GEEK LABS: Introduction to Git

Raphael Bronfman-Nadas

They say GitHub is taking over the world. They say it gains power every day. No fair. That was my plan. Well, I will have to start from scratch. Greetings fellow evil doers. I am Dr. Tig, and I will one day use my evil plan to take over the world... Probably. If GitHub is trying to take over the world too, I should check that my evil plan is secure, we don't want them finding my brilliant ideas. My plan is actually on GitHub, so we should probably act fast.

What is Git?

Git is something called version control software, and is a super useful thing. It lets you go back in time and see how files change over time. It also lets people work together, by everyone taking turns to add things, and then mushing them together. It's kind of like google docs, but a lot more manageable. For some help installing git, you can check here: http://web.cs.dal.ca/~raphael/GetGit.

What is Git good for?

Git is great for keeping track of changes over time. Most people use it for programs. Some people use it for text documents. Lots of people use it for group work.

Git is great for when something was working, but now it is not. Figuring out what changed helps figure out why it is not working anymore.

Git allows groups to all work on their own copy of a program, without instant changes from other people. Then when everything is working, the changes can be given to everyone else.

Basically, git keeps your work organized over time.

What is GitHub?

When they are not taking over the world, GitHub is a website that hosts a bunch of git projects. Lots of open source projects, or they host private projects as well. There is a student developer pack which comes with a bunch of stuff, and is free for students and can be found here: https://education.github.com/pack.

At Dal we have a git server too, it is located at https://git.cs.dal.ca.

Task 1: Finding my Evil Plan

Part 1: Cloning

My evil plan is located on GitHub, so we need to get a copy of it first. GitHub lets you see it on the website, but downloading a copy can be done through git.

A folder using git is called a git repository, and usually, but not always is linked to a git server. Git is perfectly usable without a server, and you can use it to just keep yourself organized.

It's located here: `https://github.com/DalCSS/GeekLabsGit.git`

So to get a copy, you have to **clone** it. On your terminal, use the git command.

All git commands are very similar in structure:

```
$ git <A command word> <Arguments to that word>
```

Things get slightly more complex if you really get to using git, but that is all you need to know. Also, the `$` at the beginning of a line is just showing this is a command for the command line. You do not need to type that.

So, our command word is **clone**, and to get a copy of the plan, we use

```
$ git clone https://github.com/DalCSS/GeekLabsGit.git
```

If everything is working, you should now have a new folder with the name GeekLabsGit.

This is a local copy of the project. Any changes you accidentally make, git may warn you about, but they won't affect anyone else, so don't worry about accidentally changing things.

To go into the folder on the command line, write:

```
$ cd GeekLabsGit
```

Also, my advice is to open a normal folder window on your computer to the same location, so you can see what's happening[1].

I have included a map for you, which should help you figure out where you are in git. Take a look at it if you get lost.

You should be able to see the first part of my evil plan! It's just inside plan.txt (HAhahahHha Muahahahahaha!!!! I am so EVIL!!!)

Part 2: History

Ok, so that folder starts at the most recent version, like dropbox or something. Unlike dropbox, from here we can explore the past.

Think of Git like a time machine for points in the past where files changed. Each point in the past is called a commit. Even now, the most recent version is stamped with a number representing the last commit. The commits are all given a large number letter thing (called a hash or commit number) that acts as its name. Every commit has a comment describing what it is doing.

Let's just take a look at the history first. Our command for that is **log**. And as long as we are in the folder of our git repository, any git command, (for example `git log`) will be talking about the

[1] on mac you can use the command "`$ open .`" on windows you will probably need to look somewhere around "`C:\user\<your name>\`"

folder we are in. So, why not try the **log** command and see that it gets us the history. We see those large commit number, the author, date and a comment explaining what was changed. (You may need to press `q` to get out of the log.)

I was wondering about what happened to the rest of my plan. Seems like the commit before here added part 2. Well, at least we have a comment letting us know about it. Now, we just need to ask git to be our time machine. We just give it a destination, and away we go. Part 2 will be there if we go back to the commit before the last one, so, **checkout** the version you want to see by using, well that command. Just put that long number after the checkout for where you want to go.

That should pull the folder into the past, and you should be able to look at the plan file to find the next part of my plan.

Part 3: Branches

One of my minions has been working on the plan with me. I don't let them actually change anything in the project directly. I like to see that every part is good before I let them submit their work. So, they uses a branch to do all of there work. You can take a look at all the branches with the **branch** command. Oh, and because my minion worked on a different computer, you will need to put `-a` after **branch**, to show all the branches. You may have seen that we are on the master branch.[2] The master branch is the one we start on, and it's the main time line for all the history. Branches are useful because it is a great way to share work without changing the master time line until it is ready. Branches are like alternate time lines. They at one point were the same at the master branch, but then someone created the branch, and went off and did what they wanted with it.

The minion branch here is the basic use of a branch. My minion went off, created the branch off of the main time line. Any changes commited on the minion branch had no effect on the master branch, and the master branch has no effect on the minion.

What's kind of cool about branches, is that they can be brought back together. Like some kind of weird tree, where there is a trunk, which splits off into many branches, but all the branches come together again. While here, the minion branch is not joined back up to the master timeline.

You can **checkout** a branch the same way we went into the past before, but instead of using the large number, use just the last part of the branch name, like minion and master. Like I said, that first part just means that it also is on the Internet and you don't need to put that.

That's I think all you need to know about branches for now. Just go check out that branch and you should find the next part of the plan. The names of the branches are a bit more reasonable then the long commit numbers to find exactly where you want to go.

[2] remote/origin is just git's way of saying that this branch is from the Internet version of the project and not just your version.

Part 4: Diff

Ok... looking at the most recent comment in the **log** for the minion branch, Part 4 was hidden in "cryptic.txt".

I don't have all day to figure out what was changed. Just use git to tell us the difference between two commits.

Just use the **diff** command[3]. We need to give the command two things to compare, and then it will show `+` and `-` symbols to show what what was added and removed between the first commit, and the second. So, try and see what changed between the last two commits. If all goes well, then we should see part 4 of the evil plan.

We are nearing the end of the plan. One more part to go.

(Note to evil self: There is actually a better way of doing this specific task, by asking git to **show** a commit, then it will show the changes made in specifically that commit, but I think that is good to know **diff**, as it can be used in far more ways.)

Part 5: Branch diff

More changes were made to "`cryptic.txt`" apparently. We should compare it to the master branch. The **diff** command has a lot of uses and comparing branches is as easy as comparing commits. My advice here is try what seems right and Experiment. Although git throws errors, git will not break very often, and when it dose, nothing will be lost[4]. And with that, I think that's my entire plan found.

Task 2: Making Your Own Plan

Part 1: Creating a Repository

I know. My plan is not great. You could probably make a better plan. Hmm, why don't you set up your own evil plan on Git.

Just get out of my evil plan (`cd ..`). Just tell git that you want to create a git project here in a folder by using the command

```
$ git init <folderName>
```

where you want git to create a folder with the name *folderName*.

You should also tell git who you are, if you have not done that yet (if not, git may ask you to fill that information in, or fill it in itself, but it's probably a good idea to do that now.).

Git wants two pieces of info about you: your name, and your email.

To set your name run the command, quotation marks are important.

```
$ git config -global user.name "Your name here"
```

and to configure your email (Does not have to be a real email, git asks for this, but it won't use it.)

[3] linux has a diff command for 2 files, but git **diff** is for comparing different versions of the same file across commits

[4] There is a way to lose things, but it will warn you a lot, and you have to force it, so just for a second ignore that.

`$ git config -global user.email "email@somewebsite.com"`

And before anyone gets confused, git may by default uses vi or vim as a text editor. If you have never heard of it, now is probably not the time to start using it. My advice is to change your text editor right now. I have some basic instructions, but if you figure out something better, you can use that. You will find further down the instructions if you want to leave it default, you are free to do so.

If you are on mac or linux:

`$ git config -global core.editor "nano"`

On Linux or mac, you can use `"nano"`. That's all you need to put for the command. Nano is fairly basic, it has some controls listed at the bottom. To quit nano, press Control+x, then say y to saving then enter to accept the file name. It is probably easier than using vi for the first time, so it might be worth doing.

On Windows, I would recommend using `notepad++`, however if you have something like sublime or atom, they should work too, I however I will help setup git with `notepad++` because it is easy to set up, and it is recommended for windows on the git download site. `Notepad++` can be downloaded from here: https://notepad-plus-plus.org/download/. During the instillation, you picked, or just clicked next, a destination folder. If you picked something then you will have to put that in your command, but by default, the command should be (in one line):

```
$ git config -global core.editor "'C:/Program Files (x86)/Notepad++/notepad++.exe'
-multiInst -nosession"
```

If you don't set the editor up, vim will probably be used. vim is the new version of vi so they are basically the same thing. If you find yourself in vim, here is how to use the basics: Press '`i`' to begin typing, it should say insert somewhere on the screen when you do. Try to avoid pressing any keys before you do this. Now that it says insert, and type your message. When you are done, press escape, then type in the three characters '`:wq`' and press enter. This will save and quit the file.

Part 2: Commit

So, go wild, and create your very own first draft of your evil plan. Name it whatever you want, just make sure you save it in that folder.

Now we can quickly ask git the **status** of our project. We should see, we are setting up our Initial commit, on branch master, and we have an untracked file, the file we just created.

So, we can **add** the file to let git know we care about changes to it. Ignore anything about line endings right now. How the **status** has changed?

Great you can **commit** the changes now, after we enter a message (This is where your editor kicks in.) Your comments can be on more than one line if you want. In the text editor that should have popped up, is a big block of info on lines all starting with '`#`' signs. This means these lines will

be ignored, and your comment will be any line without '`#`' signs. Just write a comment of what changed, like "initial commit" or "added evil plan", and when you are done, save and close the text editor. When you close the text editor, git should say it created the commit, and you are good to go. Congratulations.
What is often faster, if you only have one line of comments, is to use '`-m`' followed by your comment in quotation marks after the command (example: '`git commit -m"My comment"`).
Feel free to make some more commits. It's your plan. Do what you want with it. But every time you create another commit, you need to **add** all the files you changed, or git will think you did not want those changes in that commit. Git will also not create an empty commit, so you should be warned if you forgot to do that.

Part 3: Branches

Now, lets create a branch. Why you should create a branch is up to you.

Tangent on when to use branches

When should you use branches? One master branch is enough for most things. I find that I only really use branches when I found something interesting I wanted to save, but it was not something I was trying to do. I don't know if that happens to other people, but that's how I use branches.
It is probably a good idea for experimental ideas, and for features that can be worked at independently. Like the rest of git, use them to stay organised.

Ok back to reality

To create a branch we use the **branch** command. We used this before to list all of them, but if we put a name after it, it creates a branch named that. You can switch over to that branch after it is created, like we did before.
The branch will start from where we used the branch command, so in this case it will begin with the plan we committed in our master branch.
Any commit we make on this branch will not affect the master branch, and vice versa. So create a commit here, so that it is different than the copy in master.
While you are here, change some lines around here. Add some lines of your plan, get rid of some. Make a few commits, and see how that changes things.

Part 4: Setup for things to come

By this point we will need some changes in the branch.
What we are going to do is try and have a conflict. Combining branches or working in a group can lead to multiple changes on the same file, or the same line. When that happens, git decides that it's better to ask you what to do then do something automatically.

So we can show what happens when things go wrong, and how they can be fixed. So make sure you have some changes in the minion branch, then switch back to the master.

So, all those changes in the branch are gone on the master branch, Why? Because they did not happen when the branch split off. So changing one branch wont change the others. If you wanted to, you could create more branches. Then branches that start in branches, or go back in time, then branch off of some point in the middle... But what's good to know, is you don't have to go crazy with branches. You can ignore them during your own projects most of the time.

Make some changes to the same files in the master branch. **Make sure the changes are different**, so we can simulate a conflict.

Right now we are going to try and show what happens when you try and take the work that has been done in one branch and move it to another, merging the changes into one. This happens when you merge branches, or when two people want to change the same file at the same time. Git actually does fairly well with that, but it looks scary the first time it happens. So we are going to simulate that by changing the same line you changed in your minion branch just changed.

Once you have some changes on the master branch that are different, some on the same lines, some on different, then you should be good.

Part 5: Merge and merge conflicts

Ok, everything committed? Just do **status** to make sure. Good?

Alright. Now we want to tell git to bring in the changes we made in the minion branch. Easy. We just stay in our master branch, then tell git to **merge** the minion branch. You should get an error like:

"`CONFLICT (content): Merge conflict`". If that did not happen, then you successfully merged the branches. I would go back and try and change things around to see if you could cause a merge conflict if you want to experience one safely now.

Because you got the error, git is telling you that it wants to be careful. Instead of overwriting someones changes, git asks you what to do, and when you are done, it wants you to make a **commit**.

Take a look at the plan file (Or whatever file has merge conflicts.) Depending on what you did, you probably see something slightly different, but you should see a bunch of `<<<<` with `HEAD` (to say the end of the current timeline) the changes made to the master branch, then a bunch of `=====`, followed by the changes from the minion branch, indicated by the `>>>>>> minion` or what you named the branch.

Merging means editing the file, getting rid of the part's you don't like, and mixing everything in that block together. How do you fix it? You just edit the file. Make it look the way you want, make sure to **add** it to your commit, to tell git it is taken care of the conflict, git will want you to take care of all files at the same time, then commit all the files you merged.

Optional share

Well, your evil plan looks to be about done now. All you need to do to share it, is to put it on the internet, then using any other computer that has git, you can download it, and get an entire copy of the plan, and it will work exactly like it dose here.

Most common is to use github. GitHub accounts are free, and easy to set up. Once you have logged in, in the top right corner is a plus. Just click on that, and press new repository. Give it a name, and make sure not to initialize with a README if you want to link an existing git repository to your project. If you are not planing on copying an existing project to github, then creating readme files or anything else is just fine. You should get a useful page of quick set up help if you don't have a README. This is good, we need to know where our remote git repository is. That line starts with https and ends with .git is what we need. Just copy it and we will use it in a second.

We are going to let our local folder know where our internet box is. We do that using the **remote** command. Remote means not local, and for git, it means the main place where commits get stored. Our command to do this is:

```
$ git remote add origin <That line you just copied>
```

Just replace the last part with your value. This will tell git where your remote is, then we just need to tell git which of your branches to copy. You should have 2, so you can do both if you want:

```
$ git push -u origin master
$ git push -u origin minion
```

Then everything is pushed online. Refresh the page, and you should be able to see it. If your repository is public, then anyone can see it if they go to that URL.

From here, if you were to go onto another computer, you could **clone** just like we did my repository, your repository with that same line you copied. When you have an online copy, everything still works just about the same. Only now, you can do two more important things: **push** and **pull**. The command **push**, like we used before, means taking what you have and moving it to the internet. If you put no arguments after that, so you just have your command:

```
$ git push
```

Then all the commits you have made since you last pushed, will get moved to the internet. The opposite of **push** is **pull**. The command **pull** means take the changes from the internet and move them to my machine. You may have to merge changes after you pull, if someone else changed the internet copy. But with those two commands, you can work together with other people using git, and staying organised.

```
$ git pull
```

It is common to **pull** when you start to work, and when you are done, **push** what you have changed to the internet, ready to be seen by everyone else.

The end

Well, congratulations on your plan. I look forward to you taking over the world. Enjoy Git, and I hope you consider using it in the future.

What you learned is 99% of all the commands you will ever use in git. There are shortcuts, and cool other stuff you can do with git, but every basic thing you can do with git, you have now done. Plus, you can use those fancy non terminal programs that display everything to you cleanly now that you know how git really works. Most IDE will have built-in tools for working with git too. They cannot do everything, and even those will tell you to sometimes go into the command line to fix things, if so, it's good to know what's going on.

Here is a short list of some useful extras:

`.gitignore` is a file to list files not to add to commits, like builds and meta files.

`HEAD` is a shortcut for where you are now. That's why git was saying you were in detached head mode, when you moved the head away from the top of the branck.

`^` (the character '^') added to the end of a commit, number, or HEAD, or branch, means the commit before. So to go back one commit, I can use the command: `git checkout HEAD^`

show is a command that gives informaiton about a thing. Try it on a bunch of stuff. It will give the changes in the current commit if you don't give it anything, so it is much shorter than the diff from before.

diff can take one value or even zero, instead of two. I will explain below what happens if you put in less values.

And there is a lot more, but what is the best to know is there is a `man git` (You don't need to start with git for this command), and `git help` command. You can even ask about a specific command after the help. So if you want to learn more, there is a lot to find.

A guide to all commands used

git clone <***URL***> Gets a local copy of a repository from the web.

git log Shows a list of all commits to get to where you are now.

git checkout

- **git checkout** <***commit number***> sets your folder to how it was after the numbered commit.
- **git checkout** <***branch name***> sets your folder to the branch that's name was requested.
- **git checkout HEAD^** sets your folder to how it was before the last commit.

git branch

- **git branch** gets a list of all branches downloaded to your computer.
- **git branch -a** gets a list of all branches downloaded to your computer, and on the internet.
- **git branch** <***branch name***> creates a branch with the given name.

git diff

- **git diff** shows the difference between the folder now, and how it was after the last commit.
- **git diff** <***commit number/name of branch***> shows the difference between the folder now, and how it was after the given commit. If a branch was given, it compares to the last commit with the given branch.
- **git diff** <***commit number/name of branch 1***> <***commit number/name of branch 2***> compares what changed to take the folder from the first commit, or branch end, to the second one.

git init

- **git init** creates a git repository in the current folder.
- **git init** <***folder name***> creates a folder, and creates a git repository in that folder.

git config <***parameter name***> "***Value to set***" allows you to set git settings. The user can be set with the parameters "`--global user.name`" and "`--global user.email`", and "`--global core.editor`" is the parameter that sets the text editor to use.

git add <***Files***> sets a file to be committed in the next commit. This can actually be a list of files. Command line wildcards can be used to add lots of files easily. This will not add files that match a line in the `.gitignore` file.

git status shows what is ready to be committed, and what will not be. If a file is changed but not added to the next commit, git will let you know in status.

git commit

- **git commit** opens up a text editor, asks for a commit comment, then creates a commit and adds it at the end of the active branch.
- **git commit** `-m "`***Commit comment***`"` creates a commit and adds it to the end of the active branch with the comment given.
- **git commit** `-amend` takes the last commit, and whatever is staged for the next commit, and asks for a comment. This replaces the last commit. Basically, you are adding to and changing the last commit.

git merge <***branch name***> merges the changes from the given branch into the active branch. This may lead to merge conflicts, which can be fixed with a commit.

git remote add origin <***URL***> sets the remote origin for the repository after the repository is created.

git push

- **git push** takes all commits you have made since last pushing, and sends a copy to the remote server.
- **git push -u origin** <***Branch name***> takes a local branch, and pushes it and all commits on it to the remote server.

git pull makes a local copy of all changes on the server since your last pull. If you were at the end of a branch that had more commits added to it, then those commits are added to your current folder. This can lead to merge conflicts which need to be resolved with a commit.