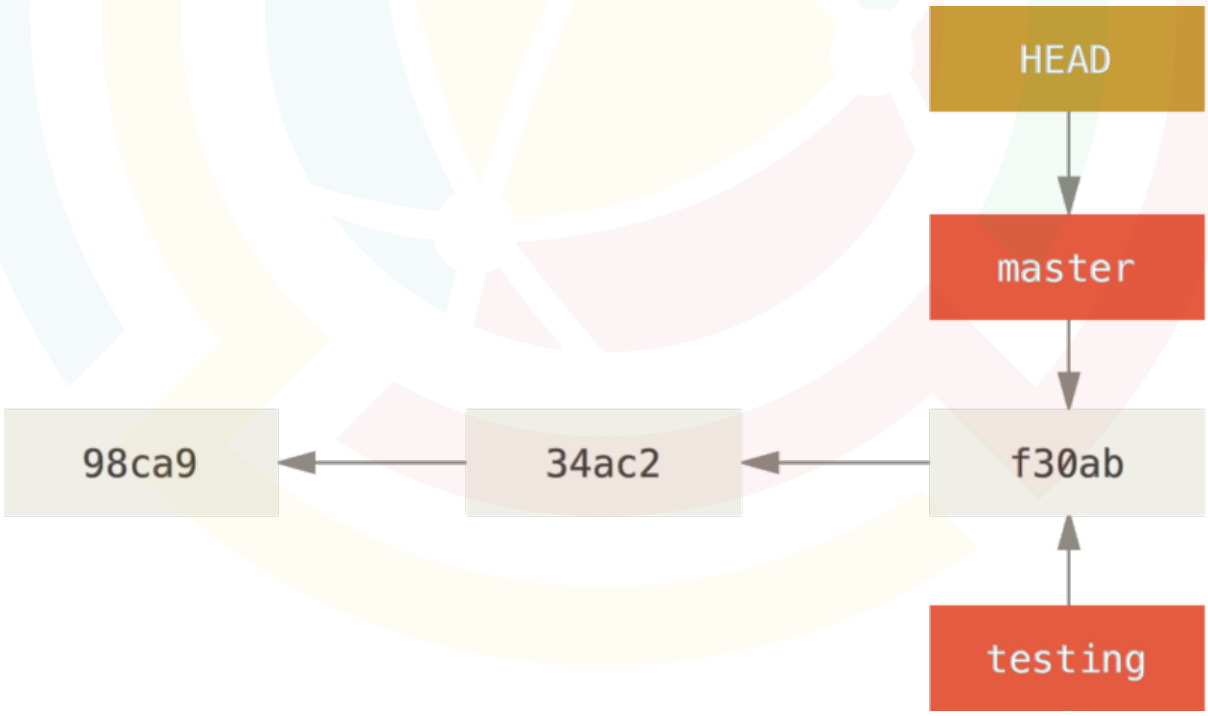
It keeps a **special pointer** called **HEAD**.

HEAD is a pointer to the latest commit id and is **always moving**, not stable.

```
# git show HEAD
```

In Git, this is a pointer to the local branch you're currently on.

In this case, you're still on master. The git branch command only created a new branch — it didn't switch to that branch.



This command shows you **where the branch pointers are pointing:**

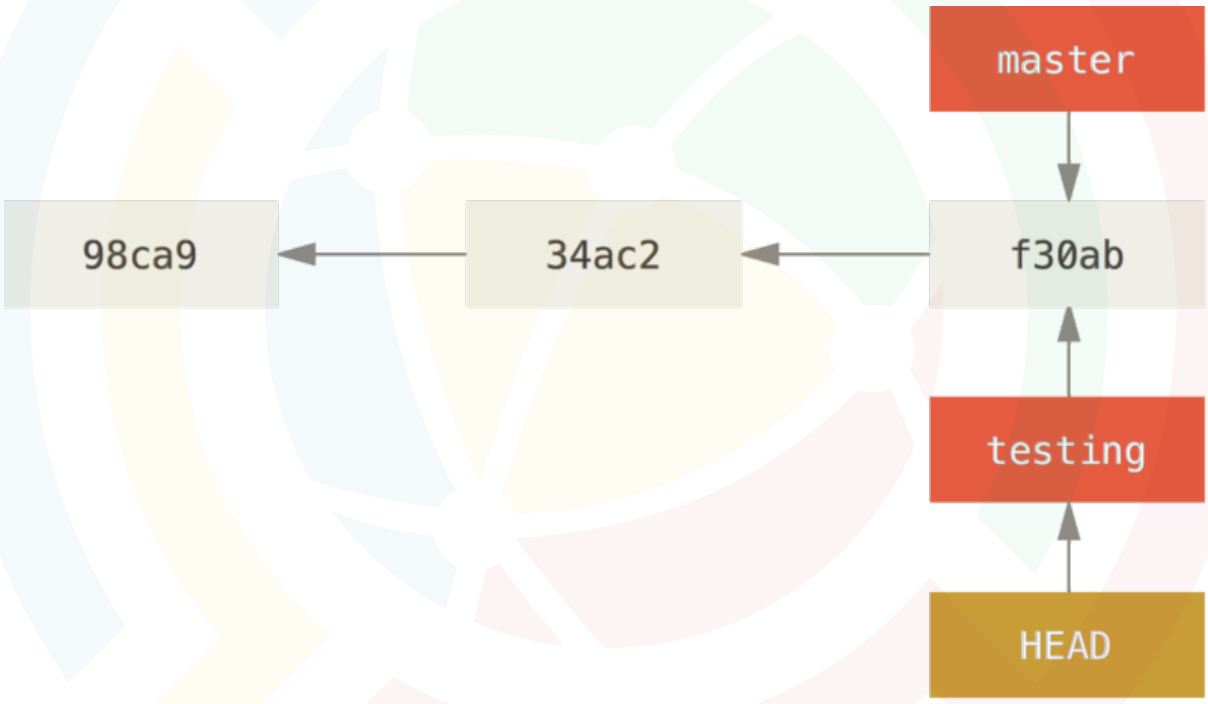
 # git log --oneline --decorate

You can see the "**master**" and "**testing**" branches that are right there next to the f30ab commit.

To **switch to an existing branch**, you run the git checkout command.

git checkout testing

This **moves HEAD** to point to the **testing** branch.



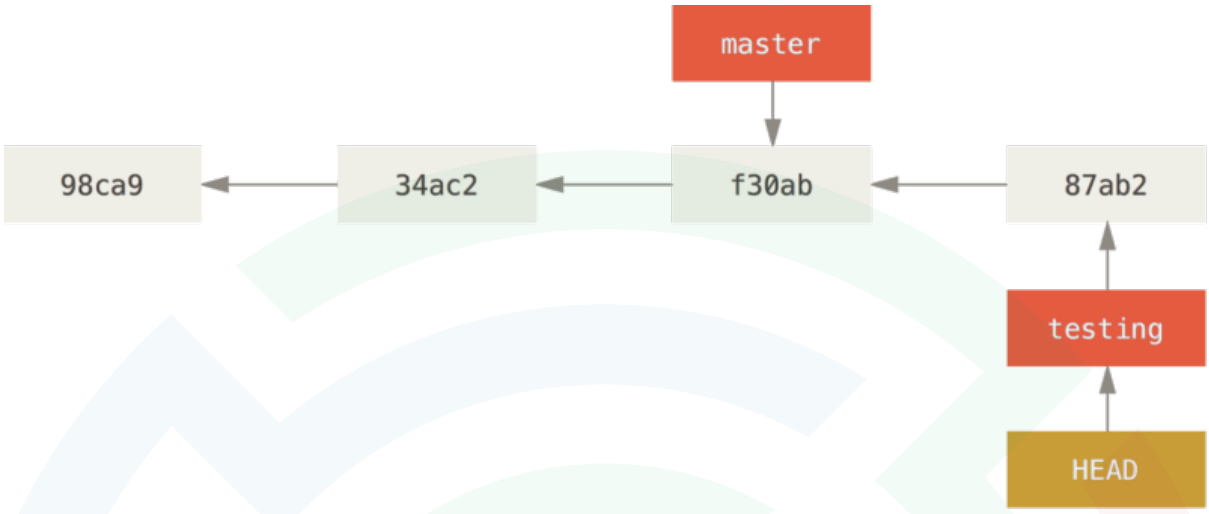
What is the significance of that?

Well, let's do another commit:

vim test.rb

git commit -a -m 'made a change'

 # git log --oneline --decorate

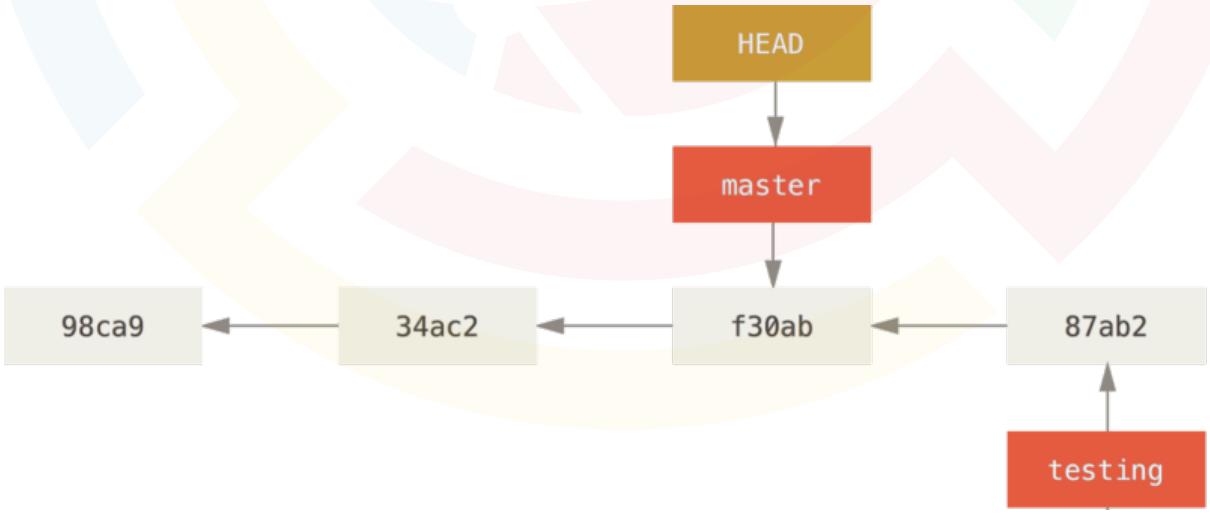


The HEAD branch moves forward when a commit is made.

This is interesting, because now your **testing** branch has **moved forward**, but your master branch still points to the commit you were on when you ran git checkout to switch branches.

Let's switch back to the master branch:

```
# git checkout master
```



HEAD moves when you checkout.

That command(git checkout master) did two things. It **moved** the **HEAD** pointer back to point to the **master branch**, and it **reverted the files** in your working directory **back to the snapshot** that **master points to**.

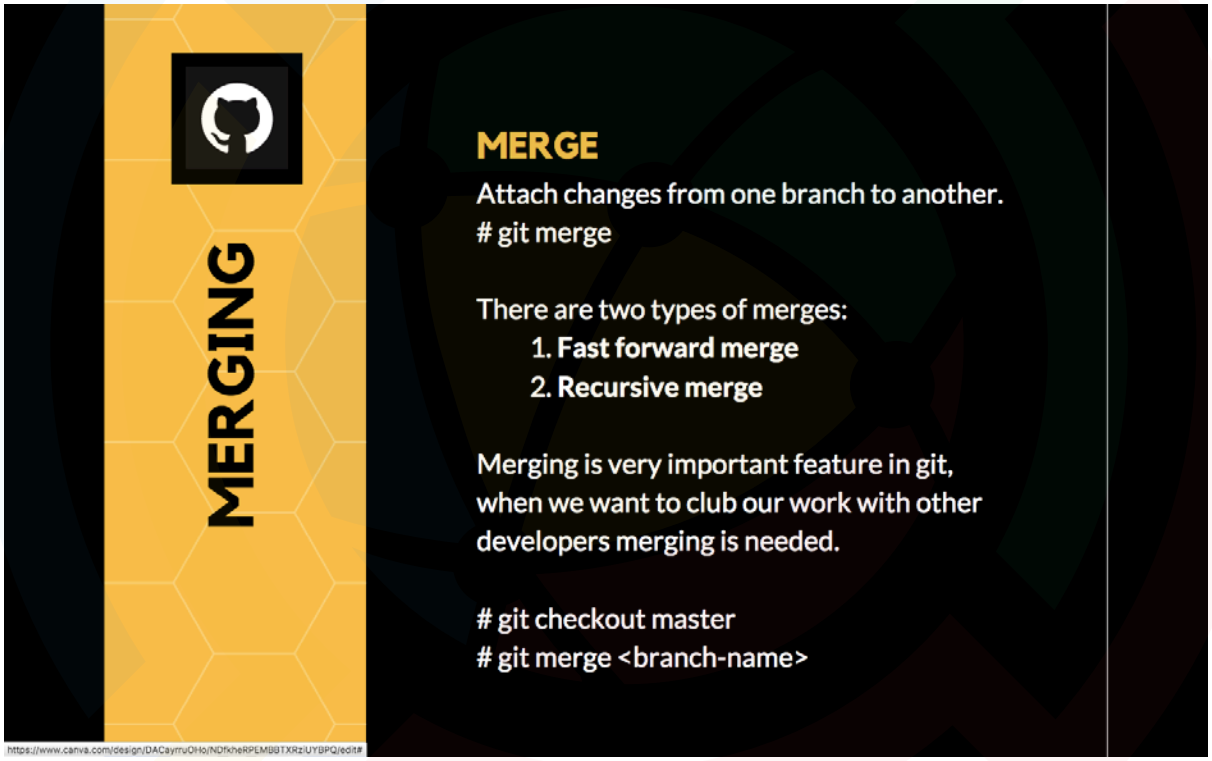
To see all available branches

 # git branch -a

 # git reflog {short logs}

Merging

==========



Git Merge Conflict

==========

A merge conflict happens when two branches both modify the same region of a file and are subsequently merged.

Git don't know which of the changes to keep, and thus needs human intervention to resolve the conflict.

Showing the example **R&D**, **Training** and **Consulting**.

Automatic merge failed

Example showing Merge Conflict

==========

On master branch

vi services.html { Add Two Dummy services like Research & Development }
git branch training
git branch consulting

git checkout training

vi services.html { We provide training }
git add && git commit
git checkout master
git merge training

git checkout consulting
vi services.html { We provide Consulting add in same line}
git add && git commit
git checkout master
git merge consulting

Automatic merge failed

we do get a merge conflict here, open the services.html in vi and resolve the conflicts that occurred.

git add .
git commit -m “Conflict resolved”

Git ignore

==========

It’s a list of files you want git to ignore in your working directory.

It's usually used to avoid committing transient files from your working directory that aren't useful to other collaborators such as temp files IDE's create, Compilation files, OS files etc.

A file should be ignored if any of the following is true:

- **The file is not used by your project**
- **The file is not used by anyone else in your team**
- **The file is generated by another process**

.gitignore

==========

https://www.gitignore.io/

If you want some files to be ignored by git **create a file .gitignore**

*.bk

*.class

 # Ignore all php files

*.php

 # but not index.php

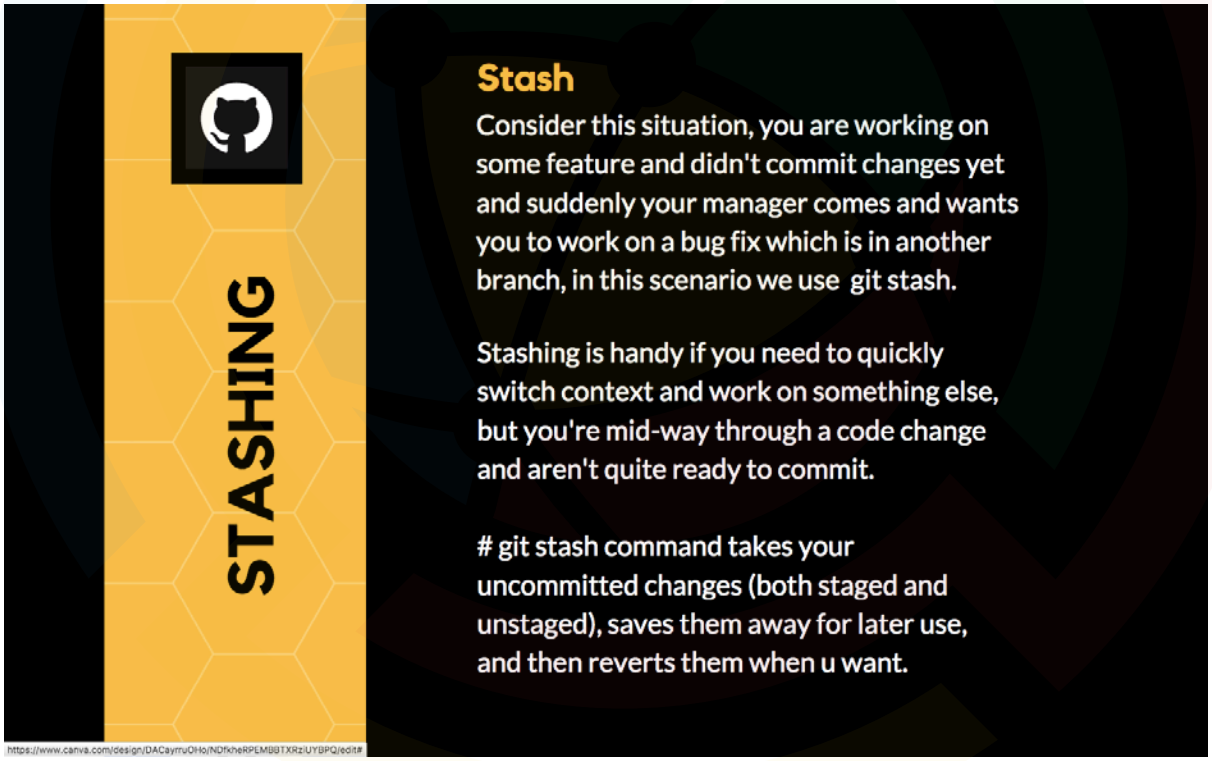
!index.php

 # Ignore all text files that start with aeiou

[aeiou]*.txt

Stashing

==========



 # git stash

 # git stash list

 # git stash apply {apply the top most stashed changes}

 # git stash apply stash@{2} {apply particular stashed changes}

 # git stash show <stash>

 # git stash pop {apply 2nd stash and remove it}

 # git stash pop stash@{2} {pop the stash at 2nd reference}

 # git stash drop stash@{3} {remove the stash}

 # git stash clear {Delete all stash entries}

Cherry Picking
==========

Cheery picking in Git is designed to apply some commit from one branch to another branch.

You can just revert the commit and cherry-pick it on another branch.

Tagging

==========

In release management we are working as a team and I'm working on a module and whenever I'm changing some files I'm pushing those files to remote master.

Now I have some 10 files which are perfect working copy, and I don't want this files to be messed up by my other team members, these 10 files they can directly go for release.

But if I keep them in the repository, as my team is working together, there is always a chance that, somebody or other can mess that file, so to avoid these we can do **TAGGING**.

You can tag till a particular commit id, imagine all the files till now are my working copies:

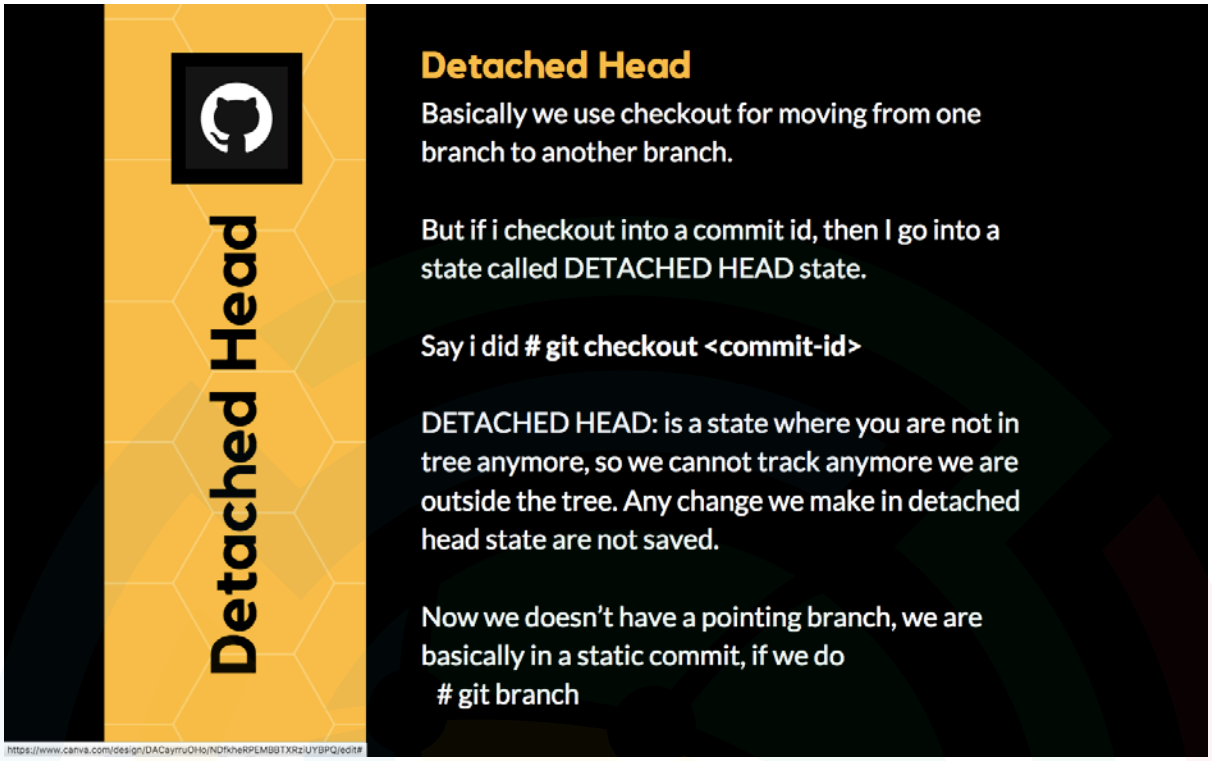
 # git tag 1.0 -m “release 1.0” <commit_id>

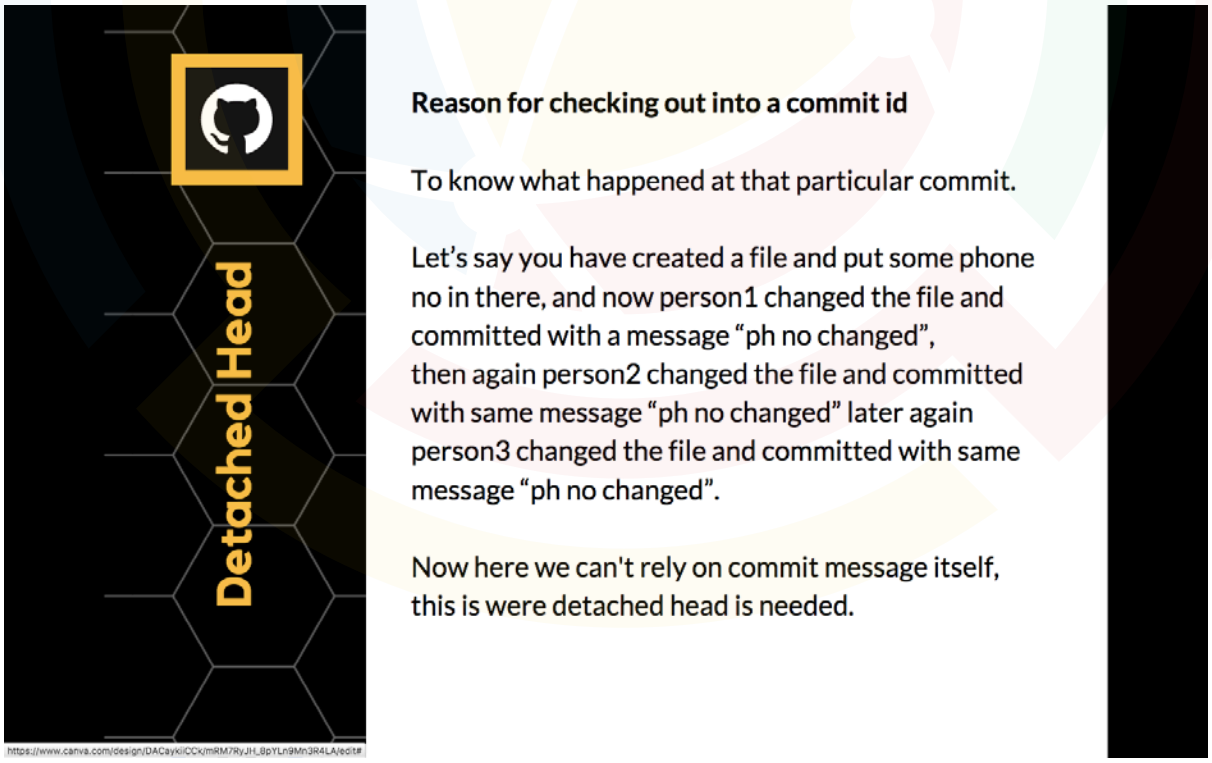
 # git show 1.0

 # git push --tags

Goto GitHub and see release click on it, you can download all the files till that commit.

TAGGING helps you in **release management**.


Detached Head
Detached Head
Basically we use checkout for moving from one branch to another branch.
But if i checkout into a commit id, then I go into a state called DETACHED HEAD state.
Say i did # git checkout <commit-id>
DETACHED HEAD: is a state where you are not in tree anymore, so we cannot track anymore we are outside the tree. Any change we make in detached head state are not saved.
Now we doesn't have a pointing branch, we are basically in a static commit, if we do
git branch


Detached Head
Reason for checking out into a commit id
To know what happened at that particular commit.
Let's say you have created a file and put some phone no in there, and now person1 changed the file and committed with a message "ph no changed",
then again person2 changed the file and committed with same message "ph no changed" later again person3 changed the file and committed with same message "ph no changed".
Now here we can't rely on commit message itself, this is were detached head is needed.