GEEK LABS: Introduction to Git

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.

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 at https://education.github.com/pack

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. (I mean, it's public, and 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.

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 all you need to know.

So, our command word is **clone**.

So, 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 <evil plan folder name>.

> You should be able to see my face evil plan I put up to hide what my true plan was. It's just inside that folder plan.txt (HAhahahHha Muahahahahaha!!!! I am so EVIL!!!)

Part 2: history.

Ok, so that folder is the most recent version. But we can always go back in time.

Think of Git like a time machine, only to 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 the last commit. The commits are all given a large number letter thing that is that point in times name.

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, (so that would be "git **log**") be talking about the 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.

I was wondering about what happened to the first part of my plan. It got deleted. 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. That file will be back if we go to the commit before it, so, **checkout** the version you want to see by using, well that command. Just put that long number after the checkout.

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

Part 3: Branches

One of my minion 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, he uses a branch to do all of his 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. It is the one we start on. It is the main timeline for all the history, and in most cases, it will be the only one you need to use, however, branches are useful to work on something you are unsure of, and don't want to get in everyone's way.

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. Branches can be brought back together in the master or even a different branch. Like some kind of weird tree, where there is a trunk, that has many branches. Then the branches all weave together again. So, a strange tree.

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. Yes. The names of the branches are a bit more reasonable then the long commit numbers. Also, you just need the last part of the branch name most of the time.

Part 4: Diff

Ok… apparently the minion changed something in this last commit to add part 4 to the "cryptic.txt".

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

Just use the **diff** command (Note, linux has a diff command for 2 files, but git **diff** is for comparing different versions of the same file across commits). We need to give the command two things to compare, and then it will show + and – symbols to show what was added and removed between the first

commit, and the second. So, just see what changed between the last two commits? And we should be find our answer.

Good, 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 it's 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. I think we talked about everything you need for this. Experiment. Although git throws errors, you cannot mess up with it. (There is a way too, but it will warn you a lot, and you have to force it, so just for a second ignore that.)

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. Create your own folder, and just tell git that you want to create a git project here in this folder by using the command **init** when you are in the folder you want to use.

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. See how the **status** has changed?

Great you can **commit** the changes now, after we enter a message (if you have never set it, git opened your default text editor. Probably vi. Press i to begin typing, escape then ":wq" to finish. Or use **–m** followed by your comment in quotation marks after the command), and git will save those changes, just like the commits in my repository. So, you can diff, and go back to those changes any time you want.

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 want to change too.

Part 3: Branches

Now, let your minion create a branch to use. I mean, you could do this yourself if you don't have one.

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 now, 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.

I mean, step 2 could use some work…

Part 4: Setup for things to come.

You can send your minion away now. We won't need them, they did their work, and you can take over. So if you don't mind returning to the master branch?

So, all those changes your minion made are gone on the master branch, or any other, because well, they did not happen when those branches were created. 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, or even more things. 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. One master branch is enough for most things.

So, something that will be a problem, is two people (or you & you) will want to change the same file at the same time. Git actually dose fairly well with that. It's just bad with two people changing the same line, and it's not sure how to mix the two lines together. So we are going to simulate that by changing the same line your minion just changed.

Now, make some change to your plan here that is different to the changes made in the minion branch.

Try and change the same step you changed in the branch to something else, and to show how good git regularly is with merging, why not change a line you did not change in the minion branch.

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 it to **merge** the minion branch. You should get an error like "`CONFLICT (content): Merge conflict`", then everything worked. Take a look at the plan file. 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, Then commit all the files you merged.

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. Git is great for group projects as although everything you did (Except **clone**, and the **branch –a**, was completely in your folder. No internet needed. But git works great online.

While you can push a repository to github to make it online, (https://help.github.com/articles/adding-an-existing-project-to-github-using-the-command-line/)

You can just create an empty repository on github, then clone it like we did with my repository. Then all you need to know, is that everything works the same. But to update the internet version, you have to **push** your changes to the internet, and **pull** other changes. Mostly this means, when you want to commit something, you **pull**, to just make sure there is nothing new. Merge, if needed, then **push** your changes out to the server. That's basically it. There are shortcuts, and cool other stuff you can do with git, but this is more than enough. Plus, you can use those fancy non terminal programs that display everything to you cleanly. While I think you can use one for most things, they are all built on top of the version of git you just used. 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 are some helpful extras to know:

HEAD is a shortcut for the last commit on a branch. So it's an easy way to jump back to the top.

"^" added to the end of a commit, number, HEAD, or branch, means the commit before. (This gets a bit confusing when there are multiple commits before, like merging, so 1,2,3… after selects which of the commits you were talking about)

show is a command that gives information 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 instead of two, and it will assume the other thing to compare it to is the current commit you are at. With nothing it shows the changes since your last commit.

commit can have **–m** followed by a quoted comment. Just a bit less annoying then having to use the text editor.

whatchanged is like **log** but only for a file given after.

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**. You can even ask about a specific command after the help. So if you want to learn more, there is a lot to find.