GIT

Git Basics

The major difference between Git and any other VCS (Subversion and friends included) is the way Git thinks about its data. Conceptually, most other systems store information as a list of file-based changes. These systems (CVS, Subversion, Perforce, Bazaar, and so on) think of the information they keep as a set of files and the changes made to each file over time.

Git doesn?t think of or store its data this way. Instead, Git thinks of its data more like a set of snapshots of a miniature filesystem. Every time you commit, or save the state of your project in Git, it basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot. To be efficient, if files have not changed, Git doesn?t store the file again, just a link to the previous identical file it has already stored. Git thinks about its data more like a stream of snapshots.

Local operations

Most operations in Git only need local files and resources to operate ? generally no information is needed from another computer on your network. Because you have the entire history of the project right there on your local disk, most operations seem almost instantaneous.

To browse the history of the project, Git doesn?t need to go out to the server to get the history and display it for you ? it simply reads it directly from your local database. This means you see the project history almost instantly. If you want to see the changes introduced between the current version of a file and the file a month ago, Git can look up the file a month ago and do a local difference calculation, instead of having to either ask a remote server to do it or pull an older version of the file from the remote server to do it locally. You can work offline and commit happily until you get to a network connection to upload.

Integrity

Everything in Git is check-summed before it is stored and is then referred to by that checksum. This means it?s impossible to change the contents of any file or directory without Git knowing about it. This functionality is built into Git at the lowest levels and is integral to its philosophy. You can?t lose information in transit or get file corruption without Git being able to detect it.

The mechanism that Git uses for this checksumming is called a SHA-1 hash. This is a 40-character string composed of hexadecimal characters (0?9 and a?f) and calculated based on the contents of a file or directory structure in Git. A SHA-1 hash looks something like this:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

You will see these hash values all over the place in Git because it uses them so much. In fact, Git stores everything in its database not by file name but by the hash value of its contents.

Git project sections

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

1. The Git directory is where Git stores the metadata and object database for your project. This is the most important part of Git, and it is what is copied when you clone a repository from another computer.
1. The working directory is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.
1. The staging area is a file, generally contained in your Git directory, that stores information about what will go into your next commit. It?s sometimes referred to as the ?index?, but it?s also common to refer to it as the staging area.

Git lifecycle status

Each file in your working directory can be in one of two states: tracked or untracked. Tracked files are files that were in the last snapshot; they can be unmodified, modified, or staged. Untracked files are everything else ? any files in your working directory that were not in your last snapshot and are not in your staging area. When you first clone a repository, all of your files will be tracked and unmodified because you just checked them out and haven?t edited anything.

As you edit files, Git sees them as modified, because you?ve changed them since your last commit. You stage these modified files and then commit all your staged changes, and the cycle repeats.

1. You modify files in your working directory.
1. You stage the files, adding snapshots of them to your staging area.
1. You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

If a particular version of a file is in the Git directory, it?s considered committed. If it has been modified and was added to the staging area, it is staged. And if it was changed since it was checked out but has not been staged, it is modified.

Git installation

Before you start using Git, you have to 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.

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/.

Installing on Linux

If you want to install Git 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
```

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

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

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.

GIT first time setup

Git comes with a tool called `git config` that lets you get and set configuration variables that control all aspects of how Git looks and operates. These variables can be stored in three different places:

1. `/etc/gitconfig` file: Contains values for every user on the system and all their repositories. If you pass the option `--system` to git config, it reads and writes from this file specifically.
1. `~/.gitconfig` or `~/.config/git/config` file: Specific to your user. You can make Git read and write to this file specifically by passing the `--global` option.
1. `config` file in the Git directory (that is, `.git/config`) of whatever repository you?re currently using: Specific to that single repository.

Each level overrides values in the previous level, so values in `.git/config` trump those in `/etc/gitconfig`.

On Windows systems, Git looks for the `.gitconfig` file in the `$HOME` directory (`C:\Users\$USER for most people`). It also still looks for `/etc/gitconfig`, although it?s relative to the MSys root, which is wherever you decide to install Git on your Windows system when you run the installer. If you are using Git for Windows 2.x or later, there is also a system-level config file at `C:\Documents and Settings\All Users\Application Data\Git\config` on Windows XP, and in `C:\ProgramData\Git\config` on Windows Vista and newer. This config file can only be changed by git `config -f <file>` as an admin.

Identity

The first thing you should do when you install Git is to set your user name and email address. This is important because every Git commit uses this information, and it?s immutably baked into the commits you start creating:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Again, you need to do this only once if you pass the --global option, because then Git will always use that information for anything you do on that system. If you want to override this with a different name or email address for specific projects, you can run the command without the --global option when you?re in that project.

Editor

Now that your identity is set up, you can configure the default text editor that will be used when Git needs you to type in a message. If not configured, Git uses your system?s default editor, which is system dependant.

If you want to use a different text editor, such as Emacs, you can do the following:

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

While on a Windows system, if you want to use a different text editor, such as Notepad++, you can do the following:

On a x86 system

```
$ git config --global core.editor "'C:/Program Files/Notepad++/notepad++.exe' -multiInst -nosession"
```

On a x64 system

```
"'C:/Program Files (x86)/Notepad++/notepad++.exe' -multiInst -nosession"
```

Checking your settings

If you want to check your settings, you can use the `git config --list` command to list all the settings Git can find at that point:

```
$ git config --list
user.name=John Doe
user.email=johndoe@example.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

You may see keys more than once, because Git reads the same key from different files (`/etc/gitconfig` and `~/.gitconfig`, for example). In this case, Git uses the last value for each unique key it sees.

You can also check what Git thinks a specific key?s value is by typing `git config <key>`:

```
$ git config user.name
John Doe
```

Getting help

Help list for git commands:

```
 git help
```

Help for specific commands `git help commandName`:

```
git help config
```

Getting a git repository

You can get a Git project using two main approaches. The first takes an existing project or directory and imports it into Git. The second clones an existing Git repository from another server.

Initializing a repo in an existing directory

Create a project directory `mkdir projectName`

To start to track an existing project in Git, go to this project `cd projectName` and type:

```
git init
```

This creates a new subdirectory named .git that contains all of your necessary repository files ? a Git repository skeleton. At this point, nothing in your project is tracked yet. To list the new folder just type:

```
ls -a
. .. .git
```

Inspecting a repository

The main tool you use to determine which files are in which state is the `git status` command. If you run this command directly after a clone, you should see something like this:

```
$ git status
On branch master
nothing to commit, working directory clean
```

Create a new file in your working directory:

```
git status
echo 'This is my first file.' > myfile.txt
git status
```

Git log

The git log command displays committed snapshots. It lets you list the project history, filter it, and search for specific changes. While git status lets you inspect the working directory and the staging area, git log only operates on the committed history.

`git log`

Tracking new files, check status and differences

In order to begin tracking a new file, you use the command git add. To begin tracking the README file, you can run this:

```
git add myfile.txt
git status
```

If you want to track all the existing files in the working directory:

```
git add .
```

If we modify the file, our new changes aren't staged yet.

```
echo "A change to this file." >> myfile.txt
git status
```

```
git diff # Diff between unstaged vs staged changes
git diff --cached # Diff between staged changes vs. commit
```

Commit changes

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 explicity ask it to.

A commit at this point will commit the file as it was when we added it, without the later changes.

```
git commit -m "Initial commit. myfile has one line."
git status
```

Add changes to the last commit

```
git add myfile.txt
git commit --amend -m "Initial commit. myfile has two lines."
```

In case we want to commit a snapshot of all changes in the working directory, we will use the flag `-a`. This only includes modifications to tracked files (those that have been added with git add at some point in their history).

```
git commit -a -m "My new commit"
```

To unstage a change that we don't want to commit

```
echo "Don't really need this line." > myfile.txt
git add myfile.txt
git reset HEAD myfile.txt    # Unstage.
git checkout myfile.txt      # Delete change.
```

To commit changes directly without staging:

```
echo "The third change to this file." >> myfile.txt
git commit -a -m "A new version of myfile"
```

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).

Revert commit

The `git revert` command undoes a committed snapshot. But, instead of removing the commit from the project history, it figures out how to undo the changes introduced by the commit and appends a new commit with the resulting content. This prevents Git from losing history, which is important for the integrity of your revision history and for reliable collaboration.

```
git revert <commit>
```

Reverting should be used when you want to remove an entire commit from your project history.

```
# Commit a snapshot
git commit -m "Make some changes that will be undone"

# Revert the commit we just created
git revert HEAD
```

Reset

`git reset` is a dangerous method. When you undo with `git reset` (and the commits are no longer referenced by any ref or the reflog), there is no way to retrieve the original copy?it is a permanent undo.

Remove the specified file from the staging area, but leave the working directory unchanged. This unstages a file without overwriting any changes.

```
git reset <file>
```

Reset the staging area to match the most recent commit, but leave the working directory unchanged. This unstages all files without overwriting any changes, giving you the opportunity to re-build the staged snapshot from scratch.

```
git reset
```

Reset the staging area and the working directory to match the most recent commit.

```
git reset --hard
```

Move the current branch tip backward to <commit>, reset the staging area to match, but leave the working directory alone. All changes made since <commit> will reside in the working directory, which lets you re-commit the project history.

```
git reset <commit>
```

Move the current branch tip backward to <commit> and reset both the staging area and the working directory to match. This destroys not only the uncommitted changes, but all commits after <commit>, as well.

```
git reset --hard <commit>
```

You should never use git reset <commit> when any snapshots after <commit> have been pushed to a public repository. After publishing a commit, you have to assume that other developers are reliant upon it.

Removing a commit that other team members have continued developing poses serious problems for collaboration.

Removing files

To remove a file from Git, you have to remove it from your tracked files (more accurately, remove it from your staging area) and then commit. The `git rm` command does that, and also removes the file from your working directory so you don?t see it as an untracked file the next time around. The file will be removed the next commit:

```
git rm myfile.txt
```

To remove from Staged and maintain in the working directory:

```
echo "The first line of the second file" > myfile2.txt
git add myfile2.txt
git commit -m "A new file myfile2" #only possible to commit -a to already tracked files.
git rm --cached myfile2.txt
```

The file will be interpreted as untracked

Ignoring files

Often, you?ll have a class of files that you don?t want Git to automatically add or even show you as being untracked. These are generally automatically generated files such as log files or files produced by your build system. In such cases, you can create a file listing patterns to match them named `.gitignore`.

Create in your working directory the .gitignore file:

```
touch .gitignore
```

If you already have a file checked in, and you want to ignore it, Git will not ignore the file if you add a rule later. In those cases, you must untrack the file first, by running the following command in your terminal:

```
git rm -r --cached
```

Here is an example `.gitignore` file:

```
$ cat .gitignore
*.[oa]
*~
```

The first line tells Git to ignore any files ending in ?.o? or ?.a? ? object and archive files that may be the product of building your code. The second line tells Git to ignore all files that end with a tilde (~), which is used by many text editors such as Emacs to mark temporary files. You may also include a log, tmp, or pid directory; automatically generated documentation; and so on. Setting up a `.gitignore` file before you get going is generally a good idea so you don?t accidentally commit files that you really don?t want in your Git repository.

The rules for the patterns you can put in the `.gitignore` file are as follows:

Blank lines or lines starting with # are ignored.

Standard glob patterns work.

You can start patterns with a forward slash (/) to avoid recursivity.

You can end patterns with a forward slash (/) to specify a directory.

You can negate a pattern by starting it with an exclamation point (!).

To set up an ignore file as global:

```
git config --global core.excludesfile ~/.gitignore_global
```

Git checkout

The git checkout command serves three distinct functions: checking out files, checking out commits, and checking out branches.

Checking out a commit makes the entire working directory match that commit. This can be used to view an old state of your project without altering your current state in any way. Checking out a file lets you see an old version of that particular file, leaving the rest of your working directory untouched.

Return to the master branch.

```
git checkout master
```

Check out a previous version of a file. This turns the <file> that resides in the working directory into an exact copy of the one from <commit> and adds it to the staging area.

```
git checkout <commit> <file>
```

Create and check out <new-branch>.

```
git checkout -b <new-branch>
```

Update all files in the working directory to match the specified commit.

```
git checkout <commit>
```

You can use either a commit hash or a tag as the <commit> argument. This will put you in a detached HEAD state. During the normal course of development, the HEAD usually points to master or some other local branch, but when you check out a previous commit, HEAD no longer points to a branch?it points directly to a commit. This is called a ?detached HEAD? state.

Getting back to the latest commit

If you know the commit you want to return to is the head of some branch (the default is usually the `master` branch), or is tagged, then you can just

```
git checkout <branchname>
```

Another useful command is `git reflog`, to see what other commits your `HEAD` (or any other ref) has pointed to in the past.

```
git reflog
```

This is different than `git log` because it lists the whole history, and not just the strictly the antecedent events.

Git Branches

A branch represents an independent line of development. Branches serve as an abstraction for the edit/stage/commit process.

The git branch command lets you create, list, rename, and delete branches. It doesn?t let you switch between branches or put a forked history back together again. For this reason, git branch is tightly integrated with the git checkout and git merge commands.

List all of the branches in your repository:

```
git branch
```

Create a new branch called <branch>. This does not check out the new branch:

```
git branch <branch>
```

Delete the specified branch. This is a ?safe? operation in that Git prevents you from deleting the branch if it has unmerged changes:

```
git branch -d <branch>
```

Force delete the specified branch, even if it has unmerged changes. This is the command to use if you want to permanently throw away all of the commits associated with a particular line of development.

```
git branch -D <branch>
```

```
git branch -m <branch>
```

Git tagging

Git has the ability to tag specific points in history as being important. Typically people use this functionality to mark release points (v1.0, and so on).

Git uses two main types of tags: lightweight and annotated.

A lightweight tag is very much like a branch that doesn?t change ? it?s just a pointer to a specific commit.

Annotated tags, however, are stored as full objects in the Git database. They?re checksummed; contain the tagger name, email, and date; have a tagging message; and can be signed and verified with GNU Privacy Guard (GPG). It?s generally recommended that you create annotated tags so you can have all this information; but if you want a temporary tag or for some reason don?t want to keep the other information, lightweight tags are available too.

List your tags:

```
git tag
```

Search for tags with a particular pattern:

```
git tag -l 'v1.*'
```

Create an annotated tag:

```
git tag -a <tag-name> -m "YOUR MESSAGE"
#Example: git tag -a 2.0-Release -m "This is the 2.0 release version"
```

Show annotated tag:

```
git show <tag-name>
#example: git show 2.0-Release
```

Create a lightweight tag:

```
git tag <tag-name>
#example: git tag 2.0-Release-lw
```

Show lightweight tag:

```
git show <tag-name>
#example: git show 2.0-Release-lw
```

This time the command just shows the commit.

Later tagging:

You can also tag commits after you've moved past them.

```
git tag -a <tag-name> <commit-checksum>
```

Git merges

Merging is Git's way of putting a forked history back together again. The git merge command lets you take the independent lines of development created by git branch and integrate them into a single branch.

```
git merge <branch>
```

Merge the specified branch into the current branch. Git will determine the merge algorithm automatically

Git rebase

From a content perspective, rebasing really is just moving a branch from one commit to another. But internally, Git accomplishes this by creating new commits and applying them to the specified base.

```
git rebase <base>
```

Rebase the current branch onto <base>, which can be any kind of commit reference (an ID, a branch name, a tag, or a relative reference to HEAD).

GitHub?

Version control really comes into its own when we begin to collaborate with other people. We already have most of the machinery we need to do this; the only thing missing is to copy changes from one repository to another.

Systems like Git allow us to move work between any two repositories. In practice, though, it?s easiest to use one copy as a central hub, and to keep it on the web rather than on someone?s laptop. Most programmers use hosting services like GitHub?, BitBucket? or GitLab? to hold those master copies;

Fork a Repo

A fork is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project. Forks are used to either propose changes to someone else's project or to use someone else's project as a starting point for your own idea.

Fork an example repository

1. On GitHub?, navigate to your preferred repository
2. In the top-right corner of the page, click FORK
3. Now you have your own copy of the 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

List the remote connections you have to other repositories:

```
git remote
```

List the remote connections including the URL of each connection:

```
git remote -v
```

Create a new connection to a remote repository. After adding a remote, you'll be able to use <name> as a convenient shorcut for <url> in other git commands:

```
git remote add <name> <url>
```

Remove the connection to the <name> repository:

```
git remote rm <name>
```

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

```
git remote rename <old-name> <new-name>
```

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.

```
git branch -r
```

Checking out a remote branch directly will put your git in a **Detached Head state** because if you execute `git checkout <remote-branch-name>` after `git fetch <remote>` you are referencing the branch in READ-ONLY mode.

To avoid this situation you need to create a local branch which is a copy of the remote one:

```
git checkout -b <local-branch-name> <remote-branch-name>
#example: git checkout -b test origin/test
```

If you execute `git branch` you will see there is a new branch called test. Now you are able to commit your own changes and push them to your repository. Remember to sync the branch with the remote one by using `git fetch` and `git merge`

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 you usually have. This gives you a chance to review changes before integrating them into your copy of the project.

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> ** git fetch origin
```

Fetch a specific branch:

```
git fetch <remote> <branch> ** git fetch origin master
```

Merge the remote branch changes (conflicts can appear):

```
 git merge <remote/branch> ** git merge origin/master
```

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

Git push

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.

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 <remote>
```

Instead of using git merge to integrate the remote branch with the local one, use git rebase.

```
git pull --rebase <remote>
```

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.

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> <branch>
```

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> --force
```

Push all of your local branches to the specified remote:

```
git push <remote> --all
```

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

git push <remote> --tags

Keep your fork synced