

**A Short Git Primer
ver. 1.8**

by

Wes Idell

wes@idelltech.org
<http://www.idelltech.org>

I. Initial Setup

1. Get Git

You may decide to use git in either Linux or Windows. The easiest solution for a Windows installation by far has proven to be msysgit, found here: <http://code.google.com/p/msysgit/>. Inside of Linux there are several routes to go to download git depending on the flavor of Linux which you use. Msysgit is a Google Code project which initially ran inside of a CygWin bash shell, but now it also has the ability to run in a Windows command prompt (cmd.exe) or as a Win32 application (GitGUI).

2. Some Background on Git

Git's main organization category is the project. All updates, branches, etc fall under this category. Throughout the project you will see a 40-digit hexadecimal object name. Though the name is not a name that you can pronounce using conventional English, it does maintain a distinction inside of the git project between objects. The name looks like this: 60dd6a2686879a71bd908563e12834d849121434, and is a basic SHA1 hash sum of the object it represents. The purpose of a hash sum is to mark a distinct set of data, so that virtually no other data will produce the same sum unless it is exactly like the first set which produced the initial sum. The concept of hashing is used throughout the computing industry and more information can be found here: http://en.wikipedia.org/wiki/SHA_hash_functions.

Git objects fall into one of four categories: a blob, a tag, a tree, or a commit. Blobs are usually files. Trees can be thought of as a directory tree which contains other blobs and trees. Commits contain identifying things like a timestamp, author of the commit, and a pointer to previous commits. Think of them like the data-structure type - linked lists. Each object consists of two other items besides the actual content – a type and a size. Tags are usually used to mark a commit at a certain point, for example, release candidates, releases, and changes in version numbers, though they can be used to describe most anything at a point in time of the particular project.

3. Configure Git

To **place your name into git**, which will be used on subsequent commits, use the following:

```
git config user.name "[your name here]"
```

Make sure to include the quotes around all configuration values (user name, email, etc) Use the same command (but different arguments) to **put your email address into git's configuration file**:

```
git config user.email "[your email address]"
```


II. Vanilla Git Usage

1. Obtaining A Git Project and/or Git Project Updates

To use git to start, or initialize a project, you can use:

git init

To clone an existing [archive] from a [server]:

git clone [server]/[archive]

This will place the archive into the present working directory named the same as the archive you are cloning.

To pull latest changes from the server, you do not need to specify any arguments, every configuration item is stored in a git configuration file. The only thing you need to do is type:

git pull

However, before doing this, you will want to make sure you have committed changes and be prepared – if you have changes to a file that also has changed on the server (i.e. a cohort has changed the file, pushed it up, and now you are pulling those changes down) you may need to either change merge strategies or be ready to do a merge by hand, as this will produce something known as a conflict. Read more about git conflicts in the next section.

2. Adding Files to the Project

git add [directory/file]

You may also use '*' to specify that all files in a certain directory be added to the git archive.

git add can also be used to convey to git that you are only going to commit certain files. For example, if you have a situation where a .cs file has changed, as well as a connection string has changed in a .config file, you may only want to track the changes in the .cs file, and leave the changes in the .config file for your local copy of the git archive. To do this, use the git add command, followed by the file name. If this is your intent, when committing, use:

git commit

as opposed to:

git commit -a

which **commits all changes which are currently tracked.**

When you are finished with what you need to do here, you may **push the changes up to the server**, but only in the instance that your local copy is in the most up-to-date state – it is always a good idea to do a git pull to be on the safe side.

git push

will take care of pushing your changes to the server. It is important to note that git push only pushes items which are tracked and have been committed to the latest version of the archive.

III. Conflicts

When it is more than you working in a particular git tree (and sometimes even when you are the sole developer), you are going to run into conflicts when merging (and this includes pulling as well).

Conflicts can happen, for example, when file A is edited by both developer x and developer y. when developer x pulls y's changes, a conflict happens. There are various strategies included into git's logic for dealing with merge conflicts, however none of these are 100% fail-proof. In the case of a failed instance of strategy experimentation, one can easily **recover to a point before the merge:**

git reset --hard HEAD^

Let's assume that no merge strategy works in your case. What then? Well you need a conflict resolution application (or at least one which has this as a feature). In fact, conflicts are so much an expected part of your experience with git that a specific command has been included in git which will **open your preferred merge tool after a conflicted merge:**

git mergetool

More can be found out by browsing to the git-mergetool manual page, at:
<http://www.kernel.org/pub/software/scm/git/docs/git-mergetool.html>

Git can use several of the *nix merge tools. From git-mergetool(1) Manual Page:
“Valid merge tools are: kdiff3, tkdiff, meld, xxdiff, emerge, vimdiff, gvimdiff, ecmerge, and opendiff.”

WinMerge fits my needs when working inside of Windows. It will allow you to open the conflicted file, make the necessary changes inside of a very nice interface, in my humble opinion. After the file has been successfully resolved, you may continue with committing the changes to your repository.

See also:

WinMerge - <http://winmerge.org/>

IV. Tags

The way to label a project's existence in a particular point in time is by the use of tags. In my work, I use tags to specify that a project has reached a specific version, or other milestone specific to the particular project in which I'm working on.

To label the commit as a version, or to generically name the release, you may use:

```
git tag [tag_text]
```

If you have **tagged the committed archive and you want everyone to see the name you have given this particular version**, you will need to push with the tags option:

```
git push --tags
```

Otherwise, your tagging is going to remain private to your personal copy of the project.

V. Removing Tracked Files

1. **Deleting Files From the Project**

If you find that **files are in the project which you do not want** (i.g. dll, pdb, etc), you need to first

```
git rm --cached [file]
```

Make certain to specify '--cached', or else it will delete the file entirely from the disk and the archive.

If you also are having problems with the files being tracked even though you have deleted them, you will need to place an entry into the .gitignore file. This file is found in the root of the git archive. It is a text file, with each line representing a file to ignore. The use of asterisk is valid to specify all matches - *.dll, *.pdb, etc.

2. **Deleting files from the Disk and the Project**

To remove the file from being tracked in the project and remove it from the disk (backup files come to mind), use:

```
git rm [file]
```

Remember to always commit your changes (and eventually push them to the main project archive).

VI. Using in Conjunction with GPG for Increased Security

1. Prerequisites:

- *A short working knowledge of GPG*
(<http://www.gnupg.org/documentation/manuals.en.html>)
- *a recent GPG installation* (<http://www.gnupg.org/download/>)
- *a key generated for the email address which you have in git's configuration*
(*user.email*) [to establish this, type:
git config user.email "[your email address for which you have a key]"]

2. Usage:

This is a very easy sub-component of git to use, and a piece of knowledge which may save your integrity, face, and possibly job in the long run.

To sign your tag with a gpg key, use git tag like so:

```
git tag -s [tag name] -m "[tag message]"
```

If your **existing key does not match the email address**, but you wish to use this key anyway, use:

```
git tag -u [email address] [tag name] -m "[tag message]"
```

3. Verification

If you wish to **verify that a tag belongs to a certain user**, please use:

```
git tag -v [tag name]
```

This will use your local installation of gpg to verify a signature, granted you have the user's key added to your key ring.

Editorial

It is good practice for developer's to always have each other's gpg keys to discuss proprietary ideas, or any information which one does not want stolen and possibly used malevolently.

VII. Conclusion

Git is one of the most powerful software change management systems which I have ever had the pleasure of using. I've never had problems with it losing information – in fact the only thing that has caused me grief is usage and commands, which is why I have written this short primer.

I'm no where finished with this, but I hope that it does come as a helpful document to help you get on your way with git. Any comments can be emailed to Wes <wesidell@gmail.com>.

© Wes Idell, 2009