Git Hub Tutorial
by Alex Dobson

Getting Started - Setting up Git

First thing is first, you have to get Git! You are going to want to do go to http://git-scm.com/downloads and download the latest version for your operating system. (Don't be alarmed if it has "snow-leopard" in the Mac installer name, it still works perfectly fine!)

If you are ever lost while using Git, type:

```
$ git --help
```

This will tell you any information you could possible need to know about the usage. You can also get help for a specific command. For example. if you type:

```
$ git add --help
```

you will get so much documentation on the add command it will make your head spin! So morale of the story, use "--help" to your advantage!

Now that you have Git installed, you have to set up your Global Variables so that Git can identify you. First, lets set up your name. Your name is added by Git to any commits you make so others can see you made changes. Setup your name by typing:

```
$ git config --global user.email "Your Name Here"
```

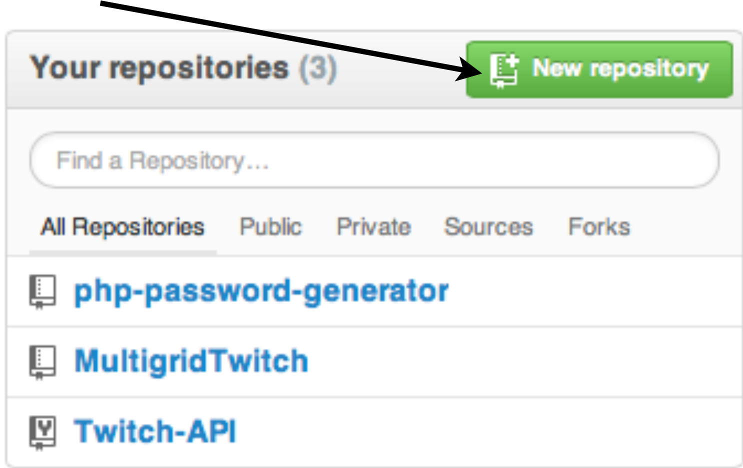
Next you will want to setup your Git email. Again, just like the Git *user.name* variable, the GIT user.email variable is used when you make commits. The command to set your email is:

```
$ git config --global user.email "Your Email Here"
```

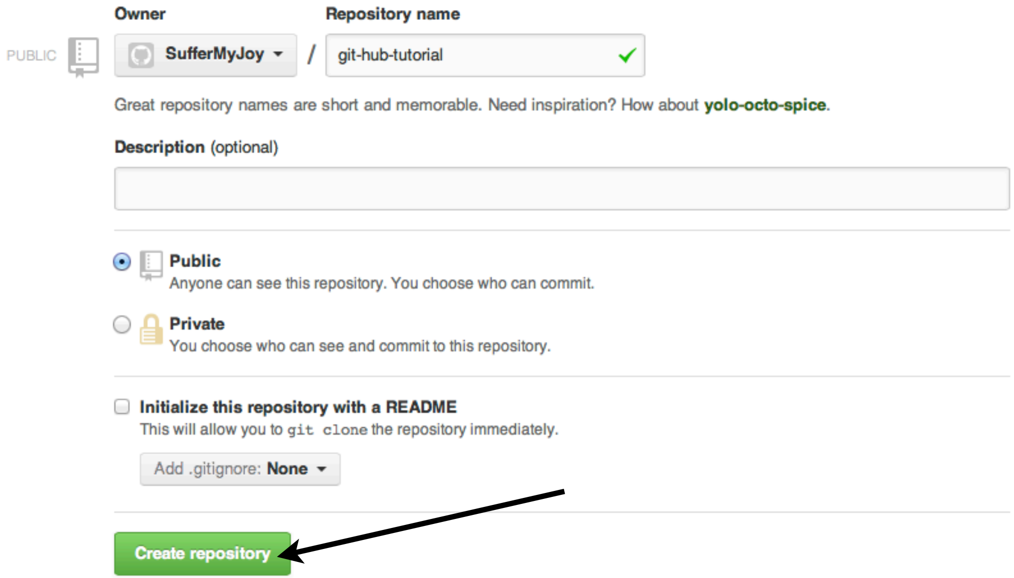
Whoooo! Git is all setup! You can do some password caching at this point so that you don't have to type in your password as much, but I haven't ever bothered. The only time Git asks you for your password is the first time you want to grab a repository. From then on while you are working on that repository, it remembers your password.

Creating a Repository

Time to make a repository! OH YEAH! First, make sure you have made a Git Hub account. Sign in, then click on “New Repository” which is located in the bottom right hand corner of your landing page:



Fill out the information and click “Create Repository” when you are done. I never create the “README” with the repository. If you create it now, you may run into some errors later on down the road because the files on your local repository don’t match those on your Git Hub repository.



Well that was relatively easy wasn't it?! We have our first repository created and working. There is only one problem, the repository doesn't exist on our local machine! Our next step is to get that online repository onto our local machine.

First, I recommend creating a directory "~/Development" or something like that. Then create a folder in "~/Development" for our newly created repository. You will probably want to do this with each different project you do.

For example, the repository I just made was called "git-hub-tutorial" so I made a folder "~/Development/git-hub-tutorial", which will contain all my files for this project.

Once you have created your folder, you need to tell Git that this folder is actually a repository. You can do this by calling:

```
$ git init
```

which after being called will display:

```
# Initialized empty Git repository in /Users/alex/Development/git-hub-tutorial/.git/
```

Congrats, your local folder is now a Git repository! By calling the commit command, Git installs the necessary files in "/.git" for the repository to work. Note that you should never have to edit these files directly... EVER!

The next step is to add your project files. You can add whatever files you want to the repository using BBEdit, TextMate, or whatever text editor you like. Use VIM for all I care! When I was working through this tutorial, I made a file called "README" using BBEdit and saved it to "~/Development/git-hub-tutorial/README", but of course you are free to do whatever you want!

Now that our file has been created, it is important to talk about how Git sees files. Git always sees files as in a particular stage, which tells Git how much change has occurred to the file. In Git, there are **3** stages that a file can be in:

- **Untracked**: these are neither staged to be committed or actually committed
- **Staged**: these have been added to the staging area, but have not yet been committed to the repository, this extra layer is here to give the user more flexibility
- **Committed**: these have been merged with the repository

So we've created our README file and put it into our repository folder, but what is its status? You can check by running:

```
$ git status
```

We can always use this command when we want to see the status of our files. When I ran this command, it returned:

```
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#        README
nothing added to commit but untracked files present (use "git add" to track)
```

As you can see, the README file is untracked. Let's say we're ready for it to be added to our repository's staging area. We can do this by running the command:

```
$ git add README
```

This will add README to the staging area, where it will wait to be committed to the repository. Now if we run:

```
$ git status
```

it will return

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#        new file:   README
#
```

Yay! We did it! We added the file to the staging area! However, it is still not a part of our local repository yet, because the file has not been committed. Lets change that right now! In order to add the file to the local repository we use the command:

```
$ git commit -m "Your Comment Here"
```

This will add the commit the files on the staging area. The comment is of course so you can look back and see what has happened over the lifetime of the projects development. You can see different changes that were made and why! If we run

```
$ git status
```

then we can see that the file has been committed. The output this time is:

```
# On branch master
nothing to commit, working directory clean
```

That is good news, that means that our repository consists of everything in our working directory. Whoop! Now lets take a look at how to add it to our remote Git Hub Repository.

First, to add a remote repository we have to run the command:

```
$ git remote add origin https://github.com/username/repo-name.git
```

or in my specific case for this tutorial:

```
$ git remote add origin https://github.com/SufferMyJoy/git-hub-tutorial.git
```

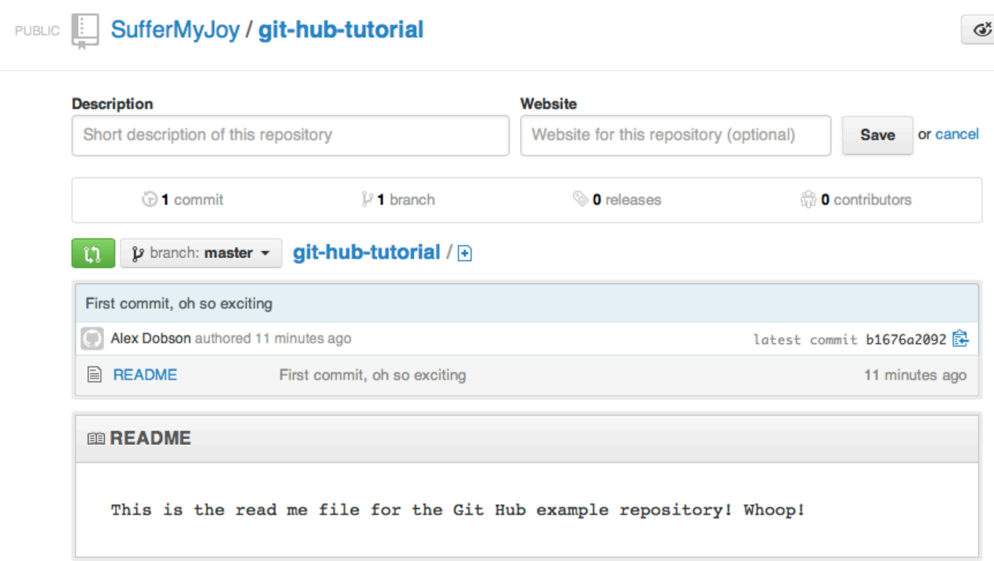
Once we run this command, we have a working copy of the online repository that we can work made named “origin”. To commit our local repository to the online repository, we simply run the command:

```
$ git push origin master
```

which will push our local commits to the repository located on the Git Hub server. Awesome! When you perform this action, if you skipped password caching, it will ask for your Git Hub username and password. Fill those out obviously! So once we run this command and type in our information. Once the command runs it will look like this:

```
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 283 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/SufferMyJoy/git-hub-tutorial.git
 * [new branch]      master -> master
```

Now if we go to our repository online, we will see that the file has indeed been added online and is available for team members (or whoever else):



And thats it, you are done making your first repository! As long as you can remember the steps that got you to this point, you will be rocking out repositories in no time!

General Workflow

While working on a project with Git, you almost always end up falling into a particular workflow. This workflow comes after you get everything setup and start actual work on the project. Both your Git Hub and your local repositories should be setup and linked together. Once that is done, the workflow you will most likely follow is:

1. Edit/Add the files you want to change. This stage can take a while, edit until everything is working or you reach some sort of milestone.

2. Once everything is working and you feel you're ready to make a commit, add it to the staging area. Again, this task is done by executing the command:

```
$ git add -all
```

which will add all of the changed files to the staging area of your local repository.

3. Once everything has been added, perform a commit. Be sure to include a detailed description of what you changed because that is what will show up on Git Hub as well. You perform a commit to your local repository by performing the command:

```
$ git commit -m “Your Comment Here”
```

4. Now that you have committed your code to your local repository, all that is left to do is to push it to your Git Hub account. This is one by executing:

```
$ git push origin master
```

and remember that once the origin has been created once, you do not have to create it again. You will also only have to enter your password on the first push, it will be saved so you won't need password for subsequent pushes. (Hence why I do not cache my password)

5. Thats it! Thats pretty much the workflow for working with Git and Git Hub, so get working on some sweet projects! Can't wait to see what you can do :)

This is my general workflow when I work on a project. However, I know there is better uses for the staging area, but I don't use that. I pretty much just throw stuff to the staging area and then immediately commit when I'm ready. The best way to know when its time to do a commit, is to think of a version change. Is the project really ready to go from 1.2 to 1.3? If the answer is yes, then its time for a commit.

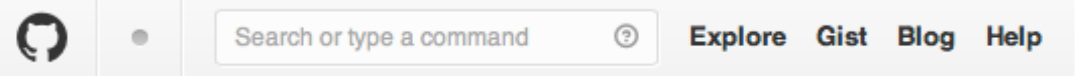
Forking a Repository

So now you think you're all fancy, huh? You got your repository up and running, and your working away on it, making changes to your files, adding them to the staging area, committing them, and finally pushing them to your remote repository.

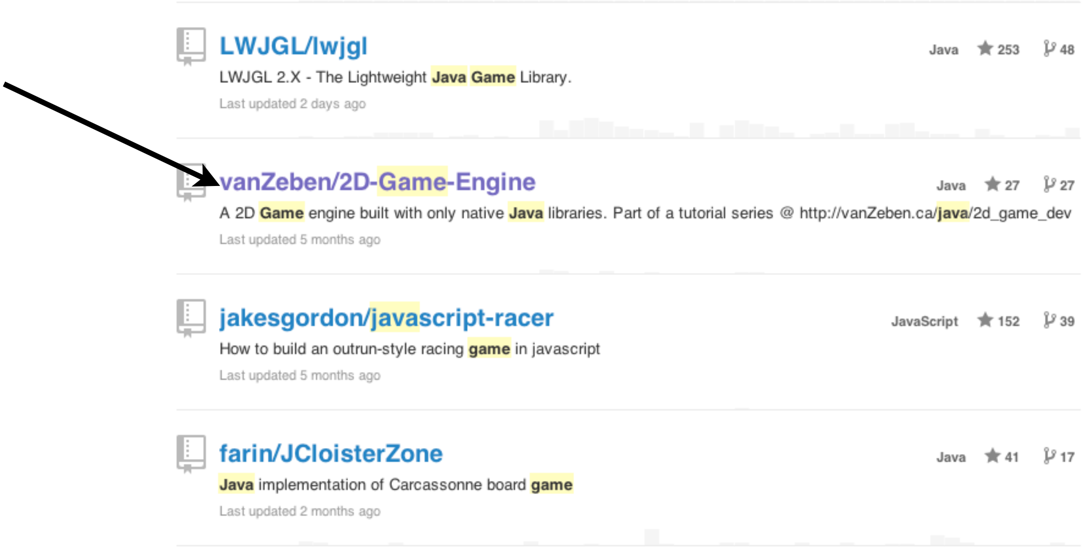
WHAT NOW?!

Now comes collaboration. You will undoubtedly find yourself in need of someone else to help you with your code. Maybe you find a project that you want to help work on. But how are two of you going to make changes and not have gross overwrite errors? The answer, my very good friend, is to FORK!

The first step of the forking process takes place on the Git Hub site. Navigate your way there using IE and sign in. Find a project you want to Fork on Git Hub. You can do this via the search bar up at the top of the Git Hub landing page which looks like:



From there you can search for particular users, projects, or whatever you want to type in! Once you find something that looks worth working on, you can click on the repository name. When I was looking around for this tutorial, I searched for Van Zeben's 2D Java Game engine, made famous by his YouTube tutorials which are available here: http://www.youtube.com/user/DesignsbyZephyr if you are interested!



When I clicked on the repository from Git Hub, it brought me right to the repository page. On this page in the top right hand corner, you can find a series of options available to you, one of which is “Fork”. You want to click this!

After forking the project it presents you with an awesome looking loading screen that notifies you that there is a copy of the repository being made especially for you! How nice! Once that is complete, it will redirect you to the copied repository, now on your own Git Hub account. You now have a working copy of the repository in your own Git Hub, so now lets get that onto your local machine so you can start working!

When we copy a repository from Git Hub, it will create a folder for the project by itself. So we simply need to change to our ~/Development directory.

```
$ cd ~/Development/
```

Now you have to clone the repository to your local machine. You do this, surprisingly enough, with a command called clone:

```
$ git clone https://github.com/username/repo-name.git
```

which will output:

```
Cloning into '2D-Game-Engine'...
remote: Counting objects: 133, done.
remote: Compressing objects: 100% (89/89), done.
remote: Total 133 (delta 49), reused 104 (delta 20)
Receiving objects: 100% (133/133), 28.49 KiB | 0 bytes/s, done.
Resolving deltas: 100% (49/49), done.
Checking connectivity... done
```

It is important to note that rather than create our own local repository and then push to our Git Hub repository, we could always take this clone approach to our projects. When a repository is cloned, it automatically creates a default remote called origin, just like the one we made ourselves in the previous part of the tutorial. Again, when cloning, this will create a folder in ~/Development/repo-name, so you DO NOT have to make the folder. Once it is cloned simply cd into the new directory and you will have a working repository.

Another important note is that origin is NOT linked to the original repository, but rather to our own fork. In order to keep track of the original repository, we have to create another remote, which I usually call upstream by:

```
$ git remote add upstream https://github.com/original-author/repo-name.git
```

** Please remember to put in the original author name, for my example I wrote

```
$ git remote add upstream https://github.com/vanZeben/2D-Game-Engine.git
```

Awesome! You are now ready to start work on your fork of the repository! When you make changes to your repository now, you can simply commit them using:

```
$ git commit -m "Your Comment Here"
```

and then you can push them online using:

```
$ git push origin master
```

and your changes will be made to your copy of the repository on your Git Hub. If you want to get the newest changes made by the original author, all you have to do is fetch their repository:

```
$ git fetch upstream
```

and then merge their repository with your own using:

```
$ git merge upstream/master
```

These two commands will effectively put your fork up to date with the original authors while keeping the changes you have made.

Note that you can use the pull command which is the combination of fetch and merge. However, fetch and merge tends to have better results in terms of stability. After fetching the original authors repository, you can review the code to make sure nothing will conflict with what you have been working on. This way when you decide to merge them, you know nothing will go wrong. With the pull command, the merge will happen automatically and problems can arise.

Creating a Branch

Branches are one of the most useful features of Git, and allow multiple developers to work on the same project all at once without any fear of compatibility issues. Getting a branch up and running only requires two simple commands, first:

```
$ git branch your-branch-name
```

which will create a branch called “your-branch-name”. After that, you simply have to make “your-branch-name” the active branch by running:

```
$ git checkout your-branch-name
```

Now I know what your thinking. Two commands is one too many commands, and Git agrees with you. Alternatively to the two above commands, you can run:

```
$ git checkout -b your-branch-name
```

which will create a new branch called “your-branch-name” and then make it the active branch. Pretty nifty shortcut!

Now lets say that you are working on your new branch called “your-branch-name”, but you need to make a couple of quick changes to the master so that it can be push out (say like on a Website project where some fast CSS changes need to be made). How would you go about doing this?

Switching between branches in Git is actually super easy. All you need to do is the checkout command. If you type:

```
$ git checkout master
```

then the master branch will become your active branch. You can then make your quick CSS changes, add and commit them, and then switch back to the “your-branch-name” branch using:

```
$ git checkout your-branch-name
```

Once you're done working on your branch and you are ready to combine it back up with the master, you need to make the master your active branch:

```
$ git checkout master
```

then merge your branch with the master:

```
$ git merge your-branch-name
```

Note that since your active branch is the master, you only need one argument for the merge command. Now you can delete your branch with:

```
$ git branch -d your-branch-name
```

Note that when you switch between branches, the files that you work on, otherwise known as the "working copy", are updated to reflect the changes in the new branch. If you have changes you have not yet committed, Git makes sure you do not lose them. Git is also very meticulous during merges and pulls to ensure you don't lose any changes! **HOWEVER**, when in doubt about losing something, commit it! That way you can't lose it!

Pull Requests

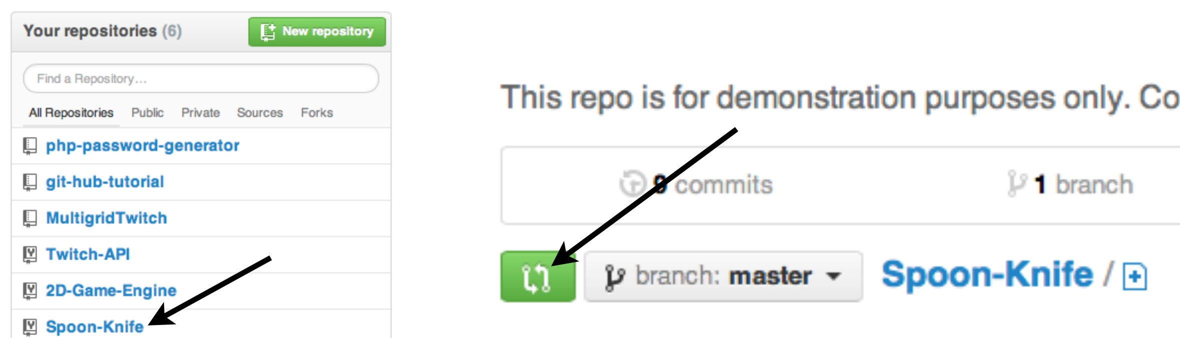
In Git Hub, there are two basic ways to work collaboratively with other people. First, there is the “Fork and Pull Model”, where a user forks a existing repository and pushes changes to their personal copy of the repository. The changes can then be pulled into the main repository by the original author.

The second model is the “Shared Repository Model” which is more prevalent with small production teams working on private projects. Everyone is simply granted push access to a single shared repository and they use branching to make isolated changes.

Pull requests are predominately used in the first type of model, “Fork and Pull”. They provide a way for other works to notify the project lead or maintainer of changes they made, and then the project lead or maintainer can make the final decision on if to include those changes or not. Lets go over how to perform a pull request.

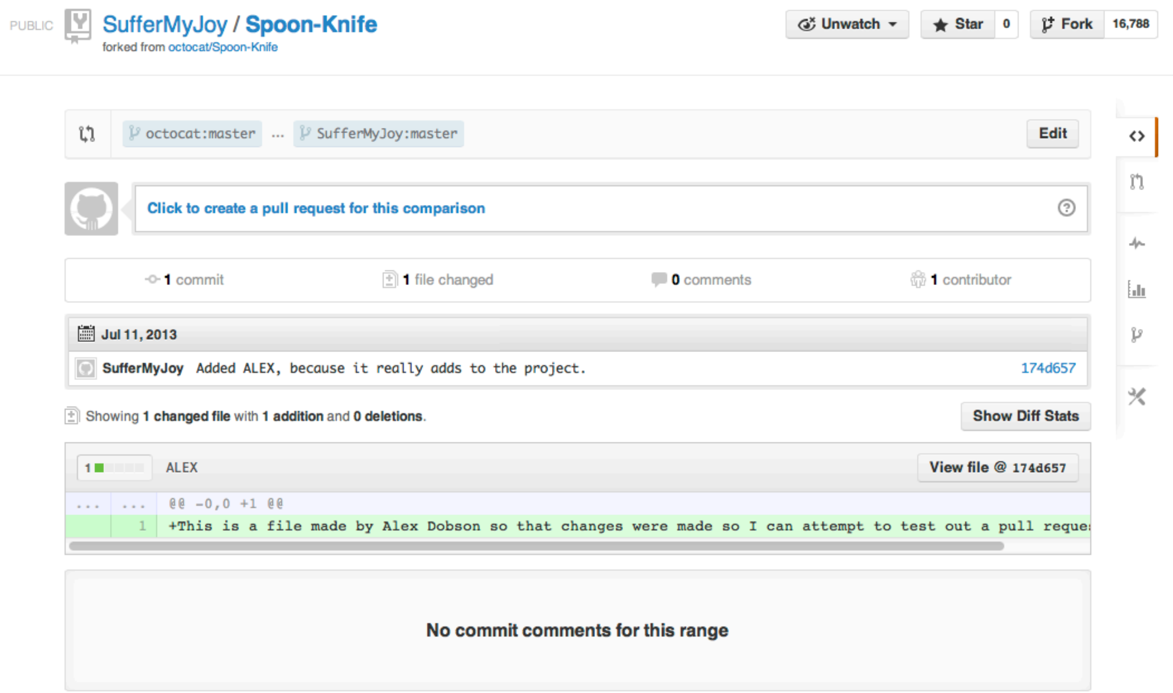
First off, you will want to create a fork of an existing project, make changes, and push those changes to your Git Hub repository. In the end, you should have a modified version of the repository on your Git Hub. Now lets get starting making a pull request.

First, login to your Git Hub account and navigate to the modified repository. For this example, I forked the Fork-Spoon repository from Octocat (the common test case for pull requests).



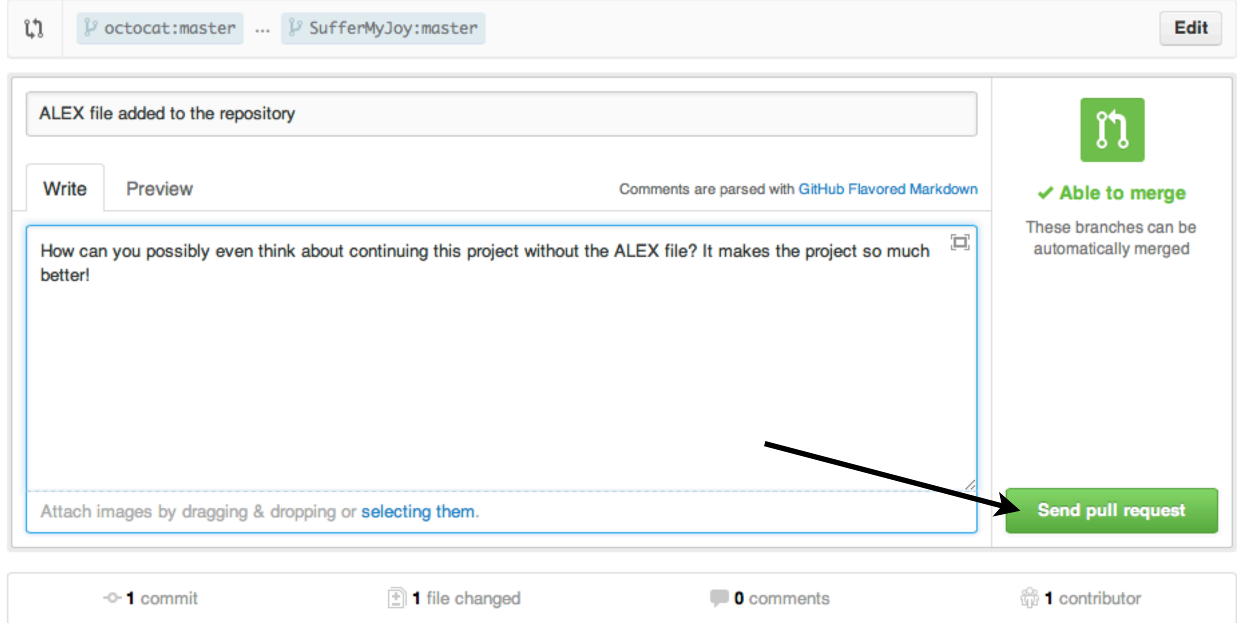
Once on the repository page, there is a small green button next to the branch selector. This button is the “Compare and Review” button. Click it! When you do, you will be taken to a page that shows the differences between the two projects.

The page looks like this:



As you can see, the file that I added to the repository is highlighted in green. From this page, you can click on the text area that reads “Click to create a pull request for this comparison” to create a new pull request.

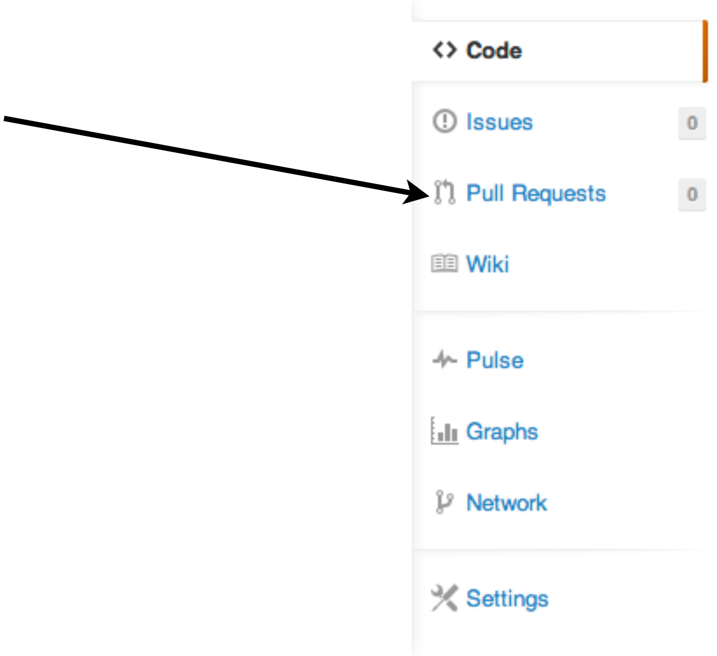
When you click on that text area, it brings you to a new page. This is where you fill out the information regarding your pull request. You give the pull request a title, and a comment that describes to the author what the changes are and why they should be used. Once this is filled out you can send the pull request with the “Send Pull Request” button located in the bottom right corner.



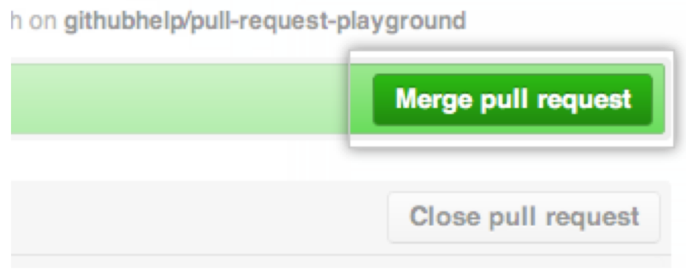
After sending the pull request, you will find yourself on the “Discussion” page for the pull quest. From here both you and the original author can post comments and discuss the changes with each other. This page is great for collaboration and really encourages discussion regarding the changes made!

And thats it for pull requests. That is the entire process to go through to make changes to someone else’s work and then request that they use those changes. But what if you’re on the other side of the coin? What if someone wants to change one of your requests? We will cover that next!

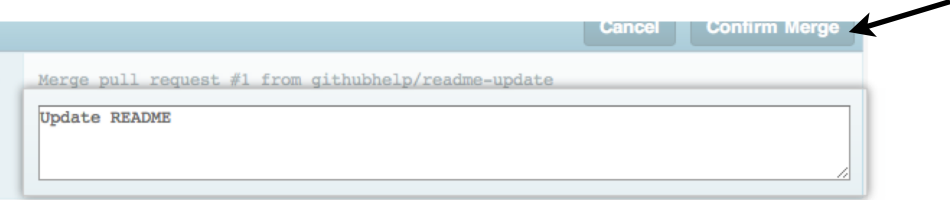
As per usual, sign into your Git Hub account to start. From there select a repository. From the repository page, you will see a variety of options on the right hand side of the page that pertain to that particular repository. One of those options is pull requests. Click it!



When you click on the pull requests link, you will be brought to a page that contains all pull requests to your projects. On the right hand side of the pull request, there will be a green button that says “Merge Pull Request”.



Clicking on the “Merge Pull Request” button will bring you to a new screen where you can enter a commit message regarding the merge.



From there you will only have to click on the “Confirm Merge” button to make the merge happen! As you can tell, Git Hub made merging pull requests from other users very easy since their goal was to be a social platform for coding. Awesome stuff!

Of course all of this can be done via the command line as well. Navigate your way to the local repository and checkout the master branch (unless of course you want to merge their changes with a different branch, in which case use that branch):

```
$ git checkout master
```

Now that you have the master as your active (working) branch, pull the desired repository from the other user’s Git Hub:

```
$ git pull https://github.com/other-user/their-repo.git branch-name
```

Now that you have their repository, you can commit the merge:

```
$ git merge branch-name
```

Now that the two repositories have been merged, all that is left to do is to push the master back to your Git Hub so it appears online (again, assuming you are working on the master branch, if you are not, change the name accordingly):

```
$ git push origin master
```