**https://stackoverflow.com/questions/2144406/how-to-make-shallow-git-submodulesALL ABOUT GIT**

GIT is distributed version control system, rather than having one single place for full version history like CSV and SVN.

Added this additional content

**GIT INSTALLATION**

1. Just go to <http://git-scm.com/download/win> and the download will start automatically

2.To get an automated installation you can use the [Git Chocolatey package](https://chocolatey.org/packages/git). Note that the Chocolatey package is community maintained.

3. [GitHub Desktop website](https://desktop.github.com/).

Another easy way to get Git installed is by installing GitHub Desktop. The installer includes a command line version of Git as well as the GUI. It also works well with PowerShell, and sets up solid credential caching and sane CRLF settings. You can download this from the [GitHub Desktop website](https://desktop.github.com/).

**CUSTOMIZE GIT ENVIRONMENT:**

1.Search in the File Explorer for \*.gitconfig,it should generally be under c:\USER\$USER

On Windows systems, Git looks for the .gitconfig file in the $HOME directory (C:\Users\$USER for most people). It also still looks for /etc/gitconfig, although it’s relative to the MSys root, which is wherever you decide to install Git on your Windows system when you run the installer.

2.Goto cmd prompt and cd to .gitconfig file location

$ git config --global user.name "John Doe"

$ git config --global user.email johndoe@example.com

$ git config --global core.editor "'C:/Program Files/Notepad++/notepad++.exe' -multiInst -nosession"

To check your config settings:

$ git config --list

USING GIT:

1.Checking the status of a branch:To find out if there are any changes in the branch or to check any untracked files use:

git status

Untracked basically means that Git sees a file you didn’t have in the previous snapshot (commit); Git won’t start including it in your commit snapshots until you explicitly tell it to do so. It does this so you don’t accidentally begin including generated binary files or other files that you did not mean to include.

### 2. Tracking New Files

In order to begin tracking a new file, you use the command git add. To begin tracking the README file, you can run this:

$ git add README

If you run your status command again, you can see that your README file is now tracked and staged to be committed:

$ git status

On branch master

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

Changes to be committed:

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

new file: README

You can tell that it’s staged because it’s under the “Changes to be committed” heading.

The git add command takes a path name for either a file or a directory; if it’s a directory, the command adds all the files in that directory recursively.

### 3. Staging Modified Files

Let’s change a file that was already tracked. If you change a previously tracked file called CONTRIBUTING.md and then run your git status command again, you get something that looks like this:

$ git status

On branch master

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

Changes to be committed:

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

new file: README

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: CONTRIBUTING.md

The CONTRIBUTING.md file appears 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. git add is a multipurpose command — you use it to begin tracking new files, to stage files, and to do other things like marking merge-conflicted files as resolved. It may be helpful to think of it more as “add precisely this content to the next commit”

$ git add CONTRIBUTING.md

$ git status

On branch master

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

Changes to be committed:

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

new file: README

modified: CONTRIBUTING.md

Both files are staged and will go into your next commit.

 At this point, suppose you remember one little change that you want to make in CONTRIBUTING.md before you commit it. You open it again and make that change, and you’re ready to commit. However, let’s run git status one more time:

$ vim CONTRIBUTING.md

$ git status

On branch master

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

Changes to be committed:

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

new file: README

modified: CONTRIBUTING.md

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: CONTRIBUTING.md

What the heck? Now CONTRIBUTING.md 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 CONTRIBUTING.md 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:

$ git add CONTRIBUTING.md

$ git status

On branch master

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

Changes to be committed:

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

new file: README

modified: CONTRIBUTING.md

### Viewing Your Staged and Unstaged Changes

If the git status command is too vague for you — you want to know exactly what you changed, not just which files were changed — you can use the git diff command.

but you’ll probably use it most often to answer these two questions: What have you changed but not yet staged? And what have you staged that you are about to commit? git diff shows you the exact lines added and removed — the patch, as it were.

Let’s say you edit and stage the README file again and then edit the CONTRIBUTING.md file without staging it. If you run your git status command, you once again see something like this:

$ git status

On branch master

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

Changes to be committed:

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

modified: README

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: CONTRIBUTING.md

To see what you’ve changed but not yet staged, type git diff with no other arguments:

$ git diff

diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md

index 8ebb991..643e24f 100644

--- a/CONTRIBUTING.md

+++ b/CONTRIBUTING.md

@@ -65,7 +65,8 @@ branch directly, things can get messy.

Please include a nice description of your changes when you submit your PR;

if we have to read the whole diff to figure out why you're contributing

in the first place, you're less likely to get feedback and have your change

-merged in.

+merged in. Also, split your changes into comprehensive chunks if your patch is

+longer than a dozen lines.

If you are starting to work on a particular area, feel free to submit a PR

that highlights your work in progress (and note in the PR title that it's

That command compares what is in your working directory with what is in your staging area. The result tells you the changes you’ve made that you haven’t yet staged.

If you want to see what you’ve staged that will go into your next commit, you can use git diff --staged. This command compares your staged changes to your last commit:

$ git diff --staged

diff --git a/README b/README

new file mode 100644

index 0000000..03902a1

--- /dev/null

+++ b/README

@@ -0,0 +1 @@

+My Project

It’s important to note that git diff by itself doesn’t show all changes made since your last commit — only changes that are still unstaged. If you’ve staged all of your changes, git diff will give you no output.