Git: A Brief Introduction

> "My name is Linus Torvalds and I am your god."
> — Linus Torvalds

Note: Although Linus did put a lot of work into creating one of the most widely adopted, open source operating systems to date. To his credit, he doesn't actually take much of it when discussing his contribution to the development of git[1]*.*

For more *interesting* quotes by Linus, see this wiki page.

1. Introduction

This document is meant to be a very brief introduction to git, specifically how to use git in combination with the Plutus Pioneer Course [1]. For super advanced techno(logical) vikings, I will include some links to additional references at the end (or perhaps a forum where you can go to argue about things would be more appropriate, and more inline with what I like to call: 'the Linus style' — *tongue-in-cheek*).

2. What Is Git?

Firstly, I think there is often a misunderstanding in relation to what Git actually is. It is important to recognise that **(at the very least)** Git and GitHub are two different things. The same comparison can be made between Git and BitBucket or *(to a lesser extent)* Git and GitLab.

To provide a quick explanation, Git is a set of algorithms and data structures that allows for peer-to-peer software development and collaboration. It is a way of sharing a software project amongst many people who can contribute to it.

An interesting thought experiment for anyone new to Git and open source software development: how can we stop malicious actors from introducing bugs and backdoors into software for their own personal gain?)

It is interesting to note that whilst Git was developed in the mid 2000s - it does use what is essentially a Blockchain (or at the very least a 'linked-list-esque' data structure, which is similar to a Blockchain) to manage software projects. Whilst, GitHub (and other similar services) are platforms which host a Git repository (for all intents and purposes, you can think

of a repo as the ledger, but in the simplest terms, it's the codebase) for you. These services are centralised, but it should be noted that you can maintain a git repo (project) through the use of peer-to-peer network communication.

At this juncture, it seems appropriate to make a comparison between distributed ledger systems, blockchain consensus algorithms and Git. Git is essentially a distributed ledger, everybody should have (if they have pulled down the latest code) the same copy of the codebase. However, the primary difference here is that there is an incredibly low probability of Byzantine actors [2] due to the implementation of the peer-to-peer web of trust model [3].

2.1 The Web Of Trust Model

Instead of having to deal with a distributed system where the Byzantines Generals Problem [2] is a very real threat, you may modify 'the system' to use an alternative security model that minimises threat (the system itself may not require a large degree of security considerations, although I wouldn't ever make that assumption).

In this case, you only allow access to the codebase to immediate trusted parties (say, you have six developers who you know you can trust), in turn they also have their own trusted parties. Since nobody really has anything to gain from acting maliciously (within a *potentially* trusted group of individuals — *never underestimate the power of human incompetency*), the probability of a Byzantine actor tends towards zero. Thus, Byzantine Fault Tolerance [2] isn't something you really have to worry about. Therefore, it follows that consensus (everybody has the same copy of the codebase) is quite easily reached without the use of immediate monetary incentives.

Note: since this document is contained within a repository about, for lack of a better word, cryptocurrencies, it seemed appropriate to draw some correlations between Git and DLTs + Blockchain Consensus Algorithms.

3. The Typical Case

The way in which open source projects (and private projects ran by private companies) are *typically* developed is through the use of a service such as GitHub. Open source software is available to view by anyone and, for the most part, anybody can contribute (subject to review and the project maintainers acceptance). The process of contributing to an open source project requires some prerequisite knowledge of how Git works, which we cover get into in §5.7. Private companies keep their proprietary code restricted, for obvious reasons[2], thus

Continue Writing This At A Later Date...

4. Basic Git Commands and Repositories

During this section we are gong to run through the very basics of git. Firstly, how to configure

a basic git user, downloading (cloning) repositories and finding out who has changed what.

4.1 Git: Simple Configuration

When you contribute to any project, whether it is open source or closed source, it's always important (and polite) to let people know who you actually are. This can be accomplished by specifying she settings (your name and your e-mail address) within the console. We do this using the following commands:

1. `git config --global user.name "Jonathon Dilworth"`
2. `git config --global user.email "jon@dilworth.dev"`

Note: the addition of the --global flag will set these values for any commits you push, on any project. To keep it project specific, simply remove the --global flag.

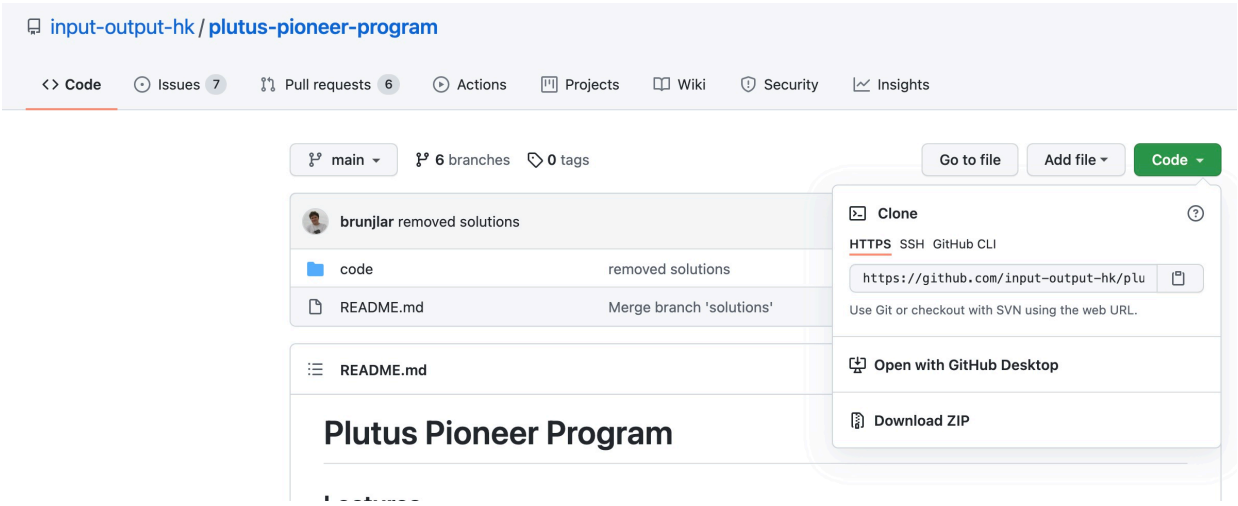
Read More Here

4.2 Git: Clone

When you copy an open source Git repository from a centralised service such as GitHub, you won't need explicit access[3] to the repository. You simply navigate to the repo, look for the green button that says: 'clone', select it (it is a drop down, of sorts) and use either HTTPS or SSH. If in doubt, HTTPS should work just fine. Copy the address provided for you by the drop down window, open your favourite console, navigate to where you want to clone the repo and enter:

```
git clone https://github.com/input-output-hk/plutus-pioneer-program.git
```

All should go well and you should now have a file called: plutus-pioneer-program within your chosen directory. **Welcome to the Plutus Pioneer team**.



Read More Here

4.3 Git: Log

Configuration variables, such as the one above are extremely useful because sometimes you'll find that you may need to know who pushed a specific commit. This is possible by running `git log` within the repository directory.

As you can see from the image below, this provides you with the commit hash (which we'll learn more about in §5.4) some information about the branch the author has been operating on [§5.6], the author name and e-mail, the date and the commit message.

```
commit 19b35191bdadff26371d4d7621d520bb2ed3000d (HEAD -> main, origin/main, orig
Author: Jon Dilworth <jon@dilworth.dev>
Date:   Thu Aug 12 08:46:15 2021 +0100

    paper updates, progress on git primer and progress on L4, which is LONG. I'm

commit 6170507cf6c256a5a59e651e6c72a72fdb8a8bd3
Author: Jon Dilworth <jon@dilworth.dev>
Date:   Wed Aug 11 15:21:27 2021 +0100

    lecture five, README update and an introduction to Git added.

commit ef5d3de367a695286ad7a6a35b9f37304f8bd1fe
Author: Jon Dilworth <jon@dilworth.dev>
Date:   Tue Aug 10 20:32:09 2021 +0100

    temporarily pausing lecture four, moving forwards, added more papers, added

commit 9bea30ab05cd00c311ca12b8df5da7ca695d0738
Author: Jonathon Dilworth <jon@dilworth.me>
Date:   Tue Aug 10 19:38:49 2021 +0100

    Added license
```

This is helpful because it allows us to easily track who made certain changes.

Read More Here

5. What Is A Repository?

Up until this point, it has been assumed that you understand the basic concept of a repository (often referred to as a repo). However, there is a little more to a repository than may meet the eye.

It just looks like any other file right? Except it has some fairly special properties (hidden .git folder[s]) which allow for useful functionality. For example, the pointer that references the current 'instance of code' (commit) you are working from is stored within these types of folder. You don't really need to know the ins and outs of what resides within a .git folder (or, if you do,

you're reading the wrong piece of documentation, this is intended for absolute beginners). It's just important you know these things exist and that it's essentially how git is able to implement functionality that you would expect from a version control system (VCS).

If you wanted to just stick with **really basic**, a repository is essentially just a folder that contains shared code, that you can collectively modify under a set of constraints.

Read More Here

5.1 Data Structures

According to Wikipedia (the fountain of all knowledge):

> Git has two data structures:
> 1. A mutable index (also called stage or cache) that caches information about the working directory and the next revision to be committed; and an immutable, append-only object database.
> 2. The index serves as a connection point between the object database and the working tree.

Note: An immutable, append only, object database? That sounds familiar, kind of, right?

For now (as a beginner), I don't think you need to worry too much about the underlying data structures, but it is a good thing to do, to check out under the hood. If you're going to use branches (which we will briefly discuss in §5.6, then, for sure, take some extra time to really understand what it means to create a new branch, make modifications, open a pull request and merge your changes into another branch, which will likely eventually find its way to staging and then production.

You may be trying to get an intuition about git for the first time right now, in which case, some of the more complicated elements of the underlying data structures and algorithms that power this version control system (VCS). This may put you off investing the time to learn it. However, regardless of what you do, if you want to develop software of any kind, there is no getting away from version control systems, so **you will need to know this stuff**. On the other hand, the way I see it is that there are more pressing matters at hand. For example, I keep asking myself: **WHY THE HELL AM I DOING WRITING A BASIC GUIDE FOR GIT WHEN I SHOULD BE LEARNING HASKELL!?!??!?** Ultimately it was due to a few questions that popped up on the discord. I thought: well, if three or four questions have popped up, I might as well throw together a very basic guide, as that is all that's really required for participation in the pioneer program. But, I should really know better. Things always take longer than you have planned, which is why you triple your time estimate; because believe me, that then becomes a more realistic delivery date. **Remember that all you junior devs!**

Note: I do also enjoy helping others and writing, so when Lars' voice gets a little bit too much for me, I take a step back and start writing about git! That was a joke by the way, Lars has an

incredible voice. The fifth lecture during the second iteration was delivered brilliantly and eloquently, I really enjoyed that lecture. The fourth lecture, however, it was like an information overload! Sorry!

Some More Interesting Reads:

1. Git Is A Purely Functional Data Structure
2. Git Internals
3. *Note: We'll slowly add more to this list over time...*

5.2 Git: Pull

This is pretty simple actually. Running `git pull` within a git repo directory will simply pull down the latest contributions everybody has made. It can get a little hairy if you're all an unorganised mess and you're editing the same files. Then you get something called a merge conflict §5.7.

Read More Here

5.3 Git: Add

Again, this is simple and easy to understand. Whenever you change something in your local repo, you can run a command called `git status` and it shows you which files you've changed, created or deleted. These files need 'adding' to the next commit. So, there are two ways of doing this.

1. The Cautious Way:

This is what I do, because sometimes other files and found their way into my local repo and they need removing. Alternatively, something isn't ready yet to commit and push. Although, I don't worry about that in this project because it's essentially a documentation project.

So, the cautious way is adding files one at a time. This is accomplished by running the following command: `git add [INSERT RELATIVE FILE PATH HERE]`.

2. The... "Ahhh, everything will be just fine" Method:

Simply run `git add .` and all the changes you've made will be appended to your commit. One of the reasons why this may be risky at times is, well, for a number of reasons really. But, the one I have seen happen (thankfully the repo was not open source), the dev accidentally pushed a bunch of passwords up to the repo (keys are usually protected, as you would typically store keys in a folder that the project manager has **made sure** resides within the .gitignore file). So yeah, maybe just watch out if you're new to this!

Read More Here

5.4 Git: Commit

Once you have a bunch of new changes together using the `git add` command, you can bundle them together in the form of a commit. A commit will be seen by other individuals when they run `git log`, they can then checkout to that commit and look at your changes. Alternatively, if they're using GitHub / a centralised method of hosting the repo, it's usually a lot easier to view commits and the changes that you have made. It's made easy through the use of a web GUI.

Right, so in order to bundle all your changes together, say, at the end of the day, or when you've finished a particular feature, you run the following command:
`git commit -m "I am inserting a message here to describe what my changes do."`.

Simple As.

Read More Here

5.5 Git: Push

Right, so all your changes are only held on your own computer locally. But what about the rest of the team? They need to see your changes, they also need to pull them into their own codebases. Reason being: they might need a feature you are responsible for creating in order to move forwards with their work. Thus, you require a way of pushing your commit to the repo. To be clear, you add modified files, then you commit, then you push (and it's recommended that you perhaps try pulling before doing any of this in case there are any merge conflicts).

After having committed your changes, go ahead and run `git push`.

Read More Here

5.6 Git: Branch

*I wouldn't worry about this too much for now. Just remember if you need to change branches or load an alternative commit, you can run either: `git branch`, this shows you all the available branches, then you can `git checkout [BRANCH_NAME]` to that branch. Alternatively, you can `git log`, find an old commit hash and do the same.

Read More Here: Branch

Read More Here: Checkout

5.7 Git: Extra Reading

- Merging Branches I When you need to merge new code you have written with the rest of the codebase.
- Opening Pull Requests I Making a request to implement changes to repos.
- Detached Head I This will come in useful, you can be banging your head against the wall for a long time if you don't understand this.
- Additional Further Reading I GitHub have provided a lot of good reading material. My advice: check it out (no pun intended).

6. Using Git With The Plutus Pioneer Project

- Regularly pull down the latest changes to the program.
- Regularly pull down the latest changes to Plutus.
- When you need to revert back to a previous lecture or lesson, go to your plutus-pioneer-program folder, find the week and check the commit hash in cabal.project. You can then checkout to that previous code. You MAY have to rebuild cabal, I'm not too sure. I have had a few people ask me about this.
- When you need to get back to the main branch, you should be able to checkout to main or master (not sure what it's called). You may get a detached head, depending on what kind of changes have been pushed to the repos. Good luck with that ;) You should be able to just run `git pull origin main`, if that doesn't work, perhaps try running `git log` and checking out to the latest commit. That may not work either though... I mean, it's up to you to figure it out! That's part of the fun, but you can always hard reset. See additional reading.

References

[1] The Cardano Group, ADA. (2021).
Plutus Pioneer Pprogram by IOHK.
Available at: https://testnets.cardano.org/en/plutus-pioneer-program/
Accessed: 10/08/2021

[2] Lamport, L., Shostak, R. and Pease, M., 2019.
The Byzantine generals problem.
In Concurrency: the Works of Leslie Lamport (pp. 203-226).

[3] Baltakatei. (2020). StackExchange.
The Web Of Trust Model.
Available at: https://crypto.stackexchange.com/questions/80629/what-is-the-pgp-web-of-trust-strongset/

Footnotes

1. Believe it or not, Linus can be quite humble! See the below quotes.

> "I maintained Git for six months, no more, ... The real credit goes to others. I'll take credit for the design."
> — Linus Torvalds, Open Source Summit Europe

> I'd also like to point out that I've been doing git now for slightly over two years, but while I started it, and I made all the initial coding and design, it's actually been maintained by a much more pleasant person, Junio Hamano, for the last year and half, and he's really the person who actually made it more approachable for mere mortals. Early versions of git did require certain amount of brainpower to really wrap your mind around. It's got much much easier since. There's obviously the way I always do everything is I try to do everybody else to do as much as possible so I can sit back and sip my Piña Colada, so there has been a lot of other people involved, too.
> — Linus Torvalds, Tech Talk: Linus Torvalds on Git at Google

You can watch the Tech Talk Here.