git

CONTENTS :

GIT :

Git is a distributed version-control system for tracking changes in source code during software development. It is designed for coordinating work among programmers, but it can be used to track changes in any set of files. Its goals include speed, data integrity, and support for distributed, non-linear workflows.

Git was created by Linus Torvalds in 2005 for development of the Linux kernel, with other kernel developers contributing to its initial development. Its current maintainer since 2005 is Junio Hamano.

As with most other distributed version-control systems, and unlike most client–server systems, every Git directory on every computer is a full-fledged repository with complete history and full version-tracking abilities, independent of network access or a central server.

Git is free and open-source software distributed under the terms of the GNU General Public License version 2.

History :

Git development began in April 2005, after many developers of the Linux kernel gave up access to BitKeeper, a proprietary source-control management (SCM) system that they had formerly used to maintain the project. The copyright holder of BitKeeper, Larry McVoy, had withdrawn free use of the product after claiming that Andrew Tridgell had reverse-engineered the BitKeeper protocols. (The same incident would also spur the creation of another version-control system, Mercurial.)

Linus Torvalds wanted a distributed system that he could use like BitKeeper, but none of the available free systems met his needs. Torvalds cited an example of a source-control management system needing 30 seconds to apply a patch and update all associated metadata, and noted that this would not scale to the needs of Linux kernel development, where synchronizing with fellow maintainers could require 250 such actions at once.

For his design criteria, he specified that patching should take no more than three seconds, and added three more points:

- Take Concurrent Versions System (CVS) as an example .
- Support a distributed, BitKeeper-like workflow.
- Include very strong safeguards against corruption, either accidental or malicious.

These criteria eliminated every then-extant version-control system, so immediately after the 2.6.12-rc2 Linux kernel development release, Torvalds set out to write his own.

The development of Git began on 3 April 2005. Torvalds announced the project on 6 April;it became self-hosting as of 7 April. The first merge of multiple branches took place on 18 April.

Torvalds achieved his performance goals. on 29 April, the nascent Git was benchmarked recording patches to the Linux kernel tree at the rate of 6.7 patches per second. On 16 June Git managed the kernel 2.6.12 release.

Torvalds turned over maintenance on 26 July 2005 to Junio Hamano, a major contributor to the project. Hamano was responsible for the 1.0 release on 21 December 2005 and remains the project's maintainer.

1.1 NAMING :

Torvalds 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'. The man page describes Git as "the stupid content tracker".

“ The name "git" was given by Linus Torvalds when he wrote the very first version. He described the tool as "the stupid content tracker" and the name as (depending on your way):

- random three-letter combination that is pronounceable, and not actually used by any common UNIX command. The fact that it is a mispronunciation of "get" may or may not be relevant.

- stupid. contemptible and despicable. simple. Take your pick from the dictionary of slang.

- "global information tracker": you're in a good mood, and it actually works for you. Angels sing, and a light suddenly fills the room.

- "goddamn idiotic truckload of sh*t": when it breaks. “

2 DESIGN :

Git's design was inspired by BitKeeper and Monotone. Git was originally designed as a low-level version-control system engine, on top of which others could write front ends, such as Cogito or StGIT. The core Git project has since become a complete version-control system that is usable directly.While strongly influenced by BitKeeper, Torvalds deliberately avoided conventional approaches, leading to a unique design.

2.1 Characteristics :

Git's design is a synthesis of Torvalds's experience with Linux in maintaining a large distributed development project, along with his intimate knowledge of file system performance gained from the same project and the urgent need to produce a working system in short order.

STRONG SUPPORT FOR NON-LINEAR DEVELOPMENT :

Git supports rapid branching and merging, and includes specific tools for visualizing and navigating a non-linear development history. In Git, a core assumption is that a change will be merged more often than it is written, as it is passed around to various reviewers. In Git, branches are very lightweight: a branch is only a reference to one commit. With its parental commits, the full branch structure can be constructed.

DISTRIBUTED DEVELOPMENT :

Like Darcs, BitKeeper, Mercurial, SVK, Bazaar, and Monotone, Git gives each developer a local copy of the full development history, and changes are copied from one such repository to another. These changes are imported as added development branches and can be merged in the same way as a locally developed branch.

COMPATIBILITY WITH EXISTENT SYSTEMS AND PROTOCOLS :

Repositories can be published via Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), rsync (removed in Git 2.8.0), or a Git protocol over either a plain socket, or Secure Shell (ssh). Git also has a CVS server emulation, which enables the use of extant CVS clients and IDE plugins to access Git repositories. Subversion and svk repositories can be used directly with git-svn.

EFFICIENT HANDLING OF LARGE PROJECTS :

Torvalds has described Git as being very fast and scalable, and performance tests done by Mozilla showed that it was an order of magnitude faster than some version-control systems; fetching version history from a locally stored repository can be one hundred times faster than fetching it from the remote server.

CRYPTOGRAPHIC AUTHENTICATION OF HISTORY :

The Git history is stored in such a way that the ID of a particular version (a *commit* in Git terms) depends upon the complete development history leading up to that commit. Once it is published, it is not possible to change the old versions without it being noticed. The structure is similar to a Merkle tree, but with added data at the nodes and leaves. (Mercurial and Monotone also have this property.)

TOOLKIT-BASED DESIGN :

Git was designed as a set of programs written in C and several shell scripts that provide wrappers around those programs.Although most of those scripts have since been rewritten in C for speed and portability, the design remains, and it is easy to chain the components together.

PLUGGABLE MERGE STRATEGIES :

As part of its toolkit design, Git has a well-defined model of an incomplete merge, and it has multiple algorithms for completing it, culminating in telling the user that it is unable to complete the merge automatically and that manual editing is needed.

GARBAGE ACCUMULATES UNTIL COLLECTED :

Aborting operations or backing out changes will leave useless dangling objects in the database. These are generally a small fraction of the continuously growing history of wanted objects. Git will automatically perform garbage collection when enough loose objects have been created in the repository. Garbage collection can be called explicitly using `git gc –prune`.

PERIODIC EXPLICIT OBJECT PACKING :

Git stores each newly created object as a separate file. Although individually compressed, this takes a great deal of space and is inefficient. This is solved by the use of *packs* that store a large number of objects delta-compressed among themselves in one file (or network byte stream) called a *packfile*. Packs are compressed using the heuristic that files with the same name are probably similar, but do not depend on it for correctness. A corresponding index file is created for each packfile, telling the offset of each object in the packfile. Newly created objects (with newly added history) are still stored as single objects, and periodic repacking is needed to maintain space efficiency. The process of packing the repository can be very computationally costly. By allowing objects to exist in the repository in a loose but quickly generated format, Git allows the costly pack operation to be deferred until later, when time matters less, e.g., the end of a work day. Git does periodic repacking automatically, but manual repacking is also possible with the `git gc` command. For data integrity, both the packfile and its index have an SHA-1 checksum inside, and the file name of the packfile also contains an SHA-1 checksum. To check the integrity of a repository, run the `git fsck` command.

Another property of Git is that it snapshots directory trees of files. The earliest systems for tracking versions of source code, Source Code Control System (SCCS) and Revision Control System (RCS), worked on individual files and emphasized the space savings to be gained from interleaved deltas (SCCS) or delta encoding (RCS) the (mostly similar) versions. Later revision-control systems maintained this notion of a file having an identity across multiple revisions of a project. However, Torvalds rejected this concept. Consequently, Git does not explicitly record file revision relationships at any level below the source-code tree.

These implicit revision relationships have some significant consequences:

- It is slightly more costly to examine the change history of one file than the whole project.To obtain a history of changes affecting a given file, Git must walk the global history and then determine whether each change modified that file. This method of examining history does, however, let Git produce with equal efficiency a single history showing the changes to an arbitrary set of files. For example, a subdirectory of the source tree plus an associated global header file is a very common case.

- Renames are handled implicitly rather than explicitly. A common complaint with CVS is that it uses the name of a file to identify its revision history, so moving or renaming a file is not possible without either interrupting its history or renaming the history and thereby making the history inaccurate. Most post-CVS revision-control systems solve this by giving a file a unique long-lived name (analogous to an inode number) that survives renaming. Git does not record such an identifier, and this is claimed as an advantage. Source code files are sometimes split or merged, or simply renamed, and recording this as a simple rename would freeze an inaccurate description of what happened in the (immutable) history. Git addresses the issue by detecting renames while browsing the history of snapshots rather than recording it when making the snapshot. (Briefly, given a file in revision N, a file of the same name in revision $N-1$ is its default ancestor. However, when there is no like-named file in revision $N-1$, Git searches for a file that existed only in revision $N-1$ and is very similar to the new file.) However, it does require more CPU-intensive work every time the history is reviewed, and several options to adjust the heuristics are available. This mechanism does not always work; sometimes a file that is renamed with changes in the same commit is read as a deletion of the old file and the creation of a new file. Developers can work around this limitation by committing the rename and the changes separately.

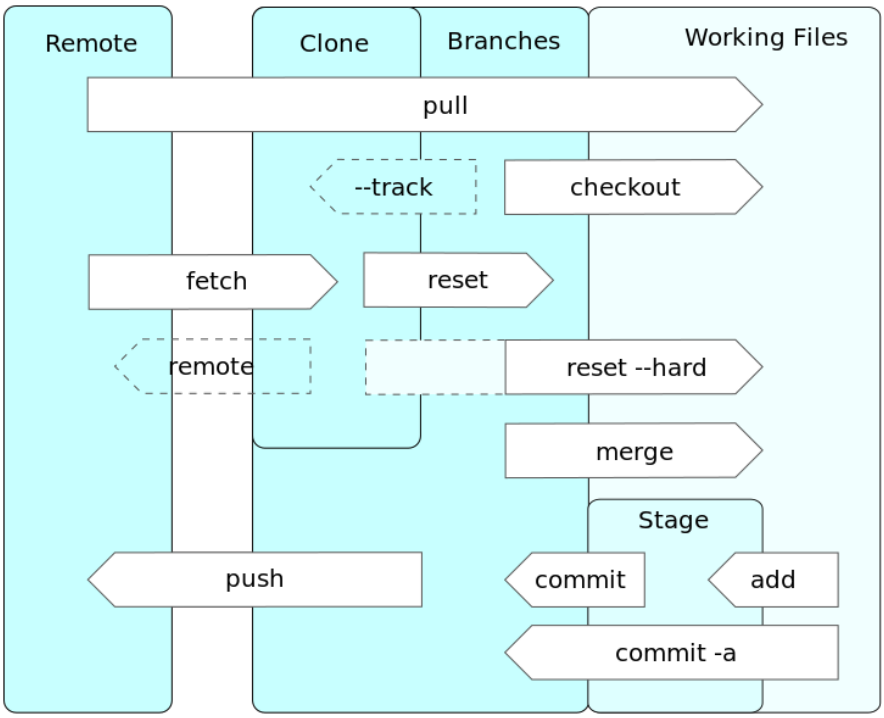
Git implements several merging strategies; a non-default strategy can be selected at merge time:

- *resolve*: the traditional three-way merge algorithm.
- *recursive*: This is the default when pulling or merging one branch, and is a variant of the three-way merge algorithm.

> ***When there are more than one common ancestors that can be used for three-way merge, it creates a merged tree of the common ancestors and uses that as the reference tree for the three-way merge. This has been reported to result in fewer merge conflicts without causing mis-merges by tests done on prior merge commits taken from Linux 2.6 kernel development history. Also, this can detect and handle merges involving renames.***
>
> — *Linus Torvalds*

2.2 DATA STRUCTURES :

Git's primitives are not inherently a source-code management system. Torvalds explains:



> In many ways you can just see git as a filesystem – it's content-addressable, and it has notion of versioning, but I really really designed it coming at the problem from the viewpoint of a *filesystem* person (hey, kernels is what I do), and I actually have absolutely *zero* interest in creating a traditional SCM system.

From this initial design approach, Git has developed the full set of features expected of traditional SCM,with features mostly being created as needed, then refined and extended over time.

Git has two data structures: a mutable *index* (also called *stage* or *cache*) that caches information about the working directory and the next revision to be committed; and an immutable, append-only *object database*.

The index serves as connection point between the object database and the working tree.

The object database contains four types of objects:

- A *blob* (binary large object) is the content of a file. Blobs have no proper file name, time stamps, or other metadata. (A blob's name internally is a hash of its content.)
- A *tree* object is the equivalent of a directory. It contains a list of file names, each with some type bits and a reference to a blob or tree object that is that file, symbolic link, or directory's contents. These objects are a snapshot of the source tree. (In whole, this comprises a Merkle tree, meaning that only a single hash for the root tree is sufficient and actually used in commits to precisely pinpoint to the exact state of whole tree structures of any number of sub-directories and files.)
- A *commit* object links tree objects together into a history. It contains the name of a tree object (of the top-level source directory), a time stamp, a log message, and the names of zero or more parent commit objects.
- A *tag* object is a container that contains a reference to another object and can hold added meta-data related to another object. Most commonly, it is used to store a digital signature of a commit object corresponding to a particular release of the data being tracked by Git.

Each object is identified by a SHA-1 hash of its contents. Git computes the hash and uses this value for the object's name. The object is put into a directory matching the first two characters of its hash. The rest of the hash is used as the file name for that object.

Git stores each revision of a file as a unique blob. The relationships between the blobs can be found through examining the tree and commit objects. Newly added objects are stored in their entirety using zlib compression. This can consume a large amount of disk space quickly, so objects can be combined into *packs*, which use delta compression to save space, storing blobs as their changes relative to other blobs.

2.3 REFERENCES :

Every object in the Git database that is not referred to may be cleaned up by using a garbage collection command or automatically. An object may be referenced by another object or an explicit reference. Git knows different types of references. The commands to create, move, and delete references vary. "git show-ref" lists all references. Some types are:

- *heads*: refers to an object locally,
- *remotes*: refers to an object which exists in a remote repository,
- *meta*: e.g. a configuration in a bare repository, user rights; the refs/meta/config namespace was introduced resp gets used by Gerrit,

3.0 Implementations :

Git is primarily developed on Linux, although it also supports most major operating systems, including BSD, Solaris, macOS, and Windows.

The first Windows port of Git was primarily a Linux-emulation framework that hosts the Linux version. Installing Git under Windows creates a similarly named Program Files directory containing the MinGW port of the GNU Compiler Collection, Perl 5, msys2.0 (itself a fork of Cygwin, a Unix-like emulation environment for Windows) and various other Windows ports or emulations of Linux utilities and libraries. Currently native Windows builds of Git are distributed as 32- and 64-bit installers.

The JGit implementation of Git is a pure Java software library, designed to be embedded in any Java application. JGit is used in the Gerrit code-review tool and in EGit, a Git client for the Eclipse IDE.

The Dulwich implementation of Git is a pure Python software component for Python 2.7, 3.4 and 3.5.

The libgit2 implementation of Git is an ANSI C software library with no other dependencies, which can be built on multiple platforms, including Windows, Linux, macOS, and BSD. It has bindings for many programming languages, including Ruby, Python, and Haskell. JS-Git is a JavaScript implementation of a subset of Git.

4 Git server :

As Git is a distributed version-control system, it could be used as a server out of the box. Dedicated Git server software helps, amongst other features, to add access control, display the contents of a Git repository via the web, and help managing multiple repositories.

Remote file store and shell access: A Git repository can be cloned to a shared file system and accessed by other persons.

It can also be accessed via remote shell just by having the Git software installed and allowing a user to log in.[58] Git servers typically listen on TCP port 9418.

4.1 OPEN SOURCE :

- Gitolite, scripts on top of git software to provide fine-grained access control.
- Gerrit, a git server configurable to support code reviews and providing access via ssh, an integrated Apache MINA or OpenSSH, or an integrated Jetty web server. Gerrit provides integration for LDAP, Active Directory, OpenID, OAuth, Kerberos/GSSAPI, X509 https client certificates. With Gerrit 3.0 all configurations will be stored as git repositories, no database required to run. Gerrit has a pull-request feature implemented in its core but lacks a GUI for it.
- Phabricator, a spin off from Facebook. As Facebook is using primarily Mercurial, the git support is not as prominent.
- Trac, supporting git, Mercurial, and Subversion with a modified BSD license.
- Rhodecode Community Edition (CE), supporting git, Mercurial and Subversion with a AGPLv3 license.
- Kallithea, supporting both git and Mercurial, developed in Python with GPL license.
- Some of the other FLOSS full solutions for self-hosting are Gogs and Gitea, both developed in Go language with MIT license

4.2 AS SERVICE :

Best known are probably GitHub and Bitbucket offerings, but many others are available, like GitLab, GerritForge, etc.

5 Adoption :

The Eclipse Foundation reported in its annual community survey that as of May 2014, Git is now the most widely used source-code management tool, with 42.9% of professional software developers reporting that they use Git as their primary source-control system compared with 36.3% in 2013, 32% in 2012; or for Git responses excluding use of GitHub. 33.3% in 2014, 30.3% in 2013, 27.6% in 2012 and 12.8% in 2011.

Open-source directory Black Duck Open Hub reports a similar uptake among open-source projects. The Stack Overflow developer survey reported in 2015 that 69.3% of developers use Git; 36.9% use Subversion; 12.2% use TFS; and 7.9% use Mercurial.

The UK IT jobs website itjobswatch.co.uk reports that as of late September 2016, 29.27% of UK permanent software development job openings have cited Git, ahead of 12.17% for Microsoft Team Foundation Server, 10.60% for Subversion, 1.30% for Mercurial, and 0.48% for Visual SourceSafe.

Since February 2017, Microsoft has been in the process of migrating Microsoft Windows development to Git, migrating from Perforce. In order to handle the size of the Windows source-code tree, Microsoft was required to develop customizations to the software, including Git Virtual File System (GVFS), a system which allows cloned repositories to use placeholders whose contents are downloaded only once a file is accessed.

5.1 Extensions :

There are many *Git extensions*, like Git LFS, which started as an extension to Git in the GitHub community and now is widely used by other repositories. The two projects are independently developed and maintained by different people, but in some point in the future a widely used extension can be merged to Git.

6 Security :

Git does not provide access-control mechanisms, but was designed for operation with other tools that specialize in access control.

On 17 December 2014, an exploit was found affecting the Windows and macOS versions of the Git client. An attacker could perform arbitrary code execution on a target computer with Git installed by creating a malicious Git tree (directory) named *.git* (a directory in Git repositories that stores all the data of the repository) in a different case (such as .GIT or .Git, needed because Git doesn't allow the all-lowercase version of *.git* to be created manually) with malicious files in the *.git/hooks* subdirectory (a folder with executable files that Git runs) on a repository that the attacker made or on a repository that the attacker can modify.

If a Windows or Mac user *pulls* (downloads) a version of the repository with the malicious directory, then switches to that directory, the .git directory will be overwritten (due to the case-insensitive trait of the Windows and Mac filesystems) and the malicious executable files in *.git/hooks* may be run, which results in the attacker's commands being executed. An attacker could also modify the *.git/config* configuration file, which allows the attacker to create malicious Git aliases (aliases for Git commands or external commands) or modify extant aliases to execute malicious commands when run.

The vulnerability was patched in version 2.2.1 of Git, released on 17 December 2014, and announced on the next day.

Git version 2.6.1, released on 29 September 2015, contained a patch for a security vulnerability (CVE-2015-7545)that allowed arbitrary code execution.

The vulnerability was exploitable if an attacker could convince a victim to clone a specific URL, as the arbitrary commands were embedded in the URL itself.

An attacker could use the exploit via a man-in-the-middle attack if the connection was unencrypted, as they could redirect the user to a URL of their choice.

Recursive clones were also vulnerable, since they allowed the controller of a repository to specify arbitrary URLs via the gitmodules file.

Git uses SHA-1 hashes internally. Linus Torvalds has responded that the hash was mostly to guard against accidental corruption, and the security a cryptographically secure hash gives was just an accidental side effect, with the main security being signing elsewhere.

7 Install Git on Linux :

Debian / Ubuntu (apt-get)

Git packages are available via apt:

1. From your shell, install Git using apt-get:

   $ sudo apt-get update
   $ sudo apt-get install git

2. Verify the installation was successful by typing `git --version`:

   $ git –version

3. Configure your Git username and email using the following commands, replacing Emma's name with your own. These details will be associated with any commits that you create:

   $ git config --global user.name "Emma Paris"
   $ git config --global user.email "Email address "

8 vim default editor git :

step 1: **git config --global core.editor “vim”**

Step 2: **git config --global merge.tool vimdiff**

step 3: **git config --global diff.tool vimdiff**

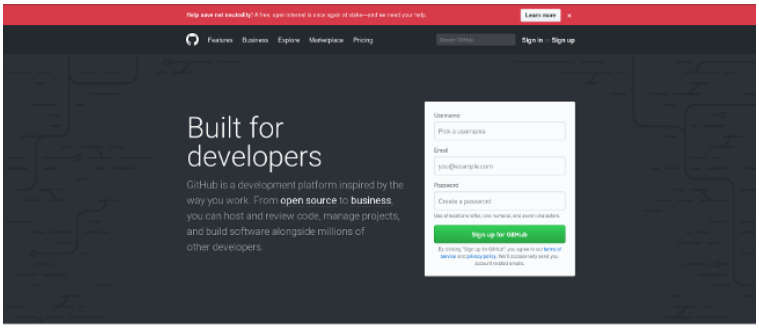
step 4: when you run the following command

git config - -list

you can see the serring are done.

9.CLONING A REPOSITORY :

Step 1 : Create a GitHub account

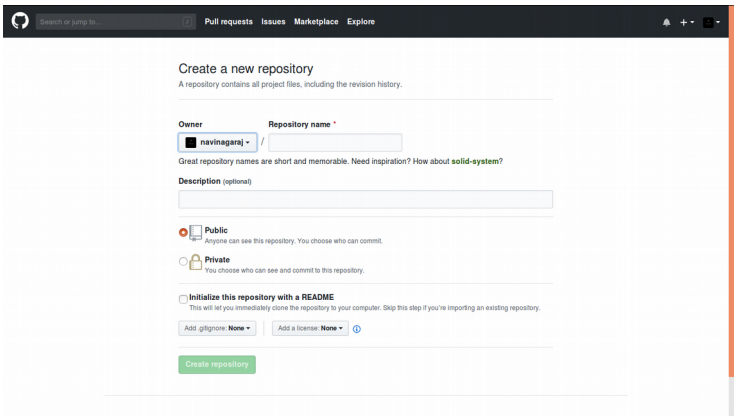


The easiest way to get started is to create an account on GitHub.com (it's free).

Pick a username (e.g., octocat123), enter your email address and a password, and click Sign up for GitHub. Once you are in, it will look something like this

Step 2 : Create a new repository

A repository is like a place or a container where something is stored; in this case we're creating a Git repository to store code. To create a new repository, select New Repository from the + sign dropdown menu (you can see I've selected it in the upper-right corner in the image above).



Enter a name for your repository (e.g, "Demo") and click **Create Repository**. Don't worry about changing any other options on this page.

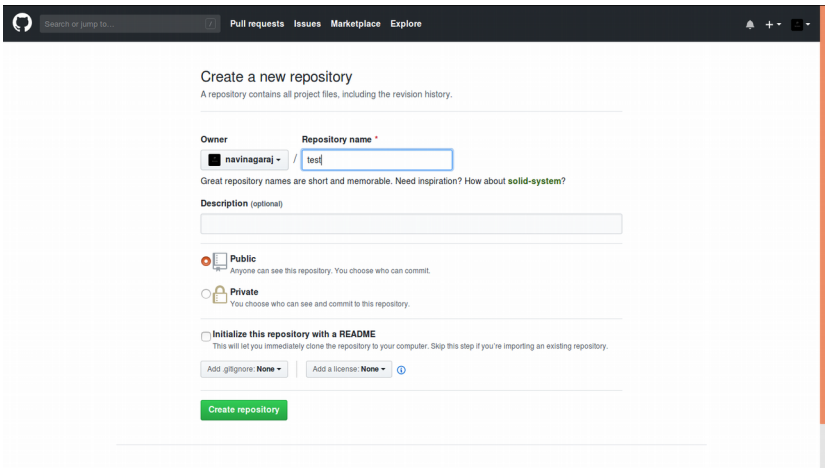
Congratulations! You have set up your first repo on GitHub.com.

Step 3 : Create a file

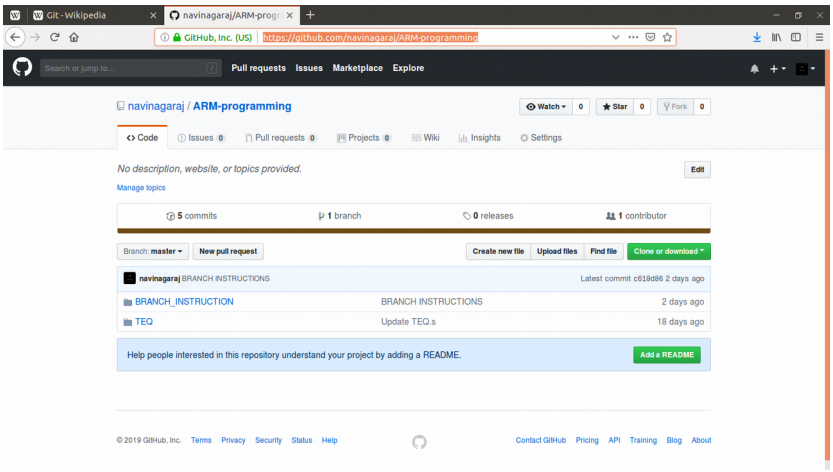
Once your repo is created, it wil look like this:

Don't panic, it's simpler than it looks. Stay with me. Look at the section that starts "...or create a new repository on the command line," and ignore the rest for now.

Open the *Terminal* program on your computer.



Step 4 : Copy a URL



copy the URL and open your terminal (ctrl+alt+t).

Step 5 : Clone a *repository*

open terminal and past the URL

$ clone “ URL past hear”.

$ ls

we created one repository has appeared in the ls its successfully executed and enter the folder are directory you can do any thing

Step 6 : 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.

$ git add file name

Step 7 : git commit

The git commit command captures a snapshot of the project's currently staged changes. Committed snapshots can be thought of as “safe” versions of a project—Git will never change them unless you explicitly ask it to. Prior to the execution of git commit, The git add command is used to promote or 'stage' changes to the project that will be stored in a commit. These two commands git commit and git add are two of the most frequently used.

$ git commit

Step 8 : git push

The git push command is used to upload local repository content to a remote repository. 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. Remote branches are configured using the git remote command. Pushing has the potential to overwrite changes, caution should be taken when pushing. These issues are discussed below.

$ git push