**Git & GitHub**

## The Three States of Git

Git has three main states that files can reside in: committed, modified and staged:

* **Committed** means that the data is safely stored in your local database.
* **Modified** means that you have changed the file but have not committed it to your database yet.
* **Staged** means that you have marked a modified file in its current version to go into your next commit snapshot.

This leads us to the three main sections of a Git project:

* **The Git Directory**
* **The Working Directory**
* **The Staging Area.**



The Git directory is where Git stores the metadata and object database for your project. This is the most important part of Git, and it is what is copied when you clone a repository from another computer.

The working tree is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.

The staging area is a file, generally contained in your Git directory, that stores information about what will go into your next commit. Its technical name in Git parlance is the “index”, but the phrase “staging area” works just as well.

The basic Git workflow goes something like this:

1. You modify files in your working tree.

2. You selectively stage just those changes you want to be part of your next commit, which adds only those changes to the staging area.

3. You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

If a particular version of a file is in the Git directory, it’s considered committed. If it has been modified and was added to the staging area, it is staged. And if it was changed since it was checked out but has not been staged, it is modified.

## Getting a Git Repository

You typically obtain a Git repository in one of two ways:

1. You can take a local directory that is currently not under version control, and turn it into a Git repository, or
2. You can clone an existing Git repository from elsewhere. In either case, you end up with a Git repository on your local machine, ready for work.

### **Initializing a Repository in an Existing Directory**

If you have a project directory that is currently not under version control and you want to start

controlling it with Git, you first need to go to that project’s directory.

$ cd /c/user/my\_project

$ git init

This creates a new subdirectory named .git that contains all of your necessary repository files — a Git repository skeleton. At this point, nothing in your project is tracked yet.

If you want to start version-controlling existing files (as opposed to an empty directory), you should probably begin tracking those files and do an initial commit. You can accomplish that with a few git add commands that specify the files you want to track, followed by a git commit:

$ git add \*.c

$ git add LICENSE

$ git commit -m 'initial project version'

We’ll go over what these commands do in just a minute. At this point, you have a Git repository with tracked files and an initial commit.

## Cloning an Existing Repository

If you want to get a copy of an existing Git repository — for example, a project you’d like to contribute to — the command you need is git clone. Instead of getting just a working copy, Git receives a full copy of nearly all data that the server has. Every version of every file for the history of the project is pulled down by default when you run git clone.

$ git clone <remote\_URL>

### **Recording Changes to the Repository**

At this point, you should have a bona fide Git repository on your local machine, and a checkout or working copy of all of its files in front of you. Typically, you’ll want to start making changes and committing snapshots of those changes into your repository each time the project reaches a state you want to record.

Remember that each file in your working directory can be in one of two states: **tracked or untracked**.

* Tracked files are files that were in the last snapshot; they can be unmodified, modified, or staged. In short, tracked files are files that Git knows about.
* Untracked files are everything else — any files in your working directory that were not in your last snapshot and are not in your staging area.

When you first clone a repository, all of your files will be tracked and unmodified, because Git just checked them out and you haven’t edited anything.

As you edit files, Git sees them as modified, because you’ve changed them since your last commit.

As you work, you selectively stage these modified files and then commit all those staged changes, and the cycle repeats.



## Git Workflow:

### **Checking the Status of Your Files**

The main tool you use to determine which files are in which state is the **git status** command.

CKP@DESKTOP-Q1TQ944 MINGW64 /e/Learn - IT Skills/GitHub/WorkSpaces/LearningGit (master)

$ git status

### **Tracking New Files**

In order to begin tracking a new file, you use the command **git add**.

$ git add file\_name

$ git add -A

$ git add –all

$ git add .

After git add, a file is staged and so it’s under the “**Changes to be committed**” heading.

### **Staging Modified Files**

If you modify a previously tracked file then git will list the file under a section named “**Changes not staged for commit**” — which means that a file that is tracked has been modified in the working directory but not yet staged. To stage it, you run the git add command. At this point, suppose you remember one little change that you want to make in that file before you commit it. You open it again and make that change, and you’re ready to commit now. At this point if you run a git status, then it can be found that **the file is listed as both staged and unstaged.**

How is that possible? It turns out that Git stages a file exactly as it is when you run the git add command. If you commit now, the version of file as it was when you last ran the git add command is how it will go into the commit, not the version of the file as it looks in your working directory when you run git commit.

***If you modify a file after you run git add, you have to run git add again to stage the latest version of the file.***

### **Ignoring Files**

Often, you’ll have a set of files that you don’t want Git to automatically add or even show you as being untracked. These are generally automatically generated files such as log files or files produced by your build system. In such cases, you can create a file listing patterns to match them named **.gitignore**.

The rules for the patterns you can put in the .gitignore file are as follows:

* Blank lines or lines starting with # are ignored.
* Standard glob patterns work, and will be applied recursively throughout the entire working
* tree.
* You can start patterns with a forward slash (/) to avoid recursivity.
* You can end patterns with a forward slash (/) to specify a directory.
* You can negate a pattern by starting it with an exclamation point (!).

Here is an example .gitignore file:

$ cat .gitignore

\*.[oa]

\*~

# ignore all .a files

\*.a

# but do track lib.a, even though you're ignoring .a files above

!lib.a

# only ignore the TODO file in the current directory, not subdir/TODO

/TODO

# ignore all files in the build/ directory

build/

# ignore doc/notes.txt, but not doc/server/arch.txt

doc/\*.txt

# ignore all .pdf files in the doc/ directory and any of its subdirectories

doc/\*\*/\*.pdf

Glob patterns are like simplified regular expressions that shells use.

* An asterisk (\*) matches zero or more characters
* [abc] matches any character inside the brackets (in this case a, b, or c)
* a question mark (?) matches a single character
* brackets enclosing characters separated by a hyphen ([0-9]) matches any character between them (in this case 0 through 9)
* You can also use two asterisks to match nested directories
* a/\*\*/z would match a/z, a/b/z, a/b/c/z, and so on

### **Committing Your Changes**

Now that the staging area is set up, run git commit to commit the changes.

$ git commit

$ git commit -m “Commit Message”

$ git commit -a -m “Commit Message”

$ git commit -v

### **Skipping the Staging Area**

### **Removing Files**

### **Moving Files**

## Working with local repositories

### **git init**

This command turns a directory into an empty Git repository. This is the first step in creating a repository. After running git init, adding and committing files/directories is possible.

**Usage**:

# make directory a git repository

$ git init

**Example**:

# change directory to codebase

$ cd /Users/computer-name/Documents/website

# make directory a git repository

$ git init

Initialized empty Git repository in /Users/computer-name/Documents/website/.git/

### **git add**

Adds files into the staging area for Git. Before a file is available to commit to a repository, the file needs to be added to the Git index (staging area). There are a few different ways to use git add, by adding entire directories, specific files, or all unstaged files.

**Usage:**

$ git add <file or directory name>

**Options:**

-A or –all or .

**Example:**

# To add all files not staged:

$ git add .

# To stage a specific file:

$ git add index.html

# To stage an entire directory:

$ git add css

### **git commit**

Record the changes made to the files to a local repository. For easy reference, each commit has a unique ID.

It’s best practice to include a message with each commit explaining the changes made in a commit. Adding a commit message helps to find a particular change or understanding the changes.

**Usage:**

# Adding a commit with message

$ git commit -m "Commit message in quotes"

**Options:**

-m

-a

**Example:**

$ git commit -m "My first commit message"

$ git commit -am “Commit Message”

Case Study:

1. Alex has pushed up his source code to branch X. Jane wants a copy of that, so clones that repo, makes her changes and pushes them up to branch X. All this time, Alex was also working on his source code locally and once done, he wants to push his changes to Branch X. When he tries to push, he gets a Reject Error saying “**Updates were rejected because the tip of your current branch is behind its remote counterpart. Merge the remote changes before pushing again**.” This happened because Jane’s push has made a commit in the remote branch which is different than the current local commit of Alex. To resolve this Alex has to do the following:

git pull => git push

**What git pull does internally? It does 2 things as below:**

1. **git fetch** : Fetch or sync our *local\_origin\_branch* with the *remote\_branch*.

However it does not update/sync our *local\_branch* yet.

1. **git merge** : Merges the *local\_origin\_branch* with *local\_branch*. Same thing as running *git merge origin/branchname,*

which performs a merge commit. After merge commit, the *local\_branch* contains both Alex’s and

Jane’s changes, whereas the *local\_origin\_branch* contains only Jane’s changes, not Alex’s. It remains that way until we do a *git push.* The push updates *local\_origin\_branch* to be at the same state as our *local\_branch.* Also pushes the changes to *remote\_branch*.

*\* When merging if it finds any conflict, then automatic merge fails and a* ***merge conflict*** *error occurs. Here we need to resolve the conflict manually.*

### **git status**

This command returns the current state of the repository.

git status will return the current working branch. If a file is in the staging area, but not committed, it shows with git status. Or, if there are no changes it’ll return nothing to commit, working directory clean.

**Usage:**

$ git status

**Options:**

-A or –all or .

**Example:**

# Message when files have not been staged (git add)

$ git status

On branch SecretTesting

Untracked files:

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

homepage/index.html

# Message when files have been not been committed (git commit)

$ git status

On branch SecretTesting

Your branch is up-to-date with 'origin/SecretTesting'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: homepage/index.html

# Message when all files have been staged and committed

$ git status

On branch SecretTesting

nothing to commit, working directory clean

### **git config**

With Git, there are many configurations and settings possible. git config is how to assign these settings. Two important settings are user user.name and user.email. These values set what email address and name commits will be from on a local computer. With git config, a --global flag is used to write the settings to all repositories on a computer. Without a --global flag settings will only apply to the current repository that you are currently in.

There are many other variables available to edit in git config. From editing color outputs to changing the behavior of git status. Learn about git config settings in the official [Git documentation](https://git-scm.com/docs/git-config).

**Usage**:

$ git config <setting> <command>

**Example**:

# Running git config globally

$ git config --global user.email "my@emailaddress.com"

$ git config --global user.name "Brian Kerr"

# Running git config on the current repository settings

$ git config user.email "my@emailaddress.com"

$ git config user.name "Brian Kerr"

### **git branch**

To determine what branch the local repository is on, add a new branch, or delete a branch.

**Usage**:

# Create a new branch

$ git branch <branch\_name>

# List all remote or local branches

$ git branch -a

# Delete a branch

$ git branch -d <branch\_name>

**Example**:

# Create a new branch

$ git branch new\_feature

# List branches

$ git branch -a

\* SecretTesting

new\_feature

remotes/origin/stable

remotes/origin/staging

remotes/origin/master -> origin/SecretTesting

# Delete a branch

$ git branch -d new\_feature

Deleted branch new\_feature (was 0254c3d).

### **git checkout**

To start working in a different branch, use git checkout to switch branches.

**Usage**:

# Checkout an existing branch

$ git checkout <branch\_name>

# Checkout and create a new branch with that name

$ git checkout -b <new\_branch>

**Example**:

# Switching to branch 'new\_feature'

$ git checkout new\_feature

Switched to branch 'new\_feature'

# Creating and switching to branch 'staging'

$ git checkout -b staging

Switched to a new branch 'staging'

### **git merge**

Integrate branches together. git merge combines the changes from one branch to another branch. For example, merge the changes made in a staging branch into the stable branch.

**Usage**:

# Merge changes into current branch

$ git merge <branch\_name>

**Example**:

# Merge changes into current branch

$ git merge new\_feature

Updating 0254c3d..4c0f37c

Fast-forward

homepage/index.html | 297 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++

1 file changed, 297 insertions(+)

create mode 100644 homepage/index.html

## Working with remote repositories

### **git remote**

To connect a local repository with a remote repository. A remote repository can have a name set to avoid having to remember the URL of the repository.

**Usage**:

# Add remote repository

$ git remote <command> <remote\_name> <remote\_URL>

# List named remote repositories

$ git remote -v

**Example**:

# Adding a remote repository with the name of beanstalk

$ git remote add origin git@account\_name.git.beanstalkapp.com:/acccount\_name/repository\_name.git

# List named remote repositories

$ git remote -v

origin git@account\_name.git.beanstalkapp.com:/acccount\_name/repository\_name.git (fetch)

origin git@account\_name.git.beanstalkapp.com:/acccount\_name/repository\_name.git (push)

Note: A remote repository can have any name. It’s common practice to name the remote repository ‘origin’.

### **git clone**

To create a local working copy of an existing remote repository, use git clone to copy and download the repository to a computer. Cloning is the equivalent of git init when working with a remote repository. Git will create a directory locally with all files and repository history.

**Usage**:

$ git clone <remote\_URL>

**Example**:

$ git clone git@account\_name.git.beanstalkapp.com:/acccount\_name/repository\_name.git

Cloning into 'repository\_name'...

remote: Counting objects: 5, done.

remote: Compressing objects: 100% (3/3), done.

remote: Total 5 (delta 0), reused 0 (delta 0)

Receiving objects: 100% (5/5), 3.08 KiB | 0 bytes/s, done.

Checking connectivity... done.

### **git pull**

To get the latest version of a repository run git pull. This pulls the changes from the remote repository to the local computer.

**Usage**:

$ git pull <branch\_name> <remote\_URL/remote\_name>

**Example**:

# Pull from named remote

$ git pull origin staging

From account\_name.git.beanstalkapp.com:/account\_name/repository\_name

\* branch staging -> FETCH\_HEAD

\* [new branch] staging -> origin/staging

Already up-to-date.

# Pull from URL (not frequently used)

$ git pull git@account\_name.git.beanstalkapp.com:/acccount\_name/repository\_name.git staging

From account\_name.git.beanstalkapp.com:/account\_name/repository\_name

\* branch staging -> FETCH\_HEAD

\* [new branch] staging -> origin/staging

Already up-to-date.

**What git pull does internally? It does 2 things as below:**

1. **git fetch** : Fetch or sync our *local origin\_repo* with the *remote\_repo* one. However it does not update/sync our *local\_repo* yet.
2. **git merge** : Merges the *local origin\_repo* with *local\_repo*. Same thing as running *git merge origin/branchname.*

### **git push**

Sends local commits to the remote repository. git push requires two parameters: the remote repository and the branch that the push is for.

**Usage**:

$ git push <remote\_URL/remote\_name> <branch>

# Push all local branches to remote repository

$ git push —all

**Example**:

# Push a specific branch to a remote with named remote

$ git push origin staging

Counting objects: 5, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (3/3), done.

Writing objects: 100% (5/5), 734 bytes | 0 bytes/s, done.

Total 5 (delta 2), reused 0 (delta 0)

To git@account\_name.git.beanstalkapp.com:/acccount\_name/repository\_name.git

ad189cb..0254c3d SecretTesting -> SecretTesting

# Push all local branches to remote repository

$ git push --all

Counting objects: 4, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (4/4), done.

Writing objects: 100% (4/4), 373 bytes | 0 bytes/s, done.

Total 4 (delta 2), reused 0 (delta 0)

remote: Resolving deltas: 100% (2/2), completed with 2 local objects.

To git@account\_name.git.beanstalkapp.com:/acccount\_name/repository\_name.git

0d56917..948ac97 master -> master

ad189cb..0254c3d SecretTesting -> SecretTesting

## Advanced Git Commands

### **git stash**

To save changes made when they’re not in a state to commit them to a repository. This will store the work and give a clean working directory. For instance, when working on a new feature that’s not complete, but an urgent bug needs attention.

**Usage**:

# Store current work with untracked files

$ git stash -u

# Bring stashed work back to the working directory

$ git stash pop

**Example**:

# Store current work

$ git stash -u

Saved working directory and index state WIP on SecretTesting: 4c0f37c Adding new file to branch

HEAD is now at 4c0f37c Adding new file to branch

# Bring stashed work back to the working directory

$ git stash pop

On branch SecretTesting

Your branch and 'origin/SecretTesting' have diverged,

and have 1 and 1 different commit each, respectively.

(use "git pull" to merge the remote branch into yours)

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: index.html

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

Dropped refs/stash@{0} (3561897724c1f448ae001edf3ef57415778755ec)

### **git log**

To show the chronological commit history for a repository. This helps give context and history for a repository. git log is available immediately on a recently cloned repository to see history.

**Usage**:

# Show entire git log

$ git log

# Show git log with date pameters

$ git log --<after/before/since/until>=<date>

# Show git log based on commit author

$ git log --<author>="Author Name"

**Example**:

# Show entire git log

$ git log

commit 4c0f37c711623d20fc60b9cbcf393d515945952f

Author: Brian Kerr <my@emailaddress.com>

Date: Tue Oct 25 17:46:11 2016 -0500

Updating the wording of the homepage footer

commit 0254c3da3add4ebe9d7e1f2e76f015a209e1ef67

Author: Ashley Harpp <my@emailaddress.com>

Date: Wed Oct 19 16:27:27 2016 -0500

My first commit message

# Show git log with date pameters

$ git log --before="Oct 20"

commit 0254c3da3add4ebe9d7e1f2e76f015a209e1ef67

Author: Ashley Harpp <my@emailaddress.com>

Date: Wed Oct 19 16:27:27 2016 -0500

My first commit message

# Show git log based on commit author

$ git log --author="Brian Kerr"

commit 4c0f37c711623d20fc60b9cbcf393d515945952f

Author: Brian Kerr <my@emailaddress.com>

Date: Tue Oct 25 17:46:11 2016 -0500

Updating the wording of the homepage footer

### **git rm**

Remove files or directories from the working index (staging area). With git rm, there are two options to keep in mind: force and cached. Running the command with force deletes the file. The cached command removes the file from the working index. When removing an entire directory, a recursive command is necessary.

**Usage**:

# To remove a file from the working index (cached):

$ git rm --cached <file name>

# To delete a file (force):

$ git rm -f <file name>

# To remove an entire directory from the working index (cached):

$ git rm -r --cached <directory name>

# To delete an entire directory (force):

$ git rm -r -f <file name>

**Example**:

# To remove a file from the working index:

$ git rm --cached css/style.css

rm 'css/style.css'

# To delete a file (force):

$ git rm -f css/style.css

rm 'css/style.css'

# To remove an entire directory from the working index (cached):

$ git rm -r --cached css/

rm 'css/style.css'

rm 'css/style.min.css'

# To delete an entire directory (force):

$ git rm -r -f css/

rm 'css/style.css'

rm 'css/style.min.css'