Getting Started With Git:

The De Facto Standard for Version Control

CONTENTS

ORIGINAL BY **MATTHEW MCCULLOUGH** AND **KURT COLLINS**
UPDATED BY **MATT RASBAND,** SENIOR SOFTWARE ENGINEER AT INTERCHANGE

Why Get Git?

Git is a postmodern version control system that offers the familiar capabilities of CVS or Subversion, but doesn't stop at just matching existing tools. Git stretches the very notion of version control systems (VCS) by its ability to offer almost all its features for use offline and without a central server. It is the brainchild of Linus Torvalds, the creator of Linux, with the first prototype written in a vitriolic two-week response to the "BitKeeper debacle" of 2005.

Today, Git is effectively the *de facto* standard for software version control and is an expected tool in every developer's toolbox. Users reference its blistering performance, usage flexibility, offline capabilities, and collaboration features as their motivation for switching.

Let's get started with Git. You'll be using it like a master in no time at all.

DISTRIBUTED VERSION CONTROL

If you are familiar with one or more traditional or centralized version control systems like Subversion, there will be several mental adjustments to make in your migration to Git. The first is that there is no central server. The second is that there is no central server. The full history of the repository lives on every user's machine that has cloned (checked out) a copy of the repository. This is the essence of a distributed version control system (DVCS).

Once over those hurdles, it is quite liberating to be able to work entirely independently, versioning any new project that you start, even in the incubation phase. The ease of setting up a new Git repository (or "repo" in common parlance) leads to setting up repos everywhere. It feels frictionless.

From there, you'll progress to the second epiphany of being able to share a repository and a changeset directly with a colleague without any complicated setup, without a check-in to a central server, without direct network connectivity, and without having to worry about firewalls getting in the way. Git has done technologically for version control what BitTorrent did for file sharing. It permanently replaced the spoke and hub structure with a peer-to-peer model, and there's no turning back. It supports transmitting binary sets of changes via USB stick, email, or in the traditional style (over a network), but amazingly, via HTTP, FTP, SCP, Samba, SSH, or WebDAV.

TAKE **CONTROL** of Your **Open Source Software**

KNOW WHAT'S IN YOUR CODE WITH FLEXNET CODE INSIGHT

MANAGE
Open Source Security and Compliance

COMPLY
with OSS Licenses and Obligations

IDENTIFY
Open Source Security Vulnerabilities

INTEGRATE
Scanning into Your DevOps Environment

FLEXERA SOFTWARE®
FlexNet® Code Insight™

Never miss evidence of open source in your applications using
FlexNet Code Insight—an end-to-end software composition analysis platform.

Getting Started

INSTALLING GIT

Git has a very light footprint for its installation. For most platforms, you can simply copy the binaries to a folder that is on the executable search `$PATH`. Git is primarily written in C, which means there is a unique distribution for each supported platform.

The canonical reference for Git installers can be found on a subpage of the official Git site. There are a number of installers available there for those who don't want to go through the hassle of doing the installation manually. In addition, also available at the official Git download site are links to older distributions of the Git binary.

You can confirm that Git is installed and available with:

```
$ git --version
git version 2.19.1
```

ESTABLISHING USER CREDENTIALS

Once you have selected a suitable distribution of Git for your platform, you'll need to identify yourself with a username and email address to Git.

In a separation of concerns most satisfying to the purist, Git does not directly support repository authentication or authorization. It delegates this in a very functional way to the protocol (commonly SSH) or operating system (file system permissions) hosting or serving up the repository. Thus, the user information provided during your first Git setup on a given machine is purely for "credit" of your code contributions.

With the binaries on your `$PATH`, issue the following three commands just once per new machine on which you'll be using Git. Replace the username and email address with your preferred credentials.

```
$ git config --global user.name "matthew.
mccullough"
$ git config --global user.email "matthew@
ambientideas.com"
$ git config --global color.ui "auto"
```

These commands store your preferences in a file named .gitconfig inside your home directory (~ on UNIX and Mac, and %USERPROFILE% on Windows).

If you are intrigued by all the potential nuances of a Git setup, here are several in-depth tutorials on setting up Git:

- Installing Git on Mac, Windows, and Linux
- Setting Up Git
- Getting Started and Installing Git

CREATING A REPOSITORY

Now that Git is installed and the user information is established, you can begin establishing new repositories. From a command prompt, change the directories to either a blank folder or an existing project that you want to put under version control. Then initialize the directory as a Git repository by typing the following commands:

```
$ git init
$ git add .
$ git commit -m 'The first commit'
```

The first command in the sequence, `init`, builds a .git directory that contains all the metadata and repository history. Unlike many other version control systems, Git uniquely stores everything in just a single directory at the top of the project — meaning no pollution in every directory.

Next, the `add` command with the dot wildcard tells Git to start tracking changes for the current directory, its files, and for all folders beneath.

Lastly, the commit function takes all the "staged" previous additions and makes them permanent in the repository's history in a transactional action. Rather than letting Git prompt the user via the default text editor, the `-m` option preemptively supplies the commit message to be saved alongside the committed files. Omitting this flag will use your system's default `$EDITOR`, falling back to `vi` as the default.

It is amazing and exciting to be able to truthfully say that you can use the basics of Git for locally versioning files with just these three commands.

CLONING EXISTING PROJECTS

An equally common use case for Git is starting from someone else's repository history. This is similar to the checkout concept in Subversion or other centralized version control systems. The difference in a DVCS is that the *entire history*, not just the latest version, is retrieved and saved to the local user's disk.

The syntax to pull down a local copy of an existing repo is:

```
# ssh protocol (requires SSH credentials to be
established):
$ git clone git@github.com:mrasband/gitrefcard.git

# https protocol:
$ git clone https://github.com/mrasband/
gitrefcard.git
```

The protocol difference often signifies access to the origin repository. Typically, HTTPS is used for read-only access (or when SSH is not configured) and SSH will be used for read-write access; however, this is not a hard-and-fast rule.

The clone command performs several subtasks under the hood. It sets up a remote (a Git repository address bookmark) named origin that points to the location *git@github.com:mrasband/gitrefcard. git*. Next, `clone` asks this location for the contents of its entire repository. Git copies those objects in a zlib-compressed manner over the network to the requestor's local disk. Lastly, clone switches to a branch named master, which is equivalent to Subversion's trunk, as the current working copy. The local copy of this repo is now ready to have edits made, branches created, and commits issued — all while online or offline.

TREEISH AND HASHES

Rather than a sequential revision ID, Git marks each commit with a SHA-1 hash that is unique to the person committing the changes, the folders, and the files comprising the changeset. This allows commits to be made independent of any central coordinating server.

A full SHA-1 hash is 40 hex characters: *b0c2c709cf57f3fa6e92ab249427726b7a82d221*.

To efficiently navigate the history of hashes, several symbolic shorthand notations can be used as listed in the table below. Additionally, any unique sub-portion of the hash can be used. Git will let you know when the characters supplied are not enough to be unique. In most cases, four or five characters are sufficient.

TREEISH	DEFINITION
HEAD	The current committed version
HEAD^1, HEAD~1	One commit ago
HEAD^^, HEAD~2	Two commits ago
HEAD~N	N Commits ago
RELEASE-1/0	User definited tag was applied to the code when it was certified for release

The complete set of revision specifications can be viewed by typing:

```
$ git help rev-parse
```

Treeish can be used in combination with all Git commands that accept a specific commit or range of commits.

Examples include:

```
$ git log HEAD~3..HEAD
$ git checkout HEAD^^
$ git merge RELEASE-1.0
$ git diff HEAD^..
```

The Typical Local Workflow

EDITING

Once you've cloned or initialized a new Git project, just start changing files as needed for your current assignment. There is no pessimistic locking of files by teammates. In fact, there's no locking at all. Git operates in a very optimistic manner, confident that its merge capabilities are a match for any conflicted changes that you and your colleagues can craft.

If you need to move a file, Git can often detect your manual relocation of the file and will show it as a pending "move." However, it is often more prudent to just directly tell Git to relocate a file and track its new destination.

```
$ git mv originalfile.txt subdir/newfilename.txt
```

If you wish to expunge a file from the current state of the branch, simply tell Git to remove it. It will be put in a pending deletion state and can be confirmed and completed by the next commit.

```
$ git rm fileyouwishtodelete.txt
```

VIEWING

Daily work calls for strong support of viewing current and historical facts about your repository, often from different, perhaps even orthogonal points of view. Git satisfies those demands in spades.

STATUS

To check the current status of a project's local directories and files (modified, new, deleted, or untracked), invoke the status command:

```
$ git status
```

DIFF

A patch-style view of the difference between the currently edited and committed files or any two points in the past can easily be summoned. The `..` operator signifies a range is being provided. An omitted second element in the range implies a destination of the current committed state, also known as HEAD:

```
$ git diff
$ git diff 32d4
$ git diff --summary 32d4..
```

Depending on the Git distribution, a utility called `diff-highlight` will be included to make diffs easier to visualize by highlighting word-level diffs instead of the default line level changes. Make sure diff-highlight is available in your `$PATH` and enable it with:

```
$ git config --global core.pager "diff-highlight |
less -r"
```

Git allows for diffing between the local files, the stage files, and the committed files with a great deal of precision.

COMMAND	DEFINITION
`git diff`	Everything unstaged (not git added) diffed to the last commit
`git diff-cached`	Everything staged (git added) diffed to the last commit
`git diff HEAD`	Everything unstaged and staged diffed to the last commit

LOG

The full list of changes since the beginning of time (or optionally, since a certain date) is right at your fingertips, even when disconnected from all networks:

```
$ git log
$ git log --since=yesterday
$ git log --since=2weeks
```

BLAME

If trying to discover why and when a certain line was added, cut to the chase and have Git annotate each line of a source file with the name and date it was last modified:

```
$ git blame <filename.ext>
```

STASHING

Git offers a useful feature for those times when your changes are in an incomplete state, you aren't ready to commit them, and you need to temporarily return to the last committed (e.g. a fresh checkout). This feature is named `stash` and pushes all your uncommitted changes onto a stack.

```
$ git stash
$ git stash --include-untracked
```

When you are ready to write the stashed changes back into the working copies of the files, simply pop them back on the stack.

```
$ git stash pop
```

ABORTING

If you want to abort your current uncommitted changes and restore the working copy to the last committed state, there are two commands that will help you accomplish this.

```
$ git reset --hard
```

Resetting with the hard option recursively discards all your currently uncommitted (unstaged or staged) changes.

To target just one blob, use the checkout command to restore the file to its previous committed state.

```
$ git checkout -- src/main/java/com.mrasband.
platform/models/Person.java
```

ADDING (STAGING)

When the developer is ready to put files into the next commit, they must be first staged with the `add` command. Users can navigate to any directory, adding files item by item or by wildcard.

```
$ git add <filename, directory name, or wildcard>
$ git add submodule1/Application.java
$ git add .
$ git add *.java
```

Specifying a folder name as the target of a `git add` recursively stages files in any subdirectories.

The `-i` option activates interactive add mode, in which Git prompts for the files to be added or excluded from the next commit.

```
$ git add -i
```

The `-p` option is a shortcut for activation of the patch** sub-mode of the interactive prompt, allowing for precise pieces within a file to be selected for staging.

```
$ git add -p
```

It's a common use case to need to exclude some files in a project from being tracked by Git. This is supported through a .gitignore file per project that is a list of *file/directory* patterns to be automatically ignored. More information is available here.

```
$ cat .gitignore
# If you find yourself ignoring temporary files
generated by your text editor
# or operating system, you probably want to add a
global ignore instead:
#   git config --global core.excludesfile '~/.
gitignore_global'
node_moles/
**/__pycache__/
/log/*
/tmp/*
!/log/.keep
!/temp/.keep
.DS_Store
```

COMMITTING

Once all the desired blobs are staged, a commit command transactionally saves the pending additions to the local repository. The default text `$EDITOR` will be opened for entry of the commit message; if unset, `vi` is the default.

```
$ git commit
```

Git requires that a message be present upon commit; leaving the message blank will abort the commit and leave the staged blobs in place.

The default editor can be changed with a Git configuration or by setting the `$EDITOR` environment variable:

```
$ git config --global core.editor <emacs|vim|subl
--wait|atom --wait>
```

To supply the commit message directly at the command prompt:

```
$ git commit -m "<your commit message>"
```

To view the statistics and facts about the last commit:

```
$ git show
```

If a mistake was made in the last commit's message, edit the commit text while leaving the changed files as-is with:

```
$ git commit --amend
```

BRANCHING

Branching superficially appears much the same as it does in other version control systems, but the difference lies in the fact that Git branches can be targeted to exist only locally or be shared with (pushed to) the rest of the team. The concept of inexpensive local branches increases the frequency in which developers use branching, opening it up to use for quick private experiments that may be discarded if unsuccessful, or merged onto a well-known branch if successful.

```
$ git branch <new branch name> <from branch>
$ git branch <new branch name>
```

CHOOSING A BRANCH

Checking out (switching to) a branch is as simple as providing its name:

```
$ git checkout <branch name>
```

Local and remote branches are checked out using the same command, but in somewhat of a radical change of operation for users coming from other systems like Subversion, remote branches are read-only until "tracked" and copied to a local branch. Local branches are where new work is performed and code is committed.

```
$ git branch <new branch> <from branch>
$ git checkout <new branch>
```

Or alternatively, in a combined command:

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

Starting with Git 1.6.6, a shorthand notation can be used to track a remote branch with a local branch of exactly the same name when no local branch of that name already exists and only one remote location is configured.

```
$ git checkout <remote and local branch name>
```

LISTING BRANCHES

To list the complete set of current local and remote branches known to Git:

```
$ git branch -a
```

The local branches typically have simple names like "master" and "experiment". Local branches are shown in white by Git's default syntax highlighting while the currently tracked branch is green with an asterisk prefix. Remote branches are prefixed by remotes and are shown in red.

MERGING

Like other popular VCSes, Git allows you to merge one or more branches into the current branch.

```
$ git merge <branch one>
$ git merge <branch one> <branch two>
```

If any conflicts are encountered, which is rare with Git, a notification message is displayed and the files are internally marked with `>>>>>>>>>` and `<<<<<<<<` around the conflicting portion of the file contents. Once manually resolved, `git add` the resolved file then commit in the usual manner.

REBASE

Rebasing is the rewinding of existing commits on a branch with the intent of moving the "branch start point" forward, then replaying the rewound commits. This allows developers to test their branch changes safely in isolation on their private branch just as if they were made on top of the mainline code, including any recent mainline bug fixes.

```
$ git rebase <source branch name>
$ git rebase <source branch name> <destination
branch name>
```

TAGGING

In Git, tagging operates in a simple manner that approximates other VCSes, but unlike Subversion, tags are immutable from a commit standpoint and must be unique. To mark a point in your code timeline with a tag:

```
$ git tag <tag name>
$ git tag <tag name> <treeish>
```

AUTOMATION WITH HOOKS

Git runs a standard set of hooks during user actions. Users can define custom scripts to be run that could be utilized to interrupt actions for code quality, spell checking, testing, etc. There are samples pre-installed in *.git/hooks*. Details are available in the Git book.

THE REMOTE WORKFLOW

Working with remote repositories is one of the primary features of Git. You can push or pull, depending on your desired workflow with colleagues and based on the repository operating system file and protocol permissions. Git repositories are most typically shared via SSH, though a lightweight daemon is also provided.

Git repository sharing via the simple daemon is introduced here. Sharing over SSH and Gitosis is documented in the Git Community Book.

REMOTES

While full paths to other repositories can be specified as a source or destination with most Git commands, this quickly becomes unwieldy and a shorthand solution is called for. In Git-speak, these bookmarks of other repository locations are called remotes.

A remote called origin is automatically created if you cloned a remote repository. The full address of that remote can be viewed with:

```
$ git remote -v
```

To add a new remote name:

```
$ git remote add <remote name> <remote git
address>
$ git remote add upstream git@github.com:mrasband/
gitrefcard.git
```

PUSH

Pushing with Git is the sending of local changes to a colleague or community repository with sufficiently open permissions as to allow you to write to it. If the colleague has the pushed-to branch currently checked out, they will have to re-checkout the branch to allow the merge engine to potentially weave your pushed changes into their pending changes.

By default, Git will only push branches, however to share tags marking immutable releases requires an additional flag:

```
$ git push --tags
```

FETCH

To retrieve remote changes without merging them into your local branches, simply fetch the blobs. This invisibly stores all retrieved objects locally in your .git directory at the top of your project structure, but waits for further explicit instructions for a source and destination of the merge.

```
$ git pull
$ git pull <remote name>
$ git pull <remote name> <branch name>
```

PULL

Pulling is the combination of a fetch and a merge as per the previous section all in one seamless action.

```
$ git pull
$ git pull <remote name>
$ git pull <remote name> <branch name>
```

In the cases in which a merge commit may not be desired and the rebasing behavior is preferred, `pull` supports a `rebase` flag.

```
$ git pull --rebase <remote name> <branch name>
```

BUNDLE

Bundle prepares binary diffs for transport on a USB stick or via email. These binary diffs can be used to "catch up" a repository that is behind otherwise too stringent of firewalls to successfully be reached directly over the network by `push` or `pull`.

```
$ git bundle create catchupsusan.bundle HEAD~8..
HEAD
$ git bundle create catchupsam.bundle
--since=10days master
```

These diffs can be treated just like any other remote, even though they are a local file on disk. The contents of the bundle can be inspected with `ls-remote` and the contents pulled into the local repository with fetch. Many Git users add a file extension of .bundle as a matter of convention.

```
$ git ls-remote catchupsusan.bundle
$ git fetch catchupsam.bundle
```

GUIS

Many graphical user interfaces have gained Git support in the last two years. The most popular IDEs and editors have excellent Git integration today.

GITK AND GIT-GUI

Standard Git distributions provide two user interfaces written in Tcl/Tk. Git-gui offers a panel by which to select files to add and commit, as well as type a commit message. Gitk offers a diagram visualization of the project's code history and branching. They both assume the current working directory as the repository you wish to inspect.

```
$ gitk
```

TOWER, SOURCETREE, AND OTHERS

There are a number of GUIs out there for Git that aren't officially released along with Git. Some of them include significant advanced functionality and are available on multiple platforms. Some of the more widely used ones include Tower, SourceTree, GitEye, and GitHub Desktop. Unfortunately, GitHub Desktop currently only works with GitHub and GitHub Enterprise repositories. However, given the wide use of GitHub as an online repository, it's likely that you'll run into the GitHub Desktop client at some point.

IDES

Java IDEs including IntelliJ, Eclipse (eGit), and NetBeans (NBGit) all offer native or simple plugin support for Git through their traditional source code control integration points. However, there are a number of other applications that also offer direct Git integration, as well. This includes applications such as Sublime Text, Atom, VS Code, Vim, and Emacs.

Numerous other platform-native GUIs offer graphically rich history browsing, branch visualization, merging, staging, and commit features.

CVS AND SUBVERSION

On the interoperability front, the most amazing thing about Git is its ability to read and write to a remote Subversion or CVS repository while aiming to provide the majority of the benefits of Git on the local copy of a repository.

CLONING

To convert a Subversion repository that uses the traditional trunk, tags, and branches structure into a Git repository, use a syntax very similar to that used with a traditional Git repository.

```
$ git svn clone --stdlayout <svn repository url>
```

Please be patient, and note the progress messages. Clones of large Subversion repositories can take hours to complete.

PUSHING GIT COMMITS TO SUBVERSION

Git commits can be pushed transactionally, one-for-one, to the cloned Subversion repository. When the Git commits are a good point for sharing with the Subversion colleagues, type:

```
$ git svn dcommit
```

RETRIEVING SUBVERSION CHANGES

When changes are inevitably made in Subversion and it is desired to freshen the Git repo with those changes, rebase to the latest state of the Subversion repo.

```
$ git svn rebase
```

ADVANCED COMMANDS

Git offers commands for both the new user and the expert alike. Some of the Git features requiring in-depth explanations can be discovered through the resources links below. These advanced features include the embedded (manpage-like) `help` and ASCII art visualization of branch merge statuses with `show-branch`. Git is also able to undo the last commit with the `revert` command, binary search for `bisect` the commit over a range of history that caused the unit tests to begin failing, check the integrity of the repository with `fsck`, prune any orphaned blobs from the tree with `gc`, and search through history with `grep`. And that is literally just the beginning.

This quick overview demonstrates what a rich and deep DVCS Git truly is, while still being approachable for the newcomer to this bold new collaborative approach to source code and version control.

Written by **Matt Rasband,** *Senior Software Engineer at Interchange*

Matt Rasband is a software engineer with a favoritism toward backend and distributed systems. He has worked in the financial services industry in a number of capacities, from Financial Advisor to Team Lead and Senior Software Engineer. Over his career he led a multi-billion-dollar company from a monolithic application to a microservices architecture. When he isn't writing Java or Python, or learning something new, he can be found in the mountains of Colorado sipping coffee, mountain biking, and spending time with his wife and their high-energy toddler.