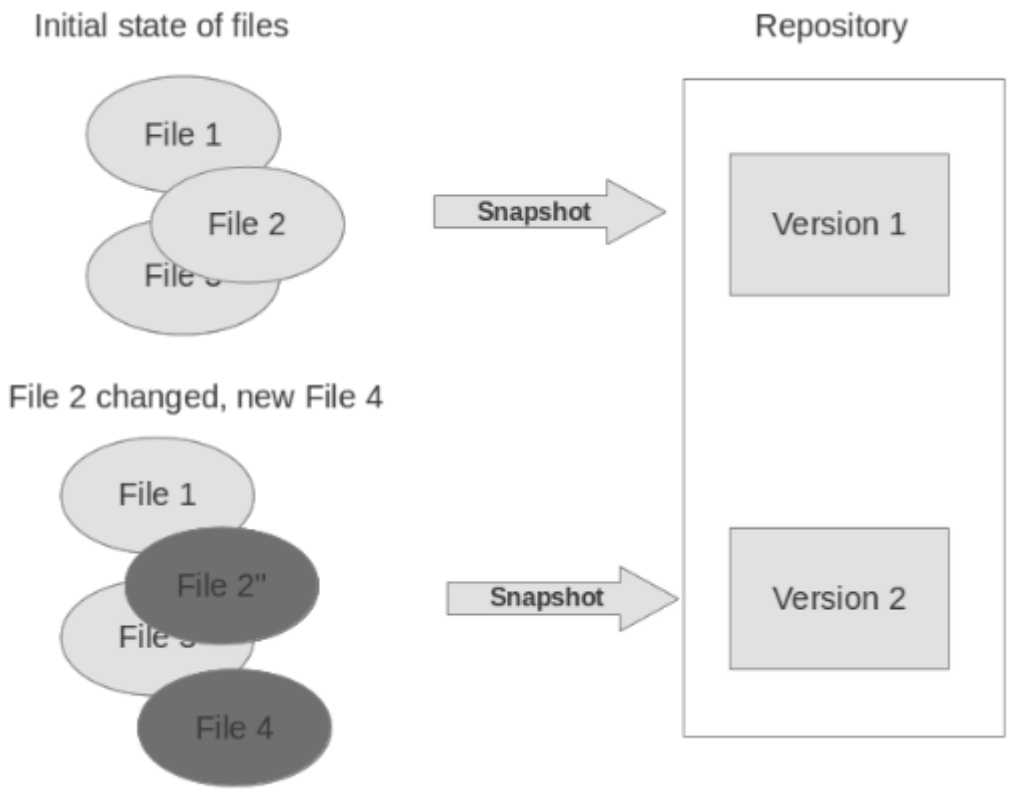
GIT REPOSITORY

Version Control Systems

Introduction VCS

Whatever model developers follow to create softwares, there is going to be lot of code that developers write or integrate in their softwares. All this code from different developers in the team has to be merged at a centralised place, which can keep track of all the versions of their code, maintain the code and even revert back in time if anything breaks.

Version control System(or revision control, or source control) is all about managing multiple versions of documents, programs, web sites, etc. A version control system (VCS) allows you to track the history of a collection of files.



Version control systems are a category of software tools that help a software team manage changes to source code over time. Version control software keeps track of every modification to the code in a special kind of database. If a mistake is made, developers can turn back the clock and compare earlier versions of the code to help fix the mistake while minimizing disruption to all team members.

Version control protects source code from both catastrophe and the casual degradation of human error and unintended consequences.

Software developers working in teams are continually writing new source code and changing existing source code. The code for a project, app or software component is typically organized in a folder structure or "file tree". One developer on the team may be working on a new feature while another developer fixes an unrelated bug by changing code, each developer may make their changes in several parts of the file tree.

When To Use VCS

Have you ever:

1. Made a change to code, realised it was a mistake and wanted to revert back?
2. Lost code or had a backup that was too old?
3. Had to maintain multiple versions of a product?
4. Wanted to see the difference between two (or more) versions of your code?
5. Wanted to prove that a particular change broke or fixed a piece of code?
6. Wanted to review the history of some code?
7. Wanted to submit a change to someone else's code?
8. Wanted to share your code, or let other people work on your code?
9. Wanted to see how much work is being done, and where, when and by whom?
10. 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.

VCS Terminologies

Basic Setup

1. Repository (repo): The database storing the files.
2. Server: The computer storing the repo.
3. Client: The computer connecting to the repo.
4. Working Set/Working Copy: Your local directory of files, where you make changes.
5. Trunk/Main: The primary location for code in the repo. Think of code as a family tree — the trunk is the main line.

Basic Actions

1. Add/Push: Put a file into the repo for the first time, i.e. begin tracking it with Version Control.
2. Revision: What version a file is on (v1, v2, v3, etc.).
3. Head: The latest revision in the repo.
4. Check out/Pull/Fetch: Download a file from the repo.
5. Check in/Push: Upload a file to the repository (if it has changed). The file gets a new revision number, and people can “check out” the latest one.

6. Check In Message: A short message describing what was changed.
7. Changelog/History: A list of changes made to a file since it was created.
8. Update/Sync: Synchronize your files with the latest from the repository. This lets you grab the latest revisions of all files.
9. Revert: Throw away your local changes and reload the latest version from the repository.

Advanced Actions

1. Branch: Create a separate copy of a file/folder for private use (bug fixing, testing, etc). Branch is both a verb ("branch the code") and a noun ("Which branch is it in?").
2. Diff/Change/Delta: Finding the differences between two files. Useful for seeing what changed between revisions.
3. Merge (or patch): Apply the changes from one file to another, to bring it up-to-date. For example, you can merge features from one branch into another. (At Microsoft, this was called Reverse Integrate and Forward Integrate)
4. Conflict: When pending changes to a file contradict each other (both changes cannot be applied).
5. Resolve: Fixing the changes that contradict each other and checking in the correct version.
6. Locking: Taking control of a file so nobody else can edit it until you unlock it. Some version control systems use this to avoid conflicts.

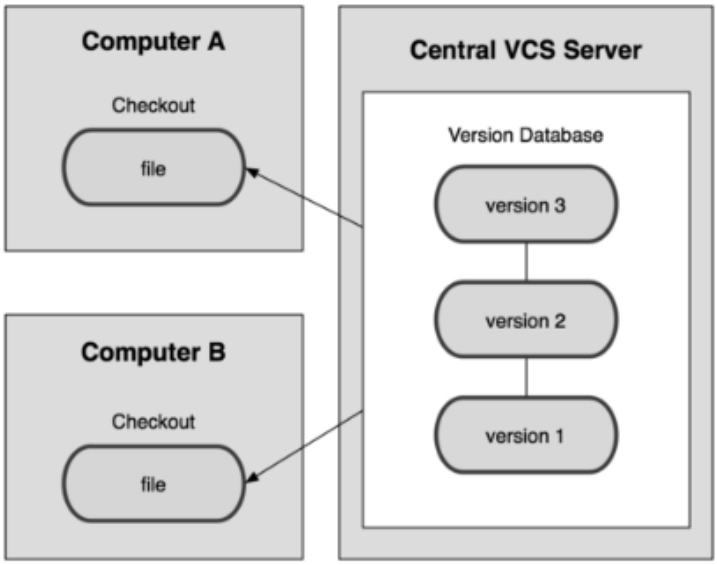
Types of version control

Localized Version Control Systems

A localized version control system keeps local copies of the files. This approach can be as simple as creating a manual copy of the relevant files.

Centralized Version Control Systems

A centralized version control system provides a server software component which stores and manages the different versions of the files. A developer can copy (checkout) a certain version from the central sever onto their individual computer. Eg: CVS,SVN etc.



• In Subversion, CVS, Perforce, etc.A central server repository (repo) holds the "official copy" of the code.

– the server maintains the sole version history of the repo

• You make "checkouts" of it to your local copy

– you make local modifications

– your changes are not versioned

• When you're done, you

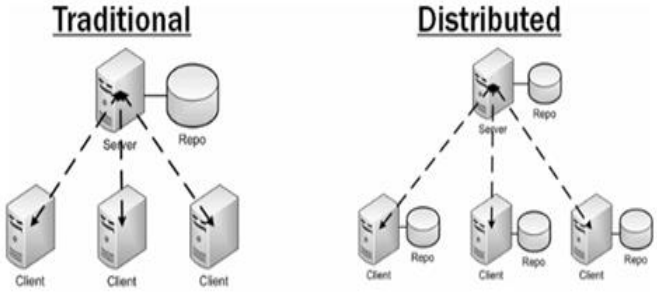
"check in" back to the server

– your checkin increments the repo's version

Distributed Version Control Systems

In a distributed version control system each user has a complete local copy of a repository on his individual computer.

Both approaches have the drawback that they have one single point of failure. In a localized version control systems it is the individual computer and in a centralized version control systems it is the server machine. Both system makes it also harder to work in parallel on different features. Eg:Git,mercurial etc.



FAMOUS VERSION CONTROL SYSTEMS.

CVS

CVS may very well be where version control systems started. Released initially in 1986, Google still hosts the original Usenet post that announced CVS. CVS is basically the standard here, and is used just about everywhere – however the base for codes is not as feature rich as other solutions such as SVN.

One good thing about CVS is that it is not too difficult to learn. It comes with a simple system that ensures revisions and files are kept updated. Given the other options, CVS may be regarded as an older form of technology, as it has been around for some time,

it is still incredibly useful for system admins who want to backup and share files.

SVN Or Subversion

SVN, or Subversion as it is sometimes called, is generally the version control system that has the widest adoption. Most forms of open-source projects will use Subversion because many other large products such as Ruby, Python Apache, and more use it too. Google Code even uses SVN as a way of exclusively distributing code.
Because it is so popular, many different clients for Subversion are available. If you use Windows, then Tortoisesvn may be a great browser for editing, viewing and modifying Subversion code bases. If you're using a MAC, however, then Versions could be your ideal client.

Git

Git is considered to be a newer, and faster emerging star when it comes to version control systems. First developed by the creator of Linux kernel, Linus Torvalds, Git has begun to take the community for web development and system administration by storm, offering a largely different form of control. Here, there is no singular centralized

code base that the code can be pulled from, and different branches are responsible for hosting different areas of the code. Other version control systems, such as CVS and SVN, use a centralized control, so that only one master copy of software is used.

As a fast and efficient system, many system administrators and open-source projects use Git to power their repositories. However, it is worth noting that Git is not as easy to learn as SVN or CVS is, which means that beginners may need to steer clear if they're not willing to invest time to learn the tool.

Mercurial

This is yet another form of version control system, similar to Git. It was designed initially as a source for larger development programs, often outside of the scope of most system admins, independent web developers and designers. However, this doesn't mean that smaller teams and individuals can't use it. Mercurial is a very fast and efficient application. The creators designed the software with performance as the core feature.

Aside from being very scalable, and incredibly fast, Mercurial is a far simpler system to use than things such as Git, which one of the reasons why certain system admins and developers use it. There aren't quite many things to learn, and the functions are less

complicated, and more comparable to other CVS systems. Mercurial also comes alongside a web-interface and various extensive documentation that can help you to understand it better.

Bazaar

Similar to Git and Mercurial, Bazaar is distributed version control system, which also provides a great, friendly user experience. Bazaar is unique that it can be deployed either with a central code base or as a distributed code base. It is the most versatile version control system that supports various different forms of workflow, from centralized to decentralized, and with a number of different variations acknowledged throughout.

One of the greatest features of Bazaar is that you can access a very detailed level of control in its setup. Bazaar can be used to fit in with almost any scenario and this is incredibly useful for most projects and admins because it is so easy to adapt and deal with. It can also be easily embedded into projects that already exist. At the same time, Bazaar boasts a large community that helps with the maintenance of third-party tools and plugins.

What is Git

By far, the most widely used modern version control system in the world today is Git. Git is a mature, actively maintained open source project originally developed in 2005 by Linus Torvalds, the famous creator of the Linux operating system kernel. A staggering number of software projects rely on Git for version control, including commercial projects as well as open source. Developers who have worked with Git are well represented in the pool of available software development talent and it works well on a wide range of operating systems and IDEs (Integrated Development Environments).

Having a distributed architecture, Git is an example of a DVCS (hence Distributed Version Control System). Rather than have only one single place for the full version history of the software as is common in once-popular version control systems like CVS or Subversion (also known as SVN), in Git, every developer's working copy of the code is also a repository that can contain the full history of all changes.

In addition to being distributed, Git has been designed with performance, security and flexibility in mind.

Why Use Git?

1. Its fast
2. You don't need access to a server
3. Amazingly good at merging simultaneous changes
4. Everyone’s using it

Install git tool on your system

Website: **https://git-scm.com/**

Windows:

Install git software

https://git-scm.com/download/win

Go to git scm download page, Select windows.

Open git installable and follow the Installation wizard, take all the default settings in the wizard.

Linux: -

```
# sudo apt-get install git (Ubuntu) or
# yum install git (Centos)
```

Local git areas & Basic Git workflow

Local Git Areas

In your local copy on git,

files can be:

– In your local repo

• (committed)

– Checked out and modified,

but not yet committed

• (working copy)

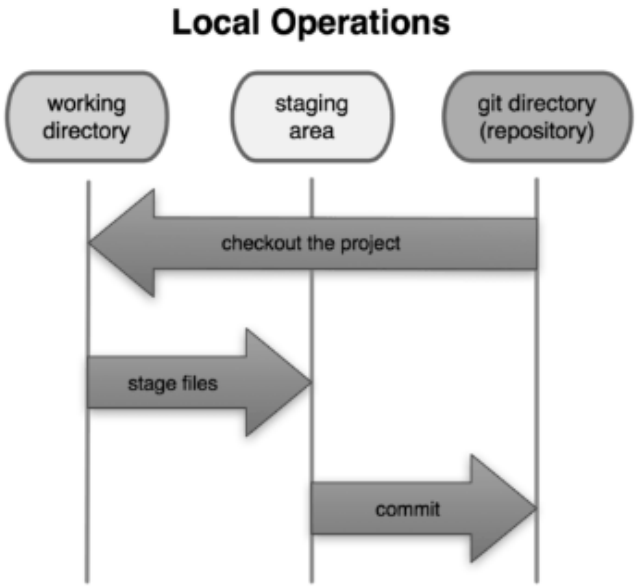
– Or, in-between, in

a "staging" area

• Staged files are ready

to be committed.

• A commit saves a snapshot of allstaged state.

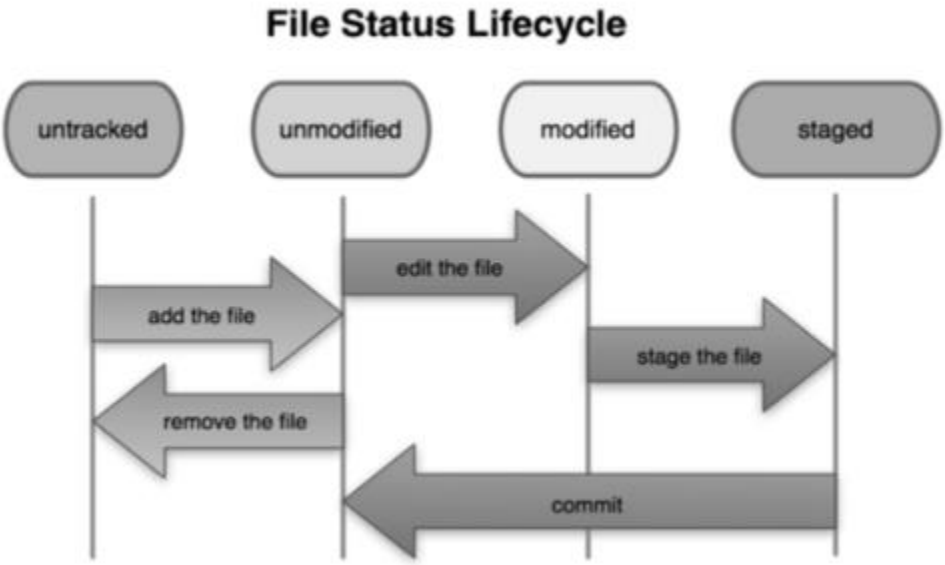


Basic Git Workflow

• Modify files in your working directory.

Stage files, adding snapshots of them to your staging area.

• Commit, which takes the files in the staging area and stores that snapshot permanently to your Git directory.



Initial Git configuration

As you read briefly in Getting Started, you can specify Git configuration settings with the git config command. One of the first things you did was set up your name and email address:

1. System level configuration
2. Global/User configuration
3. Repository / local level configuration

System Level Configuration

System level configuration are available for Entire System.
The first place Git looks for these values is in the system-wide /etc/gitconfig file, which contains settings that are applied to every user on the system and all of their repositories. If you pass the option --system to git config, it reads and writes from this file specifically.

Note: Generally this configuration is not recommended.it is always better to go with other two Global configuration and Repository level configuration

```
waheedk@waheedk-Lenovo-G580:~$ sudo git config --system user.name "Wahid Khan"
waheedk@waheedk-Lenovo-G580:~$ sudo git config --system user.email "wahid@---.com"
waheedk@waheedk-Lenovo-G580:~$ sudo git config --system --list
user.name=Wahid Khan
user.email=wahid@---.com
waheedk@waheedk-Lenovo-G580:~$ sudo cat /etc/gitconfig
[user]
        name = Wahid Khan
        email = wahid@---.com
waheedk@waheedk-Lenovo-G580:~$ sudo git config --system --edit
waheedk@waheedk-Lenovo-G580:~$ sudo git config --system --list
user.name=Waheed Khan
user.email=waheed@---.com
waheedk@waheedk-Lenovo-G580:~$
```

Global/User Configuration

Global/User configuration are available for Current LoggedIn User.

If you pass the option --global to git config, it reads and writes from ~/.gitconfig file specifically.

Note:Better to go with other two Global configuration when you want to have setup for a user.

```
waheedk@waheedk-Lenovo-G580:~$ git config --global user.name "Wahid Khan"
waheedk@waheedk-Lenovo-G580:~$ git config --global user.email "wahid@---.com"
waheedk@waheedk-Lenovo-G580:~$ git config --global --list
user.name=Wahid Khan
user.email=wahid@---.com
core.editor=vi
waheedk@waheedk-Lenovo-G580:~$ git config --global core.editor vim
waheedk@waheedk-Lenovo-G580:~$ git config --global --list
user.name=Wahid Khan
user.email=wahid@---.com
core.editor=vim
waheedk@waheedk-Lenovo-G580:~$ git config --global --edit
waheedk@waheedk-Lenovo-G580:~$ git config --global --list
user.name=Waheed Khan
user.email=waheed@---.com
core.editor=vim
waheedk@waheedk-Lenovo-G580:~$ cat ~/.gitconfig

[user]
        name = Waheed Khan
        email = waheed@---.com
[core]
        editor = vim
```

Core.Editor

By default, Git uses whatever you've set as your default text editor via one of the shell environment variables VISUAL or EDITOR, or else falls back to the vim editor to create and edit your commit and tag messages. To change that default to something else,

you can use the core.editor setting:

```
$ git config --global core.editor vim
```

Now, no matter what is set as your default shell editor, Git will fire up vim to edit messages.

Repository / Local Level Configuration

The git config command lets you configure your Git installation (or an individual repository) from the command line. This command can define everything from user info to preferences to the behaviour of a repository. Several common configuration options are listed below.

Define the author name to be used for all commits in the current repository.

```
waheedk@waheedk-Lenovo-G580:~/git-dir$ git init
Initialized empty Git repository in /home/waheedk/git-dir/.git/
waheedk@waheedk-Lenovo-G580:~/git-dir$ git config user.name "Wahid Khan"
waheedk@waheedk-Lenovo-G580:~/git-dir$ git config user.email "wahid@----.com"
waheedk@waheedk-Lenovo-G580:~/git-dir$ git config core.editor vi
waheedk@waheedk-Lenovo-G580:~/git-dir$ git config --list
user.name=Waheed Khan
user.email=waheed@---.com
user.name=Waheed Khan
user.email=waheed@---.com
core.editor=vim
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
core.editor=vi
user.name=Wahid Khan
user.email=wahid@----.com
```

```
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
        editor = vim
[user]
        name = Waheed Khan
        email = waheed@----.com
~
~
```

```
waheedk@waheedk-Lenovo-G580:~/git-dir$ cat .git/config
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
        editor = vi
[user]
        name = Wahid Khan
        email = wahid@----.com
waheedk@waheedk-Lenovo-G580:~/git-dir$ git config --edit
waheedk@waheedk-Lenovo-G580:~/git-dir$ cat .git/config
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
        editor = vim
[user]
        name = Waheed Khan
        email = waheed@----.com
waheedk@waheedk-Lenovo-G580:~/git-dir$
```

GIT OPERATIONS

1. Setting Up A Repository

This tutorial provides a succinct overview of the most important Git commands. First, the Setting Up a Repository section explains all of the tools you need to start a new version-controlled project. Then, the remaining sections introduce your everyday Git commands.

By the end of this module, you should be able to create a Git repository, record snapshots of your project for safekeeping, and view your project's history.

Git Init

The git init command creates a new Git repository. It can be used to convert an existing, unversioned project to a Git repository or initialize a new empty repository. Most of the other Git commands are not available outside of an initialized repository, so this is usually the first command you'll run in a new project.

Executing git init creates a .git subdirectory in the project root, which contains all of the necessary metadata for the repo. Aside from the .git directory, an existing project

remains unaltered (unlike SVN, Git doesn't require a .git folder in every subdirectory).

Usage

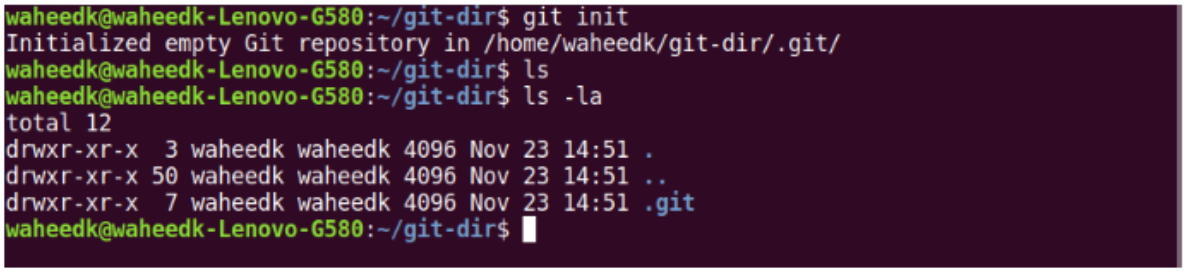
```
$ git init
```

Transform the current directory into a Git repository. This adds a .gitfolder to the current directory and makes it possible to start recording revisions of the project.

```
$ git init directoryNmae
```

Create an empty Git repository in the specified directory. Running this command will create a new folder called directory containing nothing but the .git subdirectory.

```
$ git init --bare directoryName
```



2. Git Status

git-status - Show the working tree status

Use the git status command, to check the current state of the repository.

```
waheedk@waheedk-Lenovo-G580:~/git-dir$ git status
On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)
```

The command checks the status and reports that there's nothing to commit, meaning the repository stores the current state of the working directory, and there are no changes to record.

We will use the git status, to keep monitoring the states of both the working directory and the repository.

Saving Changes

3. Git Add

The git add command adds a change in the working directory to the staging area. It tells Git that you want to include updates to a particular file in the next commit. However, git add doesn't really affect the repository in any significant way—changes are not actually recorded until you run git commit.

In conjunction with these commands, you'll also need git status to view the state of the working directory and the staging area.

Usage:

$ git add fileName

Stage all changes in file for the next commit.

$ git add directoryName

Stage all changes in directory for the next commit.

$ git add -p

When you're starting a new project, git add serves the same function as svn import. To create an initial commit of the current directory, use the following

two commands:

$ git add .

$ git commit

Example:

```
waheedk@waheedk-Lenovo-G580:~/git-dir$ echo "I am Content Of README" >README.md
waheedk@waheedk-Lenovo-G580:~/git-dir$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        README.md

nothing added to commit but untracked files present (use "git add" to track)
waheedk@waheedk-Lenovo-G580:~/git-dir$ git  add README.md
waheedk@waheedk-Lenovo-G580:~/git-dir$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   README.md
```

```
waheedk@waheedk-Lenovo-G580:~/git-dir$ git commit -m "Initial Commit for README.md File"

[master (root-commit) fed979f] Initial Commit for README.md File
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
waheedk@waheedk-Lenovo-G580:~/git-dir$ git status
On branch master
nothing to commit, working tree clean
```

4. Git Commit

The git commit command commits the staged snapshot to the project history.
Committed snapshots can be thought of as “safe” versions of a project—Git will never change them unless you explicitly ask it to. Along with git add, this is one of the most important Git commands.
While they share the same name, this command is nothing like svn commit.
Snapshots are committed to the local repository, and this requires absolutely no interaction with other Git repositories.
Usage:

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

Commit the staged snapshot. This will launch a text editor prompting you for a commit message. After you’ve entered a message, save the file and close the editor to create the actual commit. git commit -m "message-for-commit"
Commit the staged snapshot, but instead of launching a text editor, use "message-for-commit" as the commit message.

5. Git Add & Commit Other Options:

```
waheedk@waheedk-Lenovo-G580:~/git add README.md
waheedk@waheedk-Lenovo-G580:~/git-dir$ git add .
waheedk@waheedk-Lenovo-G580:~/git-dir$ git add --all
waheedk@waheedk-Lenovo-G580:~/git-dir$ git add -A
waheedk@waheedk-Lenovo-G580:~/git-dir$ git commit -m "commit-msg"
waheedk@waheedk-Lenovo-G580:~/git-dir$ git commit -am "Commit-msg"
```

Begin an interactive staging session that lets you choose portions of a file to add to the next commit. This will present you with a chunk of changes and prompt you for a command. Use y to stage the chunk, n to ignore the chunk, s to split it into smaller chunks, e to manually edit the chunk, and q to exit.

Discussion

The git add and git commit commands compose the fundamental Git workflow. These are the two commands that every Git user needs to understand, regardless of their team's collaboration model. They are the means to record versions of a project into the repository's history.

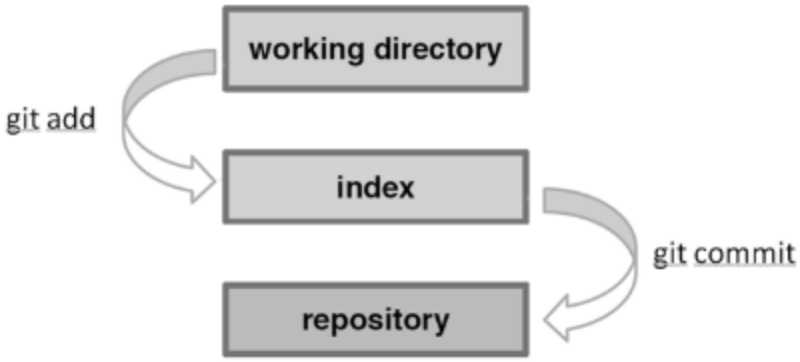
Developing a project revolves around the basic edit/stage/commit pattern. First, you edit your files in the working directory. When you're ready to save a copy of the current state of the project, you stage changes with git add. After you're happy with the staged snapshot, you commit it to the project history with git commit.

The git add command should not be confused with svn add, which adds a file to the repository. Instead, git add works on the more abstract level of changes. This means that git add needs to be called every time you alter a file, whereas svn add only needs to be called once for each file. It may sound redundant, but this workflow makes it much easier to keep a project organized.

The Staging Area

The staging area is one of Git's more unique features, and it can take some time to wrap your head around it if you're coming from an SVN (or even a Mercurial) background. It helps to think of it as a buffer between the working directory and the project history.

Instead of committing all the changes you've made since the last commit, the stage lets you group related changes into highly focused snapshots before committing it to the project history. This means you can make all sorts of edits to unrelated files, then go back and split them up into logical commits by adding related changes to the stage and commit them piece-by-piece. As in any revision control system, it's important to create atomic commits so that it's easy to track down bugs and revert changes with minimal impact on the rest of the project.



Commit History

After you have created several commits, or if you have cloned a repository with an existing commit history, you'll probably want to look back to see what has happened. The most basic and powerful tool to do this is the git log command.

```
waheedk@waheedk-Lenovo-G580:~/git-dir$ touch file{1..10}.txt
waheedk@waheedk-Lenovo-G580:~/git-dir$ git add .
waheedk@waheedk-Lenovo-G580:~/git-dir$ git commit -m "adding Empty Files"
[master 83c91ab] adding Empty Files
 10 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file1.txt
 create mode 100644 file10.txt
 create mode 100644 file2.txt
 create mode 100644 file3.txt
 create mode 100644 file4.txt
 create mode 100644 file5.txt
 create mode 100644 file6.txt
 create mode 100644 file7.txt
 create mode 100644 file8.txt
 create mode 100644 file9.txt
waheedk@waheedk-Lenovo-G580:~/git-dir$ git log
commit 83c91abcc42b9f971c10e076c0b931f2a9374669
Author: Waheed Khan <waheed@---.com>
Date:   Thu Nov 23 16:39:25 2017 +0530

    adding Empty Files

commit fb883df22b8a928d0574d126af2231900d55d61e
Author: Waheed Khan <waheed@---.com>
Date:   Thu Nov 23 16:38:23 2017 +0530

    2nd commit for modifiying README content

commit fed979f388b6f5f127f04dc6f886a456bb683c07
Author: Waheed Khan <waheed@---.com>
Date:   Thu Nov 23 15:57:51 2017 +0530

    Initial Commit for README.md File
```

Git Log & Other Options

```
waheedk@waheedk-Lenovo-G580:~/git-dir$ git log --oneline
83c91ab adding Empty Files
fb883df 2nd commit for modifiying README content
fed979f Initial Commit for README.md File
waheedk@waheedk-Lenovo-G580:~/git-dir$ git shortlog
Waheed Khan (3):
      Initial Commit for README.md File
      2nd commit for modifiying README content
      adding Empty Files

waheedk@waheedk-Lenovo-G580:~/git-dir$ git reflog
83c91ab HEAD@{0}: commit: adding Empty Files
fb883df HEAD@{1}: commit: 2nd commit for modifiying README content
fed979f HEAD@{2}: commit (initial): Initial Commit for README.md File
waheedk@waheedk-Lenovo-G580:~/git-dir$ git log --oneline |wc -l
3
waheedk@waheedk-Lenovo-G580:~/git-dir$ git log --oneline --graph
* 83c91ab adding Empty Files
* fb883df 2nd commit for modifiying README content
* fed979f Initial Commit for README.md File
```

```
waheedk@waheedk-Lenovo-G580:~/git-dir$ git log --stat
commit 83c91abcc42b9f971c10e076c0b931f2a9374669
Author: Waheed Khan <waheed@---.com>
Date:   Thu Nov 23 16:39:25 2017 +0530

    adding Empty Files

 file1.txt  | 0
 file10.txt | 0
 file2.txt  | 0
 file3.txt  | 0
 file4.txt  | 0
 file5.txt  | 0
 file6.txt  | 0
 file7.txt  | 0
 file8.txt  | 0
 file9.txt  | 0
 10 files changed, 0 insertions(+), 0 deletions(-)

commit fb883df22b8a928d0574d126af2231900d55d61e
Author: Waheed Khan <waheed@---.com>
Date:   Thu Nov 23 16:38:23 2017 +0530

    2nd commit for modifiying README content

 README.md | 2 ++
 1 file changed, 2 insertions(+)

commit fed979f388b6f5f127f04dc6f886a456bb683c07
Author: Waheed Khan <waheed@---.com>
Date:   Thu Nov 23 15:57:51 2017 +0530

    Initial Commit for README.md File

 README.md | 1 +
 1 file changed, 1 insertion(+)
```

```
waheedk@waheedk-Lenovo-G580:~/git-dir$ git log --shortstat
commit 83c91abcc42b9f971c10e076c0b931f2a9374669
Author: Waheed Khan <waheed@---.com>
Date:   Thu Nov 23 16:39:25 2017 +0530

    adding Empty Files

 10 files changed, 0 insertions(+), 0 deletions(-)

commit fb883df22b8a928d0574d126af2231900d55d61e
Author: Waheed Khan <waheed@---.com>
Date:   Thu Nov 23 16:38:23 2017 +0530

    2nd commit for modifiying README content

 1 file changed, 2 insertions(+)

commit fed979f388b6f5f127f04dc6f886a456bb683c07
Author: Waheed Khan <waheed@---.com>
Date:   Thu Nov 23 15:57:51 2017 +0530

    Initial Commit for README.md File

 1 file changed, 1 insertion(+)
```

--stat -> Show statistics for files modified in each commit.

--shortstat -> Display only the changed/insertions/deletions line from the --stat command.

Git Diff

git-diff - Show changes between commits, commit and working tree, etc

The main objective of version controlling is to enable you to work with different versions of files. Git provides a command diff to let you to compare different versions of your files.

The most common scenario to use diff is to see what changes you made after your last commit. Let's see how to do it.

```
waheedk@waheedk-Lenovo-G580:~/git-dir$ git log --oneline
a2edccd Create newfile1.txt
cc8942f Create newfile.txt
83c91ab adding Empty Files
fb883df 2nd commit for modifiying README content
fed979f Initial Commit for README.md File
waheedk@waheedk-Lenovo-G580:~/git-dir$ git diff a2edccd..cc8942f
diff --git a/newfile1.txt b/newfile1.txt
deleted file mode 100644
index 8628c4d..0000000
--- a/newfile1.txt
+++ /dev/null
@@ -1 +0,0 @@
-newfile1.txt
waheedk@waheedk-Lenovo-G580:~/git-dir$ git diff HEAD..HEAD~1
diff --git a/newfile1.txt b/newfile1.txt
deleted file mode 100644
index 8628c4d..0000000
--- a/newfile1.txt
+++ /dev/null
@@ -1 +0,0 @@
-newfile1.txt
waheedk@waheedk-Lenovo-G580:~/git-dir$
```

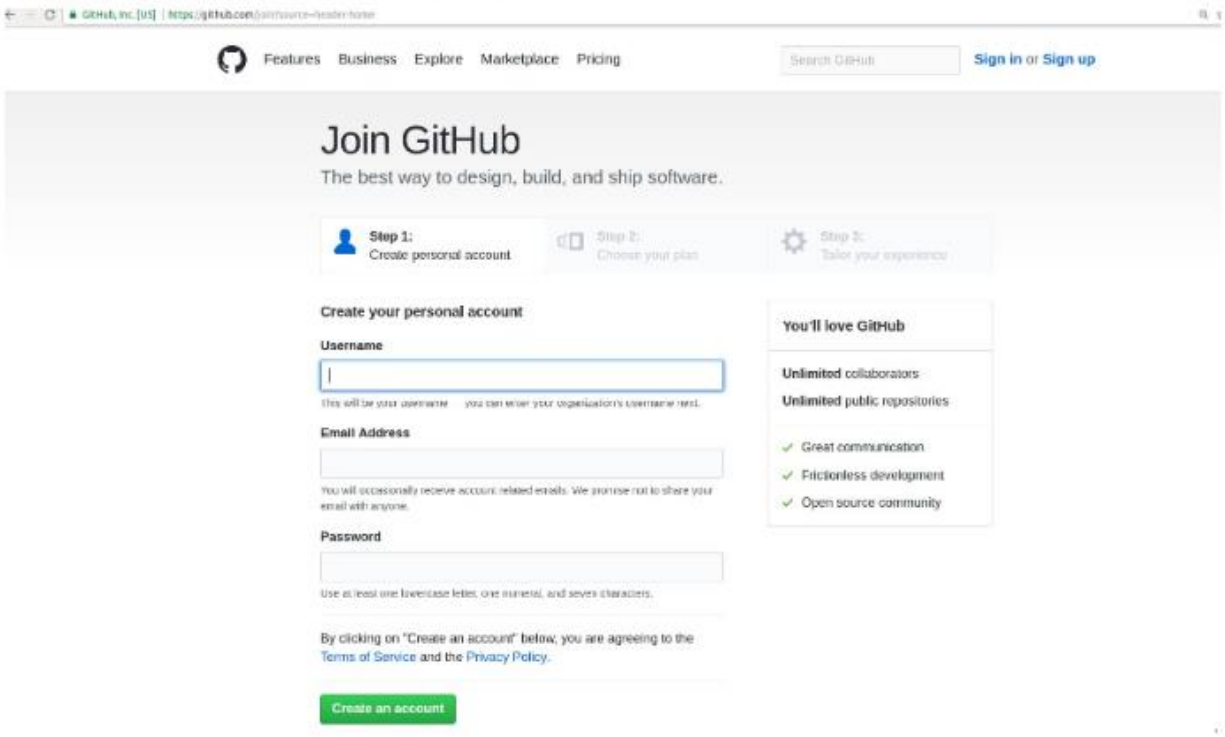
Note: Here, In git diff you can use commitId or HEAD

HEAD refers two thing in git:

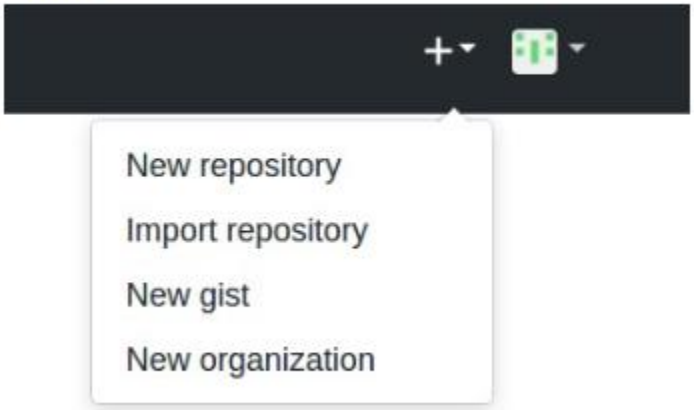
1. Top/Recent Level commit.
2. Current Working Branch.

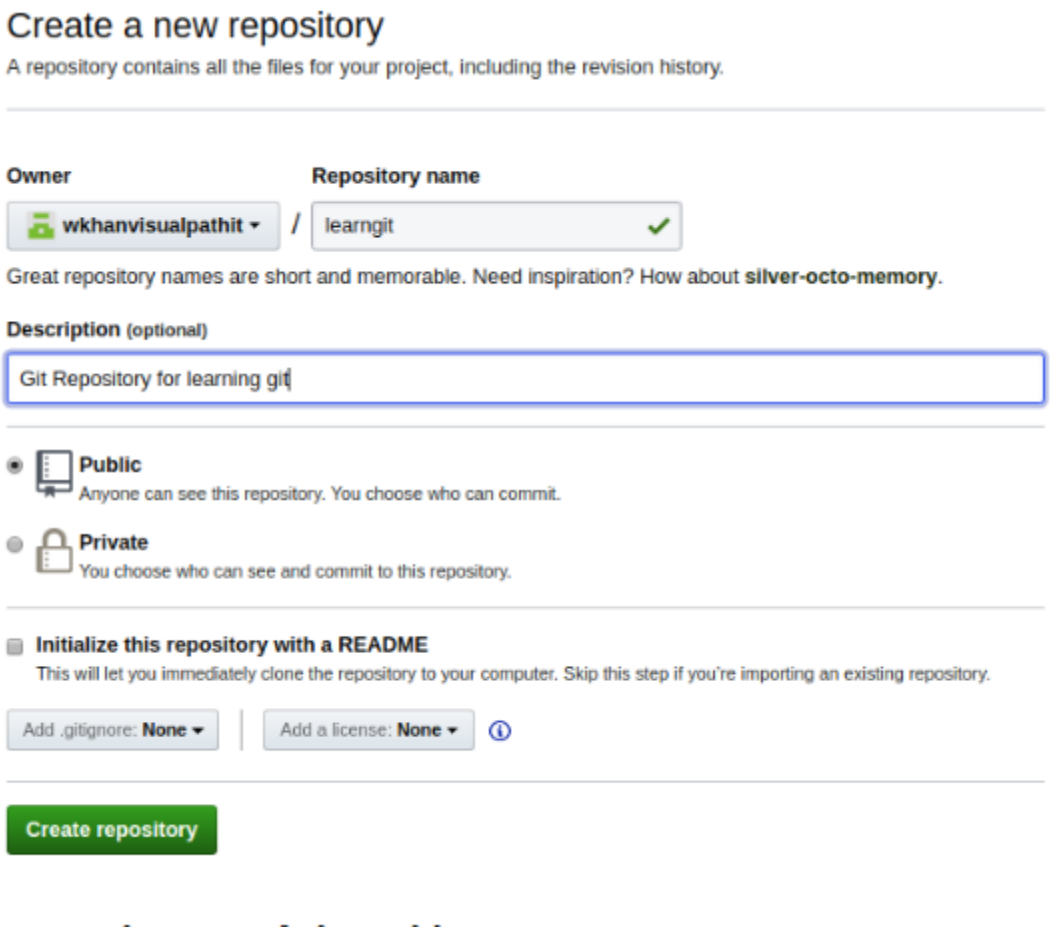
GitHub Quick Setup

1.Sign Up For The Github Account.



2. Create Public Repository.





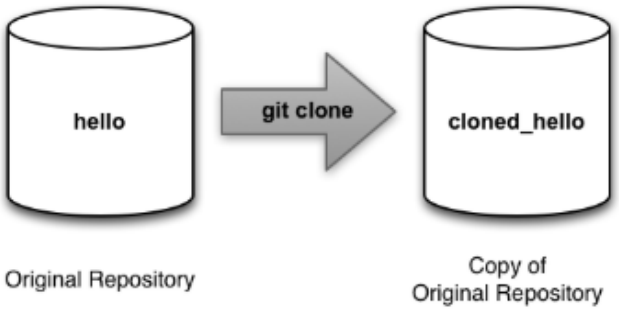
3. Copy The URL Of The Public Repo.

Git Clone

1.Clone Empty GitHub Repository:

The git clone command copies an existing Git repository. This is sort of like svn checkout, except the “working copy” is a full-fledged Git repository—it has its own history, manages its own files, and is a completely isolated environment from the original repository.

As a convenience, cloning automatically creates a remote connection called origin pointing back to the original repository. This makes it very easy to interact with a central repository.



Clone git repository on your local system

```
# git clone Git-Repo-URL
```

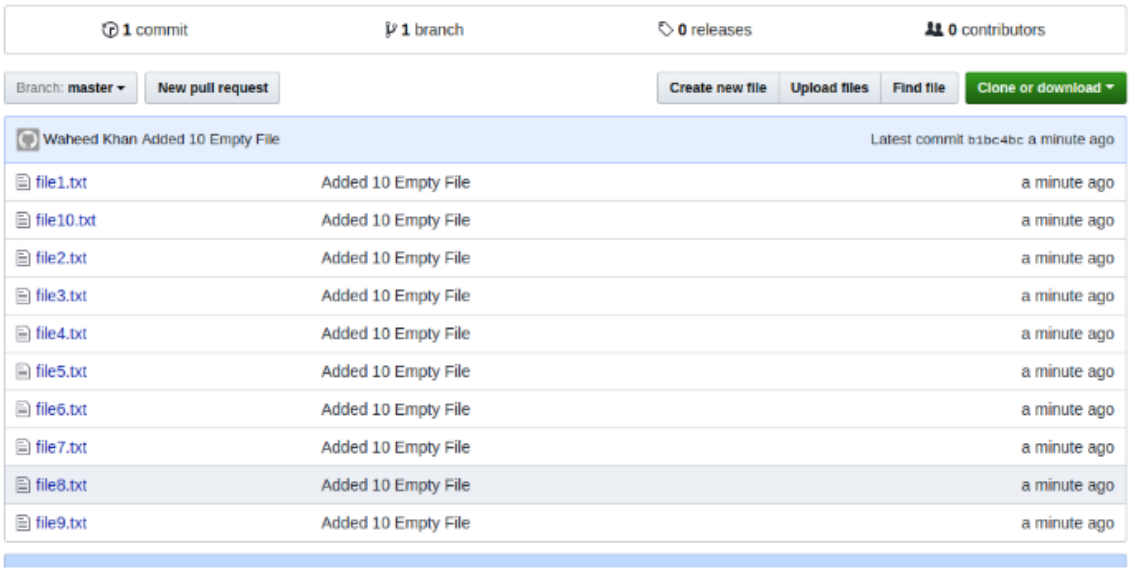
```
waheedk@waheedk-Lenovo-G580:~$ git clone https://github.com/wkhanvisualpathit/learngit.git
Cloning into 'learngit'...
warning: You appear to have cloned an empty repository.
Checking connectivity... done.
waheedk@waheedk-Lenovo-G580:~$
```

Note: Now we have just clone an empty repository,we can add the content push it to remote repository.

Example :

```
waheedk@waheedk-Lenovo-G580:~$ cd learngit/
waheedk@waheedk-Lenovo-G580:~/learngit$ touch file{1..10}.txt
waheedk@waheedk-Lenovo-G580:~/learngit$ git add .
waheedk@waheedk-Lenovo-G580:~/learngit$ git commit -m "Added 10 Empty File"
[master (root-commit) b1bc4bc] Added 10 Empty File
 10 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file1.txt
 create mode 100644 file10.txt
 create mode 100644 file2.txt
 create mode 100644 file3.txt
 create mode 100644 file4.txt
 create mode 100644 file5.txt
 create mode 100644 file6.txt
 create mode 100644 file7.txt
 create mode 100644 file8.txt
 create mode 100644 file9.txt
waheedk@waheedk-Lenovo-G580:~/learngit$ git push -u origin master
Username for 'https://github.com': wkhanvisualpathit
Password for 'https://wkhanvisualpathit@github.com':
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 251 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/wkhanvisualpathit/learngit.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
waheedk@waheedk-Lenovo-G580:~/learngit$
```

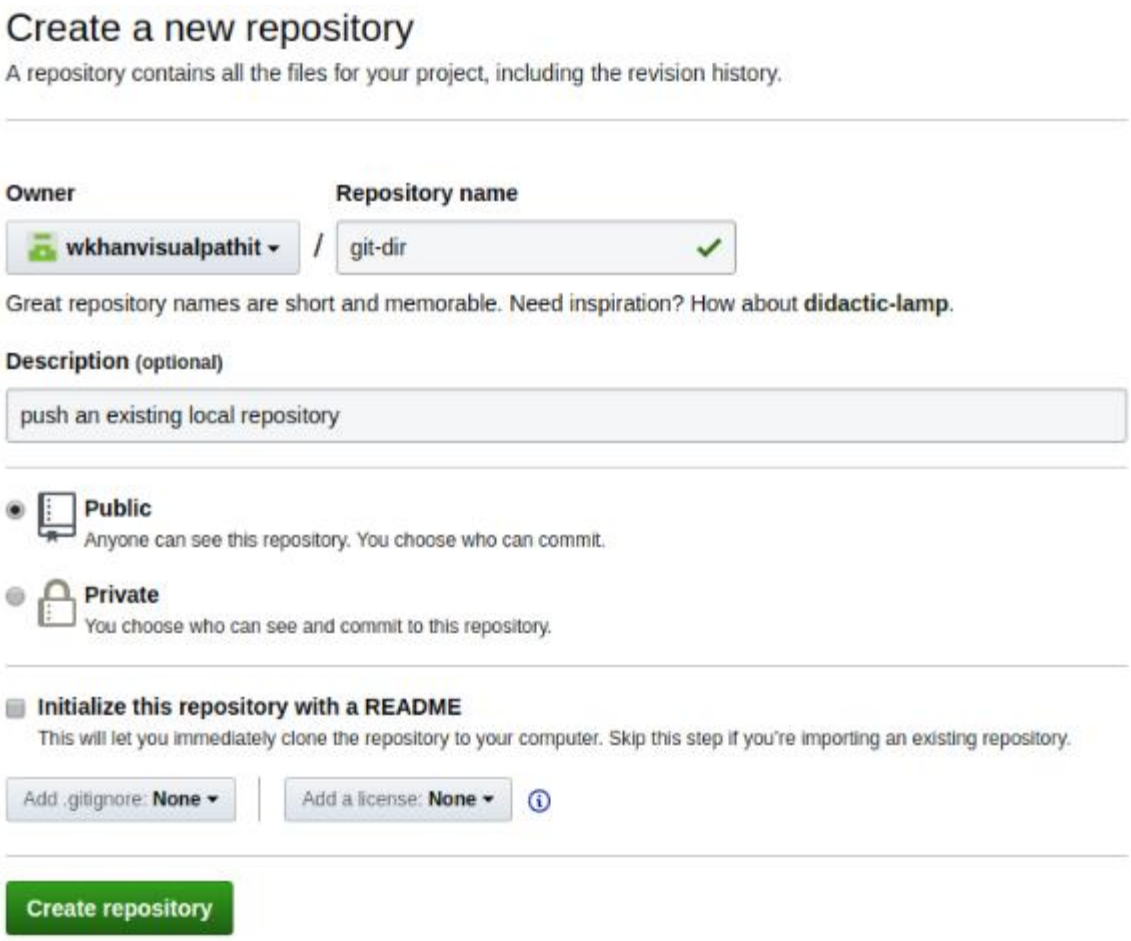
Refresh the Remote repository page and you we see the output



Push Existing Local Repository To Remote Repo

In git quick start guide we have seen to create git repository on github and then clone/pull that repo to local computer but git clone command. We will see now how to create repo locally and then later how to push it to github.

Create a public Repository and grap the remote url, add it to your local repo as bellow:


Create a new repository
A repository contains all the files for your project, including the revision history.
Owner
Repository name
wkhanvisualpathit
/
git-dir
Great repository names are short and memorable. Need inspiration? How about didactic-lamp.
Description (optional)
push an existing local repository
Public
Anyone can see this repository. You choose who can commit.
Private
You choose who can see and commit to this repository.
Initialize this repository with a README
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.
Add .gitignore: None
Add a license: None
Create repository

...or push an existing repository from the command line

```
git remote add origin https://github.com/wkhanvisualpathit/git-dir.git
git push -u origin master
```

```
waheedk@waheedk-Lenovo-G580:~/git-dir$ git log --oneline
83c91ab adding Empty Files
fb883df 2nd commit for modifiying README content
fed979f Initial Commit for README.md File
waheedk@waheedk-Lenovo-G580:~/git-dir$ git remote add origin https://github.com/wkhanvisualpathit/git-dir.git
waheedk@waheedk-Lenovo-G580:~/git-dir$ git remote
origin
waheedk@waheedk-Lenovo-G580:~/git-dir$ git remote -v
origin  https://github.com/wkhanvisualpathit/git-dir.git (fetch)
origin  https://github.com/wkhanvisualpathit/git-dir.git (push)
waheedk@waheedk-Lenovo-G580:~/git-dir$ git push -u origin master
Username for 'https://github.com': wkhanvisualpathit
Password for 'https://wkhanvisualpathit@github.com':
Counting objects: 9, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (9/9), 808 bytes | 0 bytes/s, done.
Total 9 (delta 0), reused 0 (delta 0)
To https://github.com/wkhanvisualpathit/git-dir.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
waheedk@waheedk-Lenovo-G580:~/git-dir$
```

Refresh The Browser!

3 commits | 1 branch | 0 releases | 0 contributors

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

Waheed Khan adding Empty Files — Latest commit 83c91ab an hour ago

README.md	2nd commit for modifiying README content	an hour ago
file1.txt	adding Empty Files	an hour ago
file10.txt	adding Empty Files	an hour ago
file2.txt	adding Empty Files	an hour ago
file3.txt	adding Empty Files	an hour ago
file4.txt	adding Empty Files	an hour ago
file5.txt	adding Empty Files	an hour ago
file6.txt	adding Empty Files	an hour ago
file7.txt	adding Empty Files	an hour ago
file8.txt	adding Empty Files	an hour ago
file9.txt	adding Empty Files	an hour ago

README.md

I am Content Of README new Content Added Here

Syncing

SVN uses a single central repository to serve as the communication hub for developers, and collaboration takes place by passing changesets between the developers' working copies and the central repository. This is different from Git's collaboration model, which gives every developer their own copy of the repository, complete with its own local history and branch structure. Users typically need to share a series of commits rather than a single changeset. Instead of committing a changeset from a working copy to the central repository, Git lets you share entire branches between repositories.

The commands presented below let you manage connections with other repositories, publish local history by "pushing" branches to other repositories, and see what others have contributed by "pulling" branches into your local repository.

Git Remote

The git remote command lets you create, view, and delete connections to other repositories. Remote connections are more like bookmarks rather than direct links into other repositories. Instead of providing real-time access to another repository,

they serve as convenient names that can be used to reference a not-so-convenient URL.

Usage :

$ git remote

List the remote connections you must other repositories.

$ git remote -v

Same as the above command, but include the URL of each connection.

$ git remote add "name" "url"

Create a new connection to a remote repository. After adding a remote, you'll be able to use name as a convenient shortcut for url in other Git commands.

$ git remote rm "name"

Remove the connection to the remote repository called name.

$ git remote rename "old-name" "new-name"

Rename a remote connection from old-name to new-name.

Repository URLs

Git supports many ways to reference a remote repository. Two of the easiest ways to access a remote repo are via the HTTP and the SSH protocols. HTTP is an easy way to allow anonymous, read-only access to a repository. For example:

http://host/path/to/repo.git

But, it's generally not possible to push commits to an HTTP address (you wouldn't want to allow anonymous pushes anyways). For read-write access, you should use SSH instead:

ssh://user@host/path/to/repo.git

You'll need a valid SSH account on the host machine, but other than that, Git supports authenticated access via SSH out of the box.

Examples

In addition to origin, it's often convenient to have a connection to your teammates' repositories. For example, if your co-worker, John, maintained a publicly accessible repository on dev.example.com/john.git, you could add a connection as follows:

```
$ git remote add john http://dev.example.com/john.git
```

Having this kind of access to individual developers' repositories makes it possible to collaborate outside of the central repository. This can be very useful for small teams working on a large project.

Git Fetch

The git fetch command imports commits from a remote repository into your local repo. The resulting commits are stored as remote branches instead of the normal local branches that we've been working with. This gives you a chance to review changes before integrating them into your copy of the project. Usage:

$ git fetch remoteName

Fetch all of the branches from the repository. This also downloads all of the required commits and files from the other repository.

$ git fetch remoteName branch

Same as the above command, but only fetch the specified branch.

Note: If you are performing git fetch then you have to do git merge explicitly to merge the remote branch changes to local branch shown as bellow:

```
waheedk@waheedk-Lenovo-G580:~/git-dir$ git fetch origin master
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/wkhanvisualpathit/git-dir
 * branch            master     -> FETCH_HEAD
   83c91ab..cc8942f  master     -> origin/master
waheedk@waheedk-Lenovo-G580:~/git-dir$ git merge origin/master
Updating 83c91ab..cc8942f
Fast-forward
 newfile.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 newfile.txt
waheedk@waheedk-Lenovo-G580:~/git-dir$
```

Git Pull

Merging upstream changes into your local repository is a common task in Git-based collaboration workflows. We already know how to do this with git fetch followed by git merge, but git pull rolls this into a single command.

Usage:

$ git pull remote

Fetch the specified remote's copy of the current branch and immediately merge it into the local copy. This is the same as git fetch remote followed by git merge origin/current-branch.

$ git pull --rebase remote

Same as the above command, but instead of using git merge to integrate the remote branch with the local one, use git rebase.

```
waheedk@waheedk-Lenovo-G580:~/git-dir$ git pull origin master
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
Unpacking objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
From https://github.com/wkhanvisualpathit/git-dir
 * branch            master     -> FETCH_HEAD
   cc8942f..a2edccd  master     -> origin/master
Updating cc8942f..a2edccd
Fast-forward
 newfile1.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 newfile1.txt
waheedk@waheedk-Lenovo-G580:~/git-dir$
```

Git push

Pushing is how you transfer commits from your local repository to a remote repo. It's the counterpart to git fetch, but whereas fetching imports commits to local branches, pushing exports commits to remote branches. This has the potential to overwrite changes, so you need to be careful how you use it. These issues are discussed below.

Usage:

$ git push remote branch

Push the specified branch to remote, along with all of the necessary commits and internal objects. This creates a local branch in the destination repository. To prevent you from overwriting commits, Git won't let you push when it results in a non-fast-forward merge in the destination repository.

$ git push remote --force

Same as the above command, but force the push even if it results in a non-fast-forward merge. Do not use the --force flag unless you're absolutely sure you know what you're doing.

$ git push remote --all

Push all of your local branches to the specified remote.

$ git push remote --tags

Tags are not automatically pushed when you push a branch or use the --all option. The --tags flag sends all your local tags to the remote repository.

Git Branching and Merging

A branch, at its core, is a unique series of code changes with a unique name. Each repository can have one or more branches.

By default, the first branch is called "master".

Viewing Branches

Prior to creating new branches, we want to see all the branches that exist. We can view all existing branches by typing the following:

```
$ git branch -a
```

Adding the "-a" to the end of our command tells GIT that we want to see all branches that exist, including ones that we do not have in our local workspace.

```
$ git branch -r
```

Adding the "-r" to the end of our command tells GIT that we want to see all remote branches that exist

```
User@User-PC MINGW32 ~/git-dir/learngit (master)
$ git branch
* master

User@User-PC MINGW32 ~/git-dir/learngit (master)
$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master

User@User-PC MINGW32 ~/git-dir/learngit (master)
$ git branch -r
  origin/HEAD -> origin/master
  origin/master
```

The asterisk next to "master" in the first line of the output indicates that we are currently on that branch. The second line simply indicates that on our remote, named origin, there is a single branch, also called master.

Now that we know how to view branches, it time create our first one.

Creating Branches

We are going to treat the default "master" branch as our production and therefore need to create a single branch for development, or pre-production.

To create a new branch, named development, type the following:

$ git branch branchName

```
User@User-PC MINGW32 ~/git-dir/learngit (master)
$ git branch development

User@User-PC MINGW32 ~/git-dir/learngit (master)
$ git branch
  development
* master
```

Switched To A Branch

Switched to a new branch 'development'

$ git checkout branchName

```
User@User-PC MINGW32 ~/git-dir/learngit (master)
$ git checkout development
Switched to branch 'development'

User@User-PC MINGW32 ~/git-dir/learngit (development)
$ git log --oneline
b1bc4bc (HEAD -> development, origin/master, origin/HEAD, master) Added 10 Empty File
```

Creating Branch And Switch To That Branch Same Time

Assuming we do not yet have a branch named "pre-prod", the output would be as follows: Switched to a new branch 'pre-prod'

```
User@User-PC MINGW32 ~/git-dir/learngit (development)
$ git checkout -b pre-prod
Switched to a new branch 'pre-prod'

User@User-PC MINGW32 ~/git-dir/learngit (pre-prod)
$ git branch
  development
  master
* pre-prod
```

Now that we have multiple branches, we need to put them to good use. In our scenario, we are going to use our "development" branch for testing out our changes and the master branch for releasing them to the public.

To illustrate this process, we need to switch back to our develop branch:

$ git checkout develop

Making changes to our develop branch

On this branch, we are going to modify a new blank file, named "file1.txt". Until we merge it to the master branch (in the following step), it will not exist there.

```
User@User-PC MINGW32 ~/git-dir/learngit (development)
$ vi file1.txt


User@User-PC MINGW32 ~/git-dir/learngit (development)
$ git status
On branch development
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   file1.txt

no changes added to commit (use "git add" and/or "git commit -a")

User@User-PC MINGW32 ~/git-dir/learngit (development)
$ git add .
warning: LF will be replaced by CRLF in file1.txt.
The file will have its original line endings in your working directory.

User@User-PC MINGW32 ~/git-dir/learngit (development)
$ git commit -m "file content change in branch development"
[development 5af01f0] file content change in branch development
 1 file changed, 1 insertion(+)

User@User-PC MINGW32 ~/git-dir/learngit (development)
$ git log --oneline
5af01f0 (HEAD -> development) file content change in branch development
b1bc4bc (origin/master, origin/HEAD, pre-prod, master) Added 10 Empty File
```

This file now modified on the develop branch; as we're about to find out, it doesn't exist on the master branch.

First, we are going to confirm that we are currently on the develop branch. We can do this by typing the following:

$ git branch

Merging Code Between Branches

Suppose you've decided that work is complete in "development" branch and ready to be merged into your master branch. In order to do that, you'll merge your "development" branch code into master and all the logs and data will be index in master. All you have to do is check out the branch you wish to merge into and then run the git merge command:

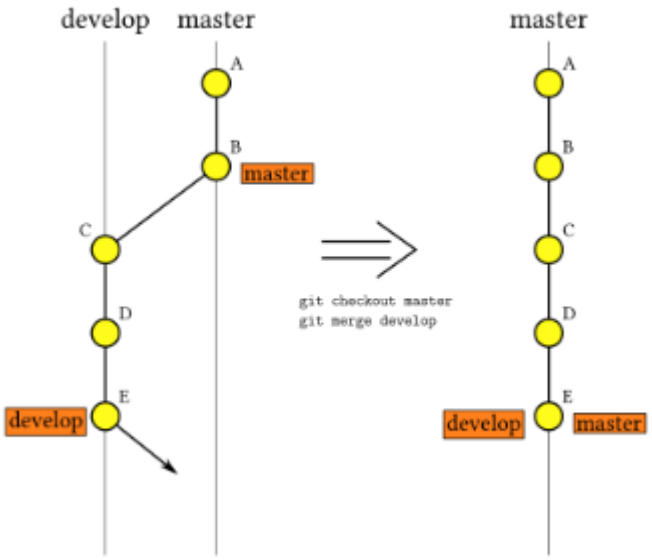
```
User@User-PC MINGW32 ~/git-dir/learngit (development)
$ git branch
* development
  master
  pre-prod

User@User-PC MINGW32 ~/git-dir/learngit (development)
$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.

User@User-PC MINGW32 ~/git-dir/learngit (master)
$ git merge development
Updating b1bc4bc..5af01f0
Fast-forward
 file1.txt | 1 +
 1 file changed, 1 insertion(+)

User@User-PC MINGW32 ~/git-dir/learngit (master)
$ git log --oneline
5af01f0 (HEAD -> master, development) file content change in branch development
b1bc4bc (origin/master, origin/HEAD, pre-prod) Added 10 Empty File
```

Type Of Merge

Once you've finished developing a feature in an isolated branch, it's important to be able to get it back into the main code base. Depending on the structure of your repository, Git has several distinct algorithms to accomplish this:
a fast-forward merge or a 3-way merge.

Fast-Forward Merge

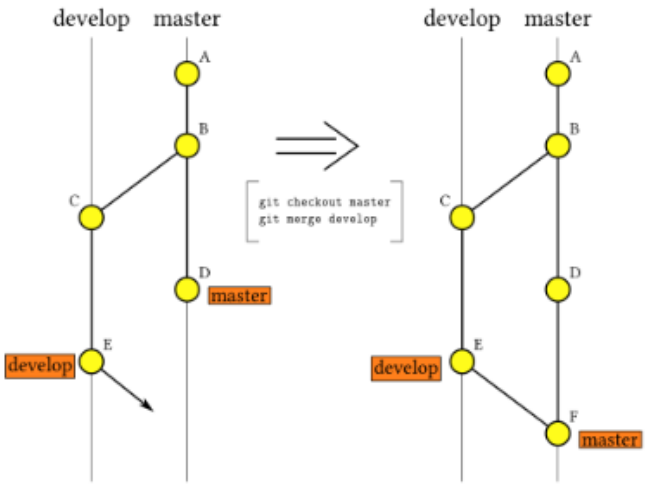
A fast-forward merge can occur when there is a linear path from the current branch tip to the target branch. Instead of "actually" merging the branches, all Git has to do to integrate the histories is move (i.e., "fast forward") the current branch tip up to the target branch tip. This effectively combines the histories, since all of the commits reachable from the target branch are now available through the current one. For example, a fast forward merge of some-feature into master would look something like the



```
User@User-PC MINGW32 ~/git-dir/learngit (development)
$ echo "developement content in branch" >development.txt

User@User-PC MINGW32 ~/git-dir/learngit (development)
$ git add .
warning: LF will be replaced by CRLF in development.txt.
The file will have its original line endings in your working directory.

User@User-PC MINGW32 ~/git-dir/learngit (development)
$ git commit -m "some new file added in development branch"
[development a8a6ea9] some new file added in development branch
 1 file changed, 1 insertion(+), 1 deletion(-)

User@User-PC MINGW32 ~/git-dir/learngit (development)
$ git log --oneline
a8a6ea9 (HEAD -> development) some new file added in development branch
e855e25 some new file added in development branch
5af01f0 (master) file content change in branch development
b1bc4bc (origin/master, origin/HEAD, pre-prod) Added 10 Empty File

User@User-PC MINGW32 ~/git-dir/learngit (development)
$ git checkout master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

User@User-PC MINGW32 ~/git-dir/learngit (master)
$ git merge development
Updating 5af01f0..a8a6ea9
Fast-forward
 development.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 development.txt
```

However, a fast-forward merge is not possible if the branches have diverged. When there is not a linear path to the target branch, Git has no choice but to combine them via a 3-way merge.

3-Way Merge



3-way merges use a dedicated commit to tie together the two histories. The nomenclature comes from the fact that Git uses three commits to generate the merge commit: the two branch tips and their common ancestor.

Content In development branch:

```
waheedk@User-PC MINGW32 ~/git-dir/learngit (development)
$ echo "new content by development branch" > development.txt

waheedk@User-PC MINGW32 ~/git-dir/learngit (development)
$ git add .
warning: LF will be replaced by CRLF in development.txt.
The file will have its original line endings in your working directory.

waheedk@User-PC MINGW32 ~/git-dir/learngit (development)
$ git commit -m "new content by development branch"
[development b6c739a] new content by development branch
 1 file changed, 1 insertion(+)
 create mode 100644 development.txt

waheedk@User-PC MINGW32 ~/git-dir/learngit (development)
$ git log --oneline
b6c739a (HEAD -> development) new content by development branch
5af01f0 (master) file content change in branch development
b1bc4bc (origin/master, origin/HEAD, pre-prod) Added 10 Empty File
```

Content In master branch:

```
waheedk@User-PC MINGW32 ~/git-dir/learngit (master)
$ git checkout master ^C

waheedk@User-PC MINGW32 ~/git-dir/learngit (master)
$ vi file1.txt

waheedk@User-PC MINGW32 ~/git-dir/learngit (master)
$ git add .

waheedk@User-PC MINGW32 ~/git-dir/learngit (master)
$ git commit -m "modified the file1.txt content"
[master 829b9b9] modified the file1.txt content
 1 file changed, 2 insertions(+)

waheedk@User-PC MINGW32 ~/git-dir/learngit (master)
$ git log --oneline
829b9b9 (HEAD -> master) modified the file1.txt content
5af01f0 file content change in branch development
b1bc4bc (origin/master, origin/HEAD, pre-prod) Added 10 Empty File
```

Note: If both the branches have its own changes that time it follow 3-way handshake and generate a merge commit as bellow:

```
waheedk@User-PC MINGW32 ~/git-dir/learngit (master)
$ git merge development
Merge made by the 'recursive' strategy.
 development.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 development.txt

waheedk@User-PC MINGW32 ~/git-dir/learngit (master)
$ git log --oneline
1dcc328 (HEAD -> master) Merge branch 'development'
829b9b9 modified the file1.txt content
b6c739a (development) new content by development branch
5af01f0 file content change in branch development
b1bc4bc (origin/master, origin/HEAD, pre-prod) Added 10 Empty File
```

Pushing Git Branches

```
waheedk@User-PC MINGW32 ~/git-dir/learngit (development)
$ git push -u origin development
Counting objects: 17, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (17/17), 1.52 KiB | 388.00 KiB/s, done.
Total 17 (delta 8), reused 0 (delta 0)
remote: Resolving deltas: 100% (8/8), completed with 1 local object.
To https://github.com/wkhanvisualpathit/learngit.git
 * [new branch]      development -> development
Branch development set up to track remote branch development from origin.

waheedk@User-PC MINGW32 ~/git-dir/learngit (development)
$ git checkout master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 5 commits.
  (use "git push" to publish your local commits)

waheedk@User-PC MINGW32 ~/git-dir/learngit (master)
$ git push -u origin master
Total 0 (delta 0), reused 0 (delta 0)
To https://github.com/wkhanvisualpathit/learngit.git
   b1bc4bc..88227f4  master -> master
Branch master set up to track remote branch master from origin.
```

Delete Local Branch

To delete the local branch use one of the following:

```
$ git branch -d branch_name
$ git branch -D branch_name
```

Note:The -d option is an alias for --delete, which only deletes the branch if it has already been fully merged in its upstream branch.

You could also use -D, which is an alias for --delete --force, which deletes the branch "irrespective of its merged status."

Delete Remote Branch

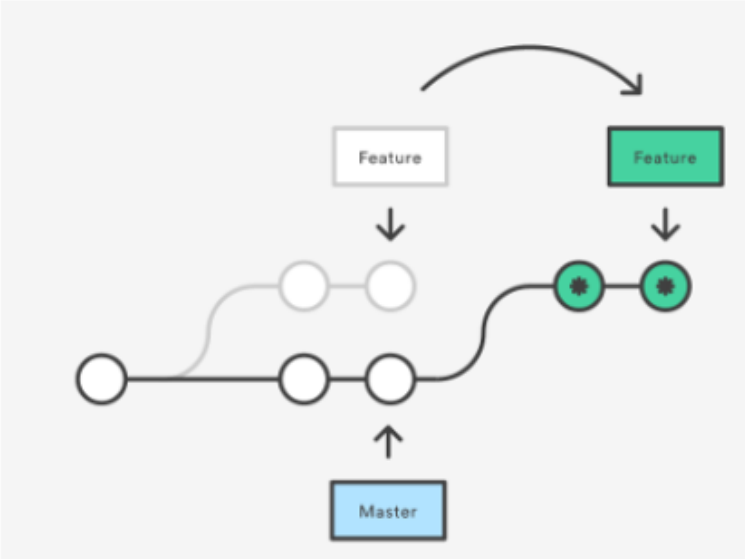
you can delete a remote branch using:

```
$ git push remote_name --delete branch_name
```

What is git rebase?

In Git, there are two main ways to integrate changes from one branch into another: the merge and the rebase. In this section you'll learn what rebasing is, how to do it, why it's a pretty amazing tool, and in what cases you won't want to use it.

Rebasing is the process of moving or combining a sequence of commits to a new base commit. Rebasing is most useful and easily visualized in the context of a feature branching workflow. The general process can be visualized as the following:



From a content perspective, rebasing is changing the base of your branch from one commit to another making it appear as if you'd created your branch from a different commit. Internally, Git accomplishes this by creating new commits and applying them to the specified base. It's very important to understand that even though the branch looks the same, it's composed of entirely new commits.

If you go back to an earlier example from Basic Merging, you can see that you diverged your work and made commits on two different branches.
The easiest way to integrate the branches, as we've already covered, is the merge command. It performs a three-way merge between the two latest branch snapshots.There while merege new extra commit is generated which wont gives us the commit history linearity but in rebase we will have commit history linearity.

Note : Git Rebase is used for git commit history linearity.

Content in master branch:

```
waheedk@User-PC MINGW32 ~/git-dir/learngit (master)
$ vi file1.txt

waheedk@User-PC MINGW32 ~/git-dir/learngit (master)
$ git add .

waheedk@User-PC MINGW32 ~/git-dir/learngit (master)
$ git commit -m "modification by master"
[master 88227f4] modification by master
 1 file changed, 2 insertions(+)

waheedk@User-PC MINGW32 ~/git-dir/learngit (master)
$ git log --oneline
88227f4 (HEAD -> master) modification by master
1dcc328 (development) Merge branch 'development'
829b9b9 modified the file1.txt content
b6c739a new content by development branch
5af01f0 file content change in branch development
b1bc4bc (origin/master, origin/HEAD, pre-prod) Added 10 Empty File

waheedk@User-PC MINGW32 ~/git-dir/learngit (master)
$
```

Content in development branch:

```
waheedk@User-PC MINGW32 ~/git-dir/learngit (master)
$ git checkout development
Switched to branch 'development'

waheedk@User-PC MINGW32 ~/git-dir/learngit (development)
$ vi development.txt

waheedk@User-PC MINGW32 ~/git-dir/learngit (development)
$ git add .

waheedk@User-PC MINGW32 ~/git-dir/learngit (development)
$ git commit -m "modification by development"
[development dd8a096] modification by development
 1 file changed, 1 insertion(+)

waheedk@User-PC MINGW32 ~/git-dir/learngit (development)
$ git log --oneline
dd8a096 (HEAD -> development) modification by development
1dcc328 Merge branch 'development'
829b9b9 modified the file1.txt content
b6c739a new content by development branch
5af01f0 file content change in branch development
b1bc4bc (origin/master, origin/HEAD, pre-prod) Added 10 Empty File
```

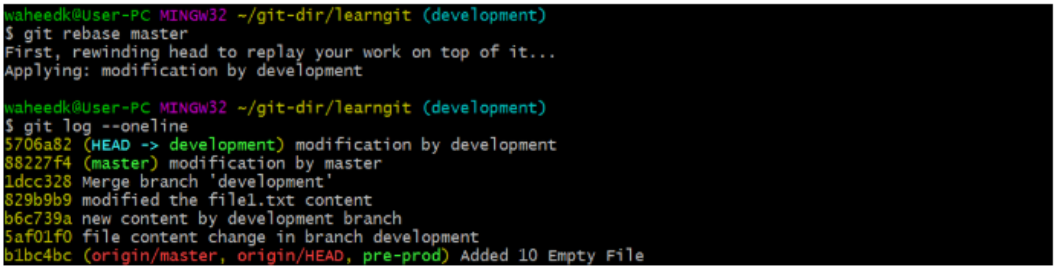
Content After rebase:

```
waheedk@User-PC MINGW32 ~/git-dir/learngit (development)
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: modification by development

waheedk@User-PC MINGW32 ~/git-dir/learngit (development)
$ git log --oneline
5706a82 (HEAD -> development) modification by development
88227f4 (master) modification by master
1dcc328 Merge branch 'development'
829b9b9 modified the file1.txt content
b6c739a new content by development branch
5af01f0 file content change in branch development
b1bc4bc (origin/master, origin/HEAD, pre-prod) Added 10 Empty File
```

Pulling Via Rebase

he --rebase option can be used to ensure a linear history by preventing unnecessary merge commits. Many developers prefer rebasing over merging, since it's like saying, "I want to put my changes on top of what everybody else has done." In this sense, using git pull with the --rebase flag is even more like svn update than a plain git pull. In fact, pulling with --rebase is such a common workflow that there is a dedicated configuration option for it:

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



Github SSH login

Generating SSH keys for github

1. Open Terminal.

2. Paste the command below, substituting in your GitHub email address.

ssh-keygen -t rsa -b 4096 -C "your_email@example.com"

3. When you're prompted to "Enter a file in which to save the key,"

Give below mentioned path

/home/USERNAME/.ssh/id_rsa_github

Note: USERNAME in above path is you Linux system user with which you have logged in.

Generating public/private rsa key pair.

Enter a file in which to save the key

(/home/USERNAME/.ssh/id_rsa):/home/USERNAME/.ssh/id_rsa_DO_github

4. Hit enter when it asks to enter passphrase

Enter passphrase (empty for no passphrase): [Type a passphrase]

Enter same passphrase again: [Type passphrase again]

Set SSH Private Key For Github.Com Login

1. Go to users ssh directory

cd ~/.ssh

ls

2. Open or create config file and update it with below mentioned content

vi config

Host github.com

HostName github.com

IdentityFile ~/.ssh/id_rsa_DO_github

User git

IdentitiesOnly yes

Add SSH Public Key In Github Account

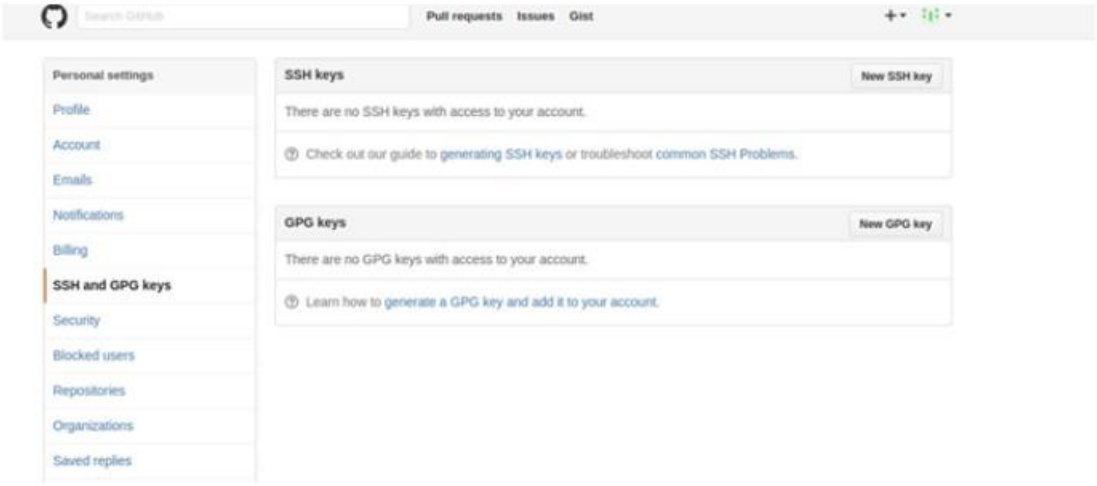
1. Copy public key

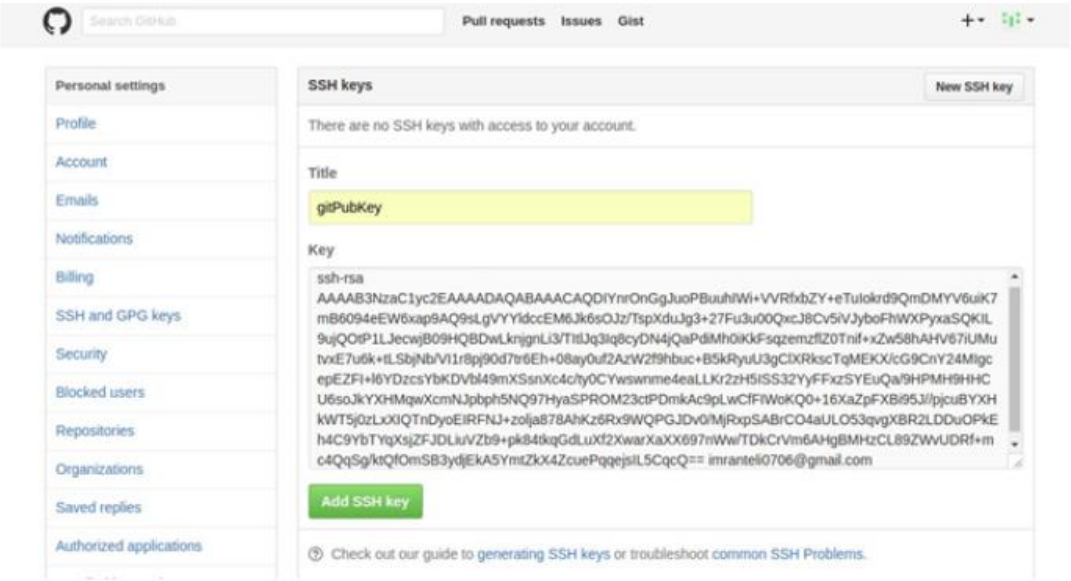
cat ~/.ssh/id_rsa_DO_github.pub

2. Login to github with the same email id you provided while create ssh keys

Click on settings => SSH and GPG keys => New SSH key => Give a name => Paste the public key content => Add SSH keys


Pull requests
Issues
Gist
Signed in as
DevImranOps
Your profile
Your stars
Explore
Integrations
Help
Settings
Sign out
Learn Git and GitHub without any code!
Using the Hello World guide, you'll create a repository, start a branch, write comments, and open a pull request.
Read the guide
Start a project


Pull requests
Issues
Gist
Personal settings
Profile
Account
Emails
Notifications
Billing
SSH and GPG keys
Security
Blocked users
Repositories
Organizations
Saved replies
SSH keys
New SSH key
There are no SSH keys with access to your account.
Check out our guide to generating SSH keys or troubleshoot common SSH Problems.
GPG keys
New GPG key
There are no GPG keys with access to your account.
Learn how to generate a GPG key and add it to your account.



3. Test the login

ssh -T git@github.com

You should get a reply as below

Hi Username! You've successfully authenticated, but GitHub does not provide shell access.

Summary

1. Version control systems are a category of software tools that help a software team manage changes to source code over time.

2. Version control software keeps track of every modification to the code in a special kind of database. 3. Git is a Distributed VCS, a category known as DVCS, more on that later. Like many of the most popular VCS systems available today, Git is free and open source.

4. Git a retain long-term change history of every file.Traceability, being able to trace each change made to the software and connect it to project management and bug tracking software such as JIRA.

5. Branching and merging. Having team members work concurrently is a no-brainer, but even individuals working on their own can benefit from the ability to work on independent streams of changes.