Getting Started with Git

Authors: Alex Huddleston, Riley Trautman

What is Git?

Torvald's fancy undo button

Git is a *system* which tracks the changes you make to files in a special folder called a *repository*. Each time you *commit* to a significant file modification, Git will help you record why you made the change. Git can also help you *revert* or undo changes if you mess up. Sometimes you're not sure which direction you want to take, so you make two *versions*. When this happens, Git can help you *branch* off and control your different project versions. When working with a team, Git can create online folders, called *remotes*, so that teams can *synchronize* their files or post *requests* for help. Lastly, if two friends make *conflicting* changes to the same file, then Git can help them *differentiate* between versions and *merge* the changes back into one file.

Creating a new repository

Everybody starts somewhere

When using a tool like GitHub or GitLab, often you won't need to create your own git repository, but we'll start from the beginning just in case. Skip to Cloning a Remote Repository if you don't care about this.

syntax: `git init [[path/to/repository]]`

example:

```
asonix@asonix-gs60 ~/Downloads/Telegram Desktop
$ git init temp/
Initialized empty Git repository in /media/windows-data/Downloads/Telegram Desk
top/temp/.git/
```

note: this command creates the folder if it doesn't already exist

Setting Remotes

Sharing is caring

Once you've created your repository, it is likely that you will want to share it with someone. To do so, you must also initialize a repository in a public location.

Once you have a public repository, you can tell your local copy where you want to sync code using the `remote add` command.

syntax: `git remote add [[remote name]] [[url]]`

example:

```
asonix@asonix-gs60 ~/git/temp
$ git remote add origin https://github.com/KayDevs/Project3.git
```

Cloning a Remote Repository

We stand on the shoulders of giants

If there is already a repository for your project, you'll want to contribute to that rather than creating a new repository. In order to make a local copy of the code so you can make changes, you'll want to `clone` the repository.

syntax: `git clone [[https://url.of/the/repository.git]]`

example:

```
asonix@asonix-gs60 ~/git
$ git clone https://github.com/Airblader/i3.git
Cloning into 'i3'...
remote: Counting objects: 31250, done.
remote: Total 31250 (delta 0), reused 0 (delta 0), pack-reused 31250
Receiving objects: 100% (31250/31250), 8.72 MiB | 366.00 KiB/s, done.
Resolving deltas: 100% (23935/23935), done.
Checking connectivity... done.
```

Creating your branch

You'll want your own copy of the code to modify to your heart's content

It is important to note that cloning the repository is only the first step to getting your own code to work on. When you first clone the repository, you will want to create your own branch so you can make changes to your code without polluting the `master branch`.

The term **branch** comes from the idea that updates to the code should be seen as a tree. The Code can exist in multiple branches (versions) simultaneously, and the branches can split or merge freely. The `master branch` is at the base of the tree, and should be considered sacred.***The*** `master branch` ***should only contain code that has been tested and is known to work.*** This way, we have the ability to distribute copies of our code that we know will not fail.

syntax: `git checkout -b [[your-branch-name]]`

example:

```
asonix@asonix-gs60 ~/git/i3 (master)
$ git checkout -b asonixdev
Switched to a new branch 'asonixdev'
```

It is typically a good idea to create a new branch for each new feature you want to add. This makes managing the repository much easier in the future.

Switching branches

When your first copy of the code is not enough

You may want to create different branches for different features and bugs you are planning to add and fix (respectively). In order to switch between branches that have already been created, you use the following: `git checkout [[branch-name]]`

`:: git checkout alexdev` changes your files to whatever is currently in your version of the branch you want to work on, in this case, alexdev

syntax: `git checkout [[branch-name]]`

example:

```
asonix@asonix-gs60 ~/git/i3 (gaps)
$ git checkout master
Branch master set up to track remote branch master from origin.
Switched to a new branch 'master'
```

Updating Your Branches

Other people write code too, you should get it

Always `pull` before you start working on new code. It will fetch any updates that have been pushed to the server on the branch you're on. It will also tell you which other branches you are tracking that are out of date and need to be pulled. You should do this on all branches you are tracking before you push commits to the server.

syntax: `git pull`

example:

```
asonix@asonix-gs60 ~/git/i3 (master)
$ git pull
remote: Counting objects: 689, done.
remote: Total 689 (delta 319), reused 320 (delta 319), pack-reused 369
Receiving objects: 100% (689/689), 182.78 KiB | 0 bytes/s, done.
Resolving deltas: 100% (537/537), completed with 119 local objects.
From https://github.com/o4dev/i3
   c56403b..c922378  gaps       -> origin/gaps
   31c33df..4c878f4  master     -> origin/master
 * [new tag]         4.8        -> 4.8
Updating c56403b..c922378
Fast-forward
 232 files changed, 9378 insertions(+), 5336 deletions(-)
```

Making changes

Programming ensues, changes are made. Whether they are right or wrong.

To check which files have been changed and need to be committed, or to see if your local code is outdated, you will want to get the repository's `status`.

syntax: `git status`

example:

```
asonix@asonix-gs60 ~/git/i3 (master)
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   Makefile

no changes added to commit (use "git add" and/or "git commit -a")
```

If you made changes, you will need to add those changes before you can commit them so Git knows what is being tracked and what to commit. To add files, just use `git add filename.extension` where filename is the name of the file and extension is the extension, if any.

```
asonix@asonix-gs60 ~/git/i3 (master)
$ git add Makefile
```

PROTIP: PLEASE DON'T ADD BIN FILES LIKE "a.out" BECAUSE YOU WILL MAKE ME MAD BECAUSE I'LL HAVE TO FIX IT BECAUSE THOSE DON'T MERGE WELL.
We'll cover methods to ensure this doesn't happen later

Committing Changes

Committing changes solidifies a group of changes as a version of the code in git

After you've added the changes so they are tracked, you can make sure everything is good by entering `git status` again and the output should tell you all the files which are being tracked. If something is not yet being tracked (like, you made a new file to be uploaded) you can 'git add' that as well or you can just ignore it if you don't want it pushed to the server.

To commit files to the LOCAL repository, use `git commit -m "enter message here on why`

`you are committing"` if you want to force a commit for whatever reason or just don't feel like adding files (not recommended) use `git commit -a`.

syntax: `git commit -m [[commit message]]`

example:

```
asonix@asonix-gs60 ~/git/i3 (master)
$ git commit -m "test commit"
[master ed9d6a7] test commit
 1 file changed, 2 insertions(+)
```

PROTIP: If you don't put a message in, the thing will yell at you and not do the commit. using -a won't require a message, but I don't recommend using it unless you just really can't get the commit to work. It involves me having to go in afterwards and fix Github ruining things.

** Minor Troubleshooting ** If you accidentally added a file to be tracked and want to remove it, use `git rm filename.extension …' (and other filenames if you want to do more). If you removed a file the normal way (rm, delete, etc.), it should be added to what is being committed automatically.

If you want to discard the changes you made (for example: someone else made them already and committed them, so your changes are useless now) use `git checkout -- filename.extension ..`

Pushing Changes To The Server

If all of that went well, all you need to do is use the push command. This pushes the commits you made on your local repository to the server's repository. This is why when you don't use the pull command often, the server won't let you push things. Your local repository needs to match what is in the server repository before you can make a push to the server, since it checks all of that and changes whatever is different. Otherwise, it'll yell at you for not having its updates.

styntax: `git push`

(authentication ensues)

Merging Branches using Pull Requests

Having branches is nice, but we need all the features in the same place (when we ship our product)

Many git frontends have a nice feature called Pull Requests, and since it helps us merge changes, we should probably use it. When you want to merge your feature branch into `master`, it is *incredibly* important that you use a pull request. To ask to be merged in, you go to the repository's GitHub or GitLab or WhatEver page, you switch to the branch you want to merge from, and you click `New Pull Request`. This will bring you to a page where you can specify what changes have been made and why this `compare` branch should be merged with the `base` branch. Once you have explained yourself thoroughly, click `create pull request`. If the author of the `base` branch thinks your changes are important and needed, they can accept the pull request.

Merging Branches Manually

For when we don't want to use GitHub's/GitLab's pull request feature, when pull requests fail, or when we aren't dealing with a sane git frontend

From here, since we pushed changes to our branch `alexdev` *and used pull requests to merge* `alexdev` *into* `master`, we want those changes on **every** feature branch to make sure we're working with updated code. To do this, we're going to use the merge command and them push those changes to the server. You have to push because when you merge it's doing that on your local repository first, then it needs to push those changes to the server.

`git merge master` merges what changes are in master into the current branch, git calls this "fast-forwarding"

syntax:

- `git merge [[branch_with_changes]]`
- `git merge [[branch_to_merge_into]] [[branch_with_changes]]`

example:

```
asonix@asonix-gs60 ~/git/i3 (master)
$ git merge master gaps
Merge made by the 'recursive' strategy.
 33 files changed, 680 insertions(+), 58 deletions(-)
```

Note: this example shows merging into master, please do not do this in any situation

Other Things of Note

Git provides some other features that can be very helpful. Most importantly, the `.gitignore` file. `.gitignore` keeps track of files you *do not* want to me committed and pushed to the repository. You can add files to `.gitignore` explicitly by stating the file path in reference to the location of `.gitignore`, or by specifying splats to cover multiple files (also in reference to the location of `.gitignore`).

A `.gitignore` may look like this:

```
a.out # to avoid commiting the compiled program
*.o # to avoid committing other binaries
*~ # to avoid committing backups of files
*.swp # to avoid committing swap files (created by vim)
.clang_complete # to avoid committing local clang configurations
tmp/* # to avoid committing anything in the local temp folder
```