How To Git

Matt McCarthy

CNU FOSS Day
12 March, 2016

1 Version Control

Simply put, version control systems (or more succinctly VCSs) track and label changes, known as revisions, to files or documents. These properties provide a plethora of capabilities such as comparing and merging revisions as well as restoring previous revisions. In turn, this enables many people to edit the same files and combine or merge their changes together as well as synchronize their versions.

In this tutorial we will use Git as our version control system. Git is a free and open source VCS that is used in many FOSS projects, most notably itself and the Linux kernel. If you do not have Git installed, you can install it from your Linux distribution's package manager or from `https://git-scm.com/downloads`.

2 Git 101

Before starting, make sure your terminal is open.

To start, we need to configure Git. To do so, we run the following two commands.
Terminal:

```
git config --global user.name "Your Name Here"
git config --global user.email "your.email@host.domain"
```

These commands tell our Git installation who we are and how to contact us so that other users know who is responsible for each commit.

2.1 Local Git

Our first task is to make a repository on our local machine. Next we want a directory in which we will store our Git repositories. For the sake of simplicity, let's just make a new folder in the home directory called `git` and then enter that directory, that is run the following.
Terminal:

```
mkdir ~/git
cd ~/git
```

Next we want to actually make the repository. To do so, we want to make a folder for the repository (`mkdir my-git-repo`) and then enter that folder (`cd my-git-repo`). We will now turn this folder into a Git repository by running
Terminal:

```
git init
```

which creates the directories and files necessary to make a folder a Git repository.

We're now going to start making changes, tracking them, and committing them. Let's begin by creating a file and telling Git to track it. Run
Terminal:

```
touch file.txt
```

this will create a file called `file.txt`. If we run
Terminal:

```
git status
```

it will list `file.txt` as an untracked file. We now need to run
Terminal:

```
git add .
```

which tracks all untracked files and tracks any changes you made. If we run `git status` again, we will see that `new file: file.txt` is in the list of changes to be committed. Lastly, to commit our changes, we run
Terminal:

```
git commit -m "Added file.txt"
```

which logs our changes and gives us a point to which we can revert. If we run `git status` once more, it will report that there is nothing to commit and that the working directory is clean. The `git add .` and `git commit -m "message here"` commands define the workflow on a single machine, that is these commands track and log each change you make to your project.

Next we'll make some changes to the project and then reset them. For now, run the following command.
Terminal:

```
echo "Subversion is the best version control software" > file.txt
```

Then track the change using `git add .` but don't commit the change just yet. Run `git status` and ensure that the change is tracked. You see, the statement we piped into `file.txt` is a lie and so we need to undo the change even though we just tracked it. To do so, we run
Terminal:

```
git reset --hard
```

Now run `git status` to make sure the change is gone. More generally, we can run `git reset --hard commit-id` to reset to any given commit. Since the old statement is gone, we will go ahead and pipe the correct statement into `file.txt`.
Terminal:

```
echo "git is the best version control software" > file.txt
```

We will now track and commit our changes in one command by running
Terminal:

```
git commit -am "Did a thing"
```

which adds all changes in *tracked* files and then commits them in one fell swoop. Furthermore, it does not add any untracked files, so any new files would be skipped by this command.

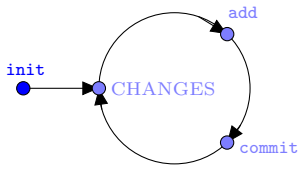

Figure 1: Workflow for Git on a single machine

2.2 Using GitHub

Now that we know how to deal with changes in a project locally, we're going to bring GitHub into the picture. To begin, if you do not have a GitHub account create one at `github.com`. Continuing with our toy project, create a repository on your GitHub account by clicking the 'New repository' button.

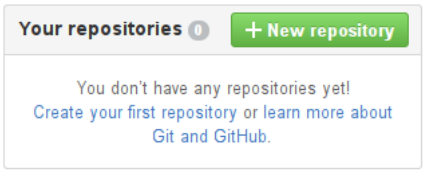

Figure 2: The 'New repository' button

For the name, call it `my-git-repo` and then create the repository. This should bring you to webpage of your new repository. The default options are fine for this project. From here copy the URL shown on your screen.

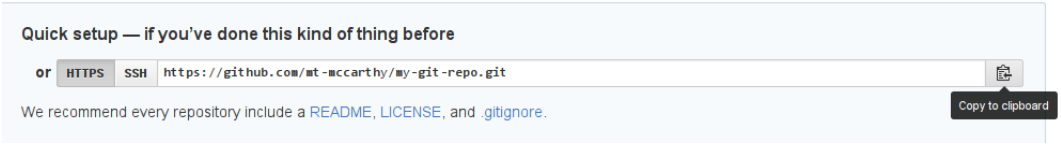

Figure 3: Click here

Our next step is to add the GitHub repository as a remote server. To do so run
Terminal:

```
git remote add origin copied-url
```

which adds a place from which Git can fetch code for this repository. Now that we have added the server, run
Terminal:

```
git push origin master
```

to push your changes to GitHub. The `git push` command requires both a place to which it can push as well as a *branch* to which we can push. The default branch of any Git repository is called the master branch. If you want to learn more about branching, check out the advanced Git section of this paper after you finish the demo.

Now that your project is on GitHub run `cd ..` to go into the parent directory and then delete your project using
Terminal:

```
rm -rf my-git-repo
```

We will now *clone* the repository from the GitHub URL. Now run
Terminal:

```
git clone copied-url
```

which creates a directory for your project and then copies all of the project's files into that directory. To check run `cd my-git-repo` and then
Terminal:

```
cat file.txt
```

and it should output the contents of the file.

When we use a `git` server, the workflow will change a little bit. Instead of just running `git add` and `git commit`, we now have to run `git push` in order to push our commits to the server. Furthermore, instead of running `git init`, we can just create an empty repository on the server and then use `git clone` to set it up on our machine.

Lastly, GitHub has a Git cheat sheet at

```
https://training.github.com/kit/downloads/github-git-cheat-sheet.pdf
```

if you wish to have a quick reference document.

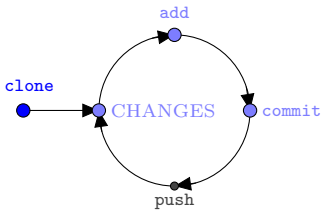

Figure 4: Workflow for Git with a Git server

3 Git Demonstration

For our demonstration, we will make a few classes that could be used for auto insurance.

3.1 Forking the Demo

In your web browser, navigate to `https://github.com/matt-mccarthy/cnu-foss-day-demo`. Once the page has loaded look for a button called 'Fork' and click it in order to fork the repository.

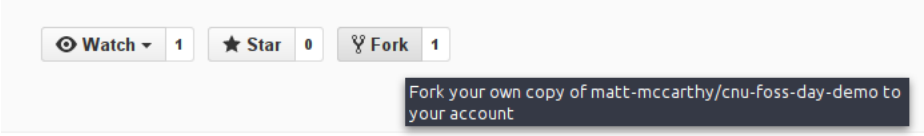

Figure 5: The Fork button

After that you should be on the page for your forked repository. From here, clone your repository and then in your terminal use `cd` to enter the repository's directory. Once you have `cd`'d into the repository, run the following command.

Terminal:

```
git remote add upstream https://github.com/matt-mccarthy/cnu-foss-day-demo.git
```

This will allow you to pull changes from the original repository into your own. At this point we say the original repository is *upstream* and your fork of it is *downstream*.

3.2 Fixing an Issue

If you navigate to `https://github.com/matt-mccarthy/cnu-foss-day-demo/issues` you will see a plethora of issues with this project. You have been assigned an issue number to fix.

Once you understand which issue you need to fix, open the file that to which the issue applies and add the code necessary to fix the issue. Once you fix your issue, you can move on to the next section.

3.3 Pull Changes from Upstream

Now that you have fixed your issue, go ahead and push your changes to your repository. After you have done that, run the command
Terminal:

```
git pull upstream master
```

pulls any changes from the original repository (called `upstream`) into your local repository and then *merges* them into your code (if you want to learn more about merging and branching check out the Advanced Git section of this paper). Sometimes, the merge operation will require manual intervention in order to succeed, but this should not be the case for this demo. Once you successfully merge, run `git status` to ensure you do not need an extra commit, and then push your changes to your repository. If you do need an extra commit, commit your changes and then push.

3.4 Pull Requests

Now that your fork is up to date, we will start talking about creating a pull request. Navigate to the GitHub page for your repository and click the "New pull request" button.

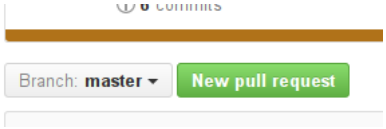

Figure 6: Pull request button

Quickly inspect the options.

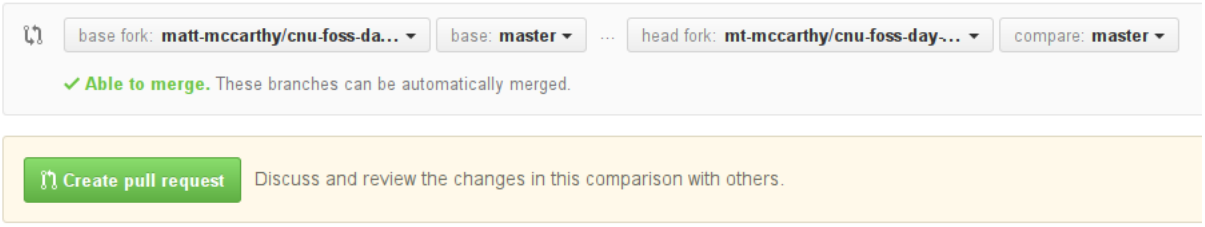

Figure 7: Proper set up

The "base fork" field should be set to `matt-mccarthy/cnu-foss-day-demo` with "base" `master`. The "base fork" field tells GitHub where we want to merge our changes, and the "base" field specifies a branch to merge. Furthermore, if you look at the "head fork" field you should see your repository listed there and again with the "compare" field set to `master`. For our purposes, GitHub did all of the work and so we can go ahead and click the "Create pull request" button. And that's the last step for you to do. All that is left is for your request to be approved or denied.

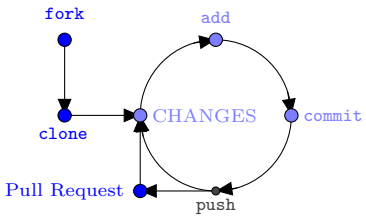

Figure 8: Workflow for a typical FOSS project using GitHub

4 Advanced Git

In this section, we will cover branches, tags, and the `.gitignore` file.

4.1 Branches

So far in this demo, we have done all of our work in a branch called master, however this is typically not the best way to do things. The more usual workflow involves creating either a branch for a new feature or release. These branches can operate independent of all other branches, which makes it easier to tackle an issue or add a new feature since the base code does not change until you decide to bring those changes in.

To start, lets see how branches work. In the demo repository, run
Terminal:

```
git checkout testing
```

which switches the branch from master to the testing branch. For this excercise, implement the `toString()` function in `Car.java`.

After that's done, we need to merge the changes you did in master into testing. To do so, we run
Terminal:

```
git merge master
```

which merges all changes you made on master into the testing branch, a necessary step to enable what we call a *fast-forward* merge when we go back into master.

Once it merges, run `git status` to make sure you do not need an extra commit to finish the merge. Now run
Terminal:

```
git push origin testing
```

to push the latest commits in the testing branch to GitHub. Now that we have brought testing up to speed, use
Terminal:

```
git checkout master
```

to switch back to the master branch. Now we run
Terminal:

```
git merge testing
```

in order to merge our testing changes into master. Once we have everything merged into master, run
Terminal:

```
git push origin master
```

in order to push your changes to master.

Next, run

Terminal:

```
git checkout -b person-toString
```

which creates a new branch called `person-toString` and immediately switches to it. For this part create a method `public String toString()` for the person class that returns a string in the format `lastname, firstname`. Once you have it implemented commit and then run

Terminal:

```
git push origin person-toString
```

which will push the `person-toString` branch to GitHub. Next switch to the master branch by running

Terminal:

```
git checkout master
```

and then run

Terminal:

```
git merge person-toString
```

which will merge the `person-toString` branch into master. Now push your master branch to GitHub by running `git push origin master`.

4.2 Tags

Git also a tag feature that is typically used to denote releases. For example, if you were to run

Terminal:

```
git checkout v0.1beta2
```

your local repository would look like how it did when you first cloned it. This is because that tag marks the commit that represents the initial version of the repository. What you need to do now is run

Terminal:

```
git tag release1
```

in order to easily switch to the current state of the repository. Once you have made your tag, go ahead and run

Terminal:

```
git checkout v0.1beta2
```

which brings you back to the way your repository originally was. Run

Terminal:

```
git checkout release1
```

in order to get back to release1.

Once you are back at release1, run

Terminal:

```
git push origin release1
```

which pushes the tag to the Git server, since git push does not push tags unless you tell it to do so.

4.3 `.gitignore`

In the demo repository, you may have seen an odd file entitled `.gitignore`. This file tells Git which files to ignore. You can specify that certain file types should be ignored or you can actually spell out the files. But this raises the question, what if you want to add an ignored file? In this case we use the command
Terminal:

```
git add -f ignored_file_name_here
```

which forces Git to track the ignored file. If you want to generate a `.gitignore` for your repository, an easy way to do so is to use `www.gitignore.io`. This website takes the languages you are using and autogenerates a `.gitignore` that ignores all of the typical temporary files that your languages produce. For example, the `.gitignore` for Java ignores `.class` and `.jar` files as well as JVM crash logs.