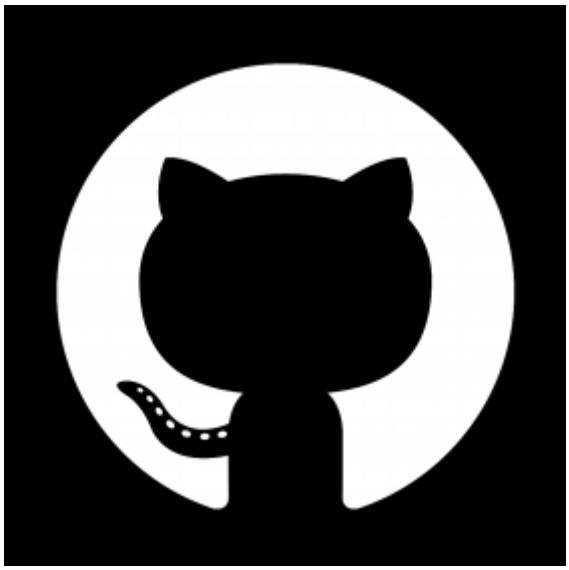
Jonathan Mines

149 Followers About Follow

The Ultimate Github Collaboration Guide

Jonathan Mines May 30, 2018 · 7 min read

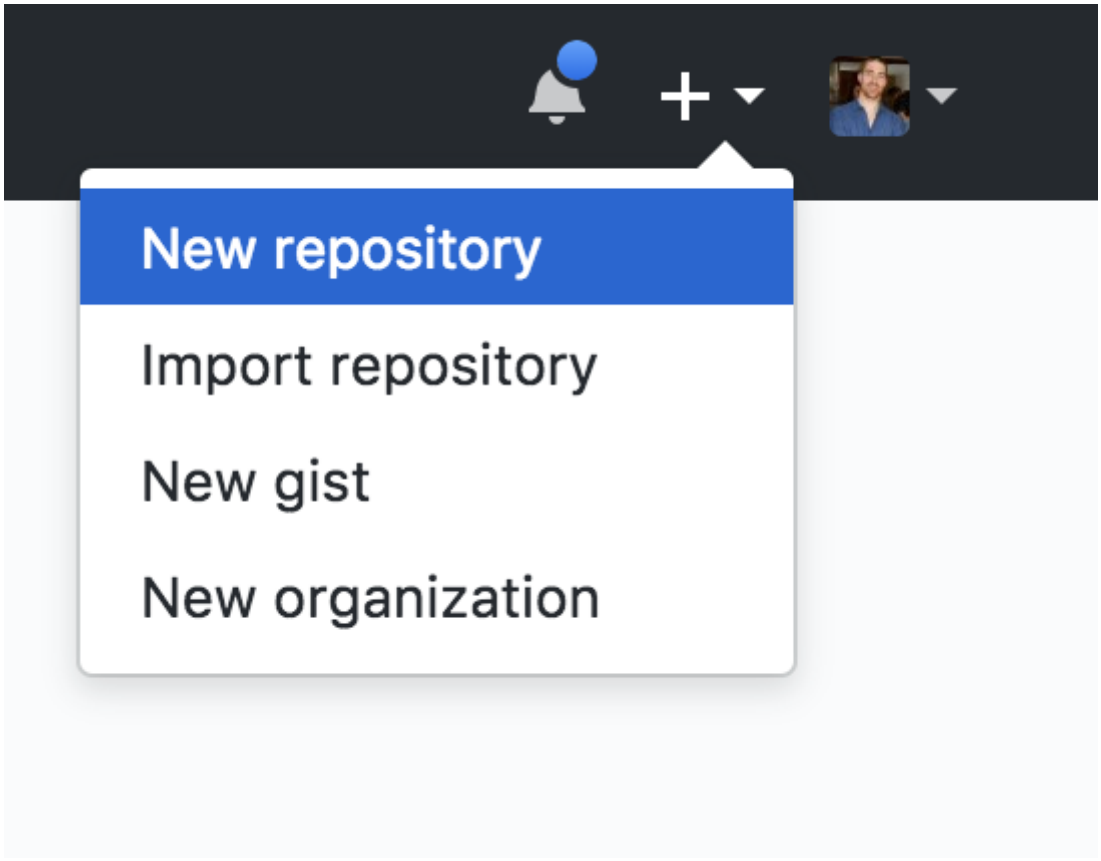


This is just one of many ways to collaborate on a project using GitHub. But it's one I would suggest if you're just starting out working with a team and haven't established a git flow yet or know where to start in establishing one.

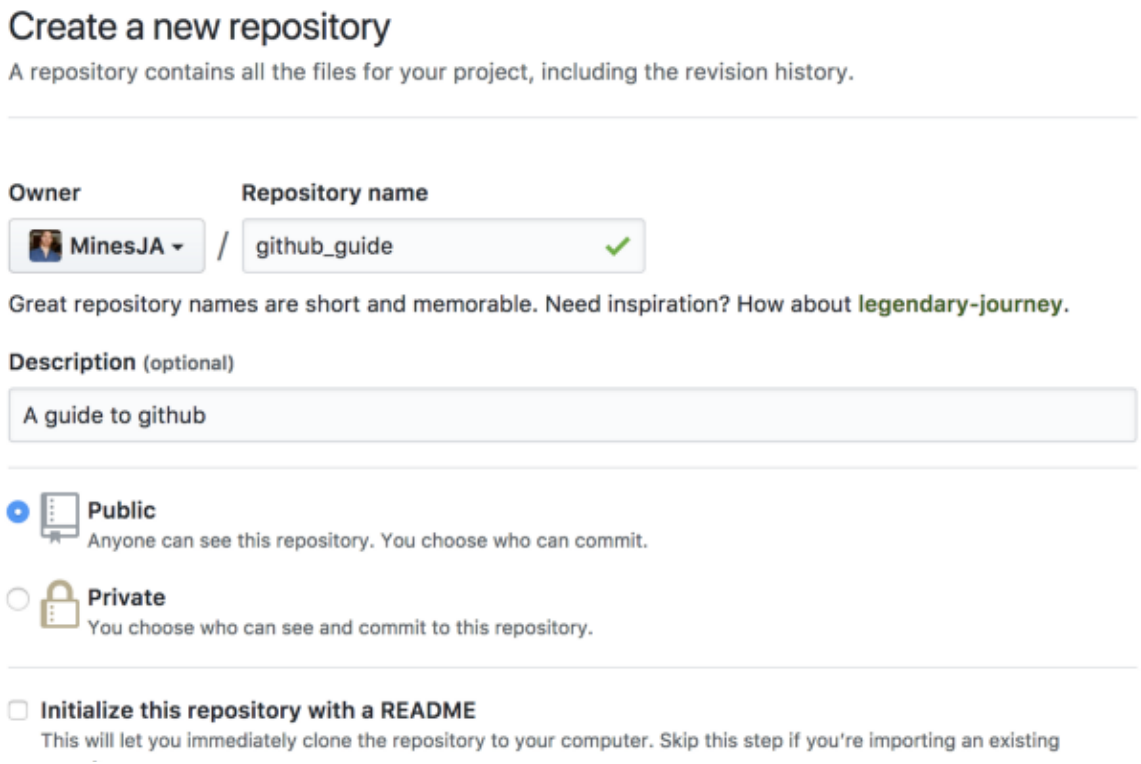
Step 1: Initialize a New Project

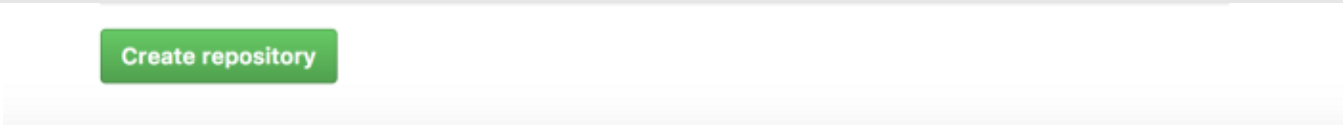
Create a new project/directory from the command line

```
$ rails new github_guide
```

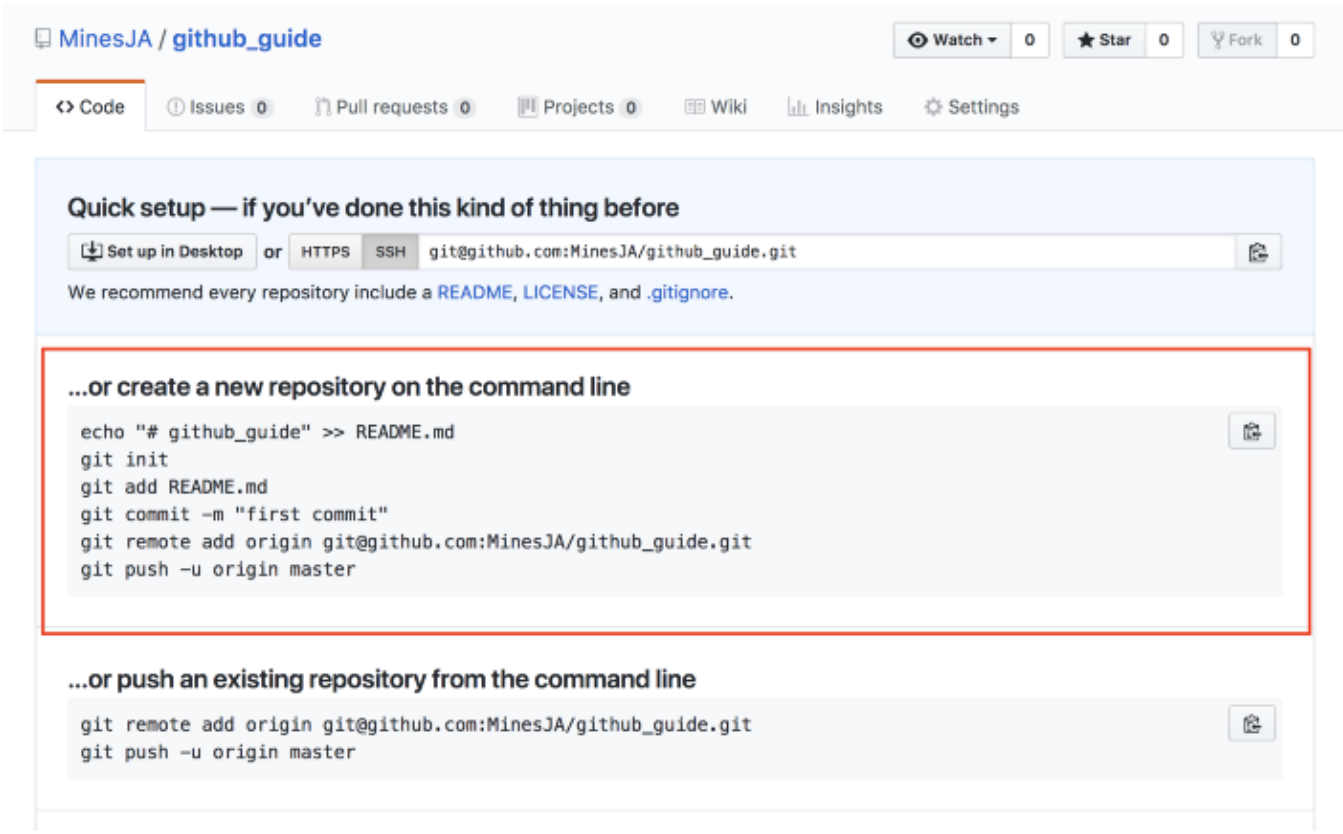


Then fill out the Repository name and the Description fields. Keep it public, and do not “Initialize this repository with a README”. Don’t change anything else. Click “Create repository”.





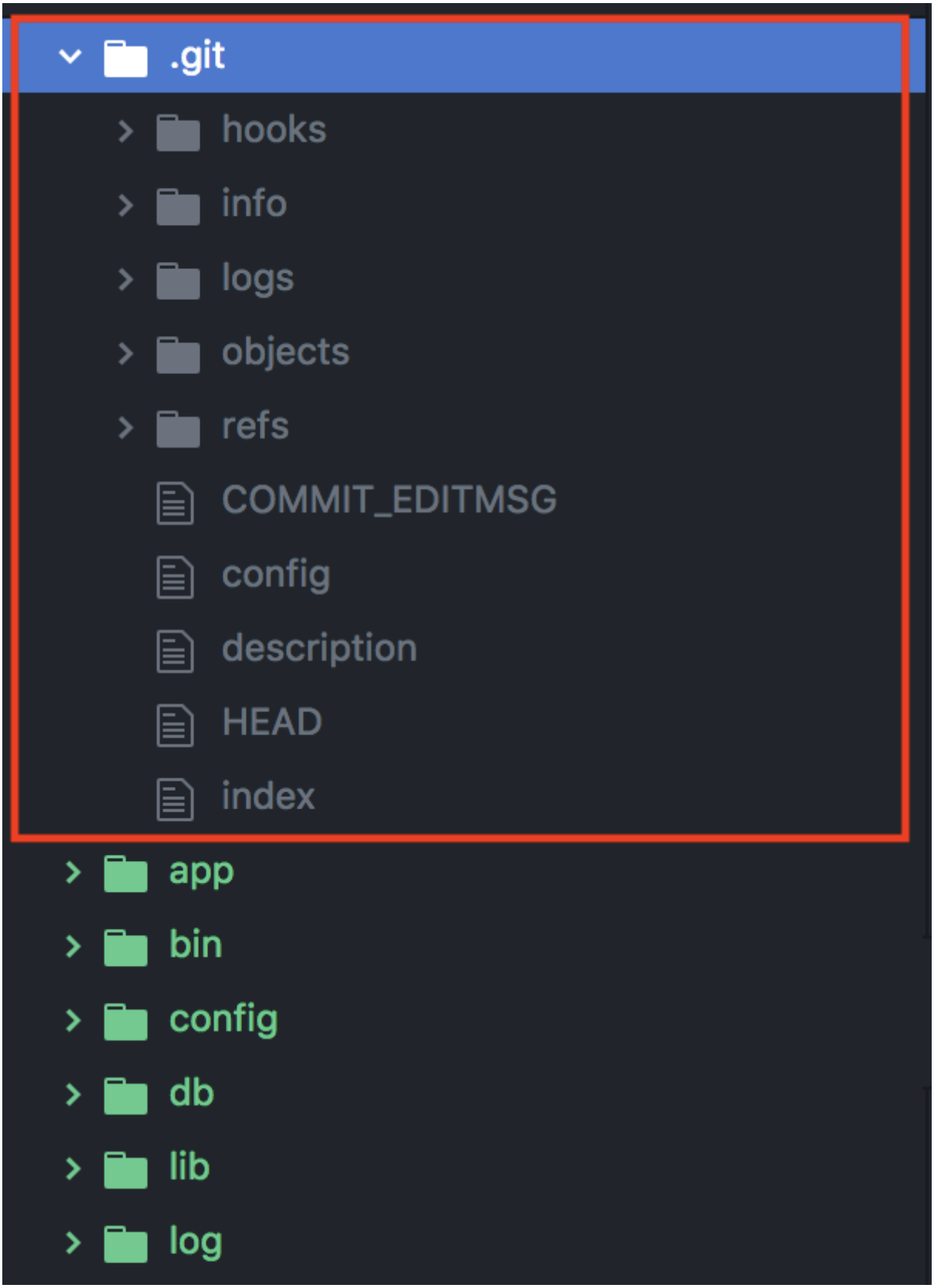
Next you'll see the setup page. These are the instructions for connecting the Repo you just created in Github (Remote) to the directory you created in your terminal (Local).

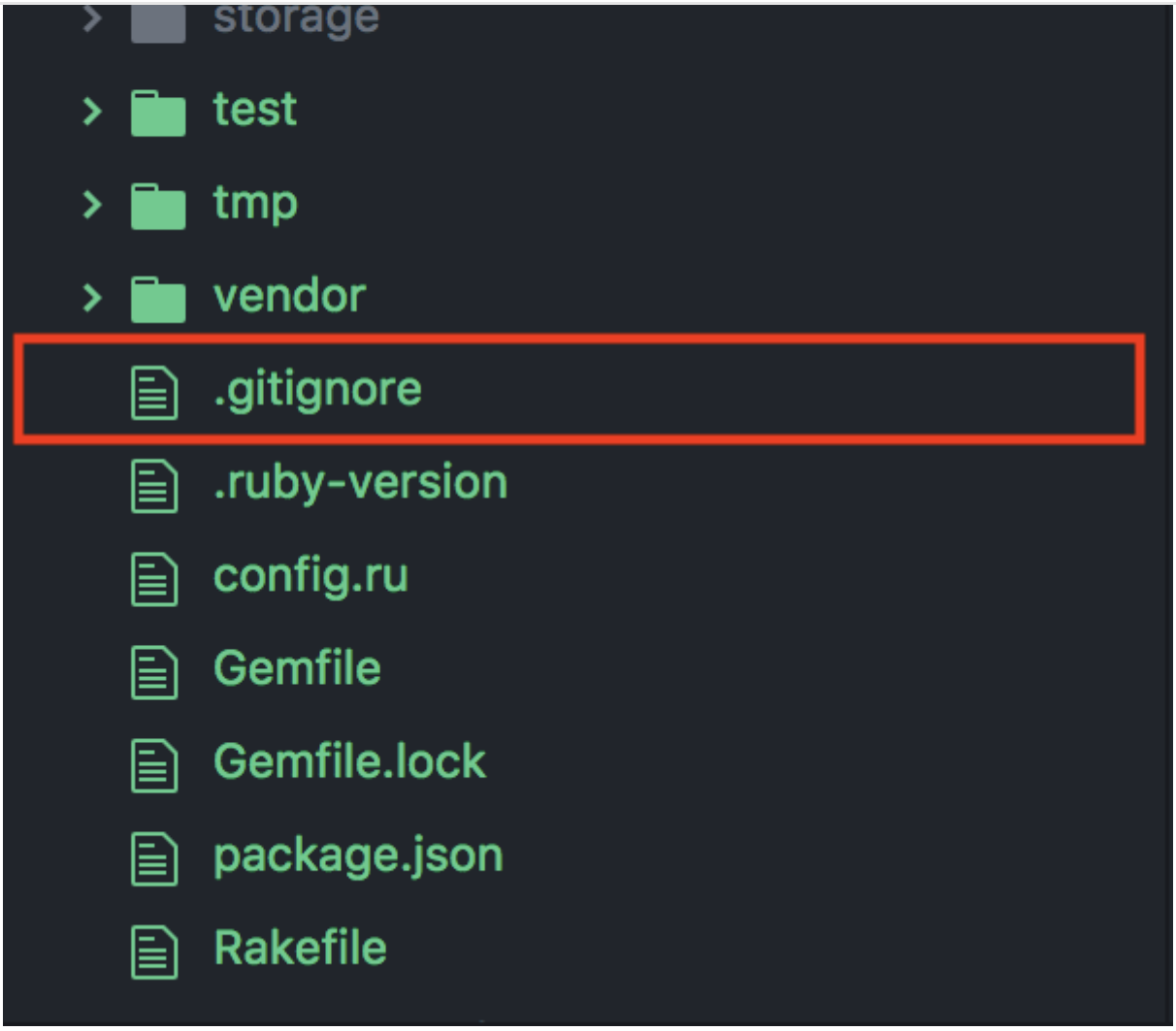


Paste the lines in the red box line by line starting with "echo…" into the terminal while you're cd'ed into the directory you just created locally. Your terminal should look like this when you're done:

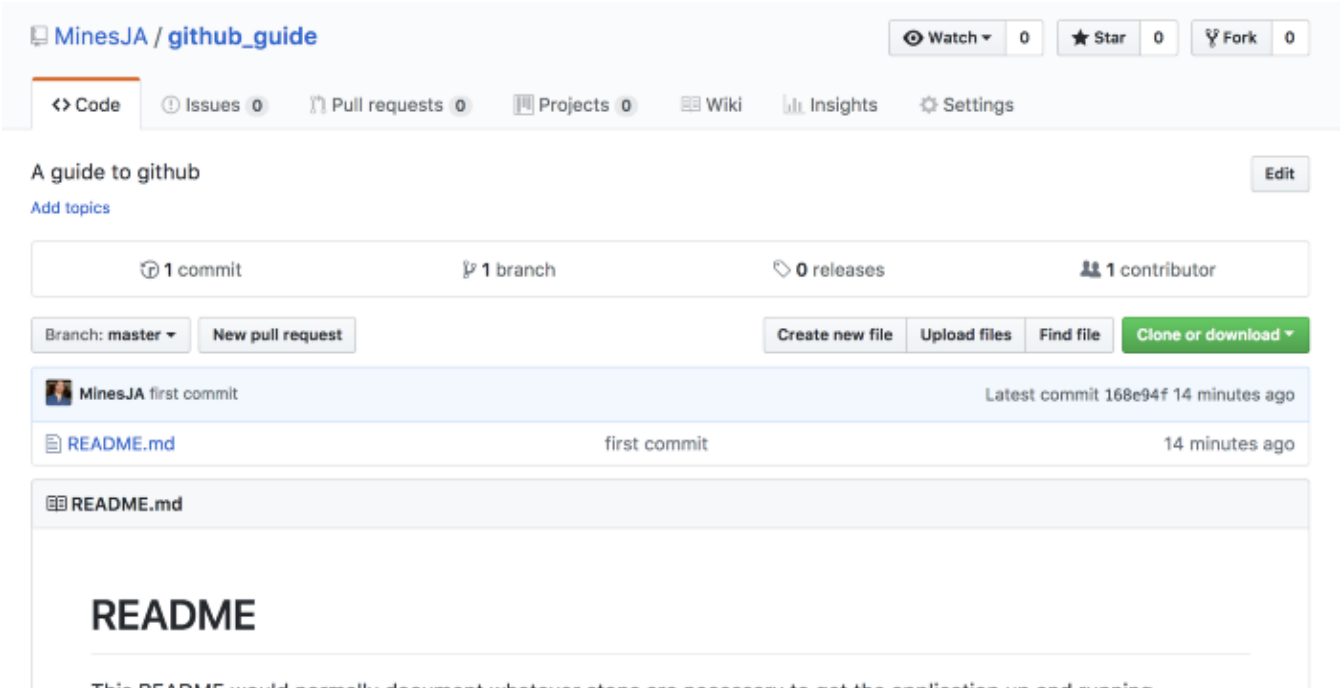
```
[13:51:37] github_guide
// ♥ echo "# github_guide" >> README.md
[13:51:49] github_guide
// ♥ git init
Reinitialized existing Git repository in /Users/jmines/Development/code/practice/github_guide/.git/
[13:51:54] github_guide
// ♥ git add README.md
[13:52:00] github_guide
// ♥ git commit -m "first commit"
[master (root-commit) 168e94f] first commit
 1 file changed, 25 insertions(+)
 create mode 100644 README.md
[13:52:04] (master) github_guide
// ♥ git remote add origin git@github.com:MinesJA/github_guide.git
[13:52:08] (master) github_guide
// ♥ git push -u origin master
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 465 bytes | 465.00 KiB/s, done.
```

This adds a '.git' folder to your repo, connects you to your remote Github Repo and also gives you a '.gitignore' file.





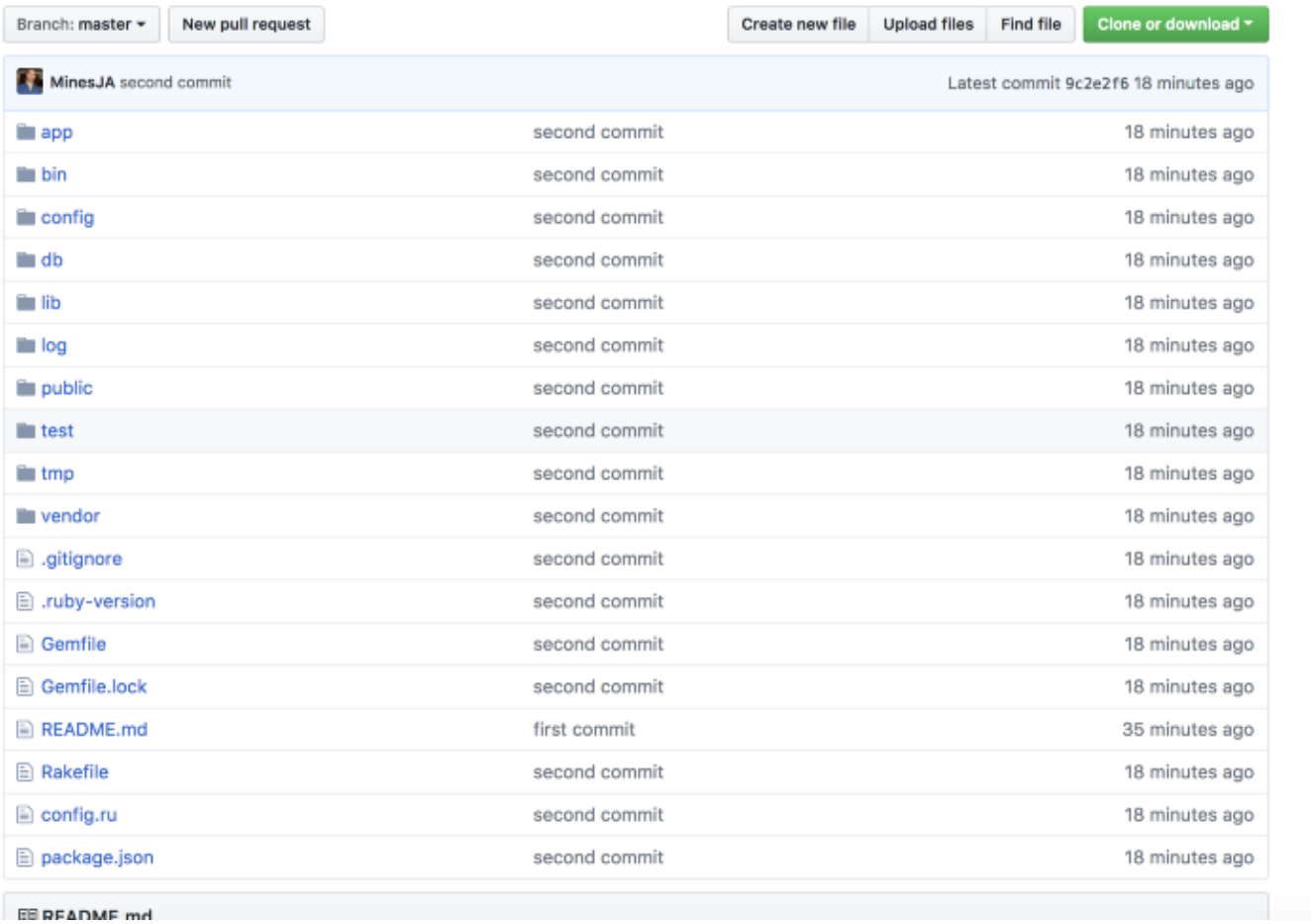
And if you go to your Github Repo page, you'll see the ReadMe that you intialized with and the reference to the first commit you made.



Now let's get this Repo up to date. Go back to your terminal and git add, git commit, and git push:

```
$ git add .
$ git commit -m "Second commit"
$ git push
```

Now check out your repo. It should have all the files you created your local directory with along with a new commit id (9c2e2f6):

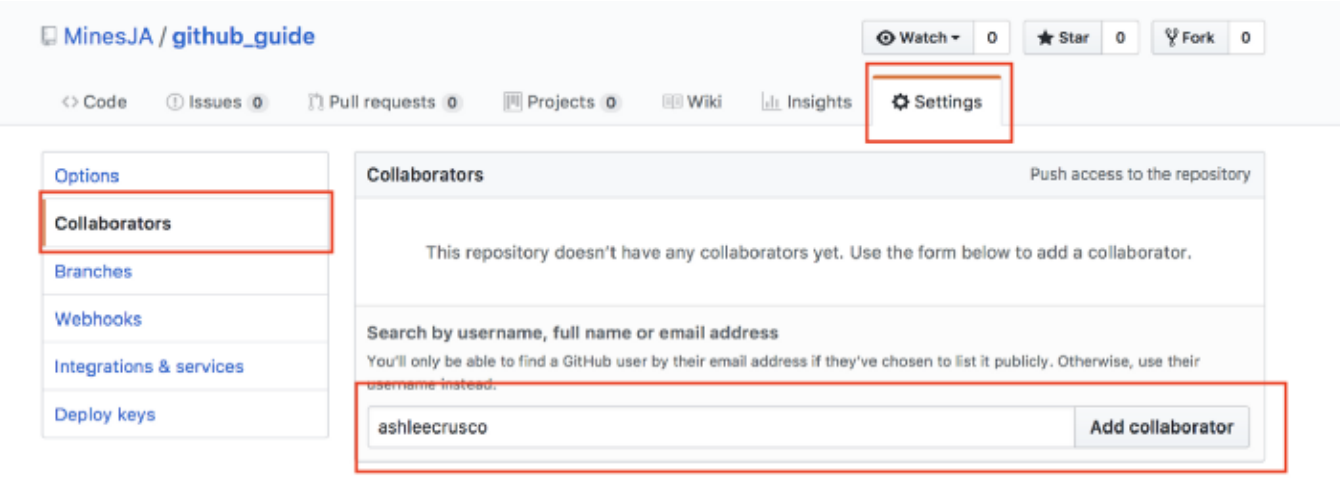


You're initialized and ready to start working!

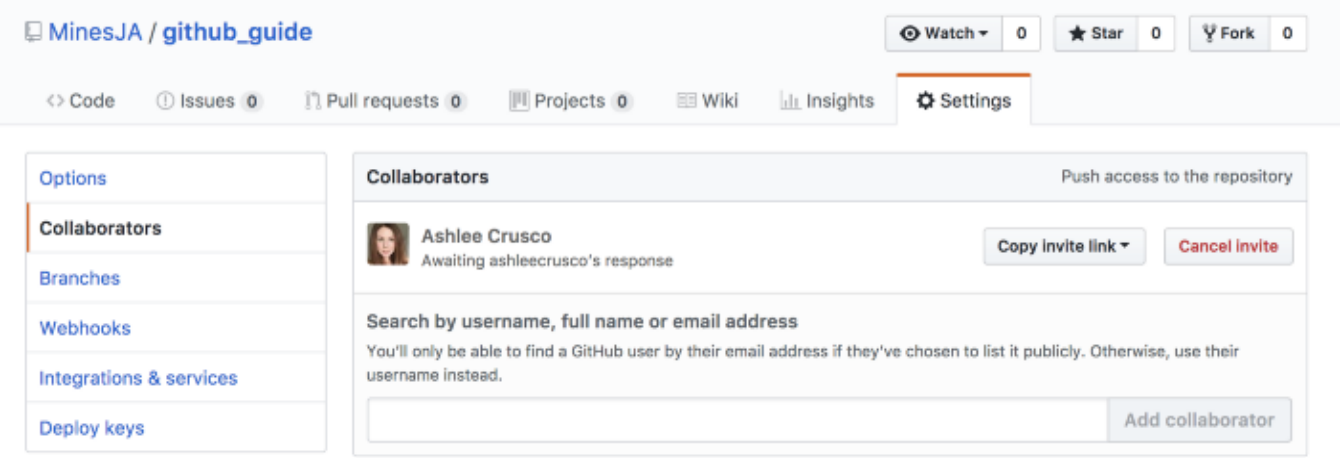
Step 2: Setup Your Team

and a ton of other destructive things so make sure you're only adding your teammates.

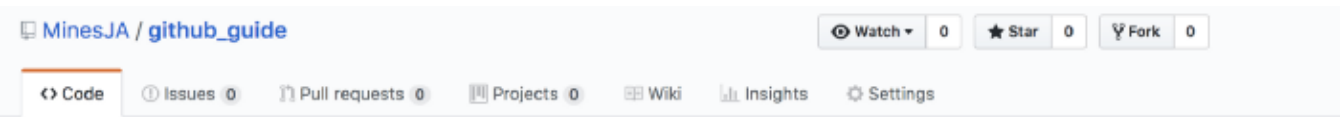
Click on the "Settings" tab of your rep, then "Collaborators" then search for Github users and add them by clicking "Add Collaborator":

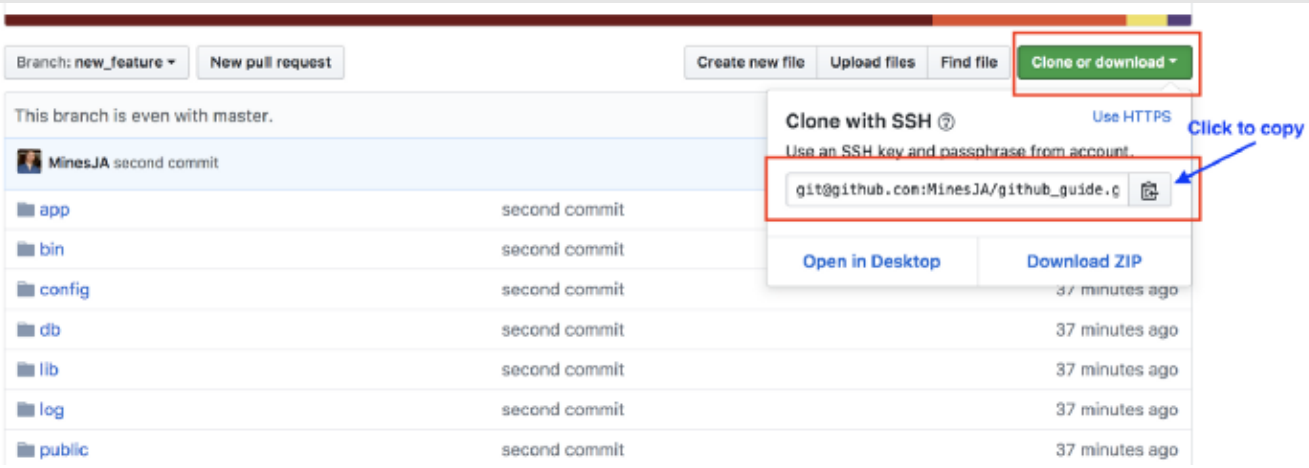


They'll receive an email letting them know you added them and will be listed as a collaborator.



If you're a collaborator, go to the Github Repo page, Git Clone the project, and cd into the directory. **Don't fork it!** Forking will copy it in a new Repo to your Github page, but you don't want that — you want to collaborate on the same Github Repo with your teammates.






```
$ git clone git@github.com:MinesJA/github_guide.git

$ cd github_guide/
```

And now you're ready to collaborate!

Step 3: Collaborating

When you're using git to work on the same project with multiple people, there's one central rule you must follow:

THE MASTER BRANCH SHOULD ALWAYS BE DEPLOYABLE

The way to keep Master deployable is to create new branches for new features and merge them into Master when they're completed. Here's how that works.

Step 3a: Branches

To start, branches should always represent features. For example, if you want to add the ability for a user to login you should probably create a branch called "user_authentication" and in that branch you should only update what you need to to enable a user to login.

It's also important when collaborating that your team picks features that don't have overlapping code. For example, you shouldn't be working on a "user_login" branch at the same time that your teammate is working on a "user_logout" branch because the

So let's say you want to create the User model. In your terminal create a new branch:

```
$ git co -b create_user
```

"co" is short for "checkout" which is used to switch between branches. Adding the "-b" and a name at the end creates a new branch and then moves into that new branch for us.

You should be able to verify this with the command:

```
$ git branch
```

Which should produce:

You're now in your new branch and can start coding away.

Note: As a general rule, you should git add frequently and git commit when you finish something that allows your code to work (ends up being a couple times an hour). For example, when you finish a method and the code base works, git commit like so:

```
$ git commit -m "Added function to allow Users to say 'Hello World'"
```

Step 3b: Submitting Pull Requests

merge them back into Master to be deployed.

Determining your Git Flow is a huge part of working in a team, but here's one Git Flow you could adopt for now:

First, determine who's going to be in charge of handling merging. The less people acting independently on merging the better so for a team of 4 it would probably behoove you to have one official "Reviewer" or "Merge Master".

Next, have everyone git push their branches:

```
$ git push
```

Now go to the Github Repo page. You should see the branch you pushed up in a yellow bar at the top of the page with a button to "Compare & pull request".

Note: Alternatively, you can select the branch in the drop-down "Branch:" menu and select the branch you just pushed up. You'll then have a "Pull request" and "Compare" button on the right hand side.

should click the “Reviewers” tab and select whoever your team decided would be the “Merge Master”. When you’re done, click “Create pull request”.

Step 3c: Merging Pull Requests

Note that if you’re a collaborator, you can merge your own pull requests. But, again, if you’re working on a team, it makes more sense to have one person do all the merging and everyone else submit “Pull Requests” and assign the “Master Merger” as a reviewer just to make sure you’re dealing with any merge conflicts one branch at a time.

So, assuming you are the one charged with taking care of all merges and someone has assigned you as “Reviewer” on a pull request, when you login to your Github you’ll notice you have a notification letting you know that someone has assigned you as a reviewer. You’ll also noticed a yellow bar indicating one of your teammates as “requested your review on this pull request.”

Go ahead and click the “Add your review” button.

This will take you to the Pull Request page. How you move forward from here is up to you and your team. If you're working together I would use every morning to sit down as a team and go through pull requests together. If you do that, you won't have to necessarily leave long-winded, detailed Review Summaries.

However, if you're working remotely, this will be your main tool for letting the requester know if they need to make changes or if you're going to merge their request.

When you click “Submit review” on the “Review changes” drop-drown your review will now exist as a comment on the pull request thread.

When you’re satisfied with the pull request, go to the bottom of the pull request and click “Merge pull request”.

You'll then see a "Pull request successfully merged and closed" message and a button to "Delete branch" which you should click.

Step 4: Rinse, Repeat

And that's pretty much it! Keep adding new branches for new features and then coming together as a team to merge them into master. Keep master clean and deployable and don't try to merge more than one branch at a time and you should be good to go.

ReadMe. Feel free to make a pull request if there's anything you want to change or if you just want to test out your pull request skills!

https://github.com/MinesJA/github_guide

Github Programming Git Coding Version Control