Git in control of your ‘Change Management’

Synopsis

Git is a free Version Control System. It is designed for use by software developers, and quite popular for that purpose. The essence of Version Control is the ability to capture the state of software documents, thus making it possible to revert to a previous state if needed. This aids in collaboration and encourages freedom to experiment.

So, if I am not a software developer, why do I care about Git?
Can you recall a time when, working on something important, you found yourself saying, "Oops! I wish I could take that back". Or, "I no longer remember what it used to be, and that would be helpful". With Version Control, like Git, you can REWIND time. By capturing snapshots along the way, your progress in a project can be revisited and modified quite freely.
You even have the ability to "cherry pick" certain changed files and MERGE them with others, all up and down the scale of time.

One of the things that makes Git special is that it is local (i.e. on your own computer) and file-based. Being local, you don't need any special servers or complex equipment. Since it is file-based, it works with ANYTHING you can create or manipulate on a computer (text documents, image files, audio or video), and you can easily share or transport your work to other computers—even as an email attachment.

When collaborating, Git allows all members of a team to work independently and easily blend together the best of each contributor's efforts into a final product.

So, now, are you beginning to think of common tasks you perform that could benefit from a version control system like Git?

If you are a photographer or digital artist, you can save your work at any state and freely experiment. If you are a programmer, you can save a program before you make big changes to improve performance.

For someone who generates documents (a teacher who creates curricula, for example), the ability to "go back in time" is immensely valuable. Or, if you have ever had a very large document that represents hundreds of hours of work and the computer suddenly “can't read the file”, a system like Git provides an inherent backup (without the tedium of making multiple copies under many different names).

With Git, you will be able to see all your changes—and even compare the differences between versions.

Managing Change

Have you ever tried to build a document in collaboration with others? By passing it around from person to person? You make changes, then your friend makes a change. While another friend is updating the document you remember some important facts that need to be added. So you make changes on YOUR copy. Now there are multiple versions of the document, all with different changes and no easy way to get them all consolidated. Working on a project with multiple people can quickly become a nightmare of 'change management'.

Software developers face precisely this dilemma all the time (as well as designers, content writers, video artists, web developers, and so on). And the system(s) that are used to help with this are call Source Code Management systems. A Source Control Management (SCM) System helps to resolve conflicts among files and conflicts within a file as a result of multiple people making changes to the same file.

Git is one of the most popular SCM systems. And it is available for anyone—and for free.

Regardless of the type of development or design work you do—front-end UI development or back-end coding—managing the change and evolution of your working projects is both important and challenging.

The goal of this document is to

- introduce you to Git
- overcome the intimidating perception that Git is very "geeky" and confusing for non-programmers
- fill a gap in almost all other resources on the web for learning (and learning about) Git.

It seems every tutorial or description of Git carries many assumptions. Sitepoint has been dedicated to providing important technical information to everyone in a form that can be easily consumed. It is our sincere hope that this little primer will help you and take you beyond those initial stumbling blocks.

Why read this document?

Perhaps you have heard of 'Git' (or Github) and it sounds like something useful or important for your work. But when you sought to learn a little more you were quickly frustrated and confounded by all the information on the web that seems to include a lot of assumptions. Assuming you already know a little about something you are trying to learn is completely counter-productive.

It is hard to argue with the impression that ALL the tutorials or guides available for Git use the Linux command-line exclusively. If you are not 'a command-line geek', this is very intimidating.

Git is a very powerful and useful tool. Like anything that offers great power, there is an equal share of complexity.

Among those who use Git on a regular basis, there is a sort of 'evangelism' movement to help lead others to understanding this great, free tool. And that is the purpose of this document. This was designed as a simple tutorial. An opportunity for you to "get your feet wet" in the Git world without any fear or trepidation.

You are NOT expected to become a Git expert as a result of reading this. You WILL NOT be tested on your "geek smarts". After reading this you should be able to—

a) decide if Git is a tool that can help your workflow
b) feel much more comfortable approaching and using Git (or Github).

If, after reading this, you feel we have fallen short, please tell us. On the other hand, if you believe we have hit the mark then be sure to tell EVERYONE ELSE you know.

What To Expect

For many first time users, it is not clear how Git can help your workflow (which is typically a cycle of Editing, Saving and Pushing changes repeatedly). To help you get a clearer sense of common Git workflows, we've provided you with a "Workflow Scenarios'' as an APPENDIX that describes 'one-person' and 'team' workflows. This APPENDIX is a separate document.

This is an introduction to Git. In order to understand how Git can help you, we will take you on a tour. The tour is carefully guided and includes the Git commands and workflow that most Web developers, Designers, and Content specialists might use.

With a simple project 'Halo Whirled', you will "learn by doing" how to use Git. The simple 'Halo Whirled' walk-through uses only the basic Git commands. Like any tour, we ask that you trust the directions will not lead you into any trouble. Don't be intimidated by typing commands directly in the Command Line. Don't worry about response(s) from your computer that we have not explicitly described. There is nothing in this set of examples that can cause any permanent harm to you, your computer or the world around you.

Setting up Git

Some "get your hands dirty" details.

The first step is to download and install Git. Git is entirely free, and can run on machines using Windows, Mac and Linux operating systems.

Based on your operating system, the specific process to install Git will vary slightly.

Rather than repeat the installation instructions that are already widely available on the web, we will instead point you to some some existing installation guides that we recommend.

To install Git on a Mac, go to http://alvinalexander.com/mac-os-x/how-install-git-mac-os-x-osx

To install Git on Windows, go to http://msysgit.github.com/

Many distributions of Linux come with GIT pre-installed, but if that's not the case for you, you can download it from http://git-scm.com/download/linux

The most common way to use Git is via the "command line". The command line is a simple program that you find on all computers. It allows you to write instructions for the computer in a very basic way, without pretty graphics, buttons and so on. You just write plain, simple code.

This can be a bit scary for a beginner, but using the command line is definitely the best way to use Git. There are “graphic user interface” (GUI) programs available as an alternative—programs that provide you with buttons to press and pretty interfaces—but we encourage you at least to learn the basics of Git using the command line. We promise to hold your hand the whole way! All the same, you can find a useful list of GUI options here: http://git-scm.com/downloads/guis

We have attempted to provide enough guidance to ensure success for you regardless of your particular operating system (Windows, Mac or Linux) or your experience with the Command Line. Once you become familiar with the operation and concepts of using Git, you are welcome and encouraged to use the GUI tools if you prefer.

Although the initial purpose—and perhaps the most often use—of Git is for software source control, every attempt has been made here to be very generic in our discussion. You should see how these functions and features can apply to ANY work you do—especially if you find yourself in an environment of constant change. If, from time to time, the discussion seems to slip into ‘software geekdom’, please accept our apology in advance. The goal of this document is to introduce you to an understandably unfamiliar realm in the hopes that it will help you to do your work better.

The HISTORY is important

The key concept in understand Git is the idea of ‘history’. Git records snapshots of the collection of files in your project. Our goal when using Git is to protect ourselves (usually FROM ourselves) by capturing snapshots of our work. More importantly, rather than just having copies of the files in use, we can review the progression of work, reassemble it, rearrange it and repeat it as desired. One of the primary benefits of doing anything with a computer is the flexibility it affords you to UNDO and REDO operations.

It will be very easy to understand all you need to know about using Git, and tapping into its great power and flexibility, if you keep in mind this concept of ‘history’.

Many times in this document there will be reference to the history. Most of the work you do in Git is related to moving through the history of your work.

Let’s Git Started!

In order to use Git—assuming you have it installed—it is necessary to set up your project.
This project could as simple as your CV (in one MS Word Document) or a complete Website and all its subfolders.

In the steps that follow, we will be working on a simple project called ‘Halo Whirled’.

As stated earlier, we will guide you to complete this project using the command line. So firstly we need to make sure you have this open.

If you are working in Windows, go to the Start Menu and choose Run. (If that doesn't work for you, you can also type 'CMD' in the SEARCH box. In Windows it is a little dependent upon which version of Windows, but in almost all cases you can select "Run" and type CMD in the box.)

In Linux, you can right-click the desktop or search the menus for words "Linux terminal", "Linux console" (various distros do it their own way).

In Mac, you can find the 'Terminal' program in the Applications > Utilities folder. (If you don't see it there, open Spotlight (the magnifying glass in the top right corner of your screen) and search for Terminal.

So now, assuming you have your command line tool open, it's time to begin using Git.

But before Git will allow you to interact with it on any project, you must set up two very simple configuration options. You will identify yourself within Git by adding your name and an email address. These can even be fabricated values (like "A User" and "User@email.com") if you prefer.

At the command line, type the following:

```
git config --global user.name "Your Name Here"
git config --global user.email "your_emai@example.com"
```

Now that you and Git have become acquainted - with a proper introduction - let's start by creating a project directory (folder) anywhere on your system, and call it 'Halo Whirled'.

In the instructions below, whatever you need to type into the command line tool will be presented in a special font like this:

```
mkdir "Halo Whirled"
```

Once you have entered the required commands, you will need to press the Enter key (Windows) or Return Key (Mac) to run the code.

Now, the first thing we want to do in the command line is decide where on your computer you want to store this project. Let's say, for example, that there's a folder on your computer called Documents, and that is where you want to store this test project. In that case, in the command line, type the following:

```
cd Documents
```

You should see a clear indication that you are in the "Documents" folder.
Depending upon your operating system (Windows, Mac, Linux) you may see other things such as a dollar sign.

So, now that we are in the Documents folder, it's time to create a folder for our project. To do this, type

```
mkdir "Halo Whirled"
```

in the command line, and hit Enter/Return. (The quotes are only really necessary in Windows, but they don't hurt in Linux or Mac)

You will notice a lot of funny commands like "mkdir". They are often shorthand for normal English instructions. For example, mkdir means "make directory". {Until more recent versions of Windows, Folders in computer systems were called **Directories** because they 'contained' a collection of folders}

Now that we have created the Halo Whirled folder, go into that folder by typing

```
cd "Halo Whirled"
```

CD means "change directory".

Now we are inside our new project folder. And this is where you would normally go to work on a project: you create a project folder and then return to it whenever you want to continue work on the project.

So far, we haven't used Git at all. So now we need to tell Git to get to work in this folder. Each command to Git consist of two words, and the first is always 'git'. To get Git going in our new project, we simply need to type

```
git init
```

You will get back a message that verifies Git is on the job and ready to start working for you with this project. So that's it: we have set up a new project, and Git is ready to help us manage it!

The Commit—setting your work in stone

The 'Commit' is fundamental to Git and source control, and is the most common action you will take.

A commit is a snapshot in time. It represents a reproducible state of your project. The various commits you make during a project constitute the project's 'history'.

Let's add some files to our project, and then look at how to tell Git which files you want it to keep track of. We will create a text file called 'TheFirstFileUnderGitSourceControl.txt'. In Linux and Mac this is as easy as typing:

```
touch TheFirstFileUnderGitSourceControl.txt
```

Now let's add some content to this file. For now, we'll add content to this file in the 'ordinary way'—by navigating on your computer to the Documents > Halo Whirled folder. Open the new TheFirstFileUnderGitSourceControl.txt file and type the year you were born. On a new line, type your favorite flavor of icecream. E.g.

```
I was born in 1842.
My favorite flavor of ice cream in vanilla.
```

Once you're done, save the file

Now let's return to the command line tool and then tell Git to start tracking the history of this new file. Type this command:

```
git add TheFirstFileUnderGitSourceControl.txt
```

Don't expect any feedback at this stage. Git has happily accepted your command and executed it instantly. Git now knows that you are going to track the progress of this file.

Let's now capture the current state of this file for all time by performing our first commit. Type the following in the command line:

```
git commit -am "Initial Commit"
```

A commit has at least two parts: the file(s) to be included and a commit 'message'. The message is just a handy label (one or more tags, a description or whatever) that we add to help us when searching through a long list of commits in the history. We will be doing that shortly.

The result of the commit command will be a little spurt of text and, if you look at it carefully, it actually shows you what was 'saved' in the commit.

Now the fun begins. Once again, open TheFirstFileUnderGitSourceControl.txt file in your text editor. Delete the text you typed earlier and add a whole lot of new text. (Copy and paste some text from anywhere you like—even from this document.) Then save it and also close it.

Now, back in your command line tool, type this:

```
git commit -am "Changed all the text"
```

You have just begun to develop a history in Git. The command above has committed the new changes to this project's history.

This example of using Git has involved only one file, but this process applies equally to any number of files and/or folders.

Checkout—stepping back in time

Let's say we now want to use Git to access a previous state of our project. To do so, we can use the 'checkout' command.

In the command line, type the following:

```
git checkout HEAD^
```

(Don't be concerned by all the scary messages Git throws at you at this point. The world is not about to end!)

The checkout command is not one you typically use, but it's handy for us at this point. To see the effect of the checkout command, open the "TheFirstFileUnderGitSourceControl.txt" file in your text editor again. You will see that it has returned to its previous state.

It's important to note here that we have not deleted any of the history of our project by running the checkout command. The commits you make to your history are permanent. There is nothing you can do in Git to make them disappear.

Normally, using checkout in this way is not how we would return to a previous state of a project. We were cheating a little there, just to keep it simple for now.

When we made our commits earlier, we added messages to our commits to make them easier to find. Below, you will learn a better way to access the various states of your project. Let's first take our project a little further and make it a bit more 'real world'.

Go to your project folder ('Halo Whirled') on your computer, and within this folder, place a new web page file, calling it index.html. (Use your preferred text editor to create this file.)

Also create a new folder called 'images' inside the 'Halo Whirled' folder.

Now go back to your command line tool to commit these changes, typing:

```
git add index.html
git add images/
git commit -am "A Simple Web Project"
```

Our little project is now the beginnings of a simple web site. Next we'll explore how to view the history of what we've done so far.

Log—reviewing the history of your project

When you want to see the entire history of your project, and locate a particular point to 'checkout`, just issue the command:

```
git log
```

Once you've done this, you may notice that each entry—regardless of the commit message you created—has a unique signature, called a 'hash', which looks something like this:

`c178771270d4`

It is generated automatically by Git, and each one uniquely identifies a particular commit. You can use these signatures to revert to the various states in the project's history.

Try it with your current project. Although the actual string you use will be different, the command will look something like this:

```
git checkout c178771270d4
```

Realize that whenever you issue the `checkout` command, the set of files that are contained in that commit are restore to whatever state they were at the time of the commit. This means any files you currently have with the same name as those in that commit will be completely overwritten (replaced).

Status

There are numerous commands that Git recognizes. Likewise, those commands can be used to do some very sophisticated things; many of the things it does are related to groups of developers working on a project - often on the very same file(s). The purpose of this document is to provide you just enough to gain an understanding and a working familiarity with Git.

When you are in the process of making changes and committing changes to save various states you can lose track of what has been updated, changed or committed. A quite valuable command in Git (and it can be argued this is the __second__ most often used command) is the `status`. Try this:

```
git status
```

Now modify one of the files that is being tracked by Git. Then issue the command again:

```
git status
```

Make another commit. Then check the status once again

Finally, add another file to your project (you can create a new text file: README), check the status:
Add the file to Git for tracking with:

```
git add <filename>
```

And then check the status again.

```
git status
```

As you can see, this command allows you to maintain your bearings on what Git believes it is tracking.

Comparing Changes (It makes no diff to me)

By now your original file, “TheFirstFileUnderGitSourceControl.txt”, should have gone through many changes - all of which have been stored in the Git history.

Make sure you have committed any recent changes. Double-check there are no outstanding changes with the status command:

```
git status
```

Look at your history with the specific intent of locating the handle to that very first commit you made:

```
git log
```

Let’s assume that looks like this (yours will vary just a little):

```
commit c178771270d4ebf3f59f33201658f2a20d60eb01
Author: Thom Parkin <Thom.Parkin@Websembly.com>
Date:   Sat Jan 26 17:18:23 2013 -0500

    Initial Commit
```

The command to `checkout` that “Initial Commit” would be:

```
git checkout c178771270d
```

But don’t do that. {If you did accidentally checkout that commit, just checkout the last one; on the top of the list}

Instead, try this command:

```
git diff c178771270d
```

The `diff` command is intended to show you the DIFFERENCES between two commits. Specifically, it will show __exactly__ what changed in each and every file. Notice the plus (+) and minus (-) symbols. They indicate files and lines in the file that were added and removed (respectively). Many times this is very valuable information. And, Git provides the capability to retrieve those changes en masse or selectively. That is where some of the more detailed commands (and the concept of merging) would apply.

Branches (Going out on a limb)

As you may begin to realize, the security provided by a SCM like Git (security in the fact that your changes are not irrevocably lost) allows the freedom to experiment. Often, in software development, a new feature or an attempt to solve a problem may lead the code development process in a bit of a

tangent. This idea of branching is so common it is part of the fundamental design philosophy of Git. Assuming you have no uncommitted changes in your project (create a new commit if necessary), try this command:

```
git checkout -b trial_by_fire
```

This creates a new `branch` that represents the current point in the history. Having a branch allows you to (much like the branch of a tree) continue from this point without affecting the base point from which this stems. Let's make this clearer by example.

Get a list of ALL the files Git is tracking in this project:

```
git ls-files
```

Now, issue the delete (`remove`) command for each file:

```
git rm <filename>
```

Then, verify you have removed everything with

```
git status
‘
```

And make a new commit:

```
git commit -am “Deleted everything”
```

Now we can restore our project:

```
git checkout master
```

Voila!!

Remember that `master` is always the name of the main branch - the one from which you always start. Everything else “branches” from that. If you form an image in your mind of a railway system (like the subway near where you live) you get the concept of branching. Each “station” on the railway is like a `commit` in the Git history. You can “travel” to any point (commit) along the system (branches) to arrive at a previous “location” (which, in our case, is a state of the set of files)

By The Way: There is no limit to the number of branches you can create. You can always see the list of branches with:

```
git branch
```

There will be an asterisk to show you which branch is current. And, in order to `checkout` a branch so you can work on it (as you might have guessed):

```
git checkout <branchname>
```

Merging

As was stated earlier, one of the principle intentions of a Source Control Management system is for multiple developers to efficiently collaborate. That often requires them to disregard the polite attitude of taking turns when making changes to files. And, as you may recall, one of the highlighted features of Git is that the entire history is held locally - right there on your machine. So, how can multiple people collaborate, making changes to the same files, when they all have their own copies of those files? Without the special help provided by Git that would be impossible.

The details of a merge a just a bit beyond the scope of this introduction..

This document would be incomplete without a mention of the `remote`.

Remotes

In order for a group to collaborate on a project they must share the Git repository. This can be done a number of ways. Github is the most notable service that provides hosting of Git repositories. An account on Github is free and , in order to support Open Source Software development, there is no limit to the number of publicly visible repositories you can host.

It is important to realize that Git and Github are two distinct entities. Although many times they are used together in the same sentence it is incorrect to use them interchangeably.

Remember that your entire project 'history' is in the Git repository. And it is local on your machine. So a remote host, like Github, provides a centralized backup of that history.

HAVE NO FEAR!!

As you may now be able to see, when your set of files (project) is under Source Control (Git), the next time you find yourself shouting "AW SNAP!!" because you made an irrevocable change. All is not lost. There are even ways to undo changes that have not yet been committed! But that is beyond the scope of this document. A search on Google or Bing will yield plenty of detailed explanations. Also, check out the resources listed at the end of this article.

And, one more important thing to remember. In the simple example you walked through here we used a single text file ("TheFirstFileUnderGitSourceControl.txt"). That was for illustration. The same process can be applied to ANY FILE on your computer!

That should be enough for you to have an understanding of what Git can do and why anyone would want to use it. There are a few more topics - that are just a bit more sophisticated - and beyond the scope of this introduction. Perhaps we can provide some more detail in a future installment.

Return the Favor

Regardless of the type of work you are doing - whether software development, web design, graphic design - it is immeasurably rewarding to contribute your talents to the world of Open Source software. Find a project (on Github) you use, or like, or believe could be better.

Use the command

`git clone` (there are instructions right on the page with each repository on Github) and get a copy for yourself to work on and improve.

Github provides instructions for submitting a "Pull Request" to have your change reviewed and included by the project maintainer.

1.5 - A GIT Mini Reference

Perhaps this summary of what you did here will be handy as a reference while you explore and learn more:

- Tell Git what files to preserve and track (for changes)
  - `git add` <file, folder or list of files (separated by spaces)>
- To store the current state so you can always return to it (see "checkout")
  - `git commit -am` "A message that will help you understand what this contains"
- To get back (restore) all the files from a previous "commit"
  - `git checkout` <an unambiguous portion of the 'hash'>
- To get the 'hash' of all commits
  - `git log`
- To interact with "remote" git repositories (like projects on Github)
  - `git push` and `git pull`

ACKNOWLEDGMENTS: *@name; designates a SitePoint® username*

1. **Author** - Thom Parkin @ParkinT,
2. **Peer Reviewers -** Ralph Mason @ralph.m;, Nuria Zuazo @molona; Steve Browning @ServerStorm; , Matt Parkin;
3. **Editor** - Linda Jenkinson @Shyflower