

# CAAM 519: Computational Science I

## Module 2: Version control and Git

Version control is an essential part of the programmer's toolkit. If you've ever developed software and wanted to:

- Develop a complicated new feature while still having a working copy of the code readily available,
- Work collaboratively on the same code base,
- Track official releases of your project, and switch seamlessly between them, or
- Isolate the point where a bug was introduced into your codebase,

taking the time to learn to use some form of version control system (VCS) would be a wise investment.

Version control tracks the changes made to a project, by different people, over time. There are a number of existing tools to choose from, and many companies have bespoke implementations that they use in-house. In recent years, the open source VCS of choice is git<sup>1</sup>, initially developed by Linus Torvalds to manage development of the Linux kernel. A big part of its success, particularly in the open source world, is Github, which offers a nice user interface for git repositories (and free hosting to boot).

### Getting git

First, let's install git and get it configured. If you're using Ubuntu, git does not come preinstalled, but you can get it readily enough via the package manager:

```
> sudo apt install git
```

If you're using a Mac, you can get git by installing the Xcode command line tools (xcode-select --install should work).

Now configure your identity with git:

```
> git config --global user.email "jah9@rice.edu"

> git config --global user.name "Joey Huchette"
```

---

<sup>1</sup> Another popular version control system is Subversion (typically abbreviated svn). It's quite possible you will encounter a project managed by subversion during your career. While the model of how changes are tracked is fairly different than in git, the basics are sufficiently similar that your skills in git should be reasonably transferable.

# Repositories

A git project is called a *repository* (or *repo* for short). While what, exactly, falls in the scope of a single project is not set in stone, a typical repository might host a single software library, all drivers used for the computational experiments appearing in a paper, or all assignments for a class (hint, hint). A project contains some number of files and directories, and tracks all changes made to those files. An atomic change is called a *commit*, and is made with reference to its parent commit (i.e. a linked list). The global history doesn't necessarily form a list of tree, however, as you can maintain multiple *branches* of commits, which can then be merged into each other.

Git is decentralized, in the sense that you can do pretty much everything locally, without a network connection. We'll discuss how to interface with remote versions of your repository later, but let's by making a new repository.

## Committing to our first repository

Make a new directory, move into it:

```
> mkdir my-first-git  
> cd my-first-git
```

To initialize a git repo whose root is your current directory, run

```
> git init  
Initialized empty Git repository in /home/joey/my-first-git/.git/
```

Now let's add a file:

```
> echo "Hello, world." > first_file.txt
```

At any point, we can check the status of our git repo:

```
> git status  
On branch master
```

```
No commits yet
```

```
Untracked files:
```

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

```
    first_file.txt
```

```
nothing added to commit but untracked files present (use "git add" to track)
```

First, we see that we are on the master branch. Your repo can have any number of branches of commits, e.g. to develop new features or track releases. The master branch is typically the mainline “development” branch.

Second, we see that `first_file.txt` file is untracked. This means that git isn’t, well, tracking it, so any changes we made will not be saved or shared with others.

Let’s add the file, and then recheck our status:

```
> git add first_file.txt
```

```
> git status
```

```
On branch master
```

```
No commits yet
```

```
Changes to be committed:
```

```
(use "git rm --cached <file>..." to unstage)
```

```
    new file:   first_file.txt
```

Our changes have been *staged* and are ready to be committed. In particular, we haven’t saved anything permanently to the git repo yet! We can do that by committing the changes:

```
> git commit -m "My first commit"
```

```
[master (root-commit) 11f5088] My first commit
```

```
1 file changed, 0 insertions(+), 0 deletions(-)
```

```
create mode 100644 first_file.txt
```

We use the `-m` flag to pass a commit message directly on the command line. If you want to write a more elaborate commit message, omit the flag and the string, and git will pop open your default text editor.<sup>2</sup>

And we're done! Our changes are permanently saved in the git repo.<sup>3</sup> We can see all commits on our current branch with

```
> git log
commit 11f5088f77da9eed47d498ea7addbc3ae5f9d08 (HEAD -> master)
Author: Joey Huchette <jah9@rice.edu>
Date:   Sat Aug 31 21:54:11 2019 -0500

    My first commit
```

Press `q` to exit.

## Branches, diffs, and merging

Let's say we want to add a cool new feature to our repo (which is much-needed, as it's a little barren right now). Instead of working on the master branch, let's make a new one to develop our changes:

```
> git branch new-branch
```

This creates a new branch, but doesn't move you to it. To switch to it, we need to do

```
> git checkout new-branch
Switched to branch 'new-branch'
```

Now let's add a new file, alter the existing file in our project, and then stage all modified files (note the new `-A` flag):

```
> echo "Lorem ipsum" > second_file.txt

> echo "Hello, foolish world." > first_file.txt

> git add -A
```

---

<sup>2</sup> You can set your default git text editor with, e.g., `git config --global core.editor vim`.

<sup>3</sup> Note that if you modify a file after staging it, you must restage that file if you want to commit your latest changes. Committing will only save changes that have been staged.

We can inspect the changes we have introduced using

```
> git diff HEAD

diff --git a/first_file.txt b/first_file.txt
index d0fdaad..3390511 100644
--- a/first_file.txt
+++ b/first_file.txt
@@ -1,1 @@
-"Hello, world."
+"Hello, foolish world."
diff --git a/second_file.txt b/second_file.txt
new file mode 100644
index 0000000..dce7167
--- /dev/null
+++ b/second_file.txt
@@ -0,0 +1 @@
+"Lorem ipsum"
```

The HEAD here is standard git terminology you will see often (typically in error messages). The HEAD points to the latest commit on the currently checked out branch; you can see it with

```
> git cat-file -p HEAD
tree 11f5088f77da9eed47d498ea7addbc3ae5f9d08
Author Joey Huchette <jah9@rice.edu> 1567783082 -0500
Committer Joey Huchette <jah9@rice.edu> 1567783082 -0500

My first commit
```

All git commits are uniquely identified with a hash, and you can check that this matches the hash from our first commit above. If we commit our changes:

```
> git commit -m "New file, and add a word to the first."
[new-branch cd47f2d] New file, and add a word to the first.
 2 files changed, 2 insertions(+), 1 deletion(-)
 create mode 100644 second_file.txt
```

Now we can observe that this same diff command doesn't show anything, since HEAD now points to this latest commit:

```
> git cat-file -p HEAD
tree cd47f2d28cd838bb501d04f1b0449c7fc8359558
Author Joey Huchette <jah9@rice.edu> 1567783962 -0500
Committer Joey Huchette <jah9@rice.edu> 1567783962 -0500
```

New file, and add a word to the first.

If we want to diff against the tip of this branch against the tip of master, we can do

```
> git diff master..new-branch
Changes to be committed:
diff --git a/first_file.txt b/first_file.txt
index d0fdaad..3390511 100644
--- a/first_file.txt
+++ b/first_file.txt
@@ -1,1 @@
-"Hello, world."
+"Hello, foolish world."
diff --git a/second_file.txt b/second_file.txt
new file mode 100644
index 0000000..dce7167
--- /dev/null
+++ b/second_file.txt
@@ -0,0 +1 @@
+"Lorem ipsum"
```

One of the nice things about VCS is that they (ideally) allow you to quickly switch among different versions of the code. For example, if our current branch is broken at the moment, we can always switch back to master with

```
> git checkout master
```

run some code, and then switch back to our new-branch for further development:

```
> git checkout new-branch
```

Let's change second\_file.txt a bit:

```
> echo "Unfinished business..." >> second_file.txt
```

As the text suggests, we might not be completely done with this change. If we want to switch back to the working master branch, git will complain about us having unsaved changes:

```
> git checkout master
error: Your local changes to the following files would be overwritten by checkout:
    second_file.txt
Please commit your changes or stash them before you switch branches.
Aborting
```

We have three options with what to do with this unfinished work. First, we could commit it as-is. This will certainly save things, but it means that our git commit history will be slightly messy (which is not something we can't fix later on; see squashing below). Second, we could just permanently delete the unsaved changes:

```
> git checkout .
```

This will be the option of choice if our changes are not particularly desirable to keep around. However, a third option is to stash our changes temporarily, and then recover them at a later time when we are ready to continue development:

```
> git stash
Saved working directory and index state WIP on new-branch: cd47f2d New
file, and add a word to the first.

> git checkout master

> git checkout new-branch

> git stash pop
On branch new-branch
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:   second_file.txt

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (461a86ee2b307d3be3d39f05a77a0774756207b9)
```

Let's finish up our work and commit it:

```
> echo "...OK, now it's done." >> second_file.txt

> git add second_file.txt

> git commit -m "Finished business."
[new-branch 13d8dac] Finished business.
1 file changed, 2 insertions(+)
```

At this point, we may consider our great new feature complete, and ready to merge into the master branch. To this, we switch to the branch we want to merge into (so, master), then merge in the other branch:

```
> git checkout master

> git merge new-branch
Updating de46650..13d8dac
Fast-forward
 first_file.txt | 2 +-
 second_file.txt | 3 +++
2 files changed, 4 insertions(+), 1 deletion(-)
create mode 100644 second_file.txt
```

Inspecting our commit log on the master branch, we see all the commits have been linked in to the master branch:

```
> git log
commit 13d8dac377f35864aae55e05e2027533c5921b8e (HEAD -> master,
new-branch)
Author: Joey Huchette <jah9@rice.edu>
Date:   Sun Sep 1 14:35:10 2019 -0500

    Finished business.

commit cd47f2d28cd838bb501d04f1b0449c7fc8359558
Author: Joey Huchette <jah9@rice.edu>
Date:   Sun Sep 1 14:20:42 2019 -0500

    New file, and add a word to the first.
```

```
commit de466506fa63eff8ccf3ac42fa7eaf91cb8f6caa
Author: Joey Huchette <jah9@rice.edu>
Date:   Sun Sep 1 14:04:33 2019 -0500
```

```
My first commit
```

## Resetting and reverting

We saw above a few options for clearing changes that we didn't want: stashing or checking out. However, both only work if the changes are in the "modified" state. If the changes were staged or committed, we will need to use `git reset`.

To illustrate, let's modify an existing file:

```
> echo "Demoing a (mixed) reset." >> second_file.txt
> git add second_file.txt
```

So, our changes have been added to the staging area. If we want to unstage the changes, we can do

```
> git reset HEAD
Unstaged changes after reset:
M    second_file.txt
```

This is a mixed reset (by default), which unstages any changes not present at the specified commit. However, the unstaged changes are not deleted, but moved to the working directory:

```
> git status
On branch master
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:   second_file.txt

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

> cat second_file.txt
"Lorem ipsum"
```

```
Unfinished business...  
"...OK, now it's done."  
Demoing a (mixed) reset.
```

There is also a `git reset --hard`, which deletes the unstaged changes:

```
> git add second_file.txt  
  
> git reset --hard # HEAD is implicit default  
HEAD is now at 13d8dac Finished business.  
  
> git status  
On branch master  
nothing to commit, working tree clean  
  
> cat second_file.txt  
"Lorem ipsum"  
Unfinished business...  
"...OK, now it's done."
```

Be very careful with hard resets, you can't undo them!

You can also use `git reset` to backtrack commits that are now unwanted by changing HEAD in the above commands to some previous commit that you want to backtrack to. However, a potentially safer way to do this is with `git revert`.

To illustrate, let's take some changes and commit them:

```
> echo "Demoing a revert." >> second_file.txt  
  
> git commit -am "This commit will soon be reverted."  
[master 7ae4075] This commit will soon be reverted.  
1 file changed, 1 insertion(+)  
  
> git revert HEAD # Opens text editor for revert commit message.  
[master 2756309] Revert "This commit will soon be reverted."  
1 file changed, 1 deletion(-)  
  
> cat second_file.txt  
"Lorem ipsum"  
Unfinished business...
```

```
"...OK, now it's done."
```

```
> git log
```

```
commit 2756309e780de0d645a07ab60cb48b5ab4a5f8b4 (HEAD -> master)
```

```
Author: Joey Huchette <jah9@rice.edu>
```

```
Date: Mon Sep 2 15:49:33 2019 -0500
```

```
Revert "This commit will soon be reverted."
```

```
This reverts commit 7ae4075c63b3813f6fc9f0e731cec475278dfe8f.
```

```
commit 13d8dac377f35864aae55e05e2027533c5921b8e (new-branch)
```

```
Author: Joey Huchette <jah9@rice.edu>
```

```
Date: Sun Sep 1 14:35:10 2019 -0500
```

```
Finished business.
```

```
...
```

The advantage of a revert over a hard reset is that you don't delete any commits or changes from your history, so you can always go back to them later if you need something from them. However, it gets slightly complicated if you want to revert multiple commits at once (though you can easily do them one-by-one).

## Merge conflicts

That merge worked seamlessly, but oftentimes you won't be so lucky. In particular, if you have two branches that have edited the same part of the same file, this will confuse git's automatic merge resolution, and you will have to go in manually to tell git which part of which changes you would like to keep.

```
> git checkout -b troublesome-branch
```

```
Switched to a new branch 'troublesome-branch'
```

```
> echo "Foo bar" > conflict.txt
```

```
> git add -A
```

```
> git commit -m "Version on branch."
```

```
[troublesome-branch 4ce70b8] Version on branch.
```

```
> git checkout master
```

```

Switched to branch 'master'

> echo "Foo baz bar" > conflict.txt

> git add -A

> git commit -m "Version on master."
[master 7e70d39] Version on master.
 1 file changed, 1 insertion(+)
 create mode 100644 conflict.txt

> git merge troublesome-branch
Auto-merging conflict.txt
CONFLICT (add/add): Merge conflict in conflict.txt
Automatic merge failed; fix conflicts and then commit the result.

```

If inspect our current status, we see that git has flagged the conflicts for us:

```

> git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both added:      conflict.txt

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

```

Indeed, it has actually changed the contents of conflict.txt to make explicit what needs to be changed:

```

> cat conflict.txt
<<<<<<< HEAD
Foo baz bar
=====
"Foo bar"
>>>>>>> troublesome-branch

```

This is git telling us that it can't automatically merge this line of the file. It shows the contents on HEAD (the tip of master) and the tip of troublesome-branch; it is our job to replace all of this with the contents we want, and then commit the changes. So: open up your text editor and fill in whatever text you would like to resolve this conflict. Then, add and commit the changes:

```
> git add conflict.txt

> git commit -m "Merge conflict resolved."
[master f6cc8dc] Merge conflict resolved.
```

## Squashing and rewriting history

Let's make a bunch of incremental commits, as you might when you are slowly working on a new feature:

```
> echo "Line 1 of 3." >> third_file.txt

> git add third_file.txt

> git commit -m "Part 1 of 3."
[master a30c705] Part 1 of 3.
1 file changed, 1 insertion(+)
create mode 100644 third_file.txt

> echo "Line 2 of 3." >> third_file.txt

> git add third_file.txt

> git commit -m "Part 2 of 3."
[master 072dcd1] Part 2 of 3.
1 file changed, 1 insertion(+)

> echo "Line 3 of 3." >> third_file.txt

> git add third_file.txt

> git commit -m "Part 3 of 3."
[master 7435761] Part 3 of 3.
1 file changed, 1 insertion(+)
```

After all this, our git log will be a bit cluttered with commits that track how we developed their contents, rather than the contents themselves:

```
> git log
commit 74357614b358d648c9a12cb5c2741077aef7c88e (HEAD -> master)
Author: Joey Huchette <jah9@rice.edu>
Date: Sun Sep 1 15:23:17 2019 -0500
```

Part 3 of 3.

```
commit 072dcd1026db52399936e25c9b56d9ec75dee85f
Author: Joey Huchette <jah9@rice.edu>
Date: Sun Sep 1 15:23:06 2019 -0500
```

Part 2 of 3.

```
commit a30c70559a48965f3346513d2cbaa72a50c7062b
Author: Joey Huchette <jah9@rice.edu>
Date: Sun Sep 1 15:22:45 2019 -0500
```

Part 1 of 3.

```
commit f6cc8dcda265cba9acca25f07428b7940743dfde
Merge: 7e70d39 4ce70b8
Author: Joey Huchette <jah9@rice.edu>
Date: Sun Sep 1 15:00:38 2019 -0500
```

Merge conflict resolved.

...

While there is nothing actively harmful about having a cluttered history, it can make it harder to track changes over time and isolate bugs that are introduced. In situations like this, you may be tempted to rewrite the git history. While this can be dangerous in general, it can make sense in situations like this. Indeed, it is often standard practice to rewrite cluttered commit history on a development branch before merging into the mainline branch.

We can rewrite the git history using the rebasing functionality. Let's say that we want to clean up all the commits from our last change where we had merge conflicts. To do this, we pick the hash from the last "clean commit" to keep, and plug it into the following command (where the `-i` means "interactive"):

```
> git rebase -i f6cc8dcda265cba9acca25f07428b7940743dfde
pick a30c705 Part 1 of 3.
```

```

pick 072dcd1 Part 2 of 3.
pick 7435761 Part 3 of 3.

# Rebase f6cc8dc..7435761 onto f6cc8dc (3 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .      create a merge commit using the original merge commit's
# .      message (or the oneline, if no original merge commit was
# .      specified). Use -c <commit> to reword the commit message.
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out

```

You can edit the first word in each row to tell git what to do with that particular commit. Commonly, you might want to keep the changes while getting rid of the commit itself, melting two commits together. In this case, fixup is your choice. Edit the contents of the file to read (modulo comments at the bottom):

```

pick a30c705 Part 1 of 3.
f 072dcd1 Part 2 of 3.
f 7435761 Part 3 of 3.

```

Saving and exiting, you get:

```

Successfully rebased and updated refs/heads/master.

```

And inspecting the logs we now see:

```
> git log
commit 742b5983e6fbe6334349112ae88a20c1fc61f137 (HEAD -> master)
Author: Joey Huchette <jah9@rice.edu>
Date:   Sun Sep 1 15:22:45 2019 -0500

    Part 1 of 3.

commit f6cc8dcda265cba9acca25f07428b7940743dfde
Merge: 7e70d39 4ce70b8
Author: Joey Huchette <jah9@rice.edu>
Date:   Sun Sep 1 15:00:38 2019 -0500

    Merge conflict resolved.
...
```

However, all the changes are still there:

```
> cat third_file.txt
Line 1 of 3.
Line 2 of 3.
Line 3 of 3.
```

We can rewrite the last commit message to be something more meaningful now:

```
> git commit --amend
```

Then type something like “One new feature, spread across three commits (now squashed).”, save, and inspect the logs to see the updated commit message.

Frequently, you will have to resolve conflicts between commits when you try to squash or fixup them using rebase. Thankfully, you can do this in much the same way as we saw above in the merge conflicts section.

## Collaboration with remotes

Up to this point, we’ve been doing everything locally. However, one of the main selling points of VCS is the ability to collaborate with other developers on the same project. To do this, we need

the concept of a *remote* version of the repo. This could be a server sitting on your advisor's computer, or the copy of your open source project hosted on Github.

To illustrate things, I'm going to show you how to interact with repos hosted on Github. I made a [stub repo for this class](#). Navigate outside of your current git directory (e.g. to your home directory) and clone a copy of the CAAM 519 repo with:

```
> git clone https://github.com/joehuchette/caam-519-f19
Cloning into 'caam-519-f19'...
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 9 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (9/9), done.

> cd caam-519-f19
```

We can inspect the remotes that are registered with this repo with:

```
> git remote -v
origin      https://github.com/joehuchette/caam-519-f19 (fetch)
origin      https://github.com/joehuchette/caam-519-f19 (push)
```

Typically origin is used to refer to the “central” instance of the repo (much like “master” is typically used for the mainline branch), though git is a completely decentralized VCS so there is no formal meaning to this.

So you now have a local copy of the repo which is hosted remotely. Say I change the repo hosted remotely on Github by adding a file. To get these changes, you will manually have ask for the changes; it will not automatically sync. To download all changes from the remote repo, run

```
> git fetch
```

This is completely nondestructive. If you want to download the changes and merge them into your current branch (say, master), run

```
> git pull origin master
```

Be ready, though: if git runs into merge conflicts pulling the remote changes, you will have to resolve them manually!

Just as you can pull changes from a remote, you can also push your changes to that remote. Let's make a small change to the README, then commit them:

```
> echo "A small, local change." >> README.md

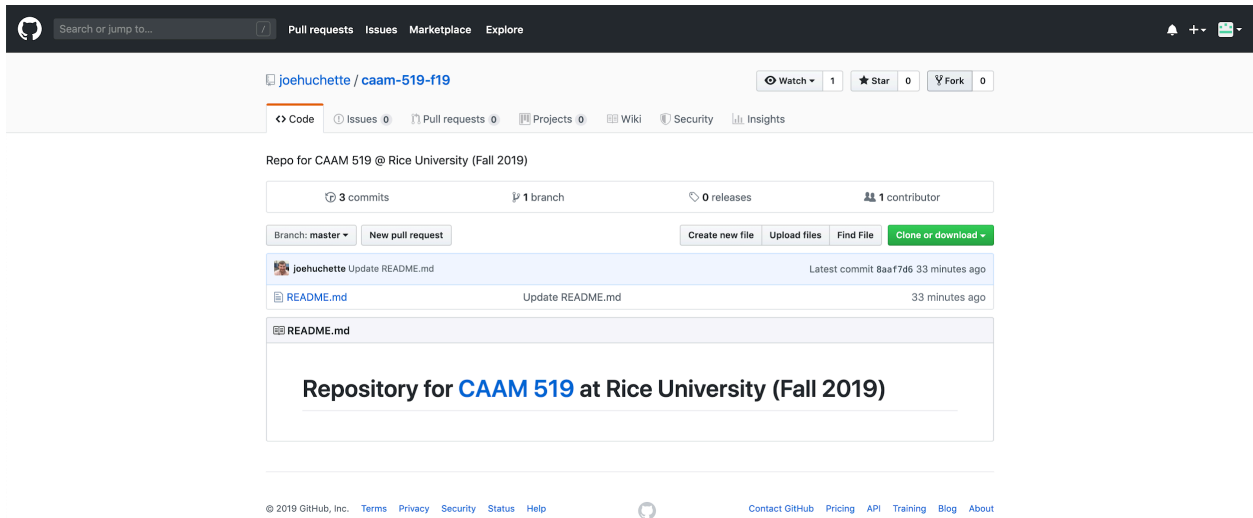
> git add README.md

> git commit -m "A small, local change to the README.md."
[master fe76672] A small, local change to the README.md.
1 file changed, 2 insertions(+)
```

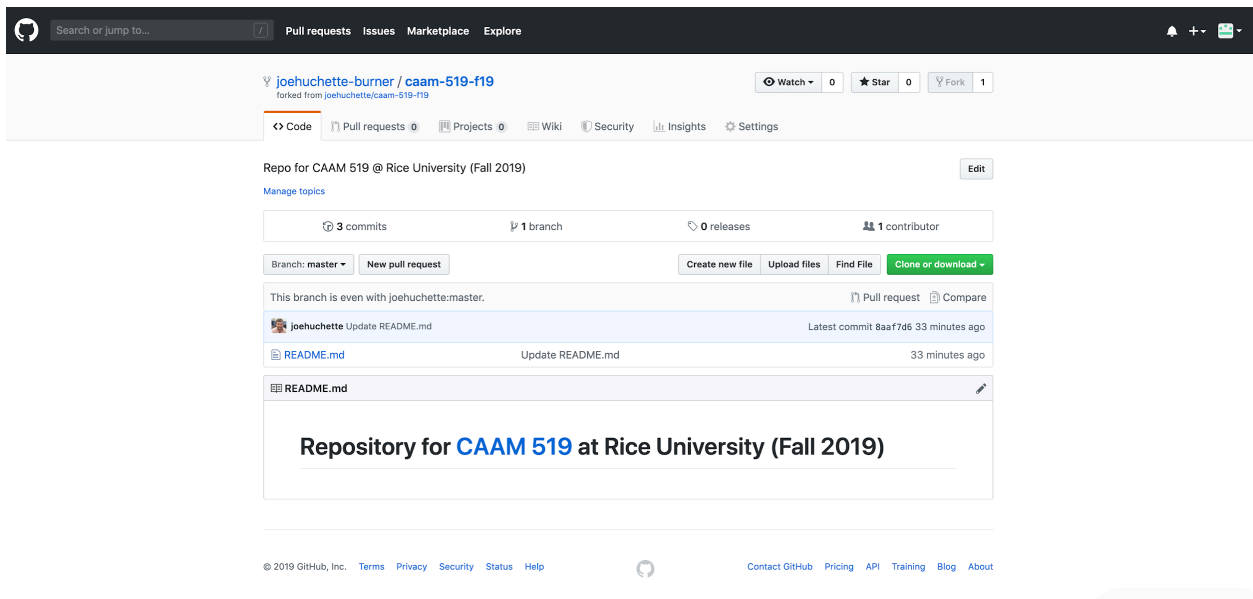
If we try to push these changes to the remote, we will be prompted for a Github login:

```
> git push origin master
Username for 'https://github.com':
Password for 'https://joehuchette-burner@github.com':
remote: Permission to joehuchette/caam-519-f19.git denied to
joehuchette-burner.
fatal: unable to access 'https://github.com/joehuchette/caam-519-f19/': The
requested URL return error: 403
```

This makes sense; we don't want any random person to be able to change our repo. What we're going to do is fork this repo. Forking is Github lingo for taking a repo that is hosted on Github and owned by someone else, and making a copy (also on Github) for which you are the owner. To do this, click the "Fork" button in the upper right:



That's all; you now have a copy of the repo where you have full permissions:



Click the green button to get the https address for the forked repo. You can then add a remote for your forked copy with:

```
> git remote add my-fork https://github.com/joeHuchette-burner/caam-519-f19.git
```

You can now push your local changes to the remote server:

```
> git push my-fork master
Username for 'https://github.com': joeHuchette-burner
Password for 'https://joeHuchette-burner@github.com':
Counting objects: 3, done.
```

```
Compressing objects: 100% (2/2), done.  
Writing objects: 100% (3/3), 288 bytes | 288.00 KiB/s, done.  
Total 3 (delta 1), reused 0 (delta 0)  
Remote: Resolving deltas: 100% (1/1), completed with 1 local object.  
To https://github.com/joehuchette-burner/caam-519-f19.git  
8aaf7d6..fe76672 master -> master
```

Now go to the github page for your fork; you should be able to see your local changes, now online!

## Other useful features

### Pull requests on Github

Github has a nice feature for reviewing potential merges between branches. I can't do it justice here, but it has useful features like comments, code reviews, diffs among branches or commits, and more. If you host your projects on Github, odds are you will end up using this feature.

### Git tags

For software projects that change over time, it's important to record different versions of the software over time. This is essential to ensuring that code you revisit in years time will still run; if you try this exercise with the master branches of all your dependencies, your chance of success is almost zero.

Git has a mechanism to label commits with handy labels called tags. This functionality can be used for a number of things, but for our purposes we will use it to tag different release points of our code.

To tag a commit with the label v1.0.0<sup>4</sup>, point HEAD to the commit you would like to tag, and run

```
> git tag v1.0.0
```

You can inspect the existing tags for the repo with

```
> git tag
```

And check them out in the same way you would check out a branch of a single commit via its hash:

---

<sup>4</sup> For guidance on how to choose version numbers, see [SemVer](#).

```
> git checkout v1.0.0
```

You will use tags to label the official “submission” for your homework assignments.

## Staging portions of a file

Sometime you will have a number of changes to a particular file, but you only want to stage part of those changes (for example, if you want to create multiple commits to track different changes on a single branch). To do this, you can use the `-p` (or `--patch`) flag to your usual staging invocation:

```
> git add -p file_with_many_changes.c
```

## Cherry picking

This operation is particularly useful if you maintain a release branch (e.g. all `v1.x.x` releases), alongside a mainline development branch on `master`. Say you have a bugfix you add to `master`, and would like to apply it to the next release. You can “cherry pick” this single commit from the `master` branch and apply it to the release branch with a single command. If `xxx` is the hash for the commit on `master`, switch to the release branch and run:

```
> git cherry-pick xxx
```