**Git Basics**

So, what is Git in a nutshell? This is an important section to absorb, because if you understand what Git is and the fundamentals of how it works, then using Git effectively will probably be much easier for you. As you learn Git, try to clear your mind of the things you may know about other VCSs, such as CVS, Subversion or Perforce — doing so will help you avoid subtle confusion when using the tool. Even though Git’s user interface is fairly similar to these other VCSs, Git stores and thinks about information in a very different way, and understanding these differences will help you avoid becoming confused while using it.

**Snapshots, Not Differences**

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 other systems (CVS, Subversion, Perforce, Bazaar, and so on) think of the information they store as a set of files and the changes made to each file over time (this is commonly described as *delta-based*version control).



Figure 4. Storing data as changes to a base version of each file.

Git doesn’t think of or store its data this way. Instead, Git thinks of its data more like a series of snapshots of a miniature filesystem. With Git, every time you commit, or save the state of your project, Git 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**.



Figure 5. Storing data as snapshots of the project over time.

This is an important distinction between Git and nearly all other VCSs. It makes Git reconsider almost every aspect of version control that most other systems copied from the previous generation. This makes Git more like a mini filesystem with some incredibly powerful tools built on top of it, rather than simply a VCS. We’ll explore some of the benefits you gain by thinking of your data this way when we cover Git branching in [Git Branching](https://git-scm.com/book/en/v2/ch00/ch03-git-branching).

**Nearly Every Operation Is Local**

Most operations in Git need only local files and resources to operate — generally no information is needed from another computer on your network. If you’re used to a CVCS where most operations have that network latency overhead, this aspect of Git will make you think that the gods of speed have blessed Git with unworldly powers. Because you have the entire history of the project right there on your local disk, most operations seem almost instantaneous.

For example, 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.

This also means that there is very little you can’t do if you’re offline or off VPN. If you get on an airplane or a train and want to do a little work, you can commit happily (to your *local* copy, remember?) until you get to a network connection to upload. If you go home and can’t get your VPN client working properly, you can still work. In many other systems, doing so is either impossible or painful. In Perforce, for example, you can’t do much when you aren’t connected to the server; and in Subversion and CVS, you can edit files, but you can’t commit changes to your database (because your database is offline). This may not seem like a huge deal, but you may be surprised what a big difference it can make.

**Git Has 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 Generally Only Adds Data**

When you do actions in Git, nearly all of them only *add* data to the Git database. It is hard to get the system to do anything that is not undoable or to make it erase data in any way. As with any VCS, you can lose or mess up changes you haven’t committed yet, but after you commit a snapshot into Git, it is very difficult to lose, especially if you regularly push your database to another repository.

This makes using Git a joy because we know we can experiment without the danger of severely screwing things up. For a more in-depth look at how Git stores its data and how you can recover data that seems lost, see [Undoing Things](https://git-scm.com/book/en/v2/ch00/_undoing).

**The Three States**

Pay attention now — here is the main thing to remember about Git if you want the rest of your learning process to go smoothly. 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 tree, and the staging area.



Figure 6. Working tree, staging area, and Git directory.

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.

The working tree 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.

The staging area is a file, generally contained in your Git directory, that stores information about what will go into your next commit. Its technical name in Git parlance is the “index”, but the phrase “staging area” works just as well.

The basic Git workflow goes something like this:

1. You modify files in your working tree.
2. You selectively stage just those changes you want to be part of your next commit, which adds *only*those changes to the staging area.
3. 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.

# Getting Started - First-Time Git Setup

## First-Time Git Setup

Now that you have Git on your system, you’ll want to do a few things to customize your Git environment. You should have to do these things only once on any given computer; they’ll stick around between upgrades. You can also change them at any time by running through the commands again.

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 applied to 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. (Because this is a system configuration file, you would need administrative or superuser privilege to make changes to it.)
2. ~/.gitconfig or ~/.config/git/config file: Values specific personally to you, the user. You can make Git read and write to this file specifically by passing the --global option.
3. 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 version 2.x or later of Git for Windows, 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.

### Your 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.

Many of the GUI tools will help you do this when you first run them.

### Your 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.

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

$ git config --global core.editor emacs

On a Windows system, if you want to use a different text editor, you must specify the full path to its executable file. This can be different depending on how your editor is packaged.

In the case of Notepad++, a popular programming editor, you are likely to want to use the 32-bit version, since at the time of writing the 64-bit version doesn’t support all plug-ins. If you are on a 32-bit Windows system, or you have a 64-bit editor on a 64-bit system, you’ll type something like this:

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

If you have a 32-bit editor on a 64-bit system, the program will be installed in C:\Program Files (x86):

$ git config --global core.editor "'C:/Program Files (x86)/Notepad++/notepad++.exe' -multiInst -nosession"

|  |  |
| --- | --- |
| Note | Vim, Emacs and Notepad++ are popular text editors often used by developers on Unix-based systems like Linux and macOS or a Windows system. If you are not familiar with these editors, you may need to search for specific instructions for how to set up your favorite editor with Git. |
| Warning | You may find, if you don’t setup your editor like this, you get into a really confusing state when Git attempts to launch it. An example on a Windows system may include a prematurely terminated Git operation during a Git initiated edit. |

### Checking Your Settings

If you want to check your configuration 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

If you ever need help while using Git, there are two equivalent ways to get the comprehensive manual page (manpage) help for any of the Git commands:

$ git help <verb>

$ man git-<verb>

For example, you can get the manpage help for the git config command by running

$ git help config

These commands are nice because you can access them anywhere, even offline. If the manpages and this book aren’t enough and you need in-person help, you can try the #git or #github channel on the Freenode IRC server (irc.freenode.net). These channels are regularly filled with hundreds of people who are all very knowledgeable about Git and are often willing to help.

In addition, if you don’t need the full-blown manpage help, but just need a quick refresher on the available options for a Git command, you can ask for the more concise “help” output with the -h or --help options, as in:

$ git add -h

usage: git add [<options>] [--] <pathspec>...

-n, --dry-run dry run

-v, --verbose be verbose

-i, --interactive interactive picking

-p, --patch select hunks interactively

-e, --edit edit current diff and apply

-f, --force allow adding otherwise ignored files

-u, --update update tracked files

-N, --intent-to-add record only the fact that the path will be added later

-A, --all add changes from all tracked and untracked files

--ignore-removal ignore paths removed in the working tree (same as --no-all)

--refresh don't add, only refresh the index

--ignore-errors just skip files which cannot be added because of errors

--ignore-missing check if - even missing - files are ignored in dry run

--chmod <(+/-)x> override the executable bit of the listed files