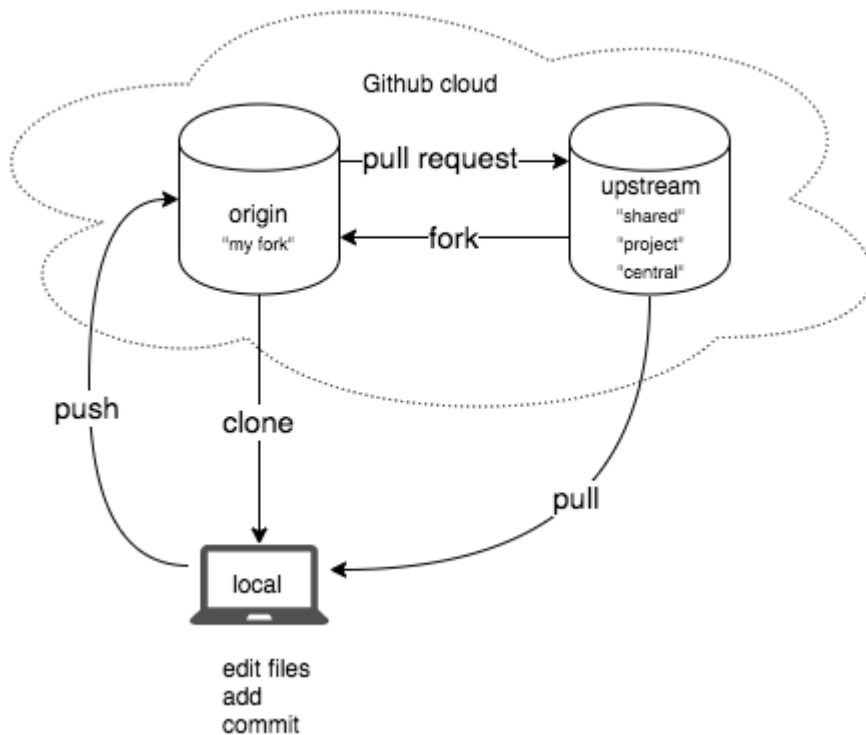


Lab 2, use github

Cheatsheet picture



Basic forking workflow setup

1. Group of three: *Maintainer*, *Contributor A*, *Contributor B*.
2. *Maintainer* creates new repository on github.
3. *Contributor A* and *Contributor B* log into github, find the new repository created by *Maintainer* and fork it.



Image showing fork button in top right

4. *Maintainer*, *Contributor A*, *Contributor B* should all clone their individual repo to their laptop. Everyone will have their own username in the URL. You can quickly copy that with this button on github.

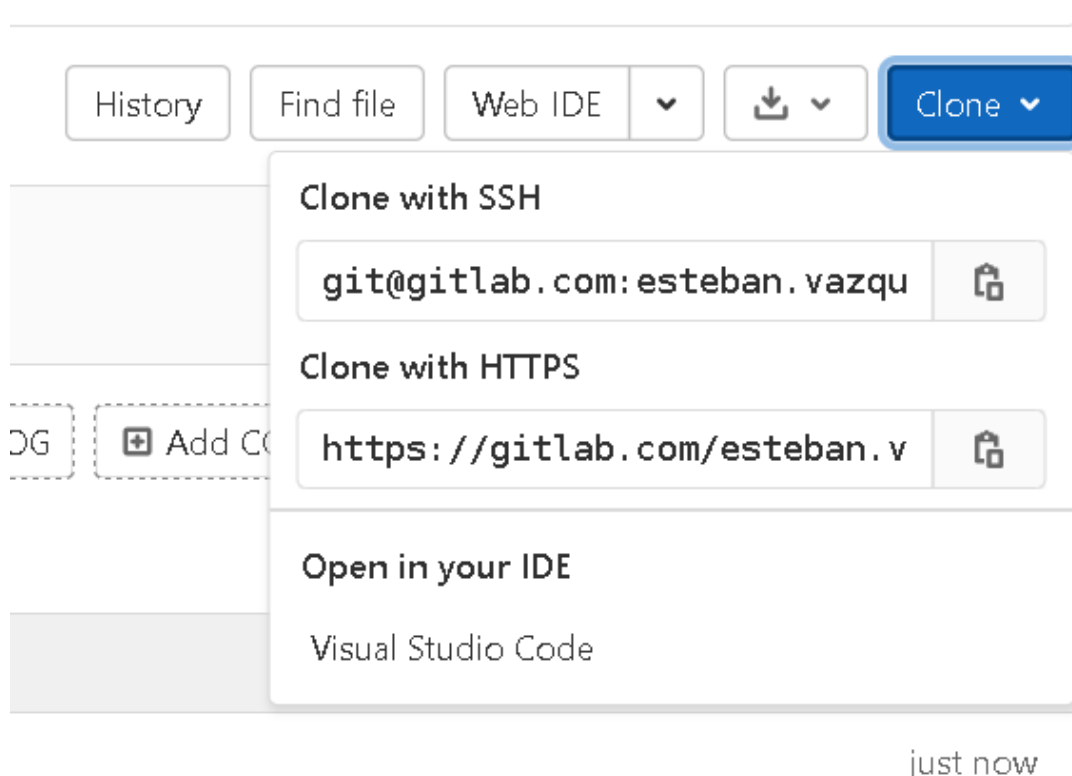


Image showing button that copies the URL

Before running `git clone`, everyone needs to ensure they are in a folder that is not inside another repo. To do this run `git status` and you *want* to see a fatal error.

```
git status
fatal: Not a git repository (or any of the parent directories): .git
```

If you do not see the fatal error, then you need to move up in your folder hierarchy until you are not in a repo. Use `cd ...`

Then clone your repository and change directory into it.

5. Establish remotes. *Maintainer* does not need to do this, since they have directly cloned the shared repository. *Contributor A* and *Contributor B* will specify the shared repository as “upstream”.

If setup properly then the `git remote -v` command will show two lines for *Maintainer*, but four lines for *Contributor A* and *Contributor B*:

Maintainer

```
git remote -v
origin <fork_repo_url> (fetch)
origin <fork_repo_url> (push)
```

Contributor A and *Contributor B*:

```
git remote add upstream https://github.com/eelrood/devOps.git
git remote -v
```

```
origin <fork_repo_url> (fetch)
origin <fork_repo_url> (push)
upstream <maintainers_repo_url> (fetch)
upstream <maintainers_repo_url> (push)
```

Note that git automatically creates the `origin` remote as whatever URL was used with `git clone`. This has the odd implication that for the *Maintainer* their `origin` is the shared repository, while for *Contributor A* and *Contributor B* their `origin` is their individual forks.

Each exercise below assumes this setup in [Basic forking workflow setup](#) as the starting point.

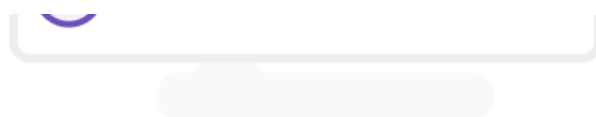
Sharing work and synchronizing

Now we are going to use `git push` and Github Merge requests, followed by `git pull` to collect work from *Maintainer*, *Contributor A*, *Contributor B* and in the shared repo, then synchronize so that everyone has a local copy of everyone's work.

1. *Maintainer*, *Contributor A*, *Contributor B* each create a file named after their role (ie `maintainer.txt`, `contributorA.txt`, `contributorB.txt`). Use whichever editor you like (e.g., `vi`, Atom, Rstudio, etc.)
2. Add and commit your new file.
3. Push your new file. By default git push uses `origin` as the destination.

For *Maintainer* the file has gone directly into the shared repo. For *Contributor A* and *Contributor B* these files are now at their fork, and will need a Merge request to move to the shared repo.

4. *Contributor A* and *Contributor B* go to github and look at their own fork. Github knows that the fork now has material that the shared repo doesn't, and will make a New Merge request button available.



Merge requests are a place to propose changes you've made to a project and discuss those changes with others

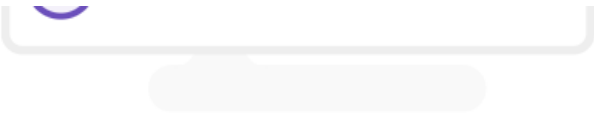
Interested parties can even contribute by pushing commits if they want to.

New merge request

Image showing New Merge request button

Since we created files with unique file names, these PRs will show up as “Able to be automatically merged”.

5. The *Maintainer* should now see a number next to the Merge request tab:



Merge requests are a place to propose changes you've made to a project and discuss those changes with others

Interested parties can even contribute by pushing commits if they want to.

New merge request

Image showing new Merge requests available

The *Maintainer* can accept both merge requests, which will merge files into the shared repo.

Now each person needs to synchronize with the shared repo, bringing the new files from the other two group members into their local repo. This uses the `pull` command.

6. *Contributor A* and *Contributor B* should pull upstream.

But *Maintainer* is pulling from their `origin`, so can use the default

7. Finally, *Contributor A* and *Contributor B* will push these new changes from the shared repo up to their forks.

I've always thought it is a little strange that a contributor's fork doesn't distribute changes from upstream, but that new work from the shared repo gets to the fork via the contributor's local repo on their laptop. After all, github knows the fork is connected to the shared repo ... but this is the roundabout way it works.

Generate merge conflict

(If starting here, confirm that your Group of three has the setup in [Basic forking workflow setup](#) as the starting point.)

The point of this exercise is to intentionally create conflicts in editing, to demonstrate how they show up in Merge requests.

1. *Maintainer* should create, add, commit, and push a file called `animals.txt` with this content:

```
lion
tiger
leopard
turtle
```

3. *Contributor A* and *Contributor B* should synchronize to receive this file.

```
git pull upstream main
```

4. *Contributor A* and *Contributor B* each edit the file by adding a different color to the first line in the `animals.txt` (so it reads `lion red` or `lion green`). They add, commit, push to their forks. *Note that these edits are incompatible, so they will generate a conflict when we try to merge them below*
5. *Contributor A* and *Contributor B* should then go to github and look at their forks. They should create a merge request by hitting the 'New Merge request' button. Note that the merge request will not say "Able to automatically merge" but create the PR anyway.
6. With everyone looking at *Maintainer's* computer, *Maintainer* should refresh the github page for the shared repository and will see two pull-requests in the pull-request tab. Accept each one, resolving any conflicts that emerge.

They can choose whatever contributions they want (the original line, the line with the color from *Contributor B* or the line with the color from *Contributor A*); however the file is after removing the `<<<< =====` and `>>>>>` lines will be what is in the shared repository.

6. Each person should then synchronize. For *Maintainer* that just means `git pull` (since they have a direct clone of upstream). For *Contributor B* and *Contributor A* they have to first get the changes in upstream, then push them to their fork.

Once more, but with branches

In the exercises above all the work was on the `master` branch. That was to keep things simple, but in reality new work should always be on a branch, usually called a (short-lived) "feature branch". Merge requests then come from the feature branch to the master branch on the shared repo. This is also true for *Maintainer* as well as *Contributor A* and *Contributor B*, the only difference for the *Maintainer* is that their merge request is from a branch inside the shared repo, while the *Contributor A* and *Contributor B* is from a branch in a different repository.

(If starting here, confirm that your Group of three has the setup in [Basic forking workflow setup](#) as the starting point.)

1. On local, create a branch (`for-pull-request`)
2. Edit a file named after your role (ie "maintainer.txt", "contributorA.txt", or "contributorB.txt"), add, commit.
3. Push the branch to origin (`git push` will fail but the error message will show you a command that will run). Note that is the fork for *Contributor A* and *Contributor B* but the shared repo for *Maintainer*, but called `origin` for everyone.

4. On github, make sure you are looking at the new branch (by choosing it from drop down menu) and open a PR from the new branch to `master` on the shared repo. All should do this (*Maintainer, Contributor A, and Contributor B*).

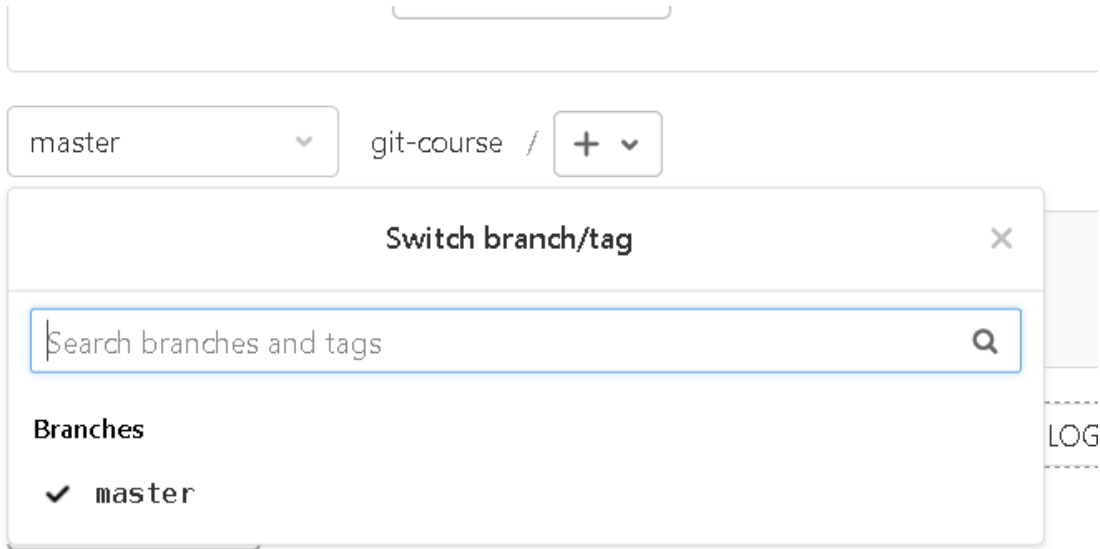


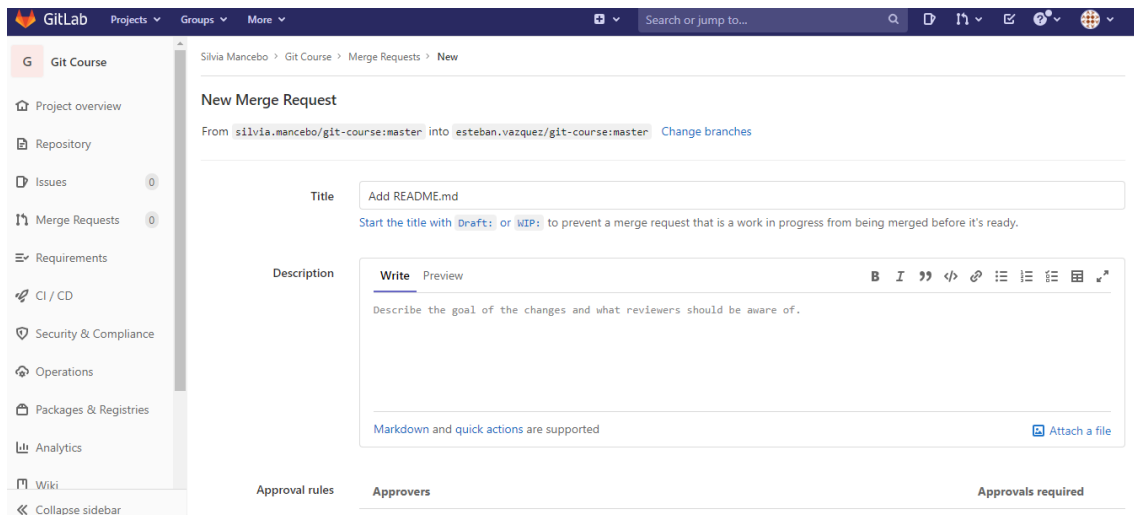
Image of drop down menu for branches, shows for-pull-request and master

Merge requests are a place to propose changes you've made to a project and discuss those changes with others

Interested parties can even contribute by pushing commits if they want to.

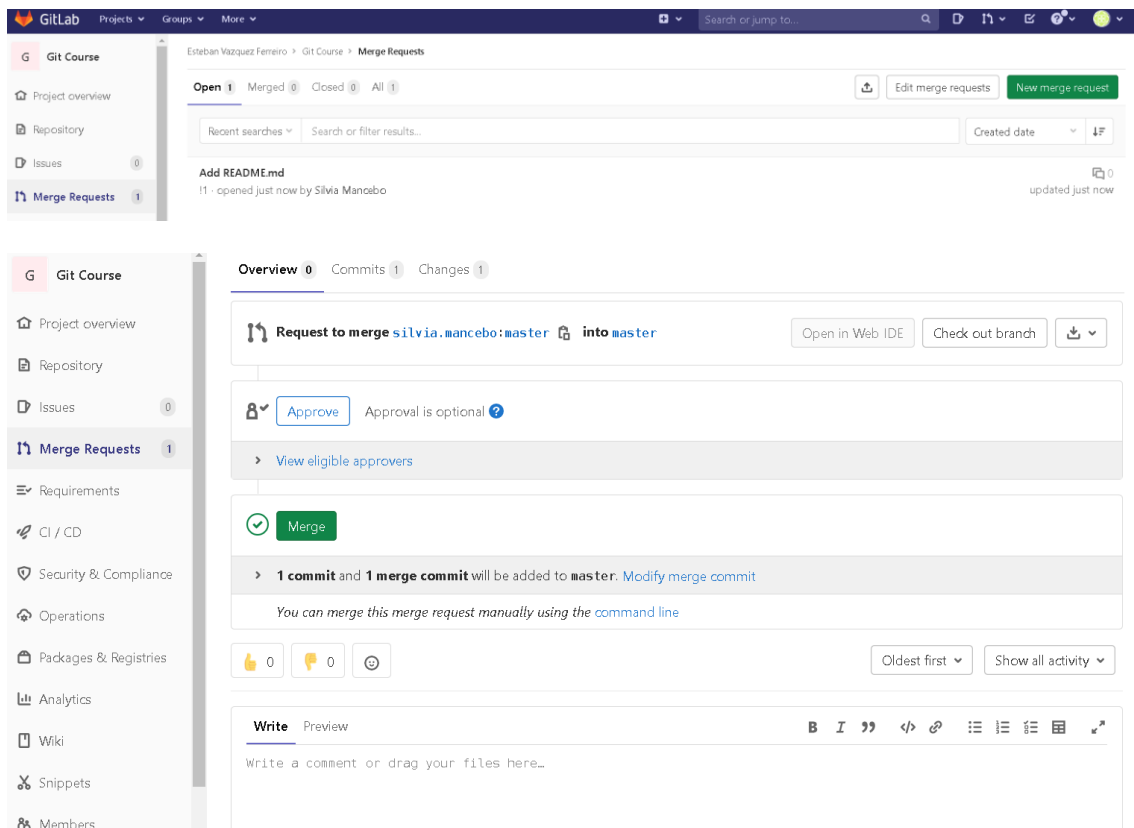
New merge request

New Merge request button



Create PR screen

5. *Maintainer* can then accept all three PRs on github.



6. For all three, *Maintainer*, *Contributor A*, and *Contributor B*, back on local, switch to master.

(Note that I think that up-to-date message is wrong because branch is behind master, hence being able to pull things, if anyone knows what's up there please tell me!)

7. Pull changes.

```
git pull
```

```
remote: Enumerating objects: 1, done. remote: Counting objects: 100% (1/1), done.  
remote: Total 1 (delta 0), reused 0 (delta 0), pack- 36cd034..6d0673d master ->  
origin/master Updating 36cd034..6d0673d Fast-forward README.md | 1 + 1 file changed,  
1 insertion(+) $ git status On branch master Your branch is up-to-date with 'origin/master'.
```

```
nothing to commit, working tree clean ``
```

8. Delete the branch locally and on github

Tags

Tags help identify important points in the version history. They are also used for releases on github. They point to a specific commit, but are human readable, rather than the hash. So they are like a bookmark which brings you directly to a specific point in the repo history.

Your task is to create three tags, `tag-animals`, `tag-colors`, `tag-noises` which reflect the merge points of different branches.

1. Have *Maintainer* create a repository, and start with the file having only animals.
2. *Maintainer* tag repo with `tag-animals`
3. *Contributor A* create a branch `colors-for-tag`. *Contributor B* and *Maintainer* contribute to that branch (`lion yellow`) via merge requests.
4. *Contributor A* merge `colors-for-tag` branch to master, then tag master as `tag-colors`.
5. *Contributor B* create branch `noises-for-tag`. *Contributor A* and *Maintainer* contribute to that branch as well. (`lion yellow roar`)
6. *Contributor B* merges that branch to master, then tag master as `tag-noises`. Others update.

Look at the history via `git log`:

```
git log --oneline --abbrev-commit --all --graph --decorate --color
```

To see the status of the repo at a tag, you can use

```
git checkout <tag_name>
```

You'll see a message about a "Detached HEAD" which sounds awful but actually means that Git wouldn't know where to put any new commits you make (since new commits are given HEAD as the parent). So tags are mostly used for read-only inspection of code at a particular point in history.

Split a merge request

Here the scenario is that you've created a merge request but it hasn't yet been accepted.

If you add additional commits to that branch then they will get added to the merge request. Remember, a merge request says, "Please come to this repo and get everything on this branch." So it's not the same as putting some commits in a zip file and mailing them, it's

more like putting things into a particular spot (like a mailbox or deaddrop in a spy movie), and telling people to pick them up from there.

So you can always add additional things before the person comes to pick them up. This is useful because if there is a conversation around the merge request then you can easily update things. For example, if someone said “Please fix a typo or pull from upstream before we consider your merge request,” you’d be able to do so without opening another PR (just add, commit, push to your branch on your fork and the PR is updated).

However, it is an issue if you *accidentally* add new commits to a branch before a merge request is accepted. Now your merge request has two sets of commits: the first set you meant to include and the second set you didn’t. This mistake is particularly easy to make if you are developing on the `master` branch in your fork (which you shouldn’t do), but also happens if you are have more than one contribution that you are working on, as when you are doing something else while waiting for a PR to be accepted. If you’ve accidentally added too many files to your merge request—something that is easy to do if you use `git add *` or some variant—you’ll also find yourself needing to remove content from your PR.

Split before submit

Things would be better if you had created a new branch for the first set of commits, then a second branch for your second set of commits, never adding either set to your master branch and following the “always work on a (short-lived) feature branch” rule. Then each set of commits would be “sent” through a different merge request.

```
    / Branch for first set.  
    /  
master -  
    \  
    \ Branch for second set
```

The grey sections below are commands, the white is output.

```
This is a git command that you should copy
```

```
This is output.
```

Below, I make a new repo, create a README file and make a commit, and edit it once making another commit:

```
git init
```

```
Initialized empty Git repository in /Users/howison/Documents/UTexas/Courses/PeerProduction/practice/.git/
```

```
touch README
```

```
git add README
```

```
git commit -m "Adding readme"
```

```
[master (root-commit) d5e1e8] Adding readme
```

```
1 file changed, 1 insertion(+)
```

```
create mode 100644 README
```

```
git log --oneline --abbrev-commit --all --graph --decorate --color
* d5eb1e8 (HEAD -> master) Adding readme
```

So now we have a single commit on a single branch `master`. To create a new file we can use `touch` (which creates an empty file here, but that's ok).

```
touch master2
git add master2
git commit -m "add master2"

[master 749454e] add master2
 1 file changed, 2 insertions(+)
git log --oneline --abbrev-commit --all --graph --decorate --color
* 749454e (HEAD -> master) add master2
* d5eb1e8 Adding readme
```

Now we have added a commit on `master` so the `HEAD -> master` shows the 'tip' of the `master` branch.

Now we can create the feature branch (`first_set_branch`) for the first set and look at the commits we've made. Using `git checkout` with `-b` creates a new branch:

```
git checkout -b first_set_branch

Switched to a new branch 'first_set_branch'

touch first_set_file
git add first_set_file
git commit -m "first set edit"

[first_set_branch d71ade9] first set edit
 1 file changed, 1 insertion(+)
 create mode 100644 first_set_file
git log --oneline --abbrev-commit --all --graph --decorate --color
* d71ade9 (HEAD -> first_set_branch) first set edit
* 749454e (master) add master2
* d5eb1e8 Adding readme
```

Now we can see that we have two branches (`master`) and `first_set_branch`. `HEAD` shows us that we are currently in the `first_set_branch` branch.

Now we're going to change back to `master`.

```
git checkout master

Switched to branch 'master'

git log --oneline --abbrev-commit --all --graph --decorate --color
* d71ade9 (first_set_branch) first set edit
```

```
* 749454e (HEAD -> master) add master2
* d5eb1e8 Adding readme
```

See how the `HEAD` changed, showing us that we're looking at the master branch. Notice also that there is a commit (`d71ade9`) that isn't in `master`.

Now we can create the second branch:

```
git checkout -b second_set_branch
Switched to a new branch 'second_set_branch'
touch second_set_file
git add second_set_file
git commit -m "first edit in second set"
[second_set_branch 9dc0a23] first edit in second set
1 file changed, 1 insertion(+)
create mode 100644 second_set_file
git log --oneline --abbrev-commit --all --graph --decorate --color
* 9dc0a23 (HEAD -> second_set_branch) first edit in second set
| * d71ade9 (first_set_branch) first set edit
|/
* 749454e (master) add master2
* d5eb1e8 Adding readme
```

Now the visualization has changed a bit. We can see that there are two branches that “come off” head after `749454e`. We can work independently, and could create merge requests for each branch separately. We are still on the second set's branch, let's add a second file to it:

```
touch second_set_file2
git add second_set_file2
git commit -m "added to second set"
[second_set_branch 04d6967] added to second set
1 file changed, 1 insertion(+)
create mode 100644 second_set_file2
git log --oneline --abbrev-commit --all --graph --decorate --color
* 04d6967 (HEAD -> second_set_branch) added to second set
* 9dc0a23 first edit in second set
| * d71ade9 (first_set_branch) first set edit
|/
* 749454e (master) add master2
* d5eb1e8 Adding readme
```

And just for fun swap back to the first branch and add a second file there:

```
git checkout first_set_branch
Switched to branch 'first_set_branch'
git log --oneline --abbrev-commit --all --graph --decorate --color
* 04d6967 (second_set_branch) added to second set
* 9dc0a23 first edit in second set
| * d71ade9 (HEAD -> first_set_branch) first set edit
|/
* 749454e (master) add master2
* d5ebl8 Adding readme
touch first_set_file2
git add first_set_file2
git commit -m "added second file to first set"
[first_set_branch b74cb32] added second file to first set
1 file changed, 1 insertion(+)
create mode 100644 first_set_file2
git add first_set_file2
git log --oneline --abbrev-commit --all --graph --decorate --color
* b74cb32 (HEAD -> first_set_branch) added second file to first set
* d71ade9 first set edit
| * 04d6967 (second_set_branch) added to second set
| * 9dc0a23 first edit in second set
|/
* 749454e (master) add master2
* d5ebl8 Adding readme
```

Now the branching in this in-terminal visualization is a bit clearer.

Ok, so this is the right way to do it: with each file/commit on the correct branch (and thus separate PRs).

Recover via splitting a PR

Pretend we had never made the two additional branches but made those commits all on the master branch. Then if a merge request opened after the first set of files `51f2622` was never accepted, our second set would have just piled on top and been added to the merge request.

```
git log --oneline --abbrev-commit --all --graph --decorate --color
* b5d4aff (HEAD -> master) added to second set
* dfa98b0 first edit in second set
```

```
* 43bc368 added second file to first set
* 51f2622 first set edit
* f945961 add master2
* 367e25a Adding readme
```

The problem here is that the top two commits (reading downward) are on `master` and should be on `second_set_branch` and the third and fourth commits are on `master` and should be on `first_set_branch`.

So our challenge is to make this linear setup look like our branched setup above. There are a few routes we can take.

First we're going to use the ability to create a new branch from an earlier point in history. By default `git checkout -b newbranch` will branch from the current HEAD, but we can tell it to branch earlier. So first we're going to create our two branches as though we'd done it before we started working, which is when HEAD was at `f945961`.

```
git checkout -b first_set_branch f945961
Switched to a new branch 'first_set_branch'
git log --oneline --abbrev-commit --all --graph --decorate --color
* b5d4aff (master) added to second set
* dfa98b0 first edit in second set
* 43bc368 added second file to first set
* 51f2622 first set edit
* f945961 (HEAD -> first_set_branch) add master2
* 367e25a Adding readme
```

So now we can see `(HEAD -> first_set_branch)` at `f945961` as the branching point for `first_set_branch`. However the commits we need on that branch are not on it, we need to bring them over from `master`.

We can do that using `git cherry-pick` pointing to each of the two commits we want to bring over. Below we can see (moving up the left) that we have a `first_set_branch` which has both of our needed commits. Note that this doesn't 'move' them from the `master` branch, but creates new commits with the same content.

```
git cherry-pick 51f2622
[first_set_branch ef2a3ee] first set edit
Date: Tue Apr 10 17:42:47 2018 -0500
1 file changed, 1 insertion(+)
create mode 100644 first_set_file
git log --oneline --abbrev-commit --all --graph --decorate --color
* ef2a3ee (HEAD -> first_set_branch) first set edit
| * b5d4aff (master) added to second set
| * dfa98b0 first edit in second set
| * 43bc368 added second file to first set
```

```

| * 51f2622 first set edit
|/
* f945961 add master2
* 367e25a Adding readme
git cherry-pick 43bc368
[first_set_branch 9f2b69a] added second file to first set
Date: Tue Apr 10 17:43:07 2018 -0500
1 file changed, 1 insertion(+)
create mode 100644 first_set_file2
git log --oneline --abbrev-commit --all --graph --decorate --color
* 9f2b69a (HEAD -> first_set_branch) added second file to first set
* ef2a3ee first set edit
| * b5d4aff (master) added to second set
| * dfa98b0 first edit in second set
| * 43bc368 added second file to first set
| * 51f2622 first set edit
|/
* f945961 add master2
* 367e25a Adding readme

```

So that's good, because now we have the right commits on `first_set_branch`. (We also have them on `master`, but that's not a problem as we'd be making our merge request from `first_set_branch` not from `master`.)

We can do the same thing to create our separate `second_set_branch`, starting by pointing back at the same branching point:

```

git checkout -b second_set_branch f945961
Switched to a new branch 'second_set_branch'
git log --oneline --abbrev-commit --all --graph --decorate --color
* 9f2b69a (first_set_branch) added second file to first set
* ef2a3ee first set edit
| * b5d4aff (master) added to second set
| * dfa98b0 first edit in second set
| * 43bc368 added second file to first set
| * 51f2622 first set edit
|/
* f945961 (HEAD -> second_set_branch) add master2
* 367e25a Adding readme
git cherry-pick dfa98b0

```

```
[second_set_branch 0d8e90d] first edit in second set
Date: Tue Apr 10 17:43:35 2018 -0500
1 file changed, 1 insertion(+)
create mode 100644 second_set_file
```

```
git cherry-pick b5d4aff
```

```
[second_set_branch 0841579] added to second set
Date: Tue Apr 10 17:43:57 2018 -0500
1 file changed, 1 insertion(+)
create mode 100644 second_set_file2
```

```
git log --oneline --abbrev-commit --all --graph --decorate --color
```

```
* 0841579 (HEAD -> second_set_branch) added to second set
* 0d8e90d first edit in second set
| * 9f2b69a (first_set_branch) added second file to first set
| * ef2a3ee first set edit
|/
| * b5d4aff (master) added to second set
| * dfa98b0 first edit in second set
| * 43bc368 added second file to first set
| * 51f2622 first set edit
|/
* f945961 add master2
* 367e25a Adding readme
```

So now we show three branches coming off at f945961: master, first_set_branch, and second_set_branch. Each of first_set_branch and second_set_branch have just the commits and files that they should:

```
git branch
```

```
first_set_branch
master
* second_set_branch
```

```
ls
```

```
README          second_set_file  second_set_file2
```

```
git checkout first_set_branch
```

```
Switched to branch 'first_set_branch'
```

```
ls
```

```
README          first_set_file  first_set_file2
```