

IX. Version Control Systems

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 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.

1. 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.

2. VCS Terminologies

Basic Setup

- **Repository (repo):** The database storing the files.
- **Server:** The computer storing the repo.
- **Client:** The computer connecting to the repo.
- **Working Set/Working Copy:** Your local directory of files, where you make changes.
- **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

- a) **Add/Push:** Put a file into the repo for the first time, i.e. begin tracking it with Version Control.
- b) **Revision:** What version a file is on (v1, v2, v3, etc.).
- c) **Head:** The latest revision in the repo.
- d) **Check out/Pull/Fetch:** Download a file from the repo.
- e) **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.
- f) **Check In Message:** A short message describing what was changed.
- g) **Changelog/History:** A list of changes made to a file since it was created.
- h) **Update/Sync:** Synchronize your files with the latest from the repository. This lets you grab the latest revisions of all files.
- i) **Revert:** Throw away your local changes and reload the latest version from the repository.

Advanced Actions

- ❖ **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?”).
- ❖ **Diff/Change/Delta:** Finding the differences between two files. Useful for seeing what changed between revisions.
- ❖ **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)
- ❖ **Conflict:** When pending changes to a file contradict each other (both changes cannot be applied).
- ❖ **Resolve:** Fixing the changes that contradict each other and checking in the correct version.
- ❖ **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.

3. Famous Version Control Systems.

1. 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.

2. SVN

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 Tortoissvn 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.

3. 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.

4. 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.

5. 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.

4. 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.

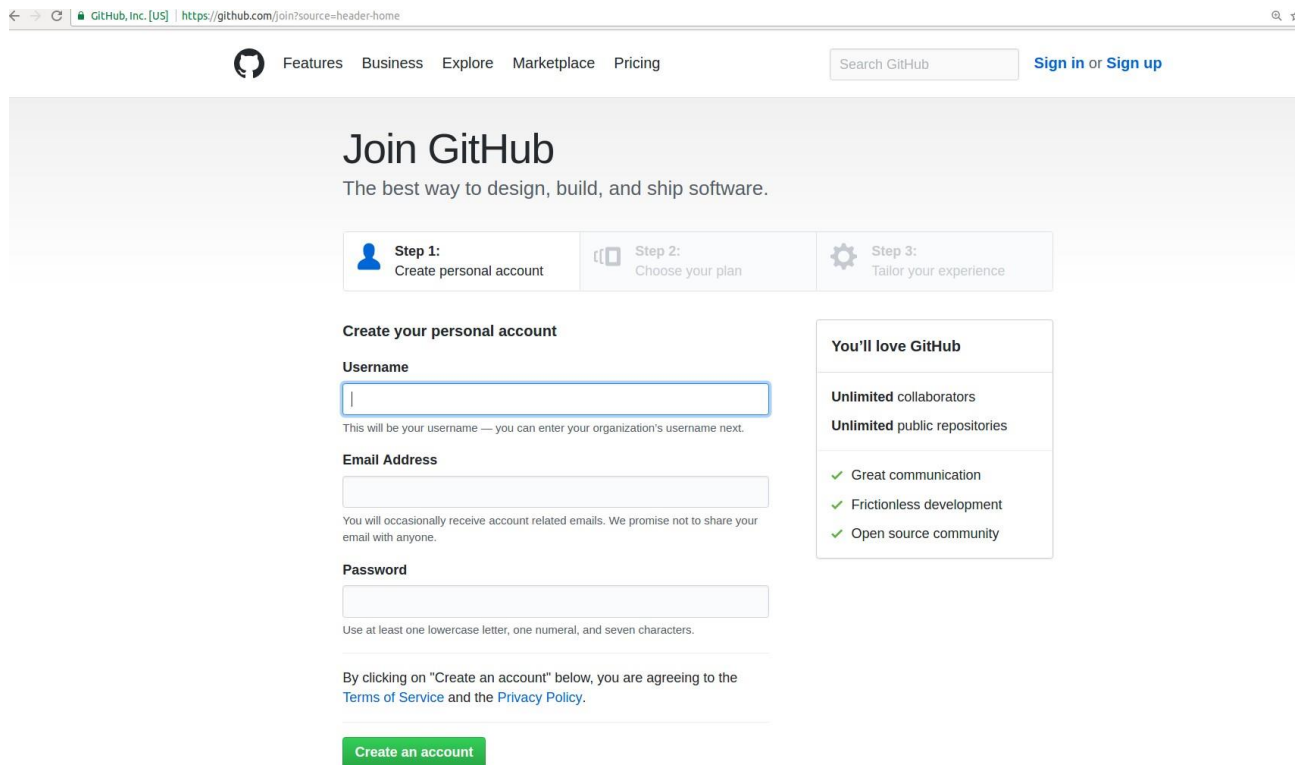
In addition to being distributed, Git has been designed with performance, security and flexibility in mind.

5. Why use git?

- Its fast
- You don't need access to a server
- Amazingly good at merging simultaneous changes
- Everyone's using it

6. Git Quick Setup

1. Sign up for the github account.



The screenshot shows the GitHub sign-up page. At the top, there's a navigation bar with links for Features, Business, Explore, Marketplace, and Pricing. A search bar and 'Sign in or Sign up' link are also present. The main heading is 'Join GitHub' with the tagline 'The best way to design, build, and ship software.' Below this, a progress bar indicates three steps: Step 1: Create personal account (active), Step 2: Choose your plan, and Step 3: Tailor your experience. The 'Create your personal account' section includes fields for Username, Email Address, and Password, each with a brief description of the requirements. To the right, a box titled 'You'll love GitHub' lists benefits: Unlimited collaborators, Unlimited public repositories, Great communication, Frictionless development, and Open source community. At the bottom, a green 'Create an account' button is visible.

Join GitHub

The best way to design, build, and ship software.

Step 1: Create personal account

Step 2: Choose your plan

Step 3: Tailor your experience

Create your personal account

Username

This will be your username — you can enter your organization's username next.

Email Address

You will occasionally receive account related emails. We promise not to share your email with anyone.

Password

Use at least one lowercase letter, one numeral, and seven characters.

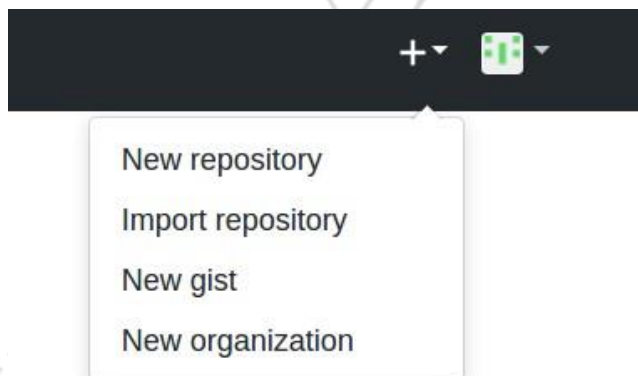
By clicking on "Create an account" below, you are agreeing to the [Terms of Service](#) and the [Privacy Policy](#).

Create an account

You'll love GitHub

- Unlimited collaborators
- Unlimited public repositories
- Great communication
- Frictionless development
- Open source community

2. Create public repository.



Create a new repository

A repository contains all the files for your project, including the revision history.

Owner **Repository name**

DevImranOps / learninggit ✓

Great repository names are short and memorable. Need inspiration? How about **musical-broccoli**.

Description (optional)

☒ **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

3. Copy the URL of the public repo.

Quick setup — if you've done this kind of thing before

or **HTTPS** **SSH**

We recommend every repository include a **README**, **LICENSE**, and **.gitignore**.

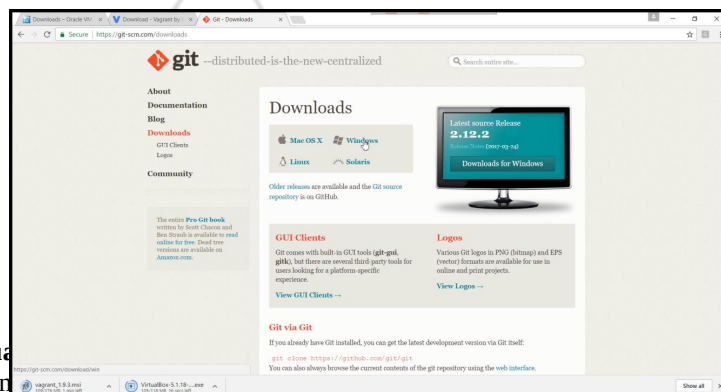
4. Install git tool on your system.

Windows:

Install git software

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

✓ Go to git scm download page, Select windows.



Visual

Flat n

Mail ID : online.visualpath@gmail.com, Website : www.visualpath.in.

No: - +91-970 445 5959, 961 824 5689 E-

- 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) or
```

5. Clone git repository on your local system

```
# git clone <Git Repo URL>
```

6. Create few directories and files in the Repo directory. (Directories should not be empty)

7. Go inside the repo directory

```
# cd <Repo name>
```

8. Update the index using the current content found in the working tree.

```
# git add .
```

9. Stores the current contents of the index in a new commit along with a log message from the user describing the changes.

```
# git commit -m "<Put some message>"
```

10. Push the changes to the github.

```
# git push origin master
```

11. Make changes to some files from github UI and pull the changes back to the local repo

imranteli first commit 97435e3 a minute ago

1 contributor

2 lines (1 sloc) 18 Bytes

1 testting git tool

Raw Blame History

Commit changes

update commit

Add an optional extended description...

☒ Commit directly to the master branch.

☐ Create a new branch for this commit and start a pull request. [Learn more about pull requests.](#)

Commit changes Cancel

git pull

12. Create new branch from master branch and pull the changes again to local repo.

Branch: master New pull request Create new file Upload files

Switch branches/tags

newtestbranch

Branches Tags

Create branch: newtestbranch from 'master'

git pull

Switch to new branch

```
# git status (Shows the current branch)
# git checkout <branch name>
# git status
```

13. Make some changes to files and push it back to the latest branch

```
# git add .
# git commit -m "<Some message>"
# git push origin <branchname>
```

14. Remove/Rename the content and push

```
# git rm <filename>
# git mv <filename> <newnameoffile>

# git commit -m "<message>"
# git push
```

Explore the UI as much as you can, check histories, commits, changes, content of files, edit file content, create multiple repositories.

Go through the below mentioned git repo and explore it, once you're through with above exercise

<https://github.com/wakaleo/gamgit>

7. Git in detail

8. Installing Git

Before you start using Git, you must make it available on your computer. Even if it's already installed, it's probably a good idea to update to the latest version. You can either install it as a package or via another installer, or download the source code and compile it yourself.

NOTE

This book was written using Git version **2.0.0**. Though most of the commands we use should work even in ancient versions of Git, some of them might not or might act slightly differently if you're using an older version. Since Git is quite excellent at preserving backwards compatibility, any version after 2.0 should work just fine.

Installing on Linux

If you want to install the basic Git tools on Linux via a binary installer, you can generally do so through the basic package-management tool that comes with your distribution. If you're on Fedora for example, you can use yum:

```
$ sudo yum install git-all
```

If you're on a Debian-based distribution like Ubuntu, try apt-get:

```
$ sudo apt-get install git-all
```

For more options, there are instructions for installing on several different Unix flavors on the Git website, at <http://git-scm.com/download/linux>.

Installing on Mac

There are several ways to install Git on a Mac. The easiest is probably to install the Xcode Command Line Tools. On Mavericks (10.9) or above you can do this simply by trying to run *git* from the Terminal the very first time. If you don't have it installed already, it will prompt you to install it.

If you want a more up to date version, you can also install it via a binary installer. An OSX Git installer is maintained and available for download at the Git website, at <http://git-scm.com/download/mac>.

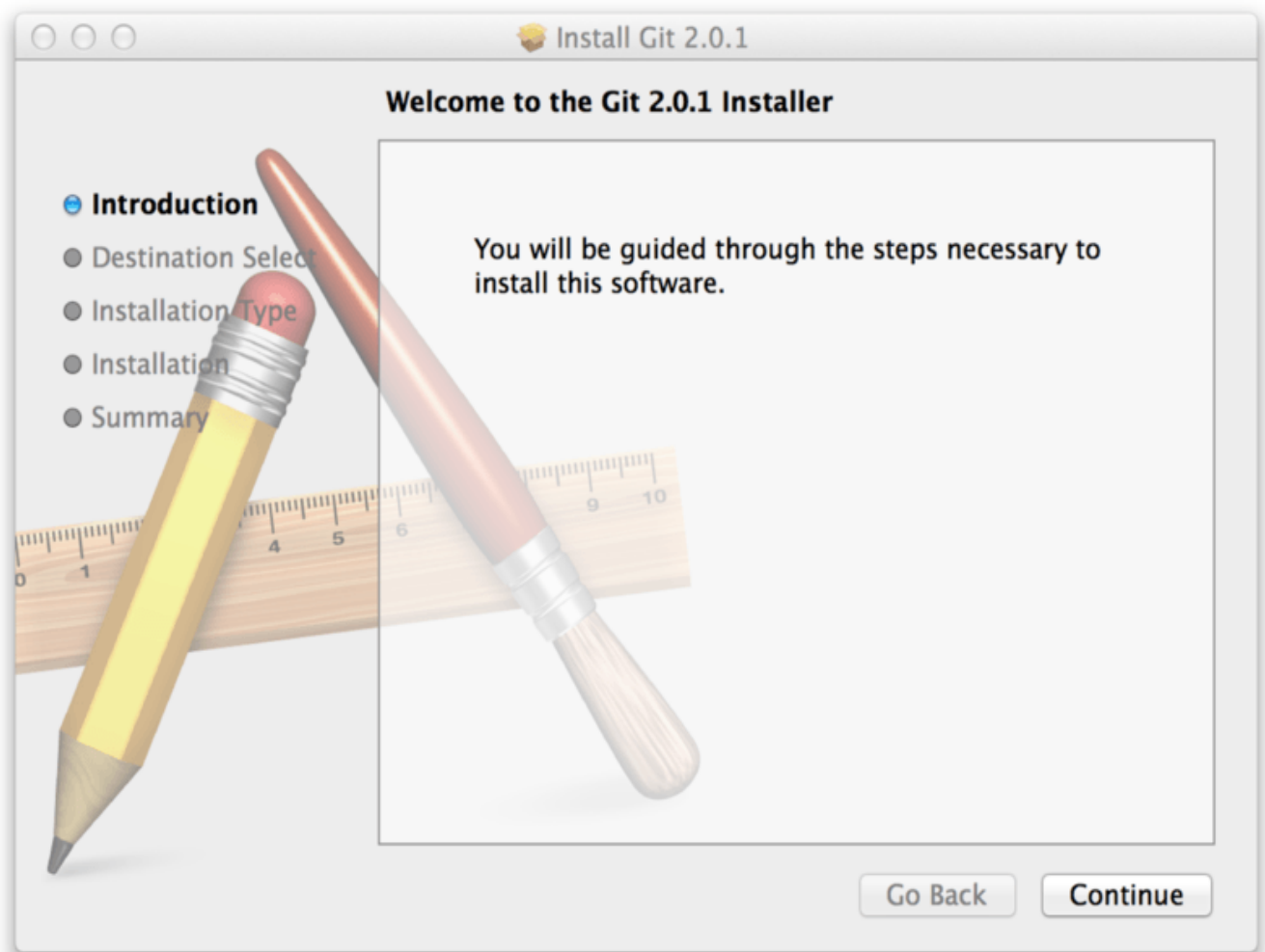


Figure 1-7. Git OS X Installer.

You can also install it as part of the GitHub for Mac install. Their GUI Git tool has an option to install command line tools as well. You can download that tool from the GitHub for Mac website, at <http://mac.github.com>.

Installing on Windows

There are also a few ways to install Git on Windows. The most official build is available for download on the Git website. Just go to <http://git-scm.com/download/win> and the download will start automatically. Note that this is a project called Git for Windows, which is separate from Git itself; for more information on it, go to <https://git-for-windows.github.io/>.

Another easy way to get Git installed is by installing GitHub for Windows. The installer includes a command line version of Git as well as the GUI. It also works well with Powershell, and sets up solid credential caching and sane CRLF settings. We'll learn more about those things a little later, but suffice it to say they're things you want. You can download this from the GitHub for Windows website, at <http://windows.github.com>.

9. 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

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.

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 .git folder to the current directory and makes it possible to start recording revisions of the project.

```
$ git init <directory>
```

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 <directory>
```

Initialize an empty Git repository, but omit the working directory. Shared repositories should always be created with the --bare flag (see discussion below). Conventionally, repositories initialized with

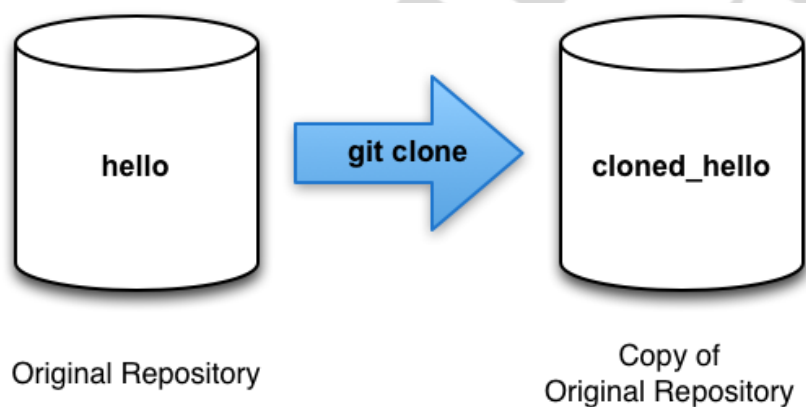
the `--bare` flag end in `.git`. For example, the bare version of a repository called `my-project` should be stored in a directory called `my-project.git`.

git clone

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.

■



Usage

```
git clone <repo>
```

Clone the repository located at <repo> onto the local machine. The original repository can be located on the local filesystem or on a remote machine accessible via HTTP or SSH.

```
git clone <repo> <directory>
```

Clone the repository located at <repo> into the folder called <directory> on the local machine.

Example

```
$ git clone https://github.com/wakaleo/game-of-life.git
```

git config

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.

Usage

```
$ git config user.name <name>
```

Define the author name to be used for all commits in the current repository. Typically, you'll want to use the --global flag to set configuration options for the current user.

```
$ git config --global user.name <name>
```

Define the author name to be used for all commits by the current user.

```
$ git config --global user.email <email>
```

Define the author email to be used for all commits by the current user.

```
$ git config --global alias.<alias-name> <git-command>
```

Create a shortcut for a Git command.

```
$ git config --system core.editor <editor>
```

Define the text editor used by commands like git commit for all users on the current machine. The <editor> argument should be the command that launches the desired editor (e.g., vi).


```
$ git config --global --edit
```

Open the global configuration file in a text editor for manual editing.

Discussion

All configuration options are stored in plaintext files, so the git config command is really just a convenient command-line interface. Typically, you'll only need to configure a Git installation the first time you start working on a new development machine, and for virtually all cases, you'll want to use the --global flag.

Git stores configuration options in three separate files, which lets you scope options to individual repositories, users, or the entire system:

<repo>/.git/config – Repository-specific settings.

~/.gitconfig – User-specific settings. This is where options set with the --global flag are stored.

\$(prefix)/etc/gitconfig – System-wide settings.

When options in these files conflict, local settings override user settings, which override system-wide. If you open any of these files, you'll see something like the following:

```
[user]
name = John Smith
email = john@example.com
[alias]
st = status
co = checkout
br = branch
up = rebase
ci = commit
[core]
editor = vim
```

You can manually edit these values to the exact same effect as git config.

Example

The first thing you'll want to do after installing Git is tell it your name/email and customize some of the default settings. A typical initial configuration might look something like the following:

```
# Tell Git who you are
$ git config --global user.name "John Smith"
$ git config --global user.email john@example.com
# Select your favorite text editor
$ git config --global core.editor vim
# Add some SVN-like aliases
$ git config --global alias.st status
$ git config --global alias.co checkout
$ git config --global alias.br branch
$ git config --global alias.up rebase
$ git config --global alias.ci commit
```

10. Saving changes

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 <file>
```

Stage all changes in <file> for the next commit.

```
$ git add <directory>
```

Stage all changes in <directory> for the next commit.

```
$ git add -p
```

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.

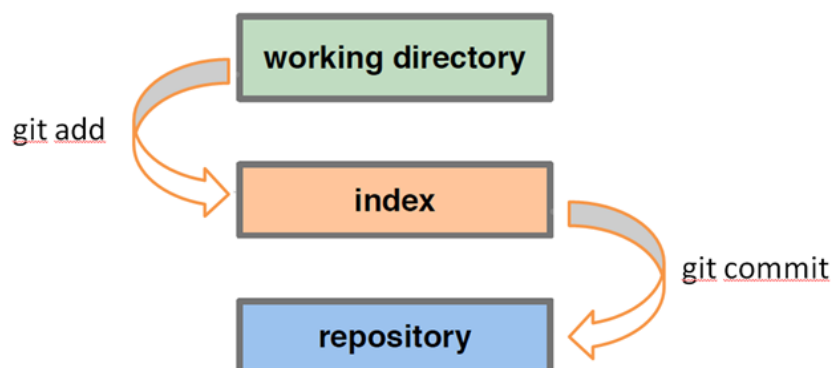
The git add command should not be confused with svn add, which adds a file to the repository. Instead, git add works on the more abstract level of changes. This means that git add needs to be called every time you alter a file, whereas svn add only needs to be

called once for each file. It may sound redundant, but this workflow makes it much easier to keep a project organized.

11. 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.



Example

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
```

Once you've got your project up-and-running, new files can be added by passing the path to git add:

```
$ git add hello.py  
$ git commit
```

The above commands can also be used to record changes to existing files. Again, Git doesn't differentiate between staging changes in new files vs. changes in files that have already been added to the repository.

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
```

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>"`

Commit the staged snapshot, but instead of launching a text editor, use `<message>` as the commit message.

```
$ git commit -a
```

Commit a snapshot of all changes in the working directory. This only includes modifications to tracked files (those that have been added with git add at some point in their history).

Example

The following example assumes you've edited some content in a file called `hello.py` and are ready to commit it to the project history. First, you need to stage the file with `git add`, then you can commit the staged snapshot.

```
$ git add hello.py
$ git commit
```

This will open a text editor (customizable via `git config`) asking for a commit message, along with a list of what's being committed:

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
```

```
#  
#modified: samplecode.py
```

Change the message displayed by samplecode.py

- Update the sayHello() function to output the user's name
- Change the sayGoodbye() function to a friendlier message

12. 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>
```

Visualpath Training & Consulting.

Flat no: 205, Nilgiri Block,Aditya Enclave, Ameerpet, Hyderabad, Phone No: - +91-970 445 5959, 961 824 5689 E-Mail ID : online.visualpath@gmail.com, Website : www.visualpath.in.

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

Discussion

Git is designed to give each developer an entirely isolated development environment. This means that information is not automatically passed back and forth between repositories. Instead, developers need to manually pull upstream commits into their local repository or manually push their local commits back up to the central repository. The git remote command is just an easier way to pass URLs to these "sharing" commands.

The origin Remote

When you clone a repository with git clone, it automatically creates a remote connection called origin pointing back to the cloned repository. This is useful for developers creating a local copy of a central repository, since it provides an easy way to pull upstream changes or publish local commits. This behaviour is also why most Git-based projects call their central repository origin.

13. 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.

14. 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 <remote>
```

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

```
$ git fetch <remote> <branch>
```

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

Discussion

Fetching is what you do when you want to see what everybody else has been working on. Since fetched content is represented as a remote branch, it has absolutely no effect on your local development work. This makes fetching a safe way to review commits before integrating them with your local repository. It's similar to svn update in that it lets you see how the central history has progressed, but it doesn't force you to actually merge the changes into your repository.

Remote Branches

Remote branches are just like local branches, except they represent commits from somebody else's repository. You can check out a remote branch just like a local one, but this puts you in a detached HEAD state (just like checking out an old commit). You can think of them as read-only branches. To view your remote branches, simply pass the -r flag to the git branch command. Remote branches are prefixed by the remote they belong to so that you don't mix them up with local branches. For example, the next code snippet shows the branches you might see after fetching from the origin remote:

```
git branch -r
# origin/master
# origin/develop
# origin/some-feature
```

Again, you can inspect these branches with the usual git checkout and git log commands. If you approve the changes a remote branch contains, you can merge it into a local branch with a normal git merge. So, unlike SVN, synchronizing your local repository with a remote repository is actually a two-step process: fetch, then merge. The git pull command is a convenient shortcut for this process.

Examples

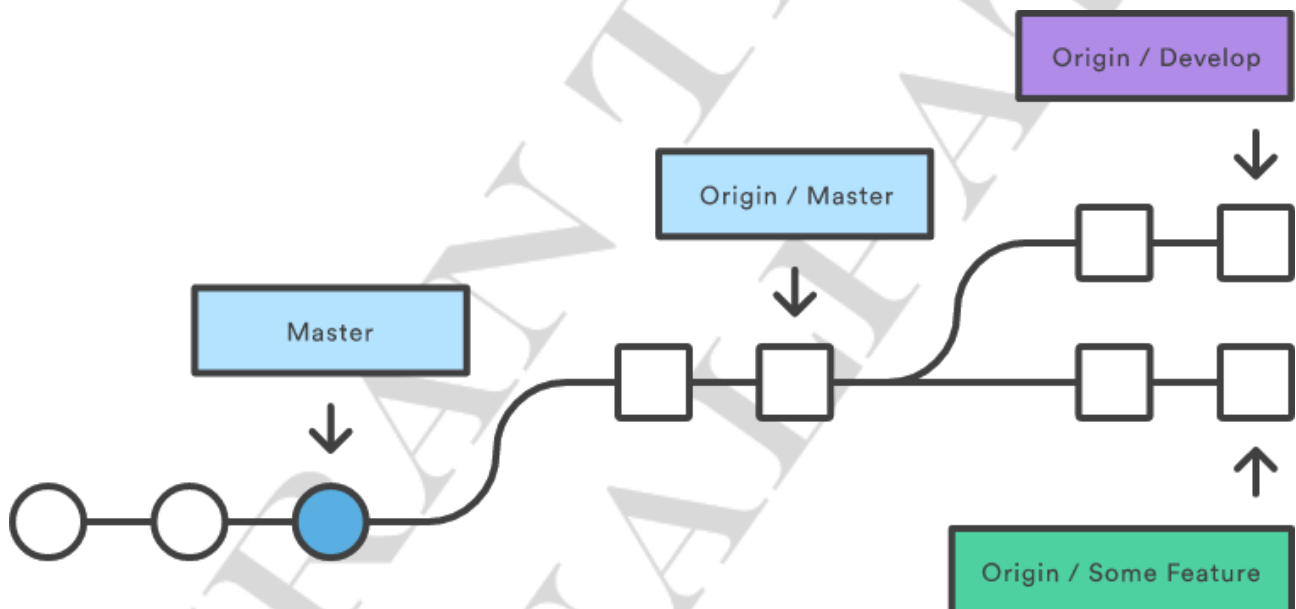
This example walks through the typical workflow for synchronizing your local repository with the central repository's master branch.

```
$ git fetch origin
```

This will display the branches that were downloaded:

```
a1e8fb5..45e66a4 master -> origin/master  
a1e8fb5..9e8ab1c develop -> origin/develop  
* [new branch] some-feature -> origin/some-feature
```

The commits from these new remote branches are shown as squares instead of circles in the diagram below. As you can see, git fetch gives you access to the entire branch structure of another repository.



To see what commits have been added to the upstream master, you can run a **git log** using **origin/master** as a filter

```
$ git log --oneline master..origin/master
```

To approve the changes and merge them into your local master branch with the following commands:

```
$ git checkout master  
$ git log origin/master
```

Then we can use **git merge origin/master**

```
$ git merge origin/master
```

The **origin/master** and **master** branches now point to the same commit, and you are synchronized with the upstream developments.

15. 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**.

Discussion

You can think of **git pull** as Git's version of **svn update**. It's an easy way to synchronize your local repository with upstream changes.

You start out thinking your repository is synchronized, but then **git fetch** reveals that **origin's** version of **master** has progressed since you last checked it. Then **git merge** immediately integrates the remote master into the local one:

16. Pulling via Rebase

The **--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 config --global branch.autosetuprebase always
```

After running that command, all git pull commands will integrate via git rebase instead of git merge.

Examples

The following example demonstrates how to synchronize with the central repository's master branch:

```
$ git checkout master  
$ git pull --rebase origin
```

This simply moves your local changes onto the top of what everybody else has already contributed.

17. 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.

Discussion

The most common use case for git push is to publish your local changes to a central repository. After you've accumulated several local commits and are ready to share them with the rest of the team, you (optionally) clean them up with an interactive rebase, then push them to the central repository.

18. 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]

```

imran@DevOps:~$ ssh-keygen -t rsa -b 4096 -C "imranteli0706@gmail.com"
Generating public/private rsa key pair.
Enter file in which to save the key (/home/imran/.ssh/id_rsa): /home/imran/.ssh/id_rsa_DO_github
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/imran/.ssh/id_rsa_DO_github.
Your public key has been saved in /home/imran/.ssh/id_rsa_DO_github.pub.
The key fingerprint is:
SHA256:AacADj0wSRR7tePN93IUS6UyLhPjrpjaFlkg/QUtMGg imranteli0706@gmail.com
The key's randomart image is:
+---[RSA 4096]-----+
|B+=+oo+ . .|
|.E.ooo.*  o|
|.o.o+o.o +|
|.o=+.+ o|
|.o.*So o|
|. .+ o|
|. . . o|
|.o . o|
|.o+ .|
+---[SHA256]-----+
imran@DevOps:~$

```

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

```

imran@DevOps: ~/.ssh
imran@DevOps:~$ cd .ssh/
imran@DevOps:~/.ssh$ ls
config id_rsa id_rsa_DO_github id_rsa_DO_github.pub id_rsa.pub known_hosts
imran@DevOps:~/.ssh$ cat config
Host github.com
  HostName github.com
  IdentityFile ~/.ssh/id_rsa_DO_github
  User git
  IdentitiesOnly yes
imran@DevOps:~/.ssh$

```

Add SSH public key in github account.

1. Copy public key

cat ~/.ssh/id_rsa_DO_github.pub

```
imran@DevOps:~/.ssh$ ls
config id_rsa id_rsa DO github id_rsa DO github.pub id_rsa.pub known_hosts
imran@DevOps:~/.ssh$ cat id_rsa DO github.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQDAQDIYnr0nGgJuoPBuuhIW1+VVRfxbZY+eTuIokrd9QmDMYV6u1K7mB6094eEW6xap9A09sLgVYYldccEM6Jk6s0Jz/TspXduJg3+27Fu3
J00QxcJ8Cv5iVJyboFhWXPyaS0KIL9uJ0tP1LJecwJB09H08DwLknjgnLi3/TitlJq3Iq8cyDN4jQaPdIMh0iKkFsqzemzflZ0Tnif+xZw58hAHV67iUMutvx7u6k+tLSbjNb/Vi1r
3pj90d7tr6Eh+08ay0uf2AzW2f9hbuc+B5kRyuU3gCLXRkscTqMEKX/cG9CnY24MIgcepEZFI+16YDzcsYbKDvbl49mXSsnXc4c/ty0CYwswnme4eaLLKr2zH5ISS32YyFFxzSYEuQa/9
4PMH9HHCu6soJkYXHMqWxcMnJpbph5N97Hya5SPROM23ctPDmkAc9pLwCfFIWoKQ0+16XaZpFXB195J//pjcubYXHkWT5j0zLxXIQTnDyoEIRFNJ+zolj a878AhKz6R9WQPGJDv0/MjR
xpSABrC04aUL053qvgXBR2LDdu0PkEh4C9YbTYqXsjZFJDliuVZb9+pk84tkqGdLuXf2XwarXaXX697nWw/TDkCrVm6AHgBMHzCL89ZWvUDRF+mc4QqSg/ktQf0mSB3ydjEkA5YmtZkX4
ZcuePqgejsIL5CqcQ== imranteli0706@gmail.com
imran@DevOps:~/.ssh$
```

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

The screenshot shows the GitHub website interface. At the top, there's a navigation bar with 'Search GitHub', 'Pull requests', 'Issues', and 'Gist'. Below this is a large banner area with the text 'Learn Git and GitHub without any code!' and buttons for 'Read the guide' and 'Start a project'. On the right side, there's a user profile dropdown menu showing 'Signed in as DevImranOps' and options like 'Your profile', 'Your stars', 'Explore', 'Integrations', 'Help', 'Settings', and 'Sign out'. The 'Settings' option is highlighted. Below the banner, the 'SSH keys' section is visible, showing 'There are no SSH keys with access to your account.' and a link to 'Check out our guide to generating SSH keys or troubleshoot common SSH Problems.' Below this, the 'GPG keys' section is visible, showing 'There are no GPG keys with access to your account.' and a link to 'Learn how to generate a GPG key and add it to your account.' On the left side, there's a sidebar with 'Personal settings' and 'SSH and GPG keys' highlighted.

Personal settings

- Profile
- Account
- Emails
- Notifications
- Billing
- SSH and GPG keys**
- Security
- Blocked users
- Repositories
- Organizations
- Saved replies
- Authorized applications

SSH keys New SSH key

There are no SSH keys with access to your account.

Title

gitPubKey

Key

```
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAQCAQDIYnrOnGgJuoPBuuhIW+VVRfbxZY+eTulokrd9QmDMYV6uiK7
mB6094eEW6xap9AQ9sLgVYYldccEM6Jk6sOJz/TspXduJg3+27Fu3u00QxcJ8Cv5iJyboFhWXPyaSQKIL
9ujQOtP1LJecwJB09HQBDwLknjgnLi3/TitJq3lq8cyDN4jQaPdiMh0iKkFsqzemzflZ0Tnif+xZw58hAHV67iUMu
tvxE7u6k+tLSbjNb/Vl1r8pj90d7tr6Eh+08ay0uf2AzW2f9hbuc+B5kRyuU3gClXRkscTqMEKX/cG9CnY24Mlgc
epEZFI+I6YDzcsYbKDVbl49mXSsnXc4c/ty0CYwswnmme4eaLLKr2zH5iSS32YyFFxzSYEuQa/9HPMH9HHC
U6soJkYXHMqWxcMnJpbph5NQ97HyaSPROM23ctPDmkAc9pLwCfFIWoKQ0+16XaZpFXBi95J//pjuBYXH
kWT5j0zLxXIQTnDyoEIRFNJ+zolja878AhKz6Rx9WQPGJDv0/MjRxpSABrCO4aULO53qvgXBR2LDDuOPkE
h4C9YbTYqSjzFJDLiuVZb9+pk84tkqGdLuXf2XwarXaXX697nWw/TDkCrV/m6AHgBMHzCL89ZWvUDRf+m
c4QqSg/ktQfOmSB3yjdJEkA5YmtZkX4ZcuePqqsIL5CqcQ== imranteli0706@gmail.com
```

Add SSH key

[Check out our guide to generating SSH keys or troubleshoot common SSH Problems.](#)

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.

```
imran@DevOps: ~
imran@DevOps:~$ ssh -T git@github.com
Warning: Permanently added the RSA host key for IP address '192.30.253.113' to the list of known hosts.
Hi DevImranOps! You've successfully authenticated, but GitHub does not provide shell access.
imran@DevOps:~$
```

19. GIT Cheat Sheet from Atlassian

Git Basics

<code>git init</code> <code><directory></code>	Create empty Git repo in specified directory. Run with no arguments to initialize the current directory as a git repository.
<code>git clone <repo></code>	Clone repo located at <repo> onto local machine. Original repo can be located on the local filesystem or on a remote machine via HTTP or SSH.
<code>git config</code> <code>user.name <name></code>	Define author name to be used for all commits in current repo. Devs commonly use --global flag to set config options for current user. +
<code>git add</code> <code><directory></code>	Stage all changes in <directory> for the next commit. Replace <directory> with a <file> to change a specific file.
<code>git commit -m</code> <code>"<message>"</code>	Commit the staged snapshot, but instead of launching a text editor, use <message> as the commit message.
<code>git status</code>	List which files are staged, unstaged, and untracked.
<code>git log</code>	Display the entire commit history using the default format. For customization see additional options. +
<code>git diff</code>	Show unstaged changes between your index and working directory +

Undoing Changes

<code>git revert</code> <code><commit></code>	Create new commit that undoes all of the changes made in <commit>, then apply it to the current branch.
<code>git reset <file></code>	Remove <file> from the staging area, but leave the working directory unchanged. This unstages a file without overwriting any changes. +
<code>git clean -n</code>	Shows which files would be removed from working directory. Use the -f flag in place of the -n flag to execute the clean.

Visualpath

Rewriting Git History

`git commit --amend`

Replace the last commit with the staged changes and last commit combined. Use with nothing staged to edit the last commit's message.

`git rebase <base>`

Rebase the current branch onto <base>. <base> can be a commit ID, a branch name, a tag, or a relative reference to HEAD. +

`git reflog`

Show a log of changes to the local repository's HEAD. Add --relative-date flag to show date info or --all to show all refs.

Git Branches

`git branch`

List all of the branches in your repo. Add a <branch> argument to create a new branch with the name <branch>.

`git checkout -b
<branch>`

Create and check out a new branch named <branch>. Drop the -b flag to checkout an existing branch.

`git merge <branch>`

Merge <branch> into the current branch.

Remote Repositories

`git remote add
<name> <url>`

Create a new connection to a remote repo. After adding a remote, you can use <name> as a shortcut for <url> in other commands.

`git fetch
<remote> <branch>`

Fetches a specific <branch>, from the repo. Leave off <branch> to fetch all remote refs.

`git pull <remote>`

Fetch the specified remote's copy of current branch and immediately merge it into the local copy. +

`git push <remote>
<branch>`

Push the branch to <remote>, along with necessary commits and objects. Creates named branch in the remote repo if it doesn't exist. +

Visualpath

git config

<code>git config --global user.name <name></code>	Define the author name to be used for all commits by the current user.
<code>git config --global user.email <email></code>	Define the author email to be used for all commits by the current user.
<code>git config --global alias.<alias-name> <git-command></code>	Create shortcut for a Git command. E.g. <code>alias.glog "log --graph --oneline"</code> will set " <code>git glog</code> " equivalent to " <code>git log --graph --oneline</code> ".
<code>git config --system core.editor <editor></code>	Set text editor used by commands for all users on the machine. <code><editor></code> arg should be the command that launches the desired editor (e.g., <code>vi</code>).
<code>git config --global --edit</code>	Open the global configuration file in a text editor for manual editing.

git log

<code>git log -<limit></code>	Limit number of commits by <code><limit></code> . E.g. " <code>git log -5</code> " will limit to 5 commits
<code>git log --oneline</code>	Condense each commit to a single line.
<code>git log --stat</code>	Include which files were altered and the relative number of lines that were added or deleted from each of them.
<code>git log -p</code>	Display the full diff of each commit.
<code>git log --author="<pattern>"</code>	Search for commits by a particular author.
<code>git log --grep="<pattern>"</code>	Search for commits with a commit message that matches <code><pattern></code> .
<code>git log <since>..<until></code>	Show commits that occur between <code><since></code> and <code><until></code> . Args can be a commit ID, branch name, HEAD, or any other kind of revision reference.
<code>git log -- <file></code>	Only display commits that have the specified file.
<code>git log --graph --decorate</code>	<code>--graph</code> flag draws a text based graph of commits on left side of commit msgs. <code>--decorate</code> adds names of branches or tags of commits shown.

git diff

<code>git diff HEAD</code>	Show difference between working directory and last commit.
<code>git diff --cached</code>	Show difference between staged changes and last commit.

git reset

<code>git reset</code>	Reset staging area to match most recent commit, but leave the working directory unchanged.
<code>git reset --hard</code>	Reset staging area and working directory to match most recent commit and overwrites all changes in the working directory.
<code>git reset <commit></code>	Move the current branch tip backward to <commit>, reset the staging area to match, but leave the working directory alone.
<code>git reset --hard <commit></code>	Same as previous, but resets both the staging area & working directory to match. Deletes uncommitted changes, and all commits after <commit> .

git rebase

<code>git rebase -i <base></code>	Interactively rebase current branch onto <base>. Launches editor to enter commands for how each commit will be transferred to the new base.
---	---

git pull

<code>git pull --rebase <remote></code>	Fetch the remote's copy of current branch and rebases it into the local copy. Uses git rebase instead of merge to integrate the branches.
---	---

git push

<code>git push <remote> --force</code>	Forces the <code>git push</code> even if it results in a non-fast-forward merge. Do not use the <code>--force</code> flag unless you're absolutely sure you know what you're doing.
<code>git push <remote> --all</code>	Push all of your local branches to the specified remote.
<code>git push <remote> --tags</code>	Tags aren't automatically pushed when you push a branch or use the <code>--all</code> flag. The <code>--tags</code> flag sends all of your local tags to the remote repo.



Visit atlassian.com/git for more information, training, and tutorials

Summary:

22. Software Development Life Cycle (SDLC) is a process used by the software industry to design, develop and test high quality softwares.
23. SDLC model are suitable for their project and it would also help the developers and testers understand basics of the development model being used for their project.
24. SDLC is an organized and time bounded way of doing Development, testing and release
25. The SDLC aims to produce a high-quality software that meets or exceeds customer expectations, reaches completion within times and cost estimates. There are various SDLC models like Waterfall, Spiral, iterative Agile and few other.
26. Version control systems are a category of software tools that help a software team manage changes to source code over time.
27. Version control software keeps track of every modification to the code in a special kind of database.
28. 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.
29. Git retain long-term change history of every file.
30. Traceability, being able to trace each change made to the software and connect it to project management and bug tracking software such as JIRA.
31. 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.