Stat 610 Lab 5: Git

October 17, 2019

Installing

First step is to install git, which you can do following the instructions at https://git-scm.com/book/en/v2/Getting-Started-Installing-Git.

Before you do that, you can check to see if it is already installed using `git -version` in the terminal.

Set up a git repository

Once you've installed git, you can set up a new repository called `test` using `git init test`.

Run the following commands:

```
cd test
ls -a
```

The first moves you to the `test` directory, and the second should show you that there is a folder called `.git`.

The objects are stored in the repository in `.git/objects`, and we can see whether we have any using:

```
find .git/objects -type f
```

Since the repository is empty, there shouldn't be any output from this command.

Create a new file

At this point, we can check the status using:

```
git status
```

which should give output

```
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

We can make a file to commit as follows:

```
echo "test file" > test.txt
```

Now if we run `git status`, we should get

```
On branch master

No commits yet

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

test.txt

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

which tells us that the file `test.txt` is in our working directory but not in the staging area. If we tried to commit now, nothing would happen because our staging area doesn't have anything in it.

To add the file `test.txt` to the staging area, we can use

```
git add test.txt
```

If we run `git status` now, we should get

```
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

new file:   test.txt
```

which tells us that `test.txt` is in the staging area, and if we would like to create a commit with that file we can.

To create the commit, use `git commit -m 'initial commit'`.

The `-m` flag is for message, and what comes after it is the *commit message*, which describes what the commit does. So when we ran the commit command, we added a snapshot of the files in the staging area to the repository, with the commit message "initial commit".

Now if you run `git status`, you should see something like

```
commit 7e5905faf3f704bb0cdafb482765970494ee0c75 (HEAD -> master)
Author: Julia Fukuyama <julia.fukuyama@gmail.com>
Date:   Thu Oct 17 09:50:11 2019 -0400
```

```
    initial commit
```

but with your information and time instead of mine.

Questions:

- What is `7e5905faf3f704bb0cdafb482765970494ee0c75`?
- What does `(HEAD -> master)` refer to?

Making branches

Let's try making branches. Remember that a branch is simply a pointer to a commit. We can make a branch that points to the the most recent commit by using the `git branch` command.

Typing `git branch` with no arguments will list the current branches, and `git branch` with a branch name will create a new branch.

Try:

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

The first command should list the branches (you should start off with just `master`), the second should create a new branch called `new-branch`, and the third command should list the branches again. The output from the third command should show you that you now have both `master` and `new-branch`:

```
$ git branch
* master
  new-branch
```

and the $*$ indicates that HEAD is pointing to the `master` branch.

If we want to switch so that HEAD points to `new-branch`, we can use `git checkout`.

Try:

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

The output from the second command should be

```
  master
* new-branch
```

indicating that HEAD now points to `new-branch`.

The next thing to do is to make some changes and commit them. Let's make a new file called `branch-test-file.txt`, add it to the staging area, and then commit the change.

```
echo "this is a file for testing out branches" > branch-test-file.txt
git add branch-test-file.txt
```

```
git commit -m "a commit on the new branch"
```

To see what's happened, let's look at the log. You can do this by typing `git log`. The output should look something like this:

```
commit a035427f43aec773cd918920204e5bc35ffa28ae (HEAD -> new-branch)
Author: Julia Fukuyama <julia.fukuyama@gmail.com>
Date:   Thu Oct 17 09:58:32 2019 -0400

    a commit on the new branch

commit 7e5905faf3f704bb0cdafb482765970494ee0c75 (master)
Author: Julia Fukuyama <julia.fukuyama@gmail.com>
Date:   Thu Oct 17 09:50:11 2019 -0400

    initial commit
```

This gives us some useful information. We see that

- There have been two commits.
- Remember that a branch is just a pointer to a commit? We see that the `master` branch poins to the commit `7e5905faf3f704bb0cdafb482765970494ee0c75`, and the `new-branch` branch points to commit `a035427f43aec773cd918920204e5bc35ffa28ae`.
- Also remember that HEAD points to a branch, and advances the branch when we made the new commit. Before the commit, `new-branch`, `HEAD`, and `master` all pointed to `7e5905faf3f704bb0cdafb4827`. Because we made commit `a035427f43aec773cd918920204e5bc35ffa28ae` when `HEAD` was pointing to `new-branch`, the `new-branch` pointer advanced when we made commit `a035427f43aec773cd9189`. Since `HEAD` was not pointing to `master`, `master` continued to point to commit `7e5905faf3f704bb0cdafb48276`.

Changing branches

Suppose we decide we don't like what we did when we made `new-branch`, and we'd like to go back to the state of the directory before. We can use `git checkout` to go back to a different commit.

Try running

```
ls
git checkout master
git branch
ls
```

- `ls` lists the files in the working directory, so you should see `branch-test-file.txt` and `test.txt`, the two files we're created.
- `git branch` should tell you that you are now on `master`.
- The second `ls` lists the contents of the working again, but now `branch-test-file.txt` should be gone. This is because we went back to commit `7e5905faf3f704bb0cdafb482765970494ee0c75`,

the commit `master` was pointing to, and at that point we had not made the `branch-test-file.txt`

Let's make and commit another file.

```
echo "master branch testing file" > master-branch-test.txt
git add master-branch-test.txt
git commit -m "a commit on the master branch"
```

Now if we run `git log`, we can see the history again, but the default is to only show the ancestors of the branch HEAD is pointing to:

```
commit e8321a77a213cbebf61bac6bfd6cd4944bdfdf2c (HEAD -> master)
Author: Julia Fukuyama <julia.fukuyama@gmail.com>
Date:   Thu Oct 17 10:08:17 2019 -0400

    a commit on the master branch

commit 7e5905faf3f704bb0cdafb482765970494ee0c75
Author: Julia Fukuyama <julia.fukuyama@gmail.com>
Date:   Thu Oct 17 09:50:11 2019 -0400

    initial commit
```

If we want to see the commits on all the branches, we can run `git log -branch`, which will give

```
commit e8321a77a213cbebf61bac6bfd6cd4944bdfdf2c (HEAD -> master)
Author: Julia Fukuyama <julia.fukuyama@gmail.com>
Date:   Thu Oct 17 10:08:17 2019 -0400

    a commit on the master branch

commit a035427f43aec773cd918920204e5bc35ffa28ae (new-branch)
Author: Julia Fukuyama <julia.fukuyama@gmail.com>
Date:   Thu Oct 17 09:58:32 2019 -0400

    a commit on the new branch

commit 7e5905faf3f704bb0cdafb482765970494ee0c75
Author: Julia Fukuyama <julia.fukuyama@gmail.com>
Date:   Thu Oct 17 09:50:11 2019 -0400

    initial commit
```

Switching between commits or branches

We can switch between commits using `git checkout`, followed by either a branch name or a commit name. Normally you will checkout using a branch name, not a commit name, but you can try it out to see that it's possible. (NB: if you check out using a commit name that doesn't correspond to a branch, you'll be in a detached HEAD state because HEAD is suppose to point

to a branch. You don't really want to commit things in this state, so checking out a commit by name is more for looking around, not for modifying files.)

Try

```
git checkout XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
git checkout master
git checkout new-branch
```

where the 40 X's are a commit number that you saw in `git log`.

Merging

Suppose now we've decided that we like the work we've done on both `master` and `new-branch`. We can use `merge` to create a new commit that has the work done on both.

```
git checkout master
ls
git merge new-branch -m "merge commit"
ls
```

After the first `ls` you should have seen only `master-branch-test.txt` and `test.txt`. After the merge, the second `ls` should show you that we now also have `branch-test-file.txt`.

Now if we look at the log with `git log`, we can see all of the commits so far:

```
commit 7bcff11a7856a79c81e960d00412b70c520952fd (HEAD -> master)
Merge: e8321a7 a035427
Author: Julia Fukuyama <julia.fukuyama@gmail.com>
Date:   Thu Oct 17 10:19:33 2019 -0400

    merge commit

commit e8321a77a213cbebf61bac6bfd6cd4944bdfdf2c
Author: Julia Fukuyama <julia.fukuyama@gmail.com>
Date:   Thu Oct 17 10:08:17 2019 -0400

    a commit on the master branch

commit a035427f43aec773cd918920204e5bc35ffa28ae (new-branch)
Author: Julia Fukuyama <julia.fukuyama@gmail.com>
Date:   Thu Oct 17 09:58:32 2019 -0400

    a commit on the new branch

commit 7e5905faf3f704bb0cdafb482765970494ee0c75
Author: Julia Fukuyama <julia.fukuyama@gmail.com>
Date:   Thu Oct 17 09:50:11 2019 -0400
```

```
initial commit
```

Notice which commits `master` and `new-branch` point to, and where `HEAD` points. Can you explain why?

Wrapping up

This is a very small introduction to git. There are many more things you can do, and the Pro Git book has a good overview. However, to understand how it all works, the most important things to understand and know how to do are:

- How to create a commit
- How to create and move between branches (and remembering that branches are just pointers to a commit)
- Understand that `HEAD` points to a certain branch, and that when you make a new commit, the branch that `HEAD` points to will be advanced to that new commit.