Git is a distributed revision control and source code management system with an emphasis on speed. Git was initially designed and developed by Linus Torvalds for Linux kernel development.

Initial Goals:

- **Speed**
- **Support for non-linear development (thousands of parallel branches)**
- **Fully distributed**
- **Able to handle large projects like Linux efficiently**

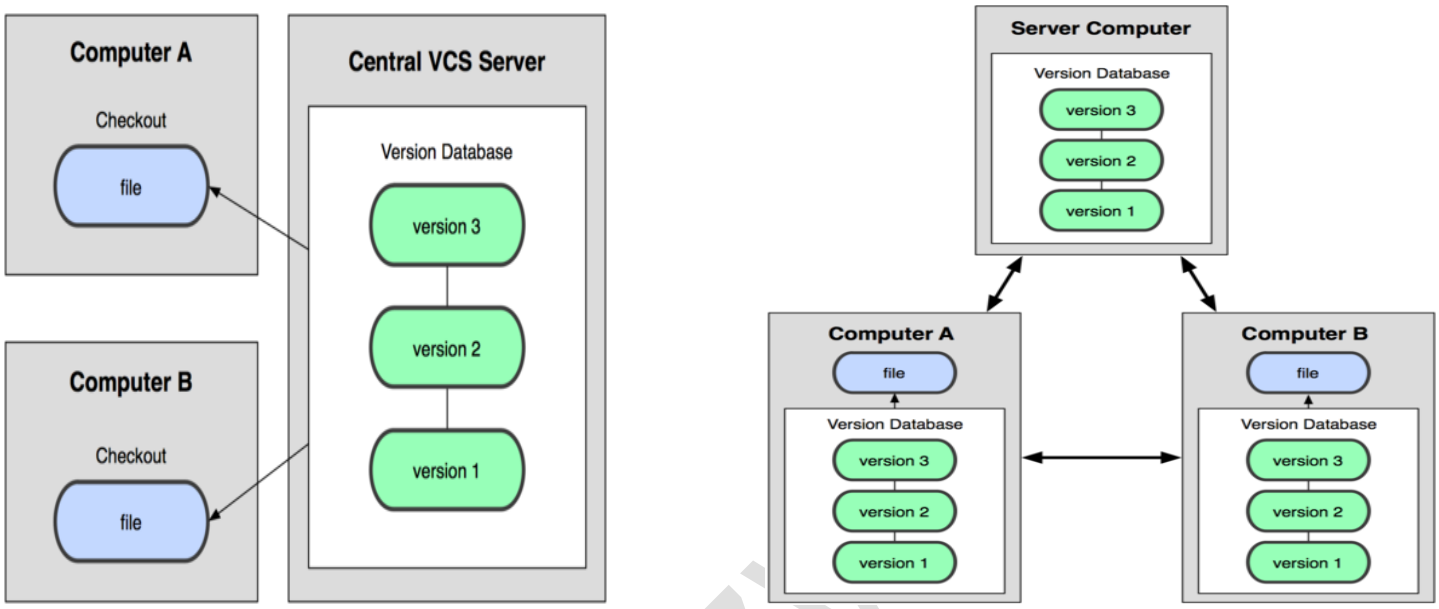
(SCM) Software configuration management is a process of tracking and controlling changes in the software

The goals of SCM are generally:

- **Teamwork**
- **Defect tracking**
- **Configuration auditing**
- **Process management**

Version Control System (VCS) is a software that helps to implement SCM in software development lifecycle.

VCS can be classified into two types:



- **Centralized version control system (CVCS)**

  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. And even in a worst case, if the disk of the central server gets corrupted and proper backup has not been taken, then you will lose the entire history of the project.

- **Distributed/Decentralized version control system (DVCS)**

  There is no server in DVCS and everything is a workspace. Every workspace will have a complete copy of the actual repository. If the central server/repository goes down, then the repository from any client can be copied back to the server to restore it.

  Git 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.

SCM Terminologies

Server: Server is a machine which holds the collection of all the changes for every product. Server contains many repositories in it.

Repository: Folder or location under server where all the changes specific to project is stored.

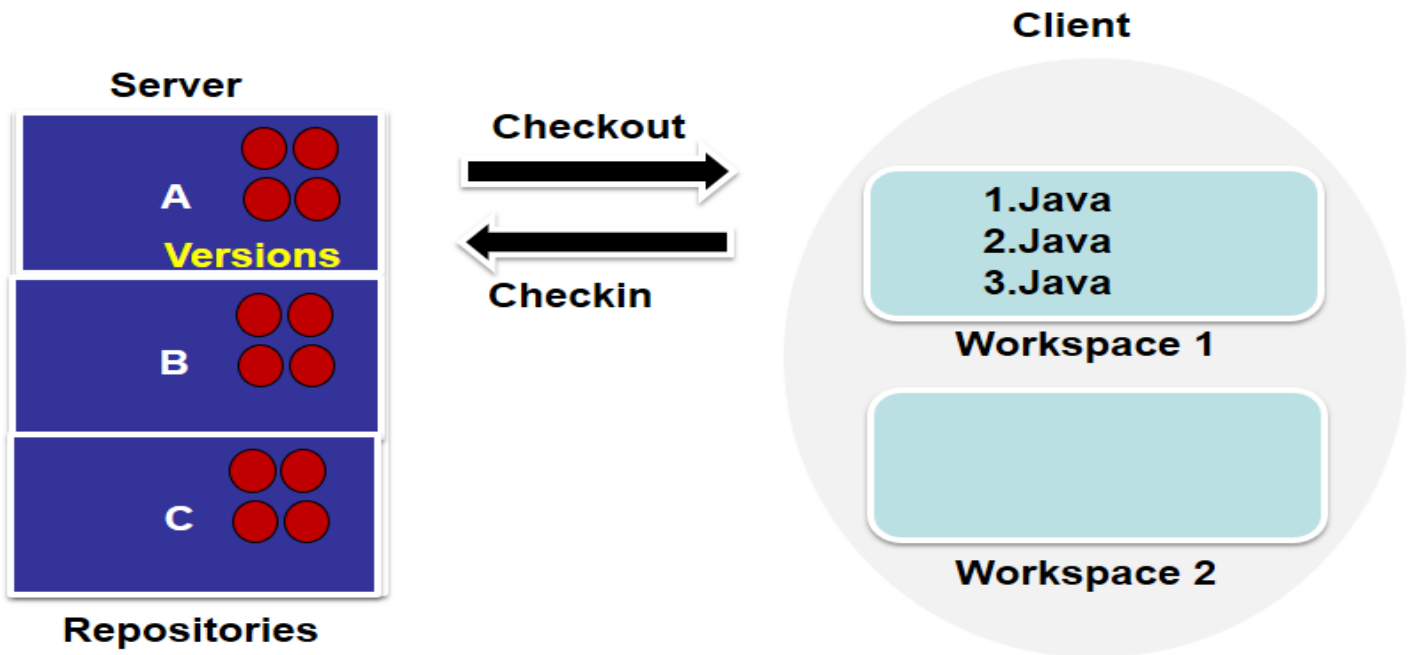
Client: Client is a software which connects to the server assisting users to perform necessary actions

Workspace: Is a folder in your local machine where we you see the files respective to a repository in the server, modify them

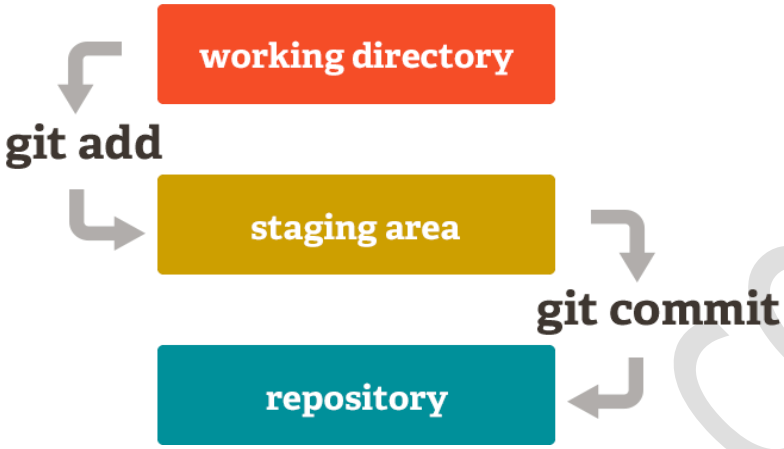
Checkout: An action which helps us to fetch the files from the repository and displays in the workspace, which are later modified

Checkin: An action where the changes made in the workspace are stored back in the repository

Versions: References created in the repository for every checkin, which holds the transaction details of who modified, what was modified and when was it modified



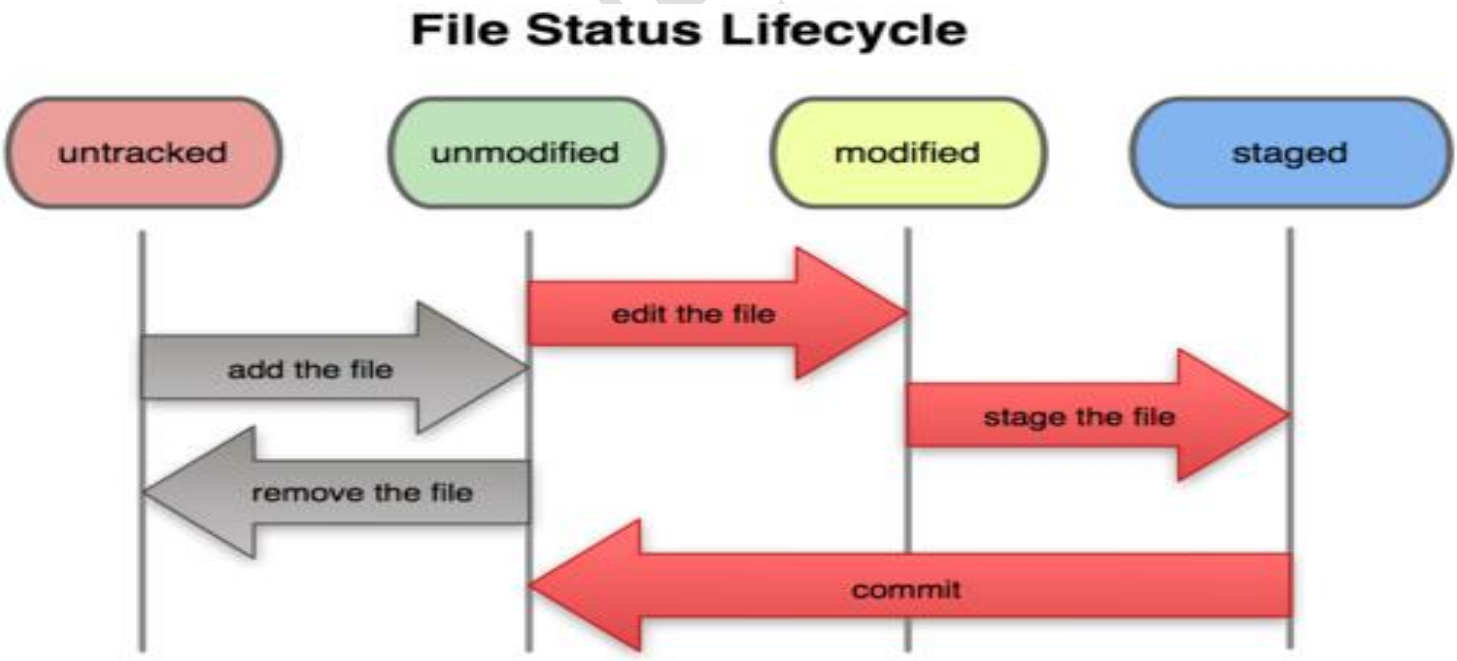
Git - local workspace has three areas:



Working directory is the folder where we see the files and modify them.

The changes are then pushed to Staging area where snapshots would be taken.

We then perform commit operation that moves the contents from the staging area and stores it into the repository by creating a commitID

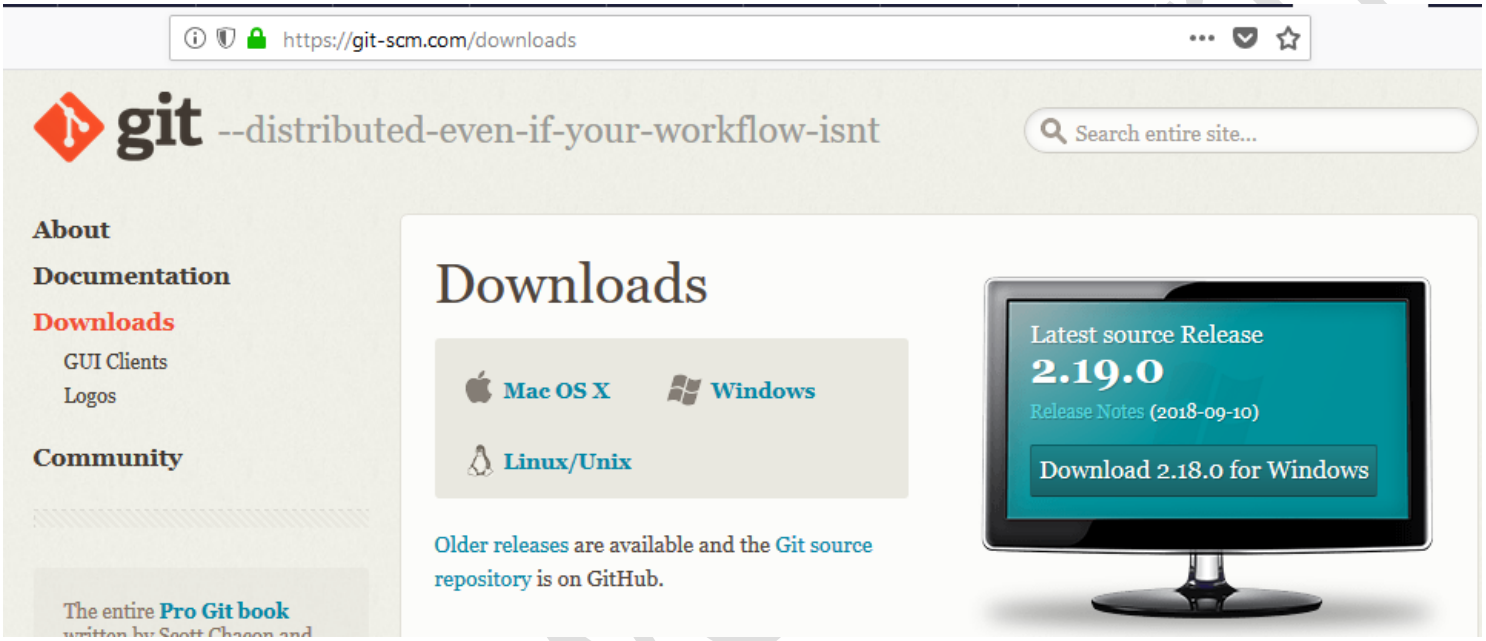


All new files are considered untracked, any files under the repository are called tracked files. However files status would be unmodified if it's a clone workspace and changes are done. Status would be modified if files are modified compared to repository content. Once the changes are

moved to staging area the file status will be staged. When these changes are checkedin the file status would be unmodified.

Git Installation

Goto https://git-scm.com/downloads and choose appropriate platform to install Git.



Installing Git on Windows

Download exe from the above mentioned page and install it on windows.

Installing Git on Linux

For Ubuntu/Debian OS, run the following as 'root'

```
$ apt install git
```

For Centos/Fedora OS

```
$ yum install git
```

Get ready to use Git on your local machine

Set the user configuration i.e name and email for Git to use when you commit.

- ✓ **Open gitbash command prompt and run the following cmds:**

```
$ git config --global user.name "Adam M"

$ git config --global user.email mailme@wezva.com

$ git config --global push.default simple

$ git config credential.helper store
```

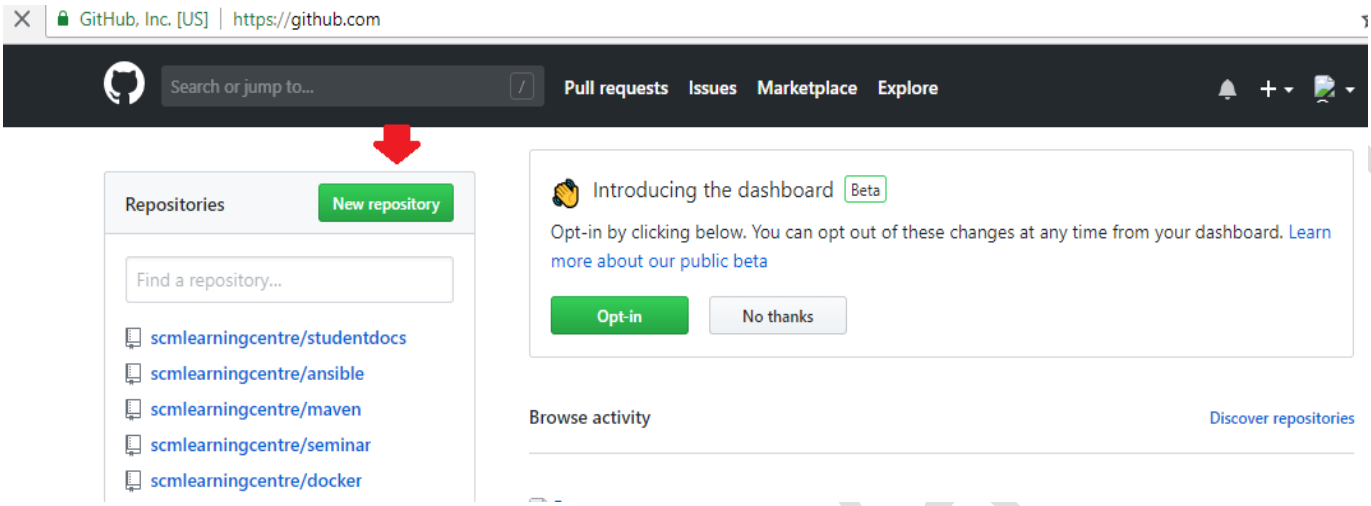
- ✓ **These will be set globally for all Git projects you work with in your machine**
- ✓ **You can also set variables on a project-only basis by not using the --global flag**

- ✓ **You can call 'git config --list' to verify these are set.**

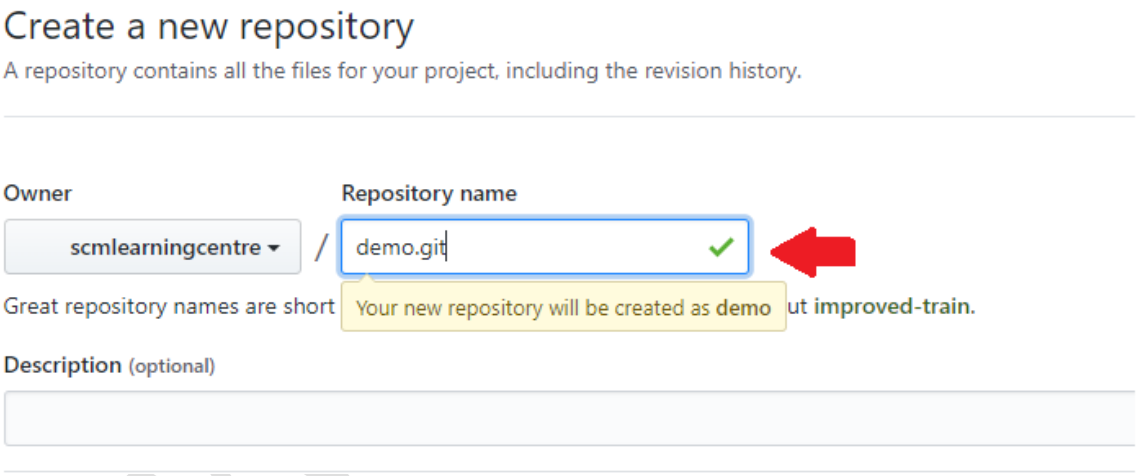
```
$ git config --list
core.symlinks=false
core.autocrlf=true
core.fscache=true
color.diff=auto
color.status=auto
color.branch=auto
color.interactive=true
help.format=html
rebase.autosquash=true
http.sslcainfo=C:/Program Files/Git/mingw64/ssl/certs/ca-bundle.crt
http.sslbackend=openssl
diff.astextplain.textconv=astextplain
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
credential.helper=manager
user.name=Adam M
user.email=mailme@wezva.com
```

Setting up Remote/Central repository using Github

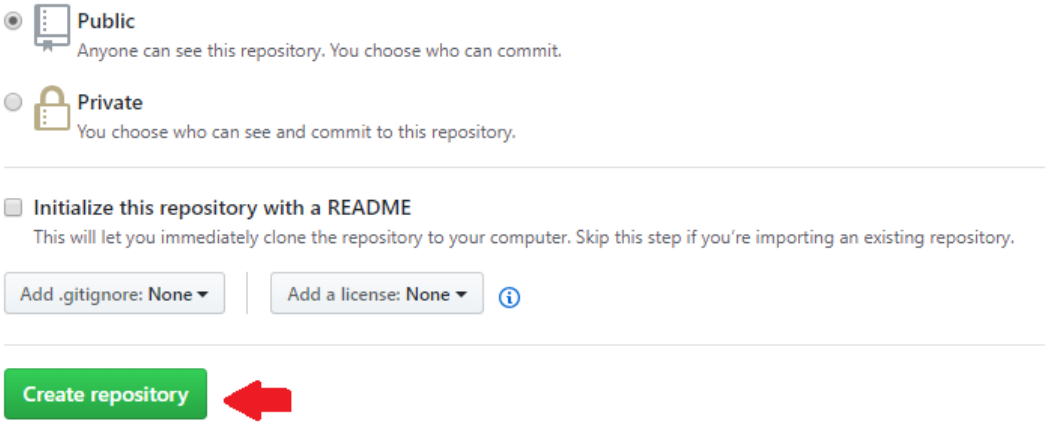
- ✓ Signup for a Github account, if you don't already have one at https://github.com/
- ✓ Click on the "New repository" button on the left side of the page



- ✓ Provide the necessary Repository name and description of the project



- ✓ Choose the project type and create repository



Setting up Remote/Central repository locally (Bare repository)

Bare repository contains the version control information and no working files. By convention the name of a bare repository should end with the .git extension

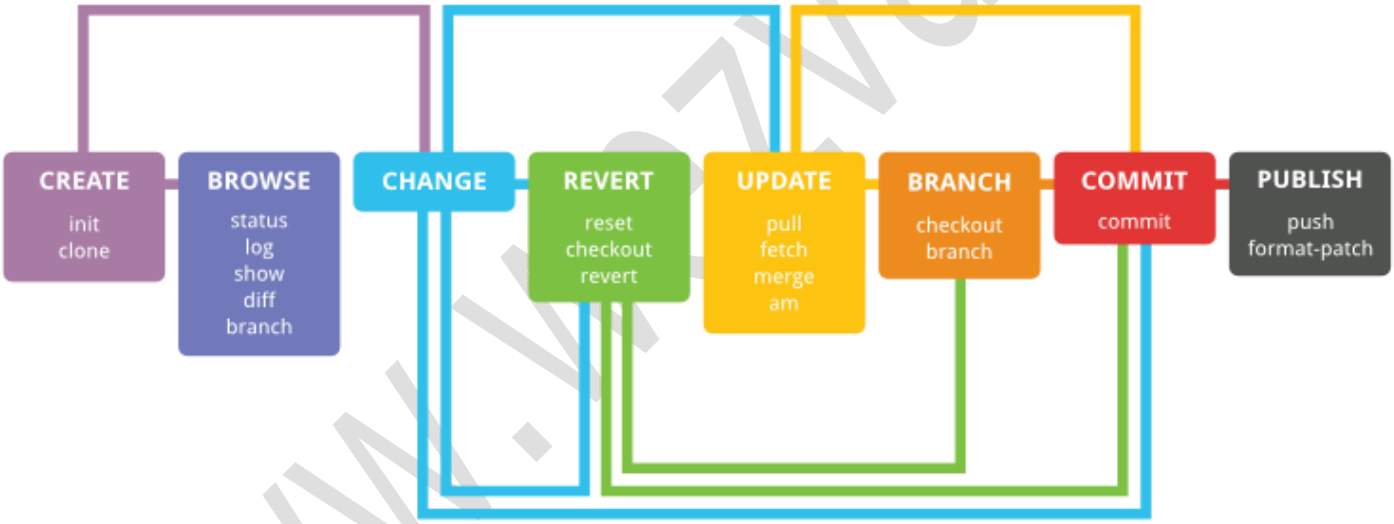
Create a new dir & run the following from it:

$ mkdir demo.git

$ cd demo.git

$ git init --bare

Git - lifecycle:



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 developers' 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.**

Git User Workflow

- Create local workspace

$ git clone <CentralRepoLocation> <LocalWorkspace>

Once the workspace is created, cd to the dir & make changes

- Show the list of all new files or modified files

$ git status

- Show the file differences of all the files which are not yet staged

$ git diff

- Add changes to the staging area

$ git add <filename> or

$ git add .

- Show the list of file differences between staging & the last file revision

$ git diff -staged

- Checkin the changes permanently to repository

$ git commit -m "<comment on why the checkin is needed>"

- Show the list of all the commit id's from latest to oldest order

$ git log # lists all the commits in full format

$ git log --oneline # lists all the commits in short format

$ git log -oneline -<num> # lists only the specified number of commit ids

- Share the modified commits from local to central repository

$ git push

- Update local workspace with the new changes available on the central repository

$ git pull

- Show the details about the commit id

$ git show <commitID>

Git Branches

A branch represents an independent line of development.

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 initially make commits, you're given a master branch that points to the last commit you made. Every time you commit, it moves forward automatically.

- Show the list of all the branches available

$ git branch

$ git branch -a # lists the branches which are in remote repository

- Create a new branch

$ git branch <NewBranchName>

- Swtich to a particular branch

$ git checkout <BranchName>

- Show difference between branches

$ git diff <Branch1> <Branch2>

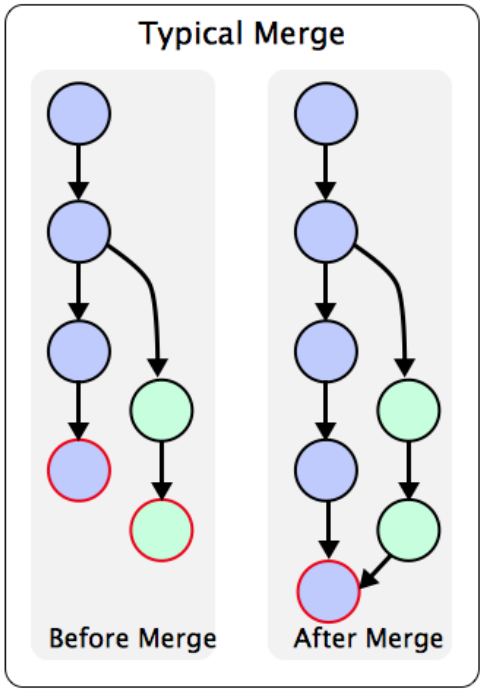
Branch Merge

Git merge will combine multiple sequences of commits into one unified history. Git merge will take the commits from a source branch and update the target branch by integrating its existing commits.

While merging branches, it's a best practice to be on the destination branch.

- Merge 2 branches

$ git diff <SourceBranch> <Desination Branch>



By default merge takes all the missing commits from source & updates the destination branch

Git Cherry-Pick is used to merge only a specific commit ID from source to destination branch

- Merge only a particular commit (Git Cherry-pick)

$ git cherry-pick <CommitID>

Resolving merge conflicts

If the two branches you're trying to merge both changed the same part of the same file, Git won't be able to figure out which version to use. When such a situation occurs, it stops right before the merge commit so that you can resolve the conflicts manually.

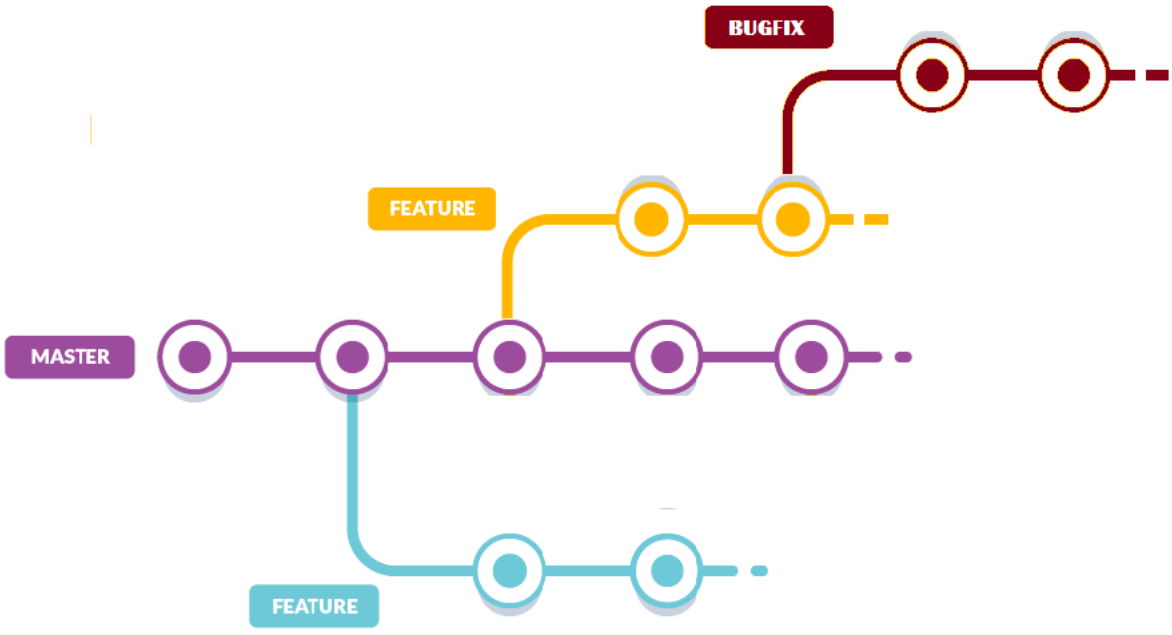
When Git encounters a conflict during a merge, It will edit the content of the affected files with visual indicators that mark both sides of the conflicted content. These visual markers are: <<<<<<<, =======, and >>>>>>>. Its helpful to search a project for these indicators during a merge to find where conflicts need to be resolved.

```
here is some content not affected by the conflict
<<<<<<< master
this is conflicted text from master
=======
this is conflicted text from feature branch
```

Branching Methodology

The Feature Branch Workflow assumes a central repository, and master represents the official project history. Instead of committing directly on their local master branch, developers create a new branch every time they start work on a new feature. Feature branches should have descriptive names, like animated-menu-items or issue-#1061. The idea is to give a clear, highly-focused purpose to each branch. Git makes no technical distinction between the master branch and feature branches, so developers can edit, stage, and commit changes to a feature branch.

In addition, feature branches can (and should) be pushed to the central repository. This makes it possible to share a feature with other developers without touching any official code



Stashing committed changes

git stash temporarily shelves (or stashes) changes you've made to your working copy so you can work on something else, and then come back and re-apply them later on.

Stashing is handy if you need to quickly switch context and work on something else, but you're mid-way through a code change and aren't quite ready to commit.

Stashing will record untracked files.

- **Create stash – take a backup & undo the change**

$ git stash

- **List all the available backup for the repository**

$ git stash list

- **Apply a specific stash from the repository**

$ git stash apply stash@{num}

- **Remove all the stash from the repository**

$ git stash clear

Undo uncommitted changes

Undoing changes doesn't just happen from only the files, since Git has 3 working areas the changes has to be reverted from all the 3 areas respectively.

Before committing any change if we need to undo the changes then use reset.

Syntax:

$ git reset <file> --[mode]

Where mode is "soft/mixed/hard"

- **Undo changes from the repository & move reference of HEAD pointer**

$ git reset --soft

- Undo changes from the repository i.e move HEAD pointer & staging area

$ git reset --mixed

- Undo changes from the repository i.e move HEAD pointer & staging area & working dir

$ git reset --hard

Revert committed changes

The git revert command can be considered an 'undo' type command, however, it is not a traditional undo operation. Instead of removing the commit from the project history, it figures out how to invert the changes introduced by the commit and appends a new commit with the resulting inverse content.

This prevents Git from losing history, which is important for the integrity of your revision history and for reliable collaboration.

A revert operation will take the specified commit, inverse the changes from that commit, and create a new "revert commit". The ref pointers are then updated to point at the new revert commit making it the tip of the branch.

$ git revert <commitID>

Traverse to an older commit

Whenever there is a need to go back to a particular commit ID & refer to the changes, then use

$ git checkout <CommitID>

- Deleting a file

$ git rm <file>

$ git commit -m “commit message”

Ignoring certain files

Git by default considers all the files under the working dir & an gitignore file specifies intentionally untracked files that Git should ignore. Files already tracked by Git are not affected; see the NOTES below for details.

Each line in a gitignore file specifies a pattern. When deciding whether to ignore a path, Git normally checks gitignore patterns from multiple sources, with the following order of precedence, from highest

Git never ignores files which are already tracked, so changes in the .gitignore file only affect new files. Commit the .gitignore to the Git repository as a best practice.

Tags

Git has the option to *tag* a commit in the repository history so that you find it easier at a later point in time.

A single commitID can have many tags, whereas a tagname cannot be attached to multiple commits.

- Create & Apply a tag

```
$ git tag -a <pattern> -m 'comment' <CommitID>
```

- Show the contents of the tag

```
$ git show <tagname>
```

- List all the available tags

```
$ git tag
```

- Delete a tag

```
$ git tag -d <tagname>
```