A Basic Introduction to Git Version Control Software

Revision History

Version	Date Complete	Person	Updates
0.1	Oct. 18, 2017	Andrew Brandt	Add init, set upstream, push, pull, add, commit, branch, merge, and merge conflict resolution
0.2	Oct. 19, 2017	Andrew Brandt	Add installing Git, .gitconfig file, and .gitignore file
0.3	Oct. 19, 2017	Andrew Brandt	Major formatting changes, added images
1.0	Oct. 19, 2017	Andrew Brandt	First release

Introduction

The purpose of this document is to demonstrate basic commands and usage of Git. By the end of this document, you will be able to:

1. Install Git and set up global variables
2. Version control files and folders in a directory
3. Create and switch branches
4. Perform merging and handle merge conflicts
5. Handle local files and remote locations
6. Set Up .gitconfig and .gitignore Files

Syntax

Normal text is for explanation to the reader.

`Monospace font indicates commands that should be typed by the reader or directory locations.`

Italics are used for term definitions and sub-headings.

Brackets should be replaced with actual names, inclusive of brackets.

Example: `"<filename>" = "test.txt"`

Git's power is well known throughout the industry, but it has a bit of a learning curve. This document is intended to help the beginner get up and running with Git quickly and help the intermediate Git user gain more confidence. Source - XKCD.

Installing Git on Windows

Installing Git on Windows is quite simple. Navigate to the download page here and download Git for Windows. Follow the installation instructions. I suggest selecting "Use Git from the Windows Command Prompt" so that you can use Git through Windows PowerShell. This is more convenient than using it through only the Git Bash command line.

Setting Up Git's Global Variables

If this is your first time using Git, there are several global variables that should be edited.

1. Ensure that Git is functioning by typing the following. An error stating "Fatal: not a git repository" should occur and be printed to terminal if you are not inside a Git repository folder:
   ```
   git status
   ```
2. Change the name that your account is known by. This is necessary so each person can see who has made what commits and is responsible for what code:
   ```
   git config --global user.name "<First Last>"
   ```
3. Change the email associated with your commits. This should be kept up to date so people have a point of contact if they have questions about a commit:
   ```
   git config --global user.email "<email@email.com>"
   ```
4. Optional: Change the default editor from vi to something more user friendly. I prefer Notepad++, but this is entirely your choice. You can also shorten this significantly by adding the location of the .exe file to your system PATH variable:
   ```
   git config --global core.editor "'C:\\Program Files\\Notepad++\\notepad.exe' -multiInst -notabbar -noPlugin -nosession"
   ```

Creating a Repository on Github

The easiest way to create a Git repository is through Github. Click the "+" in the top right corner, then add a name for your new repository.

Creating a Repository on Remote Location

Ensure that the network drive is mapped to a letter. The example we will use is the `T:` drive.

1. Move to the proper directory:
   ```
   cd T:\directory\to\initialize
   ```
2. Initialize a "bare" Git repository in the location desired. The "bare" is necessary otherwise multiple users will cause a conflict:
   ```
   git init --bare
   ```

Cloning a Repository to Local Location from Github

There are two ways to clone from Github. This guide will cover cloning using HTTPS because there is no SSH key setup required.

1. Move to the proper directory:
   ```
   cd C:\directory\to\clone\into
   ```
2. Clone the git repository:
   ```
   git clone https://github.com/username/repository-name
   ```

Cloning a Repository to Local Location from Remote Location

1. Move to the proper directory:
   ```
   cd C:\directory\to\clone\into
   ```
2. Clone the Git repository:
   ```
   git clone file://T:\location\to\clone\from
   ```

Checking the Status of Files Inside a Git Repository Folder

To find if a Git repository is ahead of the remote (data committed locally has not been pushed up to the remote) or is behind the remote (other data has been pushed to remote that is not on local), we check the status of the repository. We will get a message telling us if we are ahead of the remote repository by X commits:

1. Move to the Git repository location:
   ```
   cd C:\repository\directory\location
   ```
2. Check the status:
   ```
   git status
   ```

Adding and Committing Files Theory

Adding files for tracking and committing the files are important for tracking files through version control. There are two main parts to tracking the files: "Staging" the files and "committing" the files.

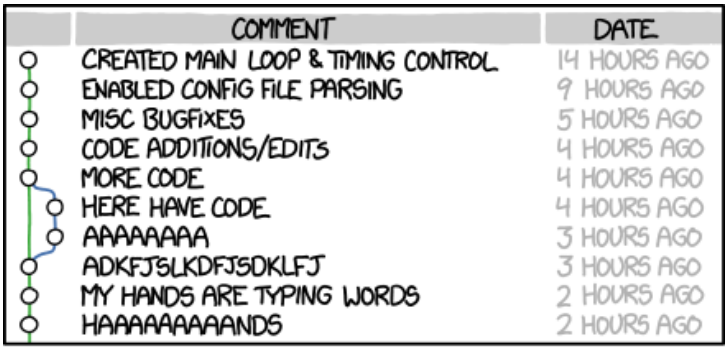
Staging: This is when the version of the file is prepared to the snapshotted and saved. Staging all files in the directory is common to save an entire snapshot of the directory at the time of the commit.

Committing: This is when the actual "save point" of the version control is created. Files can be staged, then modified, but committed files are much more challenging to edit (and is usually bad practice!).

Commit "Naming": Commits are identified by SHA-1 hash of all the information in the commit. The last 7 characters of the hash are how commits are identified. When `<commit>` is visible, simply replace this with the 7 character commit you wish to use.

A few notes on committing:

1. Commits should always be functional. Never commit just to add another save point.
2. Commit messages should be short but descriptive. There is no limit on the length of a commit message, but it should be 80 characters or less so another developer can understand quickly exactly what was accomplished with the commit.
3. Commit early and often! Even though commits should be functional, commit frequently so if a feature is broken it is easy to roll back to a previous commit where the feature is functional again.



We should have short and descriptive commit messages. This is especially true when working with other developers on the same project. Source - XKCD.

Adding and Committing Files

Now that we understand "staging" and "committing", here is how we perform the actions in Git:

1. Adding files to be staged and ready for commit:
   `git add <filename1> <filename2> ...`
   It is possible to add all files:
   `git add -A`
2. Once files are staged, we need to commit them:
   `git commit`
   This will open the default editor (probably Vim) for you to enter a commit message. If vim is not the desired editor, this can be changed through the .gitconfig file (see below). The easiest way to insert a commit message is by using a flag:
   `git commit -m "<Commit message here>"`
   We can also stage and commit all tracked files (files that are being tracked for changes) in a single command:
   `git commit -am "<Commit message here>"`

Resetting to a Previous Commit

There are three ways to reset to a previous commit.

1. We can revert and keep changes locally so a new commit will be created. Use this if you want to go a different direction, but you've already begun writing the new direction and committed it:
   `git revert <commit number>`

2. We can revert and trash all local changes since the commit we are reverting to. Use this if something got corrupted or damaged:
   `git revert --hard <commit number>`
3. We can revert and preserve all changes locally as an unstaged commit. Use this to backtrack but keep all local changes. All local changes will be put in a single new commit:
   `git reset <commit number>`

Set Upstream Push Location

For Git to work correctly, you must set a location for each push of data you perform. This must be set on every branch created and worked on.

1. Set upstream, replacing the <branch name> with the proper branch name (default branch is "master"):
   `git push --set-upstream origin <branch name>`
   Example for "master" branch:
   `git push --set-upstream origin master`
2. Now we can push our commits up to a remote location:
   `git push`

Pushing Local Commits to Remote

We want to commit on our local machine, but then push these changes up to the remote so that other people have access to the correct data. To update the remote location, we use the push command:

1. We move to the root of the Git repository:
   `cd C:\repository\directory\location`
2. Push to remote, assuming we have already set the upstream as above:
   `git push`

Pulling from a Remote to Local

There are two different methods to pull data from a remote location to a local location. Fetch will download the new data, but it will not integrate the changes into the local. Pull will both download the data and will integrate the changes into the local.

1. Fetching data from remote to local:
   `git fetch`
2. Pulling and integrating data from remote to local:
   `git pull`

Branching Theory

Git's true power is in branching, allowing people to work on different areas of the system without interfering with one another. There should be several branches in each Git repository:

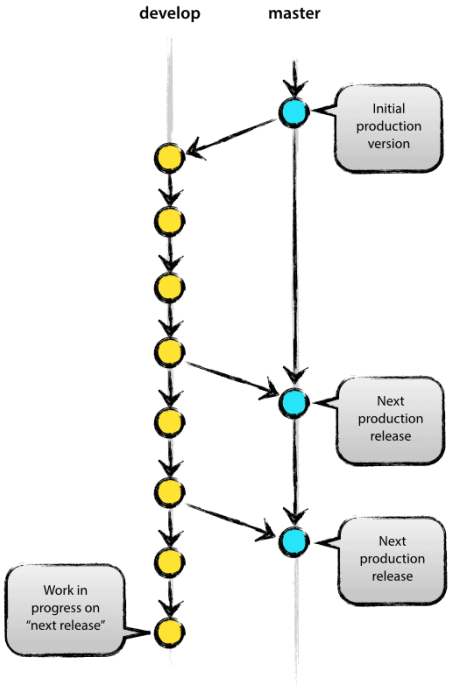
`master` branch: This branch is the master branch. This should never be worked on or pushed to by a single person. It should only be merged to when there are significant changes. It is the most stable branch in the repository.

`development` branch: This is the branch that holds working developments and feature additions. This branch can be merged into the master branch, but also should not be pushed to. This is the stable branch that holds any feature changes that have been tested and approved.

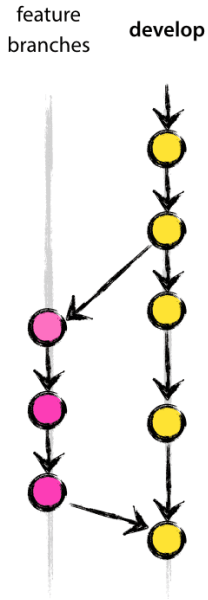
`feature` branch: This is a branch that can take any name, based on what feature is being worked on. Examples could include "gui" for creating a GUI, "kalman" for developing a Kalman filter, etc. These

branches are directly created from the development branch or created from a different feature branch to create many levels of branches.

Developers should work on a specific feature in a single branch, then merge the results back into the development branch, and finally into the master branch when the time is appropriate.



The development branch is separated from the master branch, but is merged with master at significant points. [Source - nvie.com.]



A feature branch may work off the development branch. Note the feature branch is closed when the feature is completed and merged back into the development branch. [Source - nvie.com.]

Creating a Branch

Now that we know why it's important to utilize branches, here's how we create them:

1. Check out the branch that the new branch should be created from (usually development branch):
   `git checkout <branch to create from>`
2. Create a new branch and switch to the new branch in 1 command:
   `git checkout -b <new branch name>`

Listing All Branches

It is important to be able to see all branches and which branch we are currently on.

1. List all branches, asterisk will show which branch is currently checked out:
   `git branch`
2. List all branches and extra information:
   `git branch -av`

Pushing to a New Branch

To push a new branch created locally to the remote, there are three main steps that need to be taken:

1. Set the upstream location (see above)
2. Add files to be committed (see above)
3. Commit the files (see above)

Deleting a Branch

Deleting a branch can be useful to clean up old branches that are no longer under active development or if a branch was created by accident.

1. Delete a local branch:
   `git branch -d <branch name>`

Merging Conflicts

Merging conflicts in the same file from two users is one of the most challenging functions of any version control software program. Git handles it by creating markers for differences in each file a merge conflict is found, then relies on the user to clean them up. The user has two options for cleaning up merge conflicts; use the text editor or use a built-in merge tool (like vimdiff).

1. Pull the branch from remote. A message will be printed in the terminal if a merge conflict occurs. There will be a list of files that have conflicts. This is important because it shows which files now have the differences printed in them so the user knows which files to edit.

Using Text Editor:

Go to each file that has a conflict and scroll through it, cleaning up each conflict point as necessary.

Using Merge Tool:

Open the Git merge tool and change the output file to what is desired:

Finally:

2. Add all files to be staged for a commit:
   `git add -A`
3. Commit all merged files. Now the merge conflict is resolved:
   `git commit -m "<Commit message about merging all files>"`
4. Push the newly merged and committed files to remote:
   `git push`

5. Pull the files to ensure the files were pushed successfully and the remote is up to date:
   `git pull`

Merging Branches

Merging branches is one of the most important features in using Git with many people. Branches are created for feature development, bug solving, or experimental features in isolation away from the production code. In this example, the "feature" branch will be the feature addition branch, and the "development" branch will be the branch we wish to merge into.

1. Check out the feature branch:
   `git checkout <feature branch>`
2. Check out the development branch:
   `git checkout <development branch>`
3. Merge the feature branch into the development branch (Note: this keeps all commits from the feature branch and puts them in the development branch, which is not best practice. See below for best practice.):
   `git merge <feature branch>`
4. If there are merge conflicts, resolve them following the steps above, commit on the "development" branch, then push to remote to complete the merge steps.

Merging Branches Without Fast Forwarding

Git has the option of always merging branches using a merge commit or simply sliding the pointer over onto the higher-level branch. We want to maintain structure, so we should always use the no fast forwarding flag. See the image below:

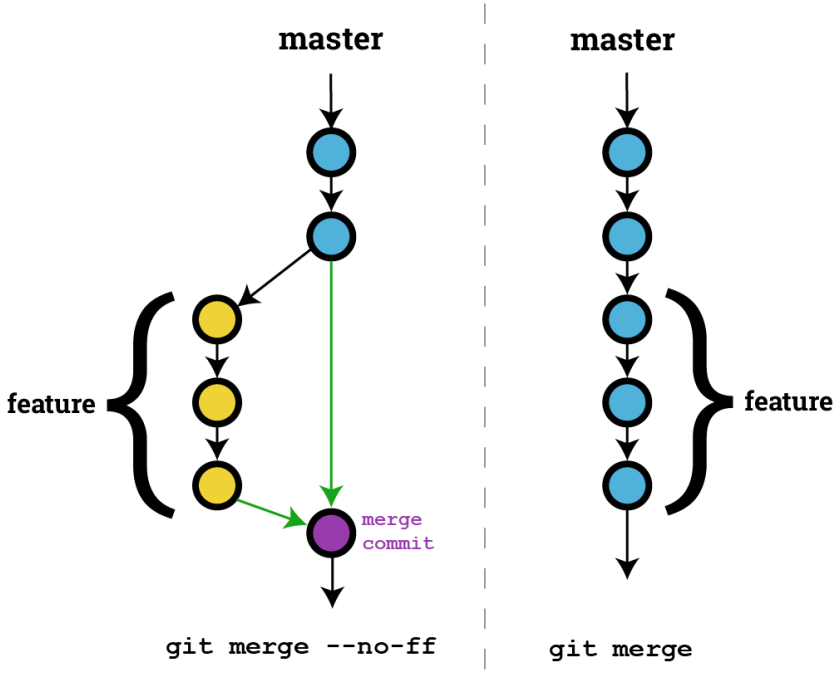


Figure showing the difference between forcing a merge commit and simply fast forwarding the branch pointer to master. Source - nvie.com.

Since we don't want to fast forward, we will use the flag no fast forward to ensure the development branch remains structured correctly with only feature additions. Arguably, this problem will never occur because the feature commits will be squashed first before being added to the development branch.

1. Checkout the branch we want to merge into:
   `git checkout <development branch>`
2. Merge branch with no fast forwarding:
   `git merge --no-ff <feature branch>`

Squashing Commits

Squashing commits in branch before merging into a higher-level branch (feature branch to development branch) is important to keeping the development branch clean of small commits and preserving only large blocks of code updates.

The Safe Way, Highly Recommended:

This is the preferred way because it is both the safer way to squash commits and because it preserves all commits on the feature branch. It is safer because the squash occurs on a separate branch away from the feature branch and the development branch. If an error is made the squash branch can be deleted and re-added.

1. Checkout the feature branch:
   `git checkout <feature branch>`
2. Create a new branch to do the squashing on off the feature branch:
   `git checkout -b <squash branch>`
3. Rebase the squash interactively from the feature branch:
   `git rebase -i <feature branch>`
4. Pick which commits to keep and which commits to squash by editing the word “pick” to “squash” in front of each commit. The “pick” commits will remain while the “squash” commits will be wrapped up into the most recently picked commit. Save the file.
5. Enter a new commit message for all the squashed commits if desired. Remember that this commit message will stand in for all squashed commits, so make it detailed if necessary. Checkout the development branch or master branch (branch to merge into):
   `git checkout <development branch>`
6. Merge the squash branch into the development or master branch:
   `git merge <squash branch>`

The Unsafe Way, Don’t Do This Unless You Know What You’re Doing:

This method will work, but it is much less safe because there is not an isolated branch that all the squashing is performed on. The commits are also squashed on the feature branch, so they will not be accessible in the future.

1. Checkout the development branch:
   `git checkout <development branch>`
2. Merge the feature branch into the master branch with the squash option:
   `git merge --squash <feature branch>`
3. Add all files to the commit message:
   `git add -A`
4. Commit all the files:
   `git commit -m "<Commit message for merge here>"`

Tags

Tags are used usually used to mark revisions and release points. These are important because they help keep clear markers of major revisions. There are two types of tags: Annotated and Lightweight. We will only cover annotated in this document because lightweight is included in an annotated tag.

1. Creating an annotated tag:
   `git tag -a <version> -m "<annotated tag message>"`
2. Viewing a tag:
   `git show <version>`

Adding and Editing a .gitignore File in a Git Repository

There may be files you do not want to add consistently to the commits. Instead of adding all files except certain extensions or files, we can create a .gitignore file that will keep certain files out. Examples of these files are .db files (such as thumbnail files), .ini files (generated by the operating system), or .class files (compiled files that should not be tracked through Git). The .gitignore file can use wildcards, so *.zip will eliminate all files with the extension .zip from being added to commits.

1. Create the .gitignore file using a text editor (Notepad, Notepad++, Vim, etc.).
2. Add the .gitignore file to the root of the Git repository directory. `git add -A` will now add all files except those in the .gitignore file.

See the end of this document for my most current .gitignore file.

Editing the .gitconfig File for Aliasing

Since Git runs from the command line, it is easy to add aliases and shortcuts for any command imaginable in Git.

The location of your .gitconfig file should be in `C:\Users\username\.gitconfig`

Open this file and you can add aliases, modify name and email, and even change colors. Below are some helpful aliases for easier use of Git:

The [alias] tag is used so Git can identify where the aliases occur. Every alias should then be indented below the alias tag, with each alias on a new line:

```
[alias]
```

Instead of typing the whole "commit -m" every time, I prefer to shorten it.

Before: `git commit -m "<Commit message>"`
Use: `git ci "<Commit message>"`
Alias: `ci = commit -m`

When adding files to staging, it is rare that you want to keep specific files out.

Before: `git add -A`
Use: `git aa`
Alias: `aa = add -A`

When using Git, you frequently check the status of your Git repository. This alias will reduce the number of keystrokes necessary for the status check.

Before: `git status`
Use: `git st`
Alias: `st = status`

Checking to see when you committed last is important because you will know how much work you have done without committing. This alias will reduce keystrokes significantly.

Before: `git log -1 HEAD`
Use: `git last`
Alias: `last = log -1 HEAD`

When adding and committing files, usually you will add all files then commit all files. This is good for when you need to know exactly what files are being committed, but if you're continuously committing all files in the directory, then it is possible to reduce these two steps down to one step. I find that this is my most frequently used Git alias.

Before:
```
git add -A
git commit -m "<Commit message>"
```
Use:
```
git cm "<Commit message>"
```
Alias:
```
cm = !git add -A && git commit -m
```

There are several different ways to visualize the branching, commits, and merging that occur in a Git repository. These aliases are different ways to visualize the progress.

Use:
```
git lg
```
Alias:
```
lg = log --all --decorate --oneline --graph
```

Use:
```
git lg1
```
Alias:
```
lg1 = log --graph --pretty=format:'%Cred%h%Creset -
%C(yellow)%d%Creset %s %Cgreen(%cr) %C(blue)<%an>%Creset' --abbrev-
commit --date=relative
```

Use:
```
git lg2
```
Alias:
```
lg2 = log --graph --abbrev-commit --decorate --
format=format:'%C(bold blue)%h%C(reset) - %C(bold green)(%ar)%C(reset)
%C(white)%s%C(reset) %C(dim white)- %an%C(reset)%C(bold
yellow)%d%C(reset)' --all
```

Closing Thoughts

This guide is in no way meant to be exhaustive. There are many more functions that can be performed by the user. To consider these further, peruse Stack Overflow, Google, or look at the Pro Git book by Scott Chacon and Ben Straub here. Many images were taken from nvie.com, specifically because their branching diagrams and explanations are clear and effective. I highly recommend reading through the website.

If you have any further questions, feel free to reach out to me at brandt@mmsi.com or brandt.andrew89@gmail.com

Thank you for reading!

My Full .gitconfig File

Just copy and paste below into your own for all my shortcuts:

```
[user]
      name = Andrew Brandt
      email = brandt.andrew89@gmail.com
[filter "hawser"]
      clean = git hawser clean %f
      smudge = git hawser smudge %f
      required = true
[filter "lfs"]
      clean = git lfs clean %f
      smudge = git lfs smudge %f
      required = true
[color]
      ui = true
[credential]
      helper = wincred
[alias]
      lg = log --all --decorate --oneline --graph
      ci = commit -m
      aa = add -A
      st = status
      last = log -1 HEAD
      cm = !git add -A && git commit -m

      lg1 = log --graph --pretty=format:'%Cred%h%Creset -
%C(yellow)%d%Creset %s %Cgreen(%cr) %C(blue)<%an>%Creset' --abbrev-
commit --date=relative

      lg2 = log --graph --abbrev-commit --decorate --
format=format:'%C(bold blue)%h%C(reset) - %C(bold green)(%ar)%C(reset)
%C(white)%s%C(reset) %C(dim white)- %an%C(reset)%C(bold
yellow)%d%C(reset)' --all

      lg3 = log --graph --abbrev-commit --decorate --
format=format:'%C(bold blue)%h%C(reset) - %C(bold cyan)%aD%C(reset)
%C(bold green)(%ar)%C(reset)%C(bold yellow)%d%C(reset)%n''
%C(white)%s%C(reset) %C(dim white)- %an%C(reset)' --all

[core]
    # if notepad++ is added to the system path
      editor = notepad++ -multiInst -notabbar -nosession -noPlugin
    # if notepad++ is not added to the system path
    # editor = 'C:\\Program Files\\Notepad++\\notepad++' -multiInst -
notabbar -nosession -noPlugin
```

My Full .gitignore File

Just copy and paste below into your own for all ignore files. Remember to add a copy of this file to each Git repository you work with:

```
# Compiled source #
###################
*.com
*.class
*.dll
*.o
*.so

# Packages #
############
# it's better to unpack these files and commit the raw source
# git has its own built in compression methods
*.7z
*.dmg
*.gz
*.iso
*.jar
*.rar
*.tar
*.zip

# Logs and databases #
######################
*.log
*.sql
*.sqlite

# OS generated files #
######################
.DS_Store
.DS_Store?
.ini
._*
.Spotlight-V100
.Trashes
ehthumbs.db
Thumbs.db
```

The Cheat Sheet

The purpose of this page is to provide a quick reference to developers who understand the theory of Git, but need to have access to all major commands in 1 location.

Pull Files from Remote and Integrate Changes
```
git pull
```

Download Files from Remote but Do Not Integrate Changes
```
git fetch
```

Stage Files to Commit
```
git add <filename1> <filename2> ...
```

Stage All Files to Commit
```
git add -A
```

Commit Files
```
git commit -m "<Commit msg>"
```

Set Upstream to Remove (Must be done for each branch)
```
git push --set-upstream origin
<branch name>
```

Push Commits to Remote
```
git push
```

List Branches
```
git branch
git branch -av
```

Checkout a Branch
```
git checkout <branch name>
```

Create and Checkout a Branch
```
git checkout -b <branch name>
```

Delete a Branch
```
git branch -d <branch name>
```

Merge Conflicts in a Branch
```
git pull
```
Open conflicting files in text editor, remove conflicts
OR
```
git mergetool
```
edit, then
```
git add -A
git commit -m "<Commit msg>"
```

Merge a Branch
```
git checkout <development branch>
git merge <feature branch to merge>
```

Squash Commits and Merge Branch
```
git checkout <feature branch>
git checkout -b <squash branch>
git rebase -i <feature branch>
```
Choose which commits to "pick" and "squash"
Enter new commit message, comment out which items to not show in commit message
```
git checkout <development branch>
git merge <squash branch>
```

Reverting Commits and Keeping Local Changes
```
git reset <commit>
```

Reverting Commits and Trashing Local Changes
```
git revert --hard <commit>
```