Getting Started With Git

BY MATTHEW MCCULLOUGH, UPDATED AND REVISED BY KURT COLLINS

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 of its features for use offline and without a central server. It is the brainchild of Linus Torvalds, with the first prototype written in a vitriolic two-week response to the "BitKeeper debacle" of 2005.

Today, developers everywhere are migrating in droves to this exciting platform. 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 if 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 commit/check-in to a central server, direct network connectivity, or 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/S, FTP/S, SCP, Samba, SSH, RSYNC or WebDAV. While some of these have been deprecated (specifically FTP and RSYNC) due to their inefficiencies, the number of synchronization options available natively for Git repos are still astounding.

GETTING STARTED

INSTALLING GIT

Git has a very light footprint for its command line installation. For most platforms, you can simple 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 at: git-scm.com/download. There are a number of installers available there for those that don't want to go through the hassle of doing the install manually. In addition, also available at the official Git download site are links to older distributions of the Git binary.

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) that is 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 in 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 "yogi.bear"
git config --global user.email "yogi@jellystonepark.org"
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, GitHub, a web-based code hosting site, offers several in-depth tutorials on setting up Git for Linux, Windows, and Mac. Here are several in-depth Git installation guides:

help.github.com/articles/set-up-git/#platform-windows
help.github.com/articles/set-up-git/#platform-mac
help.github.com/articles/set-up-git/#platform-linux

CREATING A REPOSITORY

Now that Git is installed and the user information established, you can begin creating new repositories. From a command prompt, change 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 command:

```
git init
touch README.md
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. No pollution in every directory.

Following the directory initialization, the next command creates an empty file in the directory named README.md. You can skip this part if you decided to create a repository from a directory with files in it.

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, if any exist.

Lastly, the commit function takes all 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.

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 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:

```
git clone git://github.com/matthewmccullough/hellogitworld.git
git clone http://github.com/matthewmccullough/hellogitworld.git
git clone git@github.com:matthewmccullough/hellogitworld.git
```

The protocol difference often signifies whether you have read-only or writeable access to the origin repository. The final syntax, which accesses an SSH exposed repository, is the most common write-enabled protocol.

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/matthewmccullough/hellogitworld.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 & 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:

```
64de179becc3ed324daab72f7238df1404723672
```

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, 4-5 characters are sufficient.

TREEISH	DEFINITION
HEAD	The current committed version
HEAD^	One commit ago
HEAD^^	Two commits ago
HEAD~1	One commit ago
HEAD~3	Three commits ago
:/"Reformatting all"	Nearest commit whose comment begins with "Reformatting all"
RELEASE-1.0	User-defined tag 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
```

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 file 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 newsubdir/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
```

```
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   web/app.py
        modified:   web/requirements.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        web/settings.py

no changes added to commit (use "git add" and/or "git commit -a")
```

FIGURE 1: 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..
```

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 diffed to the last commit
`git diff --cached`	everything staged diffed to the last commit
`git diff HEAD`	everything unstaged and staged diffed to the last commit

```
diff --git a/web/app.py b/web/app.py
index 2f03735..a874061 100644
--- a/web/app.py
+++ b/web/app.py
@@ -9,7 +9,7 @@ app = Flask(__name__)

 @app.route('/', methods=['GET', 'POST'])
 def index():
-    return render_template_string('<html><head><title>Hello World!</title></hea
d><body>high ho</body></html>')
+    return render_template_string('<html><head><title>Hello World!</title></hea
d><body>hidey ho</body></html>')


 if __name__ == '__main__':
diff --git a/web/requirements.txt b/web/requirements.txt
index 1a7530c..e5f657d 100644
--- a/web/requirements.txt
+++ b/web/requirements.txt
@@ -1,8 +1,17 @@
+aniso8601==1.0.0
:
```

FIGURE 2: GIT DIFF

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
```

```
commit 5613912dd930b6d82d6d816d4f603adb07186b65
Author: Kurt Collins <kurt@kurt.sx>
Date:   Wed Jul 22 03:13:59 2015 -0400

    Initial environment set up scripts.
    1. Set up the development environment and production deployment scripts.
    2. Updated the readme.

commit d8c605f7c0f3532a26ea0944d48e93011a466840
Author: Kurt Collins <kurt@kurt.sx>
Date:   Sun Jul 5 21:18:57 2015 -0400

    Adding the standard gitignore for macs and python.

commit 2ea79d487f8df2d2d7a4a1010e6b505481e68e29
Author: Kurt Collins <kurt@kurt.sx>
Date:   Sun Jul 5 21:08:04 2015 -0400

    add README
```

FIGURE 3: GIT LOG

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 came into existence:

```
git blame <filename>
```

```
5613912d (Kurt Collins      2015-07-22 03:13:59 -0400  1) # app.py
5613912d (Kurt Collins      2015-07-22 03:13:59 -0400  2)
5613912d (Kurt Collins      2015-07-22 03:13:59 -0400  3)
5613912d (Kurt Collins      2015-07-22 03:13:59 -0400  4) from flask import Flas
k
5613912d (Kurt Collins      2015-07-22 03:13:59 -0400  5) from flask import requ
est, render_template_string
5613912d (Kurt Collins      2015-07-22 03:13:59 -0400  6)
5613912d (Kurt Collins      2015-07-22 03:13:59 -0400  7)
5613912d (Kurt Collins      2015-07-22 03:13:59 -0400  8) app = Flask(__name__)
5613912d (Kurt Collins      2015-07-22 03:13:59 -0400  9)
5613912d (Kurt Collins      2015-07-22 03:13:59 -0400 10) @app.route('/', method
s=['GET', 'POST'])
5613912d (Kurt Collins      2015-07-22 03:13:59 -0400 11) def index():
00000000 (Not Committed Yet 2016-01-15 19:07:59 +0000 12)     return render_temp
late_string('<html><head><title>Hello World!</title></head><body>hidey ho</body>
</html>')
5613912d (Kurt Collins      2015-07-22 03:13:59 -0400 13)
5613912d (Kurt Collins      2015-07-22 03:13:59 -0400 14)
5613912d (Kurt Collins      2015-07-22 03:13:59 -0400 15) if __name__ == '__main
__':
:
```

FIGURE 4: GIT BLAME

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
```

You can also save your stash with a name. This is particularly useful when you want to save more than one stash to the stack.

```
git stash save "picnic_basket"
```

To write the most recently stashed changes back into the working copies of the files, simply pop them back off the stack. Keep in mind, the following command also removes the changes from the stack.

```
git stash pop
```

However, if you have multiple stashes on the stack, you can also list all of the stashes by using:

```
git stash list
```

Once you've found the stash you want to load, you can do it one of two ways: either use the apply command or the pop command as in the following example.

```
git stash pop stash{n}
git stash apply stash{n}
```

Take note of the "n" in between the brackets. The "n" corresponds to the number of the stash you want to use. You can get the number by use the git stash list command explained above. Again, if you use the git stash pop version of the command, it will remove the code you want to use from the stack. However, if you use the git stash apply version of the command, it will use the code you want and leave it on the stack to be used again at another time.

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 of 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 -- Person.java
```

ADDING (STAGING)

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

```
git add <file name, folder name, or wildcard>
git add submodule1/PrimaryClass.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
```

COMMITTING

Once all 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.

```
git commit
```

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 text while leaving the changed files as-is with:

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

Or alternatively, in a combined command:

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

There is another way, as well. The following commands:

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

This method is often used if you're already working within a branch and you want to quickly start a new branch based off of the current branch you're working with.

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.

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

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. 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 happens less with Git that with many other VCSes, a notification message is displayed and the files are internally marked with >>>>>>>>> and <<<<<<<< surrounding 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. To mark a point in your code timeline with a tag:

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

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 at: www.kernel.org/pub/software/scm/git/docs/git-daemon.html. Sharing over SSH and Gitosis is documented in the Git Community Book at: book.git-scm.com/4_setting_up_a_private_repository.html.

REMOTES

While full paths to other repositories can be specified as a source or destination with the majority of 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 address>
git remote add <remote name> git@github.com:matthewmccullough/
ts.git
```

PUSH

Pushing with Git is the same thing as sending 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.

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 fetch <remote name> git merge <remote name/remote branch>
```

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>
```

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 catchupsusan.bundle --since=10.days 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 Is-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 catchupsusan.bundle
```

GUIS

Many graphical user interfaces have gained Git support in the last two years. The most popular Ruby, Perl, and Java/JVM IDEs have between a good and great level of Git integration today.

GITK & GIT-GUI

Standard Git distributions provide two user interfaces written in Tcl/Tk. The first, Git-Gui offers a panel by which to select files to add and commit, as well as type a commit message. The latter 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.

```
git gui gitk
```

TOWER, SOURCETREE & 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 & 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 one 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 and Atom.

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

CVS, 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, branches structure into a Git repository, use a syntax very similar to that used with a traditional Git repository.

```
git svn clone --stdlayout <svn repo 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 at a good point for sharing with the Subversion colleagues, type:

```
git svn dcommit
```

RETRIEVING SUBVERSION CHANGES

When changes are inevitably made in Subversion and you want 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.

ABOUT THE AUTHOR

KURT COLLINS (@timesync) is Director of Technology Evangelism and Partnerships at Built.io. Kurt began his engineering career at Silicon Graphics (SGI) where he worked on debugging its UNIX-flavored operating system. Kurt co-founded The Hidden Genius Project along with eight other individuals in response to the urgent need to increase the number of black men in the technology industry. Mentoring young people interested in the technology industry is a personal passion of his. Today, he continues this work with tech startups and non-profits alike.

RESOURCES

OFFICIAL RELEASE NOTES AND 'MAN' PAGES

git-scm.com

kernel.org/pub/software/scm/git/docs

MANUALS, TUTORIALS

cworth.org/hgbook-git/tour

www-cs-students.stanford.edu/~blynn/gitmagic

peepcode.com/products/git

BOOKS

Pro Git by Scott Chacon (free HTML)

Version Control with Git by Jon Loeliger

Pragmatic Version Control Using Git by Travis Swicegood

BOOKMARKS

delicious.com/matthew.mccullough/git

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, research guides, feature articles, source code and more.

"DZone is a developer's dream," says PC Magazine.

DZONE, INC.
150 PRESTON EXECUTIVE DR.
CARY, NC 27513

888.678.0399
919.678.0300

REFCARDZ FEEDBACK WELCOME
refcardz@dzone.com

SPONSORSHIP OPPORTUNITIES
sales@dzone.com

ISBN-13: 978-1-936502-77-6
ISBN-10: 1-936502-77-1
50795
9 781936 502776

VERSION 1.0