






Warning! This zine is a WIP!

Thank you for taking interest in our zine! Feel free to take a look around, but remember: this is a very early draft. Let us know what you think on our socials, or by mentioning #FujoGuide.

-  [Tumblr](#)
-  [Twitter](#)
-  [Mastodon](#)

Like what you see? [Back our kickstarter!](#)



From the Author

I wrote this book as I was learning version control. It was very difficult to stay focused, but the hot men helped.

Forgive the typos and TODOs, they'll be fixed in the final edition. I only had two months to put this whole thing together (lol).

You can leave feedback at [TODO].

— Boba-tan

The story so far...

After getting isekai'd to a weird world where web development concepts are suddenly hot, Boba-tan has finally settled in at her new place in `Localhost HQ`.

There's no time to relax though because a new danger already looms at the horizon: deadlines.

The Characters



Terminal



Git



GitHub

Guest Starring



HTML



CSS



ARIA

Table of Contents

Chapter 1: Meet Git

- >_ Disaster Strikes for Boba-tan!
- >_ Introducing Git
- >_ Git & Version Control Systems (VCS)
- >_ Saving & Moving Through Your Code History
 - The Gi(s)t
 - Repositories: Where Your Code Lives
 - Commits: Code Checkpoints Made Easy
 - Putting It Together
 - *Coming Soon: Git History*
 - Log: Your Code Diary
 - Checkout: A Quick Dip Into The Past
 - Reset: Time Travel At Your Fingertips
- >_ Working In Parallel
 - The Gi(s)t
 - Branches: A Multiverse For Your Code
 - Merge (Rebase): Bringing The Timelines Back Together
 - Fixing Merge Conflicts
 - Bonus: Different Types Of Merge
 - Putting It Together

Coming Soon: Chapter 2: Meet GitHub

>_ Comic

>_ Introducing GitHub

>_ Understanding Remotes

→ What Is A Remote

→ Why You Should Use A Cloud Hosted VCS

→ Why We Chose GitHub & Alternatives

→ The Gi(s)tHub

◦ Push: Upload Your Work To The Cloud

◦ Pull: Get Your Work Back From The Cloud

◦ Clone: Start It All Over Again

→ Harnessing the Power™

◦ Automatic Deployment

◦ GitHub Actions

>_ Collaborating With Others

→ How Cloud-Hosted VCS Fosters Collaboration

◦ Forking: Customize Your Own Software

◦ Pull Requests: Working In A Community

◦ Templates: Help Others Get Started

[BONUS] When Git Gets: Mad Recovering From Failure

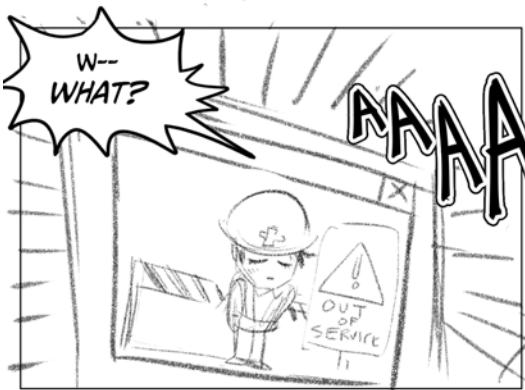
[BONUS] Commands Reference

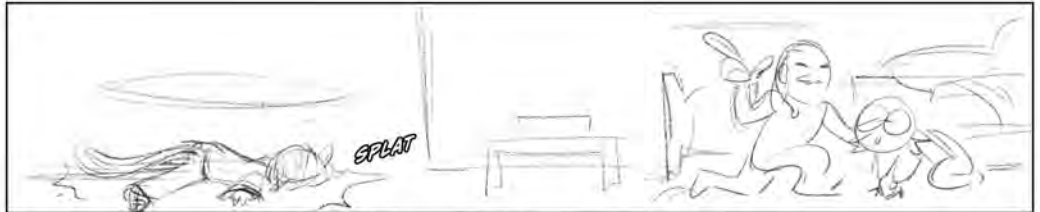
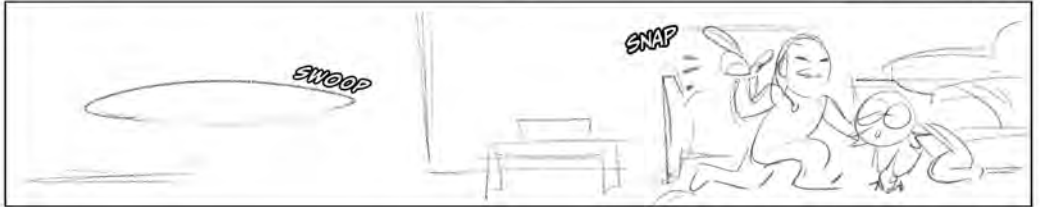
[BONUS] Glossary



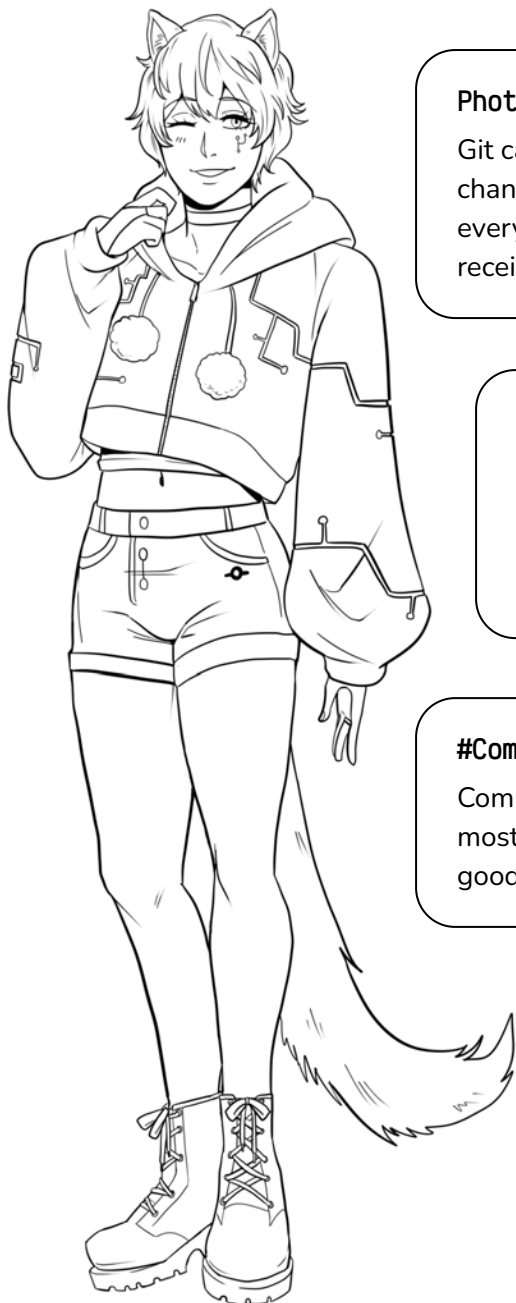
Chapter 1: **Meet Git**







Introducing Git



Photographic Memory

Git can easily remember all changes to your code, as well every real or perceived slight he's received.

Conflict Resolution

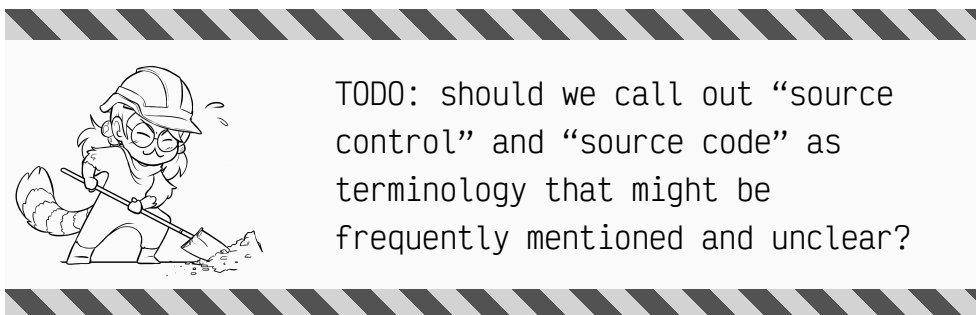
Git is adept at resolving conflicts across file changes. But watch out: he can be prone to anger when things don't go as planned.

#Committed

Committing often is one of the most effective ways to stay in Git's good graces.

Git & Version Control Systems (VCS)

Git and other **Version Control Systems** help you edit your code with confidence by making it much harder to irrevocably mess up your programs. They allow you to revisit multiple versions of your code as they existed at previous points in time and work on many features and improvements in parallel without losing access to older working version of your code. VCSs also allow multiple people to work on the same code without fear of overwriting each other's changes.



As you familiarize yourself with Git, you'll be able to:

- Move back and forth through the history of your code, comparing previous versions and (potentially) restoring edited or deleted files.
- Work on separate features in parallel, merging them into the main version of your code once they're ready.
- Figure out when an error was first introduced and what change was the culprit.

In the next section of this guide, you'll also learn how GitHub (a cloud-based Version Control System based on Git) can help you back up your code, share it with others, and collaborate together.

(Callout, character TBD) Despite being most often associated with software development, Version Control Systems can be used with any type of file: whether it's the evolving text of a future fanfiction, different versions of the same image file, or any edit to a set of files over time, version control can help you permanently keep track of changes.

While many—not all!—experienced programmers use Git through its command line interface (Terminal), more visual and interactive options are also available. You can explore our current recommendations on our companion website. [TODO: add link]



Should I use Git through the command line or a visual interface?

Command Line Interfaces (CLIs) are very popular with experienced programmers! While they might seem scary at first, they only require learning a small handful of commands to be used effectively and can be faster and more powerful than Graphic User Interfaces (GUIs).


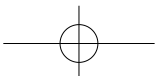

DON'T BE
AFRAID, MS
BOBA-TAN.



I'M NOT
SCARY AT ALL,
HAHA.

He was cooking.

You don't need to start your programming journey by using a CLI, nor do you ever need to use one if a GUI is available for your program of choice. However, as you progress in your journey, you may encounter programs that require one.



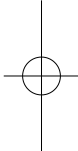
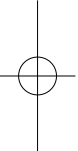
When this happens, don't panic! Take it one command at a time and remember: even long-time programmers are always looking up commands they don't use often—sometimes even those they do!

Saving & Moving Through Your Code History

The Gi(s)t

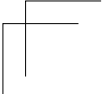
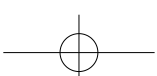
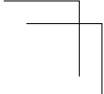
Why is version control such a big deal when you can simply save a file? Although saving a file is a useful action on its own, the power of version control is in keeping an *archived history* of your changes.

Have you ever:

- 
- 
- Rewritten part of an essay or fic, saved, and returned the next day only to realize the previous version was better?
 - Merged layers on an image, saved over the unmerged copy, and not realized it until you reopened the file later?
 - Had a program crash in the middle of a save so that you can't undo a change you decide you wanted to undo?

In all of these cases, if you had archived a prior version with version control, you could restore the prior copy, regardless of the software's "undo" limitations.

The benefits of version control are so desirable, it is a prominent feature of cloud hosting services like Google Drive, OneDrive, and Dropbox. Likewise, Git is a powerful tool with a very long memory!





TODO: We could add an image of Git promising never to use his power for evil - but you sort of don't believe him.

Repositories: Where Your Code Lives


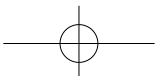

Before you can take advantage of Git's powerful memory tools, the directory your code lives in needs to be turned into a **repository**, often shortened as "repo".

When you first initialize a Git repository (by running the `git init` command or equivalent), a `.git` subdirectory is created. While you won't need to interact with it directly, this `.git` subdirectory stores all the data related to the history of the changes, as well as the actual changes made at each step. Without it, a directory is just a collection of files.



TODO: Consider using footnotes or margin notes to point out that directories and subdirectories are also called "folders".

As long as the `.git` directory remains intact, you can always recover files that were saved in it, because it keeps copies of all versions of your files. This is important to know if you're using it for larger files, as it may make your `.git` directory very large. Once something is committed to Git's memory, it will stay forever in the `.git` directory



unless you specifically remove it from Git's history. This is a more advanced process that exceeds the scope of this guide, but if you find yourself needing to do so, we recommend you search the web for "deleting a file from *.git* history".

The *.git* directory is a hidden folder by default, so you may not always see it unless you change your computer settings. However, there's nothing mystical about it; you can actually view the folder both in your file explorer and using the terminal. Although we won't go into details on that (it's not really necessary to get started), if you're interested there are other online resources explaining how to view the contents of your *.git* directory.

To ask Git to track a directory's history, you run `git init` in your CLI.

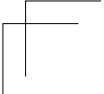
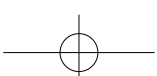
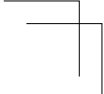


Commits: Code Checkpoints Made Easy



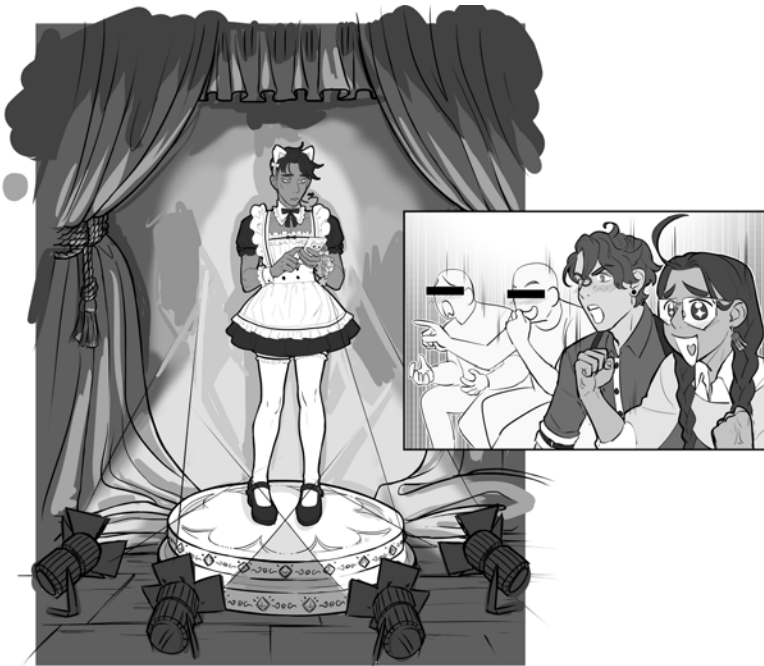
While Git keeps track of your files' histories for you, this isn't an automatic process. Instead, Git relies on **commits** to learn about new files or changes to existing ones. A commit represents a snapshot of your code at a certain point in time that Git has *committed* to its long-term memory; it's a bit like manually saving your game just before a boss fight to be able to recover your place in the game. You can create a commit using the `git commit` command.

If you're ever unsure of what Git is tracking, you can check the status of your repo with `git status`. When there are no changes, Git responds to the command to let you know. When a new file has been added, because it has never seen this file before, Git shows the file name in red and will advise the file is untracked. Files that



Git is already tracking which have modifications since the last commit will show in green.

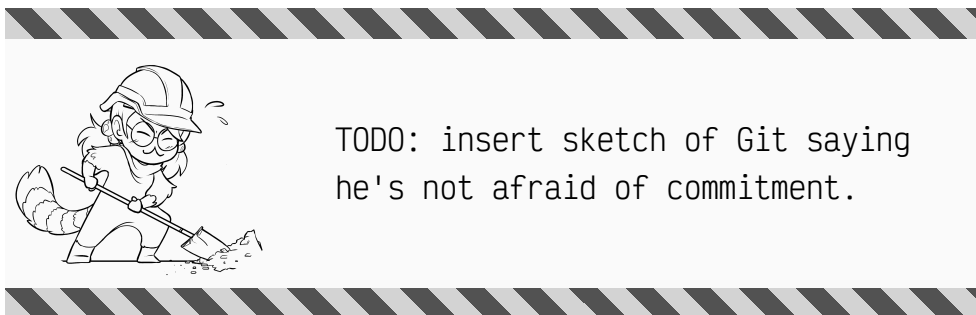
You can use the `git add` command to tell Git about the new files or changes that you wish to include in the next commit. This process is called **staging**. Since commits will only include changes that have been explicitly staged, you can easily separate unrelated files and commit them on their own. This makes your code history easier to understand and move around.



Staging changes to HTML files.

Staging the untracked red files mentioned before updates them to green and labels them as "new file" when you run `git status` again, indicating that Git is aware of them now.

Once you're ready for Git to store these changes to its memory, you would run `git commit`. Commit frequently in order to prevent potential loss of changes!



In most GUIs, a commit is represented with the circle symbol, while in a command line interface it is usually represented with an `*` unless you use certain options known as **flags**. More information about these flags appears in the section titled "Log: your code diary" and on our Git Cheat Sheet.

There may be times when you want to exclude certain files and folders from Git's memory. These might be files and folders with sensitive information, such as access keys, passwords, or user data. You can tell Git to ignore these files by creating a `.gitignore` file. This is especially important if you use a remote hosting service like GitHub and have your repository set to public. More information on `.gitignore` can be found in **Chapter 2: GitHub**.

Putting It Together

Imagine you're creating a new website for yourself and don't want to suffer the same misfortune as poor Boba-tan. You wisely decide to enlist Git's help *before* disaster strikes. In general, your workflow for your new project is:

1. Initialize the repo.
2. Create files, delete files, and make edits.
3. Stage the files.
4. Check status to verify staging.
5. Commit.
6. Go back to 2.

Note: once you start working with branches (discussed in "Branches: A Multiverse For Your Code"), this workflow changes slightly!

After opening your terminal and navigating to the folder where you want to store your repo (see our "Command Line Tips" for help) you run:

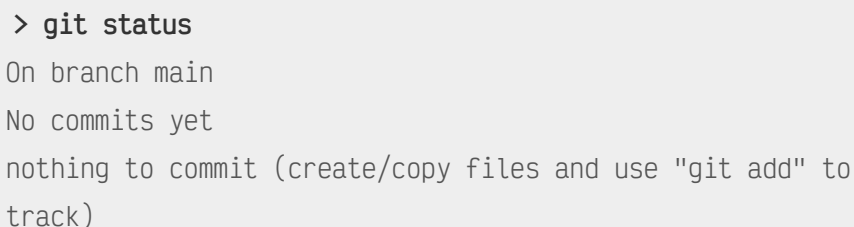
```
> git init my-website
```

```
Initialized empty Git repository in ~/my-website/.git/
```

This will both create the directory called "my-website" AND ask Git to track it for you. You could also create the directory yourself in any way you're familiar navigate your terminal into the already existing directory, and then run `git init`.

Now that you've got your repository initialized, you begin to build your website. First, you create a file called "index.html" to serve as your homepage. You spend some time working on it and saving as normal in your code editor, and then decide to take a break for lunch.

When it's time to start working again, you can't remember if you've asked Git to track the *index.html* file yet. You decide to check, so you move to your terminal and run:

A terminal window with a grey title bar and three window control buttons. The output of the 'git status' command is displayed in a monospaced font.

```
> git status
On branch main
No commits yet
nothing to commit (create/copy files and use "git add" to track)
```

You realize this means you haven't tracked *index.html* yet! You're about to start some major updates to the content and decide that you should probably save all your changes so far. You run:

A terminal window with a grey title bar and three window control buttons. The command 'git add -A' is entered at the prompt.

```
> git add -A
```

Using the -A flag stages all the files in your directory that have had changes made to them. For more useful flags, see the [Git Cheat Sheet](#)! Now that Git knows about the files you've added, you're ready to commit.

You'll first verify that your files are staged correctly by running:

A terminal window with a grey title bar and three window control buttons. The output of the 'git status' command is displayed in a monospaced font.

```
> git status
On branch main

No commits yet

Changes to be committed:
```

```
(use "git rm --cached <file>..." to unstage)
new file: index.html
```

Now that you've confirmed, you are ready to commit! You run:

```
> git commit -m "Initial commit."
[main (root-commit) e350e8c] Initial commit.
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 index.html
```

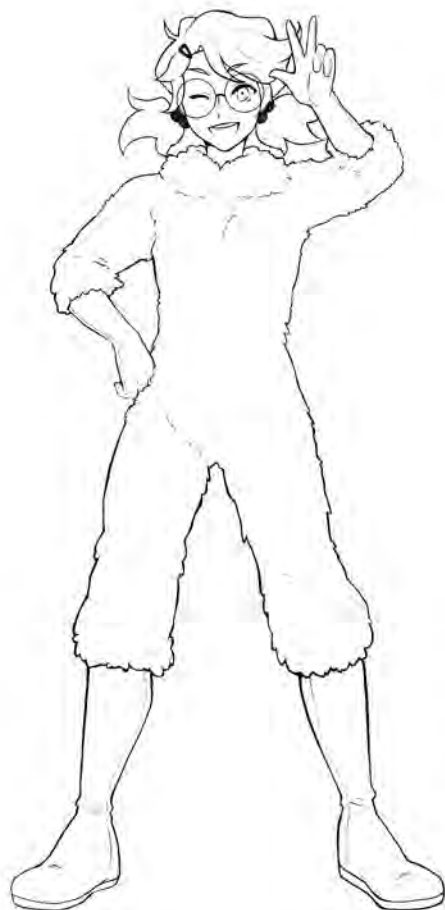
You've committed your first archive of your files to Git's memory! Every commit should have a commit message to accompany it which summarizes what you've done. Check out our "Tips for Commit Messages" cheat sheet.



How often should I commit?

There's no hard and fast rule for how often you should commit! It varies based on the project and your own personal preferences. Generally, it's a good idea to commit after you've completed a self-contained but significant change, such as successfully creating a new small feature or resolving a frustrating bug.

Don't worry about committing too frequently. Not only are small commits easier to understand and review, but it's easier to combine multiple of them together ("squashing") than it is to try to split ones that are too large!



...To Be Continued



Credits

Project Lead/Director

- Ms Boba

Producers

To be done!



Project Organizing

- Ms Boba
- Enigmalea
- Slogbait

Content Designer

- Ms Boba
- 
- 
- 



Technical Writer

- Enigmalea
- Ms Boba
- Heidi

Editor

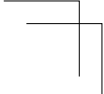
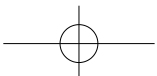
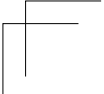
- Enigmalea



Art Directing

- Jack @brokemycrown

Character Design

- Jack @brokemycrown (Git, GitHub)
 - Sgt-Spank (Aria)
 - spillingdown (Terminal)
 - ymkse (HTML, CSS, Boba-tan)
- 
- 
- 



Additional Art & Layout

- AmkiTakk
- Cat Bathing Sun (Graphic Design)
- Catterbug
- Kiwipon
- SCUMSUCK
- Slogbait

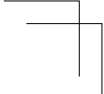
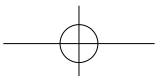
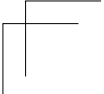


Social Media & Marketing



- Owl
- Thebiballerina

QA Testing

- Citro
 - CMDonovann
 - Elendraug
 - Enigmalea
 - Michelle
- 
- 
- 

- Yuu

Sensitivity Reading

- Moon (AdmiralExclipse)
- Miscellanium

Catboy Wrangler

- Michelle

Lead Research

- Elf Herself
- Yuu

Additional Research, Feedback, Development, and Assistance

- Candle
- Citro
- CMDonovann

- Elendraug
- Ererifan915
- MadGastronomer (Event Planning)
- Mantra
- Michelle
- Noclip
- Pamuya
- Playerprophet
- SCUMSUCK
- Thunder the Wolf
- Tovanish
- wilde_stallyn