A quick-start guide to Git

Anders Damsgaard
https://adamsgaard.dk, andersd@riseup.net

Last revision: January 1, 2017

1 What is Git?

Git is the most popular command-line tool for version control. It is most commonly used to track changes to plain-text files such as source code. When a software project is initialized as a Git *repository*, the history of the tracked files are recorded through a series of changes. When the user performs changes to the tracked files, she can choose to *commit* these changes. Git repositories can be managed through online services such as Github[1], which allows the changes to be synchronized between multiple contributers and end users. The same repository can contain multiple versions of the same files. These coexisting versions are called *branches*.

Git usage is typically very verbose and by design weighs explicitness over convenience. This ensures that its default behavior does not lead to unintended outcomes.

2 Installation

Please refer to the official documentation[2] for platform-specific instructions on how to install Git. My prefered installation method on OS X is through Homebrew[3]:

```
$ brew install git
```

On Debian-based systems, Git can be installed through the advanced package tool:

```
$ apt-get install git
```

Git has excellent built-in documentation through its man page (i.e. `man git`). For documentation on a sub-command such as `git add`, see its documentation with `man git-add`. For more complex tasks, I recommend referring to a Git handbook or searching the web (in that order).

[1] https://github.com
[2] https://git-scm.com/book/en/v2/Getting-Started-Installing-Git
[3] http://brew.sh

3 Getting started

Before using Git for version control, it is a good idea to record some information about yourself. This makes it is easy to see who specific commits can be attributed to when working on projects with multiple contributors. The user information is stored in a plain-text file in the home directory (`~/.gitconfig`), and can be created with the following commands:

```
$ git config --global user.name "John Doe"
$ git config --global user.email "john-doe-farms@aol.com"
```

4 Initializing a repository

In order to track changes to files in a directory, the directory needs to be initialized as a repository. Let's say that we want to track the changes to a file `arithmetic.c` which located in the directory `~/src/calculator`. We start off by initializing the directory as a repository:

```
$ cd ~/src/calculator
$ git init
Initialized empty Git repository in ~/src/calculator/.git/
```

Git lets us know that the directory is initialized as a new repository, and that the hidden sub-directory `.git` is used for the files related to the version control[4].

It is important to realize that just because you initialize a directory as a Git repository, the files and subdirectories are *not automatically tracked*! You need to manually specify which files inside of the directory you want to include in the version control system. There are several important reasons for this. As an example, during the compilation step many compilers create object files which are linked together to create the final executable binaries. These object files are created in machine code, and can have a significant size on the file system. If Git automatically recorded the changes and versions to all files inside of the repository, it would save all changes in the binary object files, which are of no use since they can be readily reconstructed from the source code.

5 Creating a local copy of an online repository

If you want to create a local copy of an online repository you can download a clone of it in its current stage to your local file system:

```
$ git clone git://github.com/john-doe/tractor-simulation
```

[4]This directory contains many interesting files, and I encourage you to explore it when you are more familiar with Git.

This will checkout the online repository on Github into a corresponding directory in the local directory. You can also choose to clone over other protocols such as SSH or HTTPS if you prefer.

The local repository will remember where it was cloned from. If you have write permissions to the online repository, you can upload your local commits using `git push`.

6 Adding files and commiting changes

To add a file to the version-control system inside a repository, use the following command:

```
$ git add arithmetic.c
```

One or more changes to the tracked files in a repository must be accompanied by a *commit message*. The commit message should ideally be short and descriptive of the changes that are contained in the commit. The commit messages are logged. It should be easy for a user to glance through the log of commit messages and understand the changes without reading the changes to the files themselves. This is what sets version-control systems aside from automatic backups. Only the user herself has the ability to identify when a set of changes are complete and significant, and can formulate a meaningful description of the changes in their respective context.

In case you have forgotten what you have changed in a file, use the following command:

```
$ git diff -- arithmetic.c
```

To commit all files which have been added to the repository, you can use the following command:

```
$ git commit -m "First commit of arithmetic.c"
```

If you ommit the `-m` flag and message string (i.e. simply type `git commit`), Git will open your favorite command-line editor[5]. You then write the commit message in the editor, and finalize the commit by saving and exiting the file.

If you perform subsequent edits to the file, you need to commit the new changes once again. We can either once again add the file and commit the changes:

```
$ git add arithmetic.c
$ git commit -m "Implemented multiplication"
```

Alternatively, since the file `arithmetic.c` is already added to the repository, you can commit *all* changes to *all* tracked files in the repository with a single command:

```
$ git commit -a -m "Implemented multiplication"
```

Are you not sure which files changed since the last commit? Git can show you an overview of the current state:

```
$ git status
```

[5]You can set which editor you want to use using the `EDITOR` environment variable in e.g. `~/.bashrc`.

7 Inspecting repository changes and reverting to a previous commit

Git can show you an overview of all recorded changes in the repository:

```
$ git log
commit 2745f1e3b4803f1c8728089667a18f3178cd18dc
Author: John Doe <john-doe-farms@aol.com>
Date:   Fri Sep  2 10:22:59 2016 -0700

    Implemented multiplication

commit 3329dfa1b6bfecc00353d1e9db50bcab9fb41521
Author: John Doe <john-doe-farms@aol.com>
Date:   Fri Sep  2 10:22:13 2016 -0700

    First commit of arithmetic.c
```

Git can show the changes to the files between any two commits:

```
$ git diff 3329dfa1b 2745f1e3b
diff --git a/arithmetic.c b/arithmetic.c
index e69de29..523d72b 100644
--- a/arithmetic.c
+++ b/arithmetic.c
@@ -0,0 +1,4 @@
+double multiply(double x, double y)
+{
+    return x * y;
+}
```

In case you want to roll back your most recent changes and revert the repository to a stage corresponding to an earlier commit. Each commit has an uniquely identifying string which is shown with the above command. To revert you need to supply the first 9 characters of this string:

```
$ git checkout 3329dfa1b
```

This will revert any changes contained in subsequent commits. In case you change your mind and want to go back to the most recent commit, use:

```
$ git revert HEAD
```

The special string `HEAD` refers to the most recent commit, and the commit before that is referred to by `HEAD^`, while the third-most recent commit is `HEAD^^`. These special strings are convenient, for example when you want to see what changed during the most recent commit, which can be done with the command `git diff HEAD^ HEAD filename`.

8 Branching and merging

Git allows you to have multiple versions (branches) of the same repository. The first step is to create a new branch and give it a suitable name:

```
$ git checkout -b new_interface
```

Subsequent commits are staged to the new branch `new\_interface`. You can see which branches are present in the repository with `git branch`:

```
$ git branch
  master
* new_interface
```

where `master` is the original branch. The asterisk denotes what current branch is active. You can switch between branches, which will automatically apply all relevant patches to the affected files:

```
$ git checkout master
```

To delete a branch, use `git branch -d new\_interface`. To merge another branch into your current active branch, use:

```
$ git merge new_interface
```

When merging, all commits and file changes performed in a branch are applied to the currently active branch.

9 Ignoring files

Many compilers create auxillary files which are never relevant to track in a version-control system, but clutter your repository overview when using commands such as `git status` or `git commit -a`. You can specify which files Git should ignore by their filename in a file at the root of the repository in a file named `.gitignore`. For a repository containing C code, an example `.gitignore` file could contain:

```
*.o
```

This will ignore all object files. For a LaTeX repository the file could contain:

```
*.aux
*.glo
*.idx
*.log
*.toc
*.ist
*.acn
*.acr
*.alg
```

```
*.bbl
*.blg
*.dvi
*.glg
*.gls
*.ilg
*.ind
*.lof
*.lot
*.maf
*.mtc
*.mtc1
*.out
*.xdy
*.synctex.gz
```

It is up to you to specify the contents of the `.gitignore` file. Maybe your program generates output files, which should not be tracked. Simply add their names or file type to the `.gitignore` file and never encounter them in your Git workflow again.

10 Extra: Useful shell aliases

I like to bind short aliases to the most commonly Git commands. I do this by appending the following to the `rc` file of my shell (`~/.zshrc` or `~/.bashrc`):

```
alias gs='git status | less'
alias gl='git log --graph --oneline --decorate --all'
alias ga='git add'
alias gd='git diff --'
alias gc='git commit -v'
alias gca='git commit --all --verbose'
alias gp='git push'
alias gpu='git pull'
alias gcgp='git commit --verbose && git push'
alias gcagp='git commit --all --verbose && git push'
```

Using these aliases I can quickly add a file (`ga file.c`). Alternatively, I can quickly commit all changes to all files that are already tracked in the repository (`gca`). With `gl` I can quickly see the commit tags and commit messages in short form, and scroll up and down with `j` and `k` or the arrow keys. `gs` gives me a quick overview of the changes in the current repository, and uses the same keys as `gl` for scrolling.