




Introduction to Git & SVN

Anshuman Sanghvi
Project Lead, Java
STTL



Contents


- Introduction to VCS
 - What is VCS?
 - Centralized vs Decentralized VCS
- Introducing Git & SVN
- Why Git over other VCS like SVN
- Exploring Git Theory
 - Features
 - States, Stages and Flow
 - Additional Git Concepts
 - Git Flow Revisited
- Working with Branches
 - Merge & Rebase
 - Exercise
- How to use Git in your work
 - Git Commands
 - Getting started
 - Demo
 - General Practices
 - Working on a task
- References



What is Version Control System

Need for VCS

Types of VCS



What is a Version Control System

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, maintaining the history of the evolution of the code.

What is the need for a Version Control System?

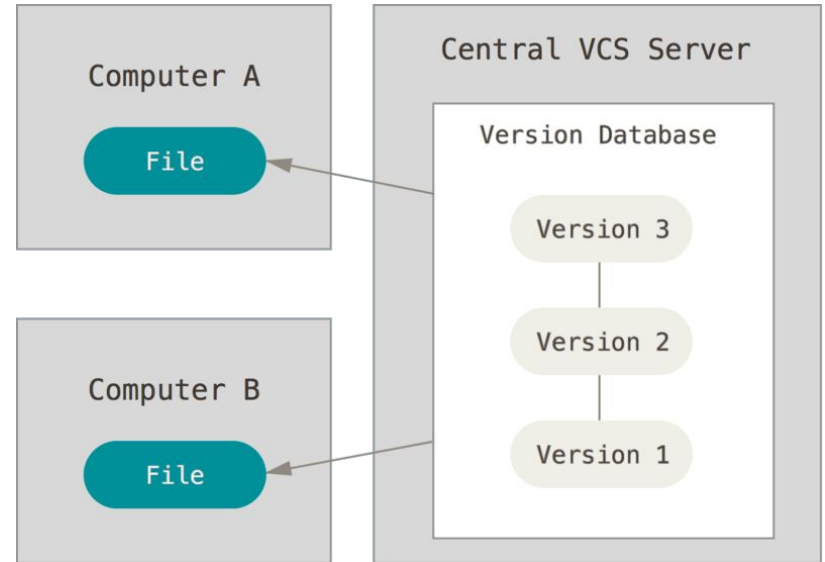
- Backup and recovery of code
 - For backing up your code
 - Reverting to a specific commit in the timeline of the code
- Traceability
 - Which commit did the bug originate from and who wrote it?
 - Keeping a record of the history and order of code changes
- to work in a team
 - working on same portion of code
 - working on parallel features simultaneously without getting affected by each other
- to have version releases like beta, alpha, v1, v2

Centralized VCS

This setup offers many advantages, especially over local VCSs. For example, everyone knows to a certain degree what everyone else on the project is doing.

Administrators have fine-grained control over who can do what, and it's far easier to administer a CVCS than it is to deal with local databases on every client.

However, this setup also has some serious downsides. The most obvious is the single point of failure that the centralized server represents.

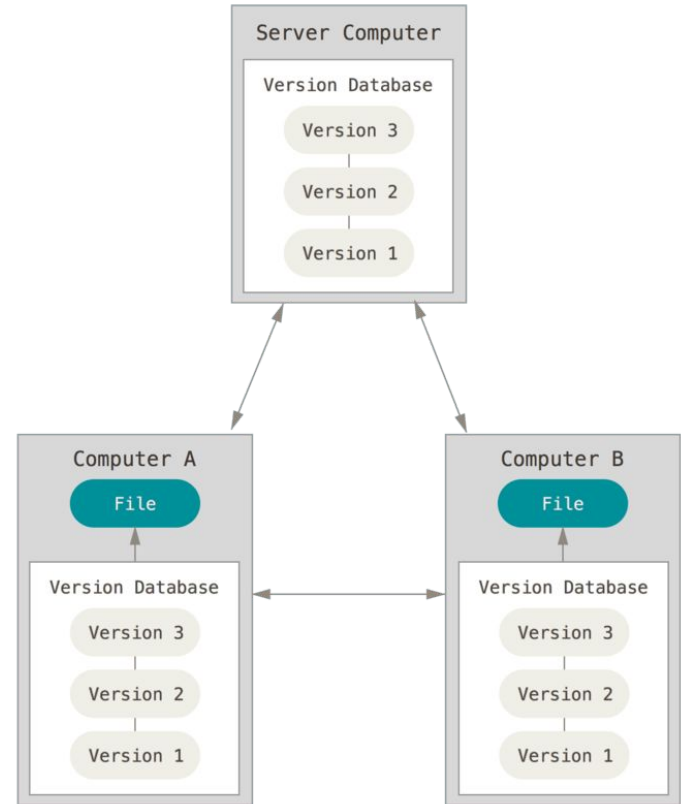


Distributed VCS

In a DVCS (such as Git, Mercurial, Bazaar or Darcs), clients don't just check out the latest snapshot of the files; rather, they fully mirror the repository, including its full history.

Thus, if any server dies, and these systems were collaborating via that server, 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.





Introducing Subversion



Introducing SVN

Apache Subversion (often abbreviated **SVN**, after its command name *svn*) is a [software versioning](#) and [revision control](#) system distributed as [open source](#) under the [Apache License](#).^[1] Software developers use Subversion to maintain current and historical versions of files such as [source code](#), web pages, and documentation. Its goal is to be a mostly compatible successor to the widely used [Concurrent Versions System](#) (CVS).

History [\[edit \]](#)

[CollabNet](#) founded the Subversion project in 2000 as an effort to write an open-source version-control system which operated much like [CVS](#) but which fixed the bugs and supplied some features missing in CVS.^[3] By 2001, Subversion had advanced sufficiently to [host its own source code](#),^[3] and in February 2004, version 1.0 was released.^[4] In November 2009, Subversion was accepted into Apache Incubator: this marked the beginning of the process to become a standard top-level Apache project.^[5] It became a top-level Apache project on February 17, 2010.^[6]

Introducing Git

Introducing Git

Git ([/git/](#))^[8] is a [distributed version control](#) system^[9] that tracks changes in any set of [computer files](#), usually used for coordinating work among programmers who are collaboratively developing [source code](#) during [software development](#). Its goals include speed, [data integrity](#), and support for [distributed](#), non-linear workflows (thousands of parallel [branches](#) running on different computers).^{[10][11][12]}

History [\[edit \]](#)

Git development was started by Torvalds in April 2005 when the proprietary [source-control management](#) (SCM) system used for Linux kernel development since 2002, [BitKeeper](#), revoked its free [license](#) for Linux development.^{[21][22]} The copyright holder of BitKeeper, [Larry McVoy](#), claimed that [Andrew Tridgell](#) had created [SourcePuller](#) by [reverse engineering](#) the BitKeeper [protocols](#).^[23] The same incident also spurred the creation of another version-control system, [Mercurial](#).

Naming [\[edit \]](#)

Torvalds sarcastically quipped about the name *git* (which means "unpleasant person" in [British English](#) slang): "I'm an egotistical bastard, and I name all my projects after myself. First '[Linux](#)', now 'git'. "^{[31][32]} The [man page](#) describes Git as "the stupid content tracker".^[33]

Why Git over other VCS?

- git is decentralized
- no single point of failure
- you can work offline.

According to a [stackoverflow](https://stackoverflow.com/survey/2021) survey in 2021,
95% of developers use git.



Git vs SVN

Git

Decentralized Model

- decentralized model
- no single point of failure
- having entire copy means you can run entire project locally and test the integration of your changes

Network usage

- network is required to only the push and pull operations
- operations are faster as they are offline.

single .git dir per repository

more concepts, difficult to learn

large binary files are make git operations slow

SVN

Centralized Model

- There is a central repository
- there is a single point of failure
- you need access to, and sync each directory in repo to test the integration of your changes with the project as a whole

Network usage

- network is required to perform svn operation
- each operation is network bound and slower

.svn dir per folder

simple to use and learn

svn only pulls files that have you are working on. Takes time to get all files and test your changes

SVN Features

SVN Features

Commits as true atomic operations (interrupted commit operations in CVS would cause repository inconsistency or corruption).

The system maintains versioning for directories and some specific file metadata (see Properties). Users can move or copy files and entire directory-trees very quickly, while retaining full revision history (as being implemented by a reference to the original object).

Versioning of symbolic links.

Native support for binary files, with space-efficient binary-diff storage.

SVN Features

Client/server protocol sends diffs in both directions.

Parsable output, including XML log output.

Open source licensed – Apache License since the 1.7 release; prior versions use a derivative of the Apache Software License 1.1.

Internationalized program messages.

File locking for unmergeable files ("reserved checkouts").

SVN Features

Path-based authorization.

Language bindings for C#, PHP, Python, Perl, Ruby, and Java.

Full MIME support – users can view or change the MIME type of each file, with the software knowing which MIME types can have their differences from previous versions shown.

Merge tracking – merges between branches will be tracked, this allows automatic merging between branches without telling Subversion what does and does not need to be merged.

Changelists to organize commits into commit groups.

SVN Theory & Concepts

SVN Concepts

Repository: A repository is the heart of any version control system. It is the central place where developers store all their work. Repository not only stores files but also the history. Repository is accessed over a network, acting as a server and version control tool acting as a client. Clients can connect to the repository, and then they can store/retrieve their changes to/from repository. By storing changes, a client makes these changes available to other people and by retrieving changes, a client takes other people's changes as a working copy.

Trunk: The trunk is a directory where all the main development happens and is usually checked out by developers to work on the project.

Tags: The tags directory is used to store named snapshots of the project. Tag operation allows to give descriptive and memorable names to specific version in the repository.

SVN Concepts

Branches: Branch operation is used to create another line of development. It is useful when you want your development process to fork off into two different directions. For example, when you release version 5.0, you might want to create a branch so that development of 6.0 features can be kept separate from 5.0 bug-fixes.

Working copy: Working copy is a snapshot of the repository. The repository is shared by all the teams, but people do not modify it directly. Instead each developer checks out the working copy. The working copy is a private workplace where developers can do their work remaining isolated from the rest of the team

Commit changes: Commit is a process of storing changes from private workplace to central server. After commit, changes are made available to all the team. Other developers can retrieve these changes by updating their working copy. Commit is an atomic operation. Either the whole commit succeeds or is rolled back. Users never see half finished commit.

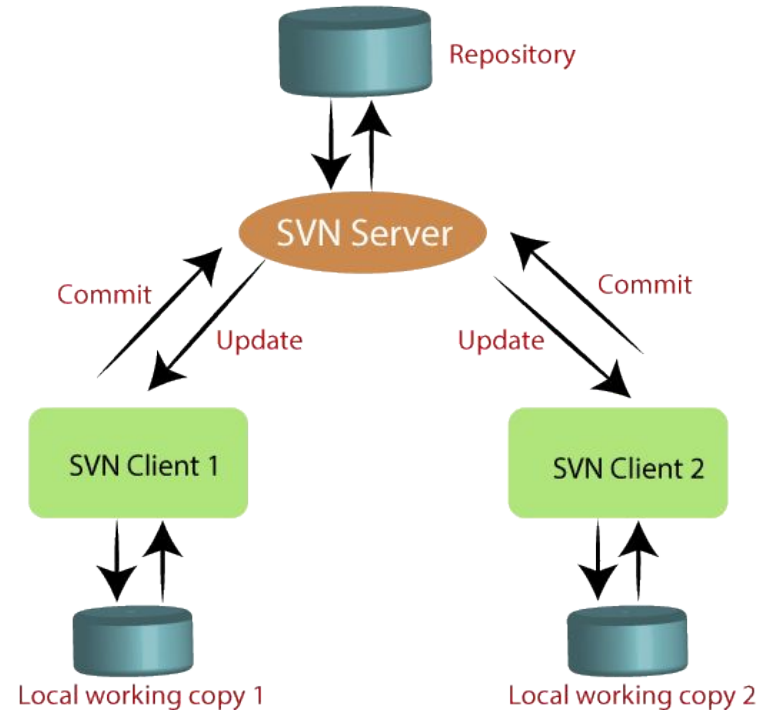
SVN Components

It has two main components:

SVN Server: Contains the master copy of the code repository. It provides remote access and read and write access to clients.

SVN Client: Is installed in the user machine. It provides an interface to talk with the server.

Files are stored under the FSFS (FileSystem atop a FileSystem) storage subsystem.





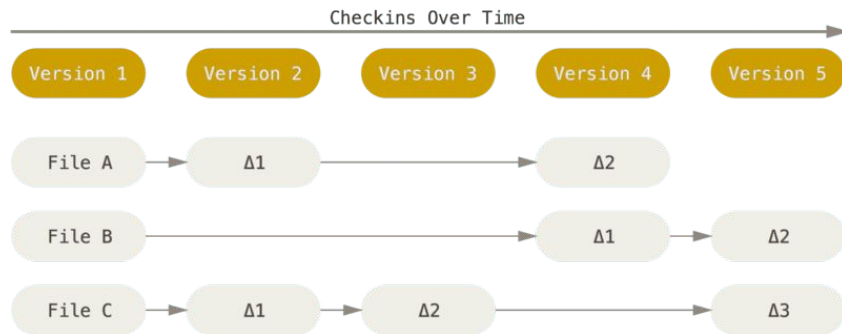
Git Features



Snapshots, not differences

The major difference between Git and any other VCS (Subversion and friends included) is the way Git thinks about its data.

Conceptually, most other systems store information as a list of file-based changes. These other systems (CVS, Subversion, Perforce, Bazaar, and so on) think of the information they store as a set of files and **the changes made to each file** over time (this is commonly described as **delta-based** version control).

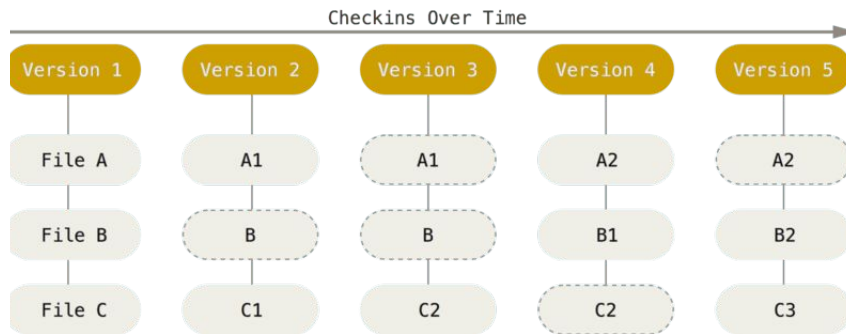


Snapshots, not differences

Git doesn't think of or store its data this way. Instead, Git thinks of its data more like a series of snapshots of a miniature file-system.

With Git, every time you commit, or save the state of your project, **Git basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot.**

To be efficient, if files have not changed, Git doesn't store the file again, just a link to the previous identical file it has already stored. Git thinks about its data more like a **stream of snapshots.**



Nearly every operation is local


- You can browse the history of a repo without being online.
- Almost all your changes can be done locally, and only finally you can sync with an online repository.

Git has integrity


- Everything in Git is checksummed before it is stored and is then referred to by that checksum. This means it's impossible to change the contents of any file or directory without Git knowing about it.
- The mechanism that Git uses for this checksumming is called a SHA-1 hash. This is a 40-character string composed of hexadecimal characters (0–9 and a–f) and calculated based on the contents of a file or directory structure in Git.
- Git stores everything in its database not by file name but by the hash value of its contents.

Git generally only adds data

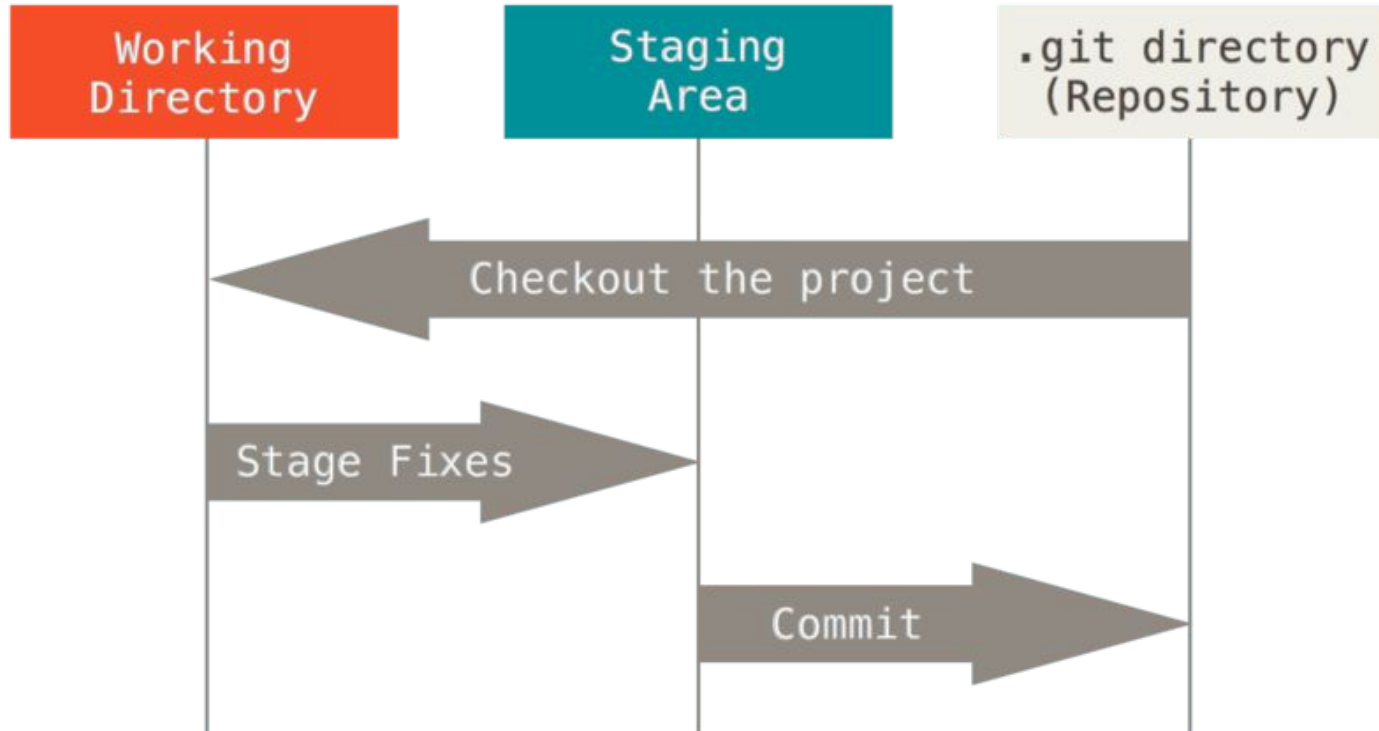
- When you do actions in Git, nearly all of them only add data to the Git database.
- It is hard to get the system to do anything that is not undoable or to make it erase data in any way.



Git Theory & Concepts

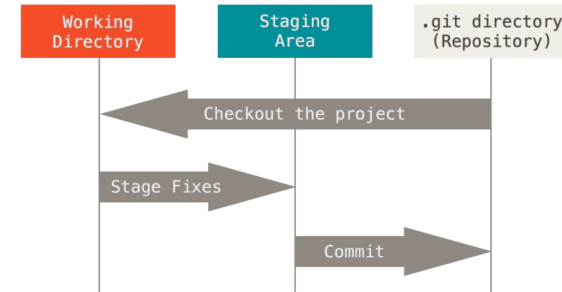


Three States of Git



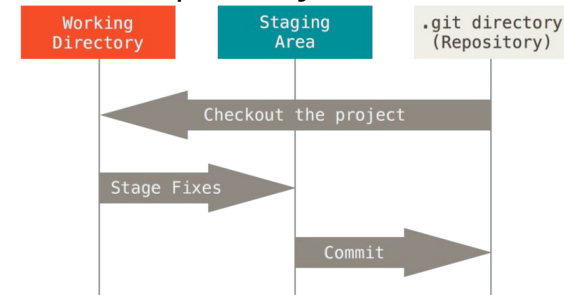
Three States of Git

- Git has three main states that your files can reside in: modified, staged, and committed:
 - **Modified** means that you have changed the file but have not committed it to your database yet.
 - **Staged** means that you have marked a modified file in its current version to go into your next commit snapshot.
 - **Committed** means that the data is safely stored in your local database.

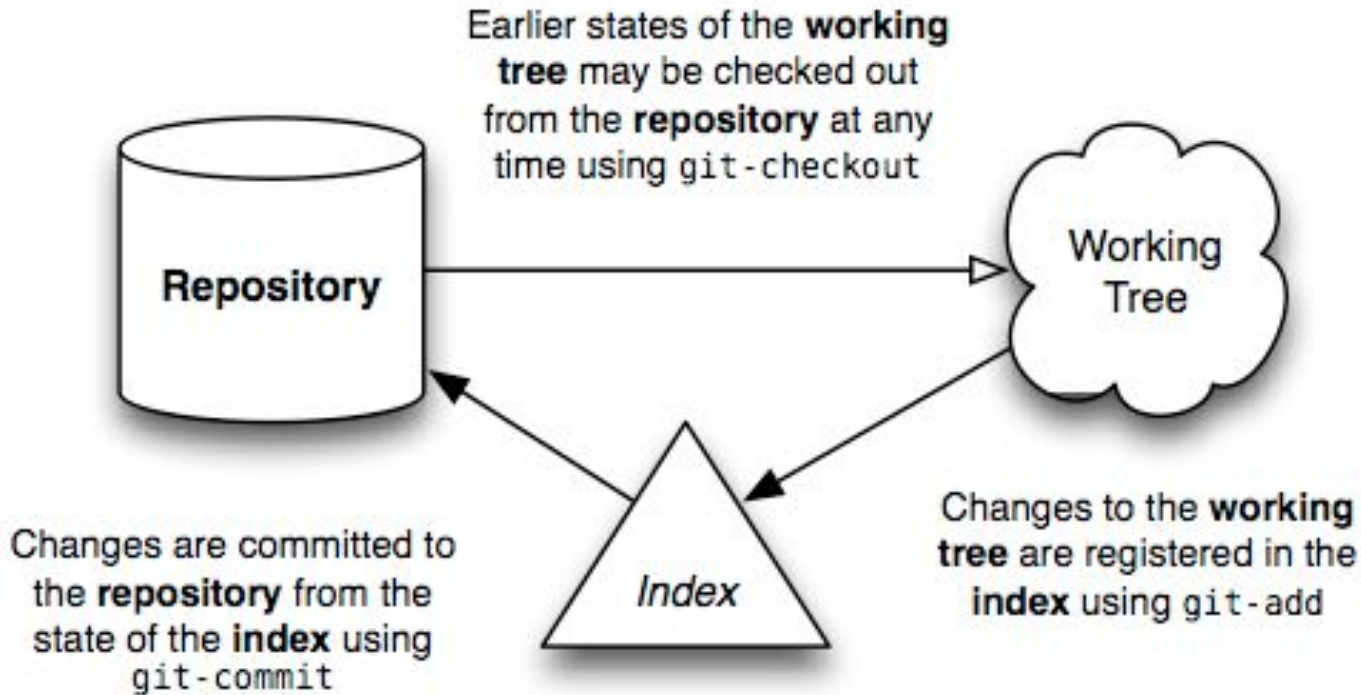


Three Sections of Git

- This corresponds to the three main sections of a Git project:
 - The **working tree** is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.
 - The **staging area** is a file, generally contained in your Git directory, that stores information about what will go into your next commit. Its technical name in Git parlance is the **"index"**, but the phrase "staging area" works just as well.
 - The **Git directory** is where Git stores the **metadata** and **object database** for your project. This is the most important part of Git, and **it is what is copied when you clone a repository** from another computer.

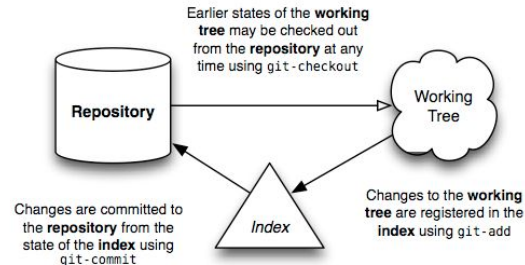


Git LifeCycle



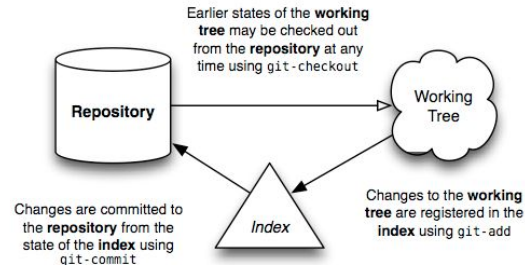
Git LifeCycle

- You modify files in your *working tree*. (If a file was changed since it was *checked out* but has not been staged, it is *modified*.)
- You selectively *stage* just those changes you want to be part of your next *commit*, which adds only those changes to the *staging area*. (If it has been modified and was added to the *staging area*, it is *staged*.)
- You do a *commit*, which takes the files as they are in the *staging area* and stores that *snapshot* permanently to your *Git directory*. (If a particular version of a file is in the *Git directory*, it's considered *committed*.)



Git LifeCycle

- The life-cycle of a file change goes through the following stages:
 - **Untracked:** Files are not ready to be part of your repository,
 - **Staging:** The file becomes part of your working tree. This means the file is ready to be committed,
 - **Committed:** The file changes are recorded in your git directory
 - **Pushed:** The file changes are synced with a remote source.



Git Concepts

Repository

A repository is a collection of commits, each of which is an archive of what the project's working tree looked like at a past date, whether on your machine or someone else's.

It also defines HEAD (see below), which identifies the branch or commit the current working tree stemmed from.

Lastly, it contains a set of branches and tags, to identify certain commits by name.

Clone

repositories can be cloned, that is you get a replica copy of the master branch

Branch

A branch is just a name for a commit, also called a reference. It's the parentage of a commit which defines its history, and thus the typical notion of a "branch of development".

A branch is a copy of the repository having exact replica of its parent.

Git Concepts

Head

A head is simply a reference to a commit object.

HEAD is used by your repository to define what is currently checked out:

If you checkout a branch, HEAD symbolically refers to that branch, indicating that the branch name should be updated after the next commit operation.

If you checkout a specific commit, HEAD refers to that commit only. This is referred to as a *detached* HEAD, and occurs, for example, if you check out a tag name.

Each head has a name. By default, there is a head in every repository called *master*. A repository can contain any number of heads. At any given time, one head is selected as the “current head.” This head is aliased to HEAD, (always in capitals) of the master branch

Git Concepts

Commit

A commit is a snapshot of your working tree at some point in time. The state of HEAD at the time your commit is made becomes that commit's parent. This is what creates the notion of a "revision history".



A commit object contains three things

- A set of files, reflecting the state of a project at a given point in time.
- References to parent commit objects.
- An SHA1 name, a 40-character string that uniquely identifies the commit object. The name is composed of a hash of relevant aspects of the commit, so identical commits will always have the same name.

Git Concepts

Commit

Every git commit has an id like 3f29abcd233fa, also called a SHA ("Secure Hash Algorithm"). A SHA refers to both:

-  the changes that were made in that commit ← see them with 'git show'
-  a snapshot of the code after that commit was made

No matter how many weird things you do with git, checking out a SHA will always give you the exact same code. It's like saving your game so that you can go back if you die 😊

You can check out a commit like this:

```
git checkout 3f29ab
```

SHAs are long
but you can
just use the
first 6 chars

Git Concepts

Index / Staging Area

You tell git to start tracking a file by moving it to staging area.

This means, you use commands like ``git add <file>`` to consider the file as a part of your local repository (before that, git is not aware of its existence), and therefore to track when it gets changed.

All files in the staging area will be added to the commit.

You can move files in and out of the staging area, to decide whether they should be included in a commit.

Working Tree

A working tree is any directory on your filesystem which has a repository associated with it (typically indicated by the presence of a sub-directory within it named `.git`).

It includes all the files and sub-directories in that directory.

Git Concepts

Remote

if you are cloning repo from a remote source, it will come with remote branch linked to your repo.

To have multiple remote means that you can have same repository that can push code to multiple remote sources. E.g: You pushed/pulled code to/from your team as well as to/from your production.

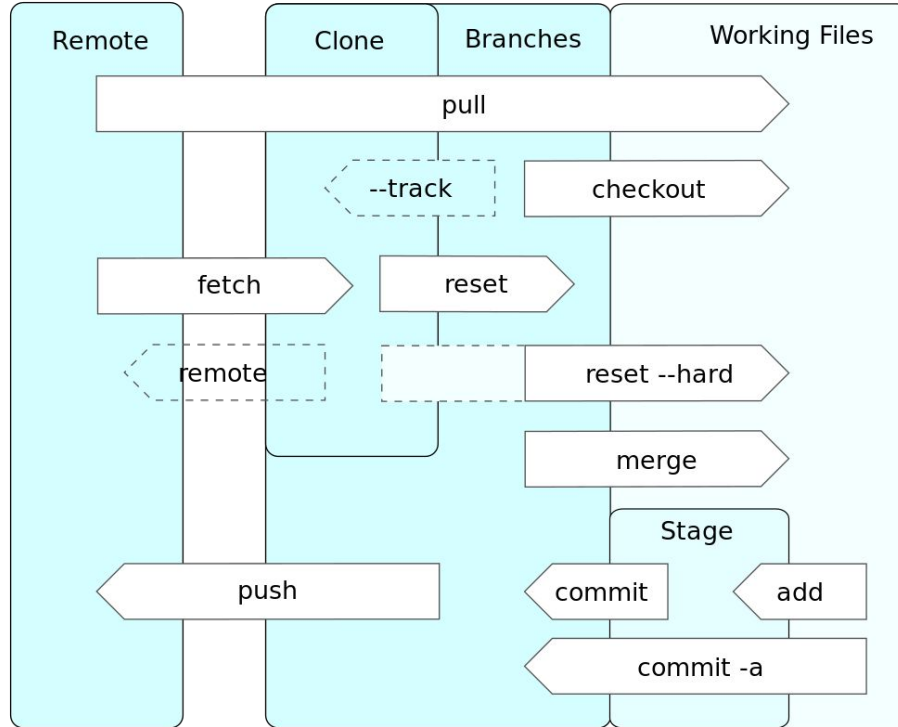
Pushing

Once you attach a remote source, you are getting capability to sync (pulling & pushing) your repository with the remote source.

Master

The mainline of development in most repositories is done on a branch typically named "**master**".

Git Flow Revisited





Working with Branches



Git Concepts

Branches

A branch is a pointer
to a commit

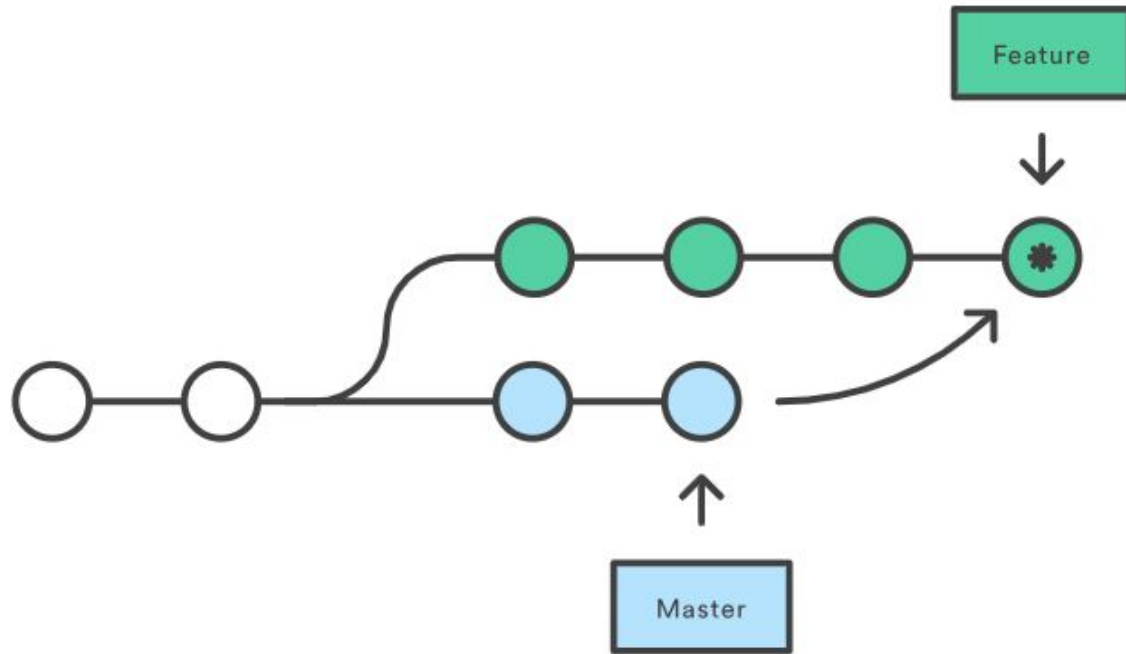
A branch in git is a pointer to a commit SHA

master \longrightarrow 2e9fab

awesome-feature \longrightarrow 3bafea

fix-typo \longrightarrow 9a9a9a

Merge

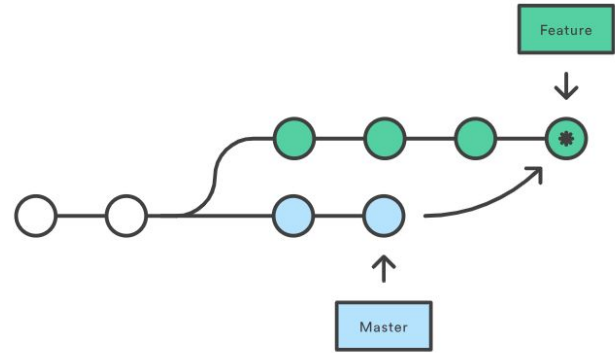


Merge

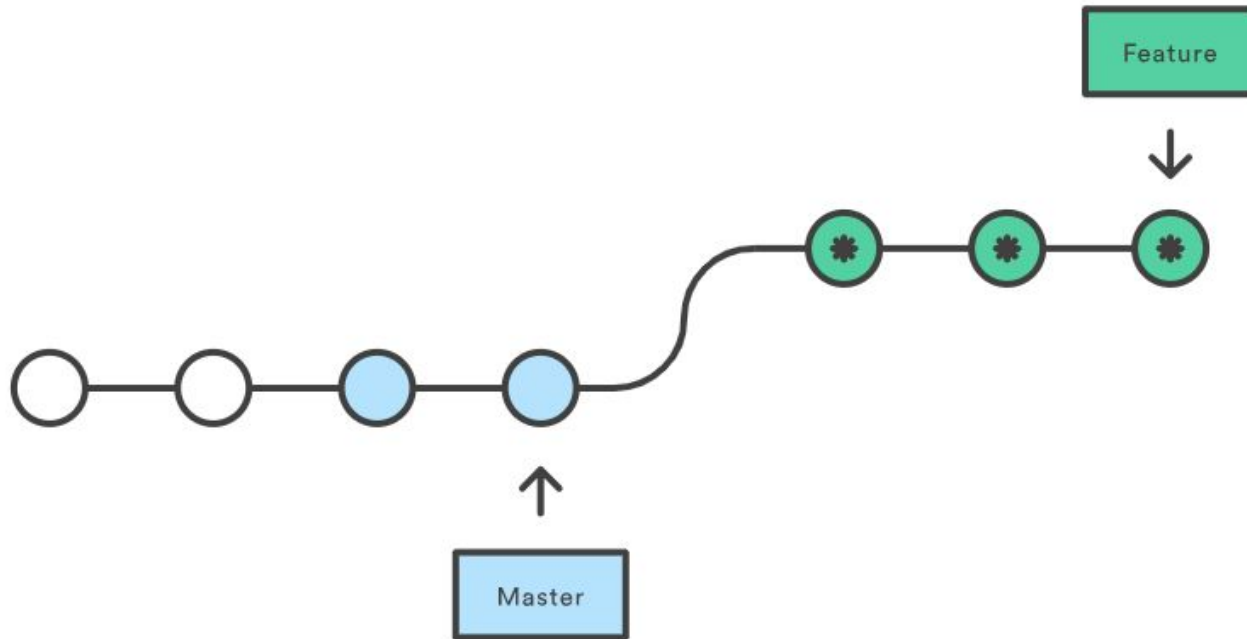
Merging takes the contents of a source branch and integrates them with a target branch. In this process, only the target branch is changed. The source branch history remains the same.

Merging is a common practice for developers using version control systems. Whether branches are created for testing, bug fixes, or other reasons, merging commits changes to another location.

Generally we merge feature branch into master



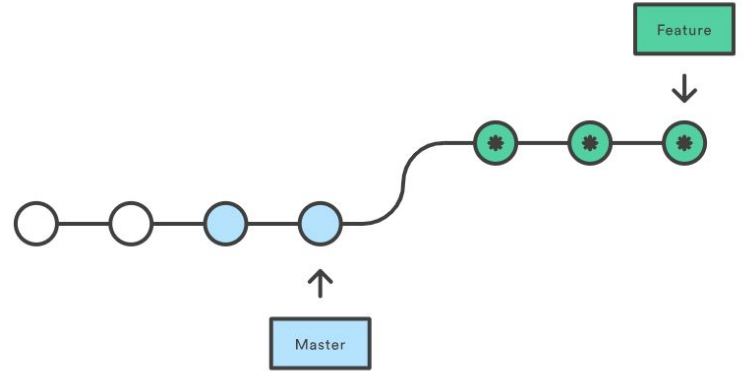
Rebase



Rebase

We pluck commits from requested branch and add it to the beginning or base of the current branch.

Generally we rebase the feature branch with commits of the master branch so that those commits are included in the feature branch.



Exercise

Some more exercises: [git immersion](#), [git exercises](#), [learn git branching](#)



How to use Git in your work



THIS IS GIT. IT TRACKS COLLABORATIVE WORK
ON PROJECTS THROUGH A BEAUTIFUL
DISTRIBUTED GRAPH THEORY TREE MODEL.

COOL. HOW DO WE USE IT?

NO IDEA. JUST MEMORIZE THESE SHELL
COMMANDS AND TYPE THEM TO SYNC UP.
IF YOU GET ERRORS, SAVE YOUR WORK
ELSEWHERE, DELETE THE PROJECT,
AND DOWNLOAD A FRESH COPY.



Git Commands

Create a Repository

From scratch -- Create a new local repository

```
$ git init [project name]
```

Download from an existing repository

```
$ git clone my_url
```

Observe a Repository

List new or modified files not yet committed

```
$ git status
```

Show the changes to files not yet staged

```
$ git diff
```

Show the changes to staged files

```
$ git diff --cached
```

Show all staged and unstaged file changes

```
$ git diff HEAD
```

Show the changes between two commit ids

```
$ git diff commit1 commit2
```

List the change dates and authors for a file

```
$ git blame [file]
```

Show the file changes for a commit id and/or file

```
$ git show [commit]:[file]
```

Show full change history

```
$ git log
```

Show change history for file/directory including diffs

```
$ git log -p [file/directory]
```

Working With Branches

List all local branches

```
$ git branch
```

List all branches, local and remote

```
$ git branch -av
```

Switch to a branch, my_branch, and update working directory

```
$ git checkout my_branch
```

Create a new branch called new_branch

```
$ git branch new_branch
```

Delete the branch called my_branch

```
$ git branch -d my_branch
```

Merge branch_a into branch_b

```
$ git checkout branch_b
```

```
$ git merge branch_a
```

Tag the current commit

```
$ git tag my_tag
```

Make a Change

Stages the file, ready for commit

```
$ git add [file]
```

Stage all changed files, ready for commit

```
$ git add .
```

Commit all staged files to versioned history

```
$ git commit -m "commit message"
```

Commit all your tracked files to versioned history

```
$ git commit -am "commit message"
```

Unstages file, keeping the file changes

```
$ git reset [file]
```

Revert everything to the last commit

```
$ git reset --hard
```

Synchronize

Get the latest changes from origin (no merge)

```
$ git fetch
```

Fetch the latest changes from origin and merge

```
$ git pull
```

Fetch the latest changes from origin and rebase

```
$ git pull --rebase
```

Push local changes to the origin

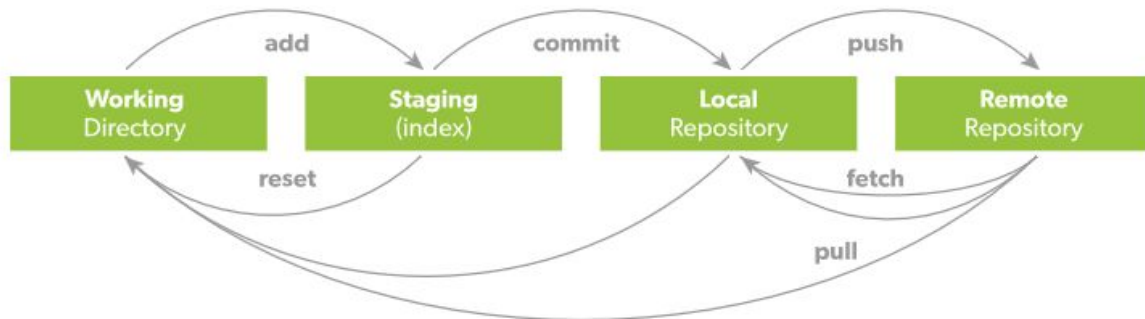
```
$ git push
```

Finally!

When in doubt, use git help

```
$ git [command] --help
```

Or visit training.github.com for official GitHub training.



Bad News

You have to know the

Frequently used commands,

What they do, and

How they map to the git flow

Good News

You dont have to type every time. There's apps for that.



Start Using Git

- Download the [latest](#) version of git
- Once installed, select Launch the Git Bash, then click on finish. The Git Bash is now launched.
- Check that git is successfully installed by typing "git --version" which should output the installed version of git.
- Configure your username and email for git.
 - git config --global user.name "Anshuman Sanghvi"
 - git config --global user.email "anshuman.sanghvi@silvertouch.com"
- Optional: [Install](#) the Git credential helper on Windows which automatically enters your credentials instead you having to type it every time.

Demo

- Initiating a repo
- Cloning a repo
- Adding commits
- Creating branches
- Merge
- Rebase
- Adding remotes
- Pull & Push

General Practices

- main branch should always be releasable
- all features have their own branches
- your branch should have small units of work as commits. (commit often)
- each commit should have a meaningful message of which overall feature does it belong to, what it does, will it have side effects, etc.
- developers make a pull request to merge feature branch with the main branch, once the feature or its milestone is ready.

General Practices

- at the pull request, your team members can review your code, and make suggestions.
- once you have fixed issues or worked on suggestions, your team members may approve your branch
- your branch can then be merged.
- some projects may have internal and external repos.

Working on a task

1. Create a (local) branch from latest origin master related to the ticket/task/bugfix
2. Create commits for logical units of work. Generally we create a single commit for a ticket.
 - 2.1 One commit per subtask if ticket is large and is split into logical subtask
3. Once commits are pushed to local branch (after due processes)
4. Push the commit to origin branch. (if pushing first time, then new branch will be created)
5. Raise internal PR of your origin feature branch against origin master.
6. Once PR comments are reviewed and changes are made, (squash the commit for the changes)

Working on a task

7. Internal PR get approved.
8. Push the commit to upstream feature branch. (if pushing first time, then new branch will be created)
9. Raise External PR raised for your upstream feature branch against upstream master branch
10. External PR review comments - address the comments (squash commit)
11. External PR gets approved.
12. Merge the branch against upstream master. (select squash commit, select yes to delete branch once merged)
13. Deploy the new repository changes to the EXP environment.

References

- [visual explanation of git concepts](#)
- [visualizing branches](#)
- [definitive book on git](#)
- [for remembering commands](#)
- [learning git by commands](#)
- [Stackoverflow: Why is Git better than SVN](#)
- [Quora: Difference between Git and SVN](#)
- [StackExchange: Differences between Git and SVN branches](#)
- [StackOverflow: How to merge commits into one?](#)
- [StackOverflow: merge commits after already starting rebase](#)
- [Merge vs Rebase](#)
- [Oh Shit, Git!?! \(How to undo most mistakes in git\)](#)
- [Git Magic \(Book on Git\)](#)

Thank you

Anshuman Sanghvi
Project Lead, Java
STTL