Introduction

What is Git?

Git is a popular version control system. It was created by Linus Torvalds in 2005, and has been maintained by Junio Hamano since then.

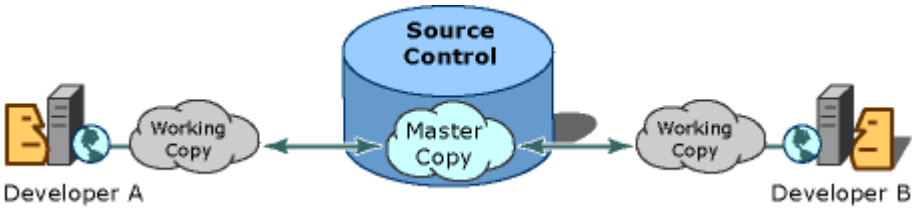
It is used for:

- Tracking code changes
- Tracking who made changes
- Coding collaboration

What is usually done using Git

- Initialize Git on a folder, making it a Repository
- Git now creates a hidden folder to keep track of changes in that folder
- When a file is changed, added or deleted, it is considered modified
- You select the modified files you want to Stage
- The Staged files are Committed, which prompts Git to store a permanent snapshot of the files
- Git allows you to see the full history of every commit.
- You can revert back to any previous commit.
- Git does not store a separate copy of every file in every commit, but keeps track of changes made in each commit!

Why Git?



- Over 70% of developers use Git!
- Developers can work together from anywhere in the world.
- Developers can see the full history of the project.
- Developers can revert to earlier versions of a project.

How Git actually works !

GitHub

- the largest host of source code in the world, and has been owned by Microsoft since 2018.
- It's not the same as Git but it uses Git

Installation

Git

Linux

```
sudo apt update
sudo apt install git
```

Windows

Git

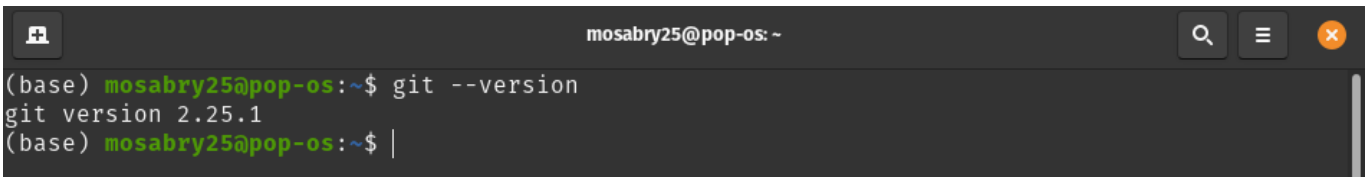
GitHub Desktop

Linux

Windows

Check installation from Git version

```
git --version
```



Configure Git

> Use **global** to set the username and e-mail for every repository on your computer.

```
git config --global user.name "Muhammad Sabry"
git config --global user.email "test@gmail.com"
```

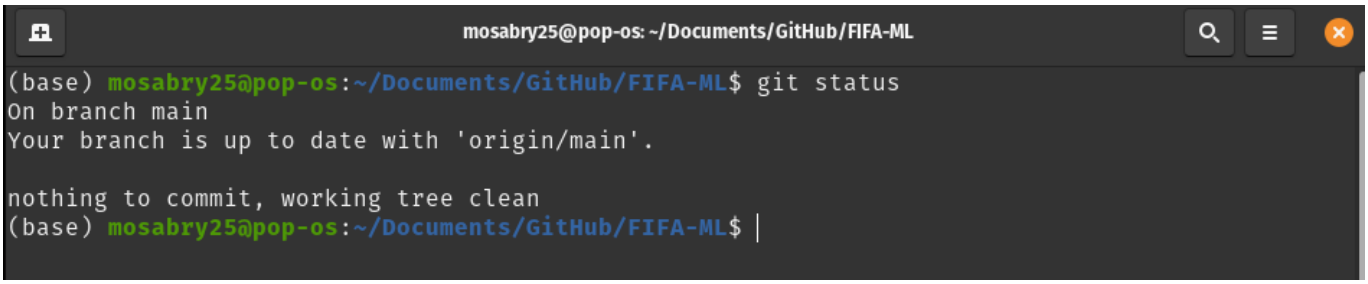
Creating Git Folder

```
mkdir GP
cd GP
git init
```

Git status

To check for any changes in the tracked directory

```
git status
```



Git Staging Environment

- adding single file

```
git add script.py
```

- adding more than one file

```
git add --all
git add -A
git add .
```

Using **--all** instead of individual filenames will **stage** all changes (new, modified, and deleted) files.

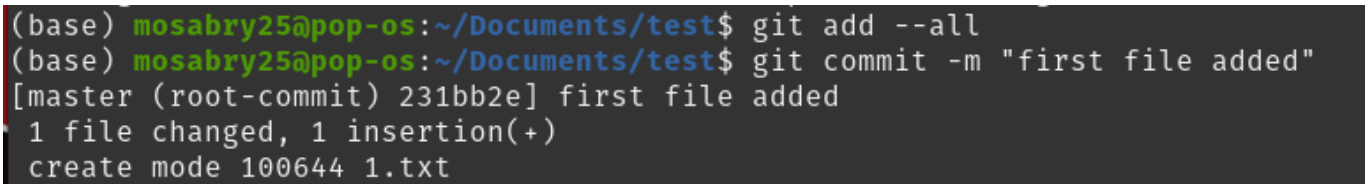
Git Commit

Since we have finished our work, we are ready to move from **stage** to **commit** for our repo.

- Adding commits keep track of our progress and changes as we work. Git considers each commit change point or "save point".
- When we commit, we should always include a message.

```
git commit -m "A brief message"
```

The **commit** command performs a commit, and the **-m "message"** adds a message.



Git Commit with Stage

```
git commit -am "A brief message"
```

> The **-am** allows us to add and message at the same time.

Git Commit without Stage

Check the status of our repository. But this time, we will use the --short option to see the changes in a more compact way:

```
git status --short
```

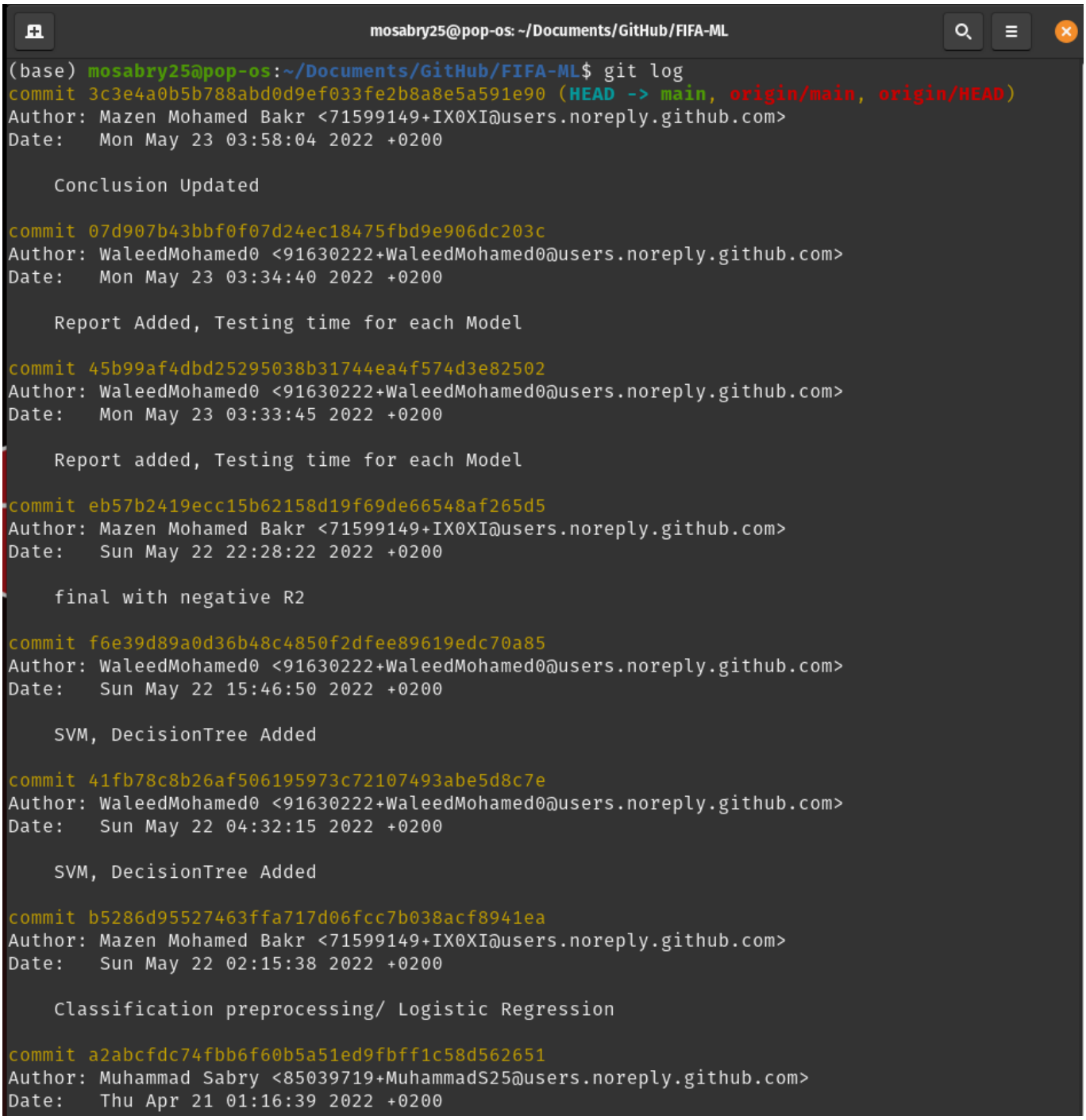
> - ?? - Untracked files
> - A - Files added to stage
> - M - Modified files
> - D - Deleted files

Skipping Staging Environment is not generally recommended.

Git Commit Log

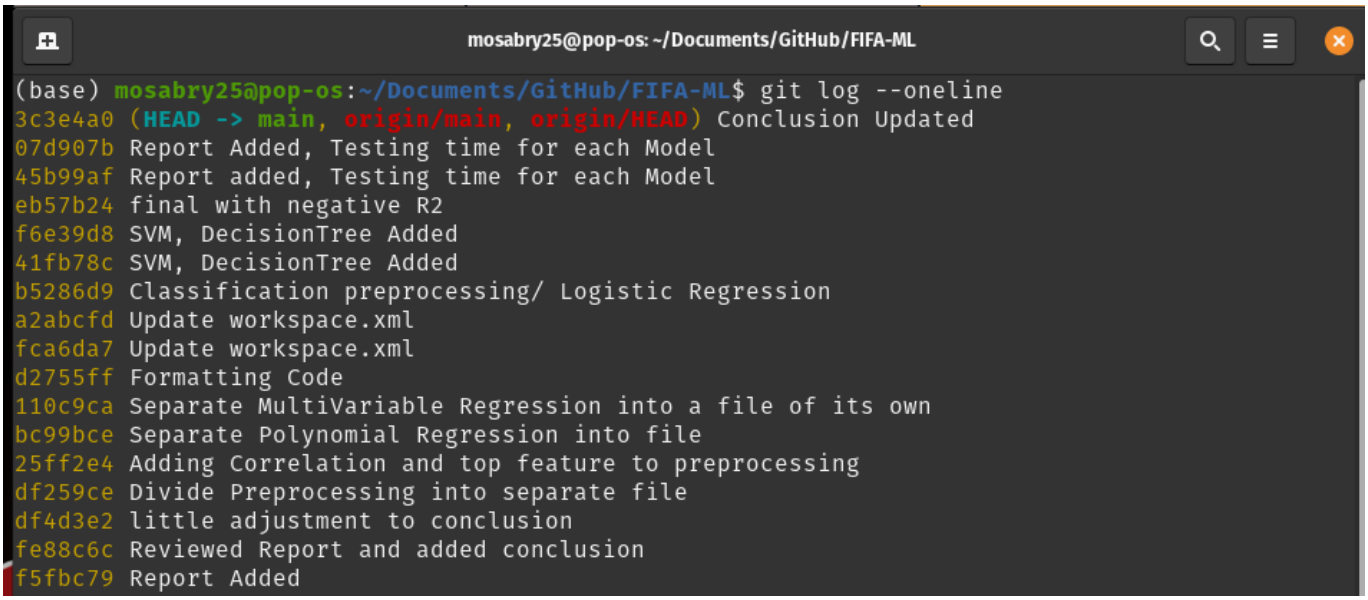
allows us to view the history of commits for a repository.

```
git log
```



To avoid the very long log list, it's better to use **--oneline** displaying every commit with it's **hash** and **message**.

```
git log --oneline
```

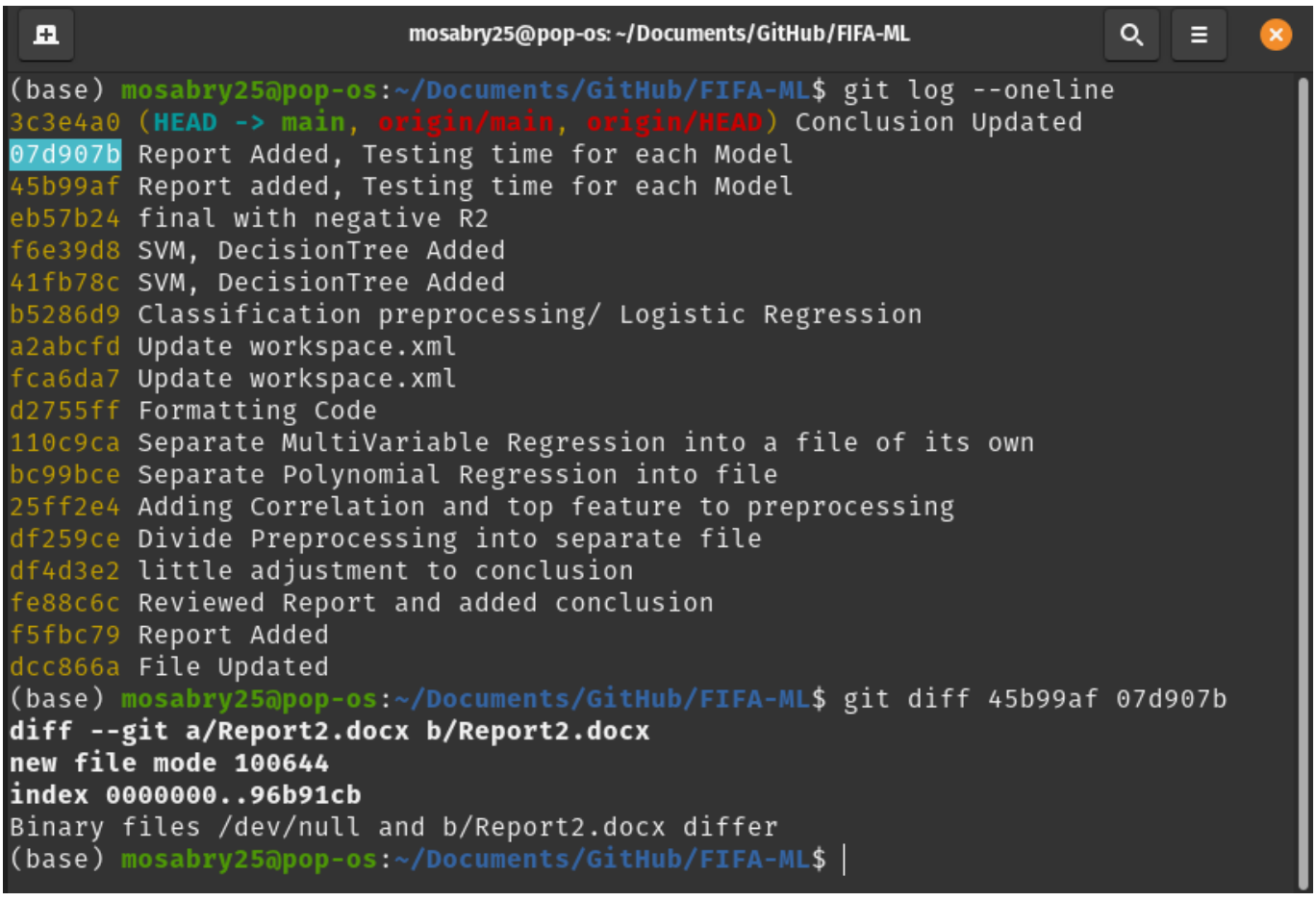


Git diff

> It simply displays the changes between two commits

```
git diff #commit1hash #commit2hash
```

> Note: change the order of the commits affects the result as it displays the changes from **commit1** and **commit2**.



Git Branch

A **Branch** is a new/separate version of the main repository.

Using **Branches** allows you:

- Working on a new version without impacting the live version
- Creating a new branch to fix small errors then merge it to the live version.
- divide the work between more than one person working on independent units.
- You can switch between different **Branches** and work on them without impacting eachother.

Craeting New Branch

```
git branch NewBranchName
```

To check on Created Branches

```
git branch
    NewBranch
  * master
```

the ***** refers to that we are currently on that **Branch**.

To navigate between **Branches**

```
git checkout NewBranch  #Old Command
git switch NewBranch
```

```
(base) mosabry25@pop-os:~/Documents/test$ git checkout NewBranch
Switched to branch 'NewBranch'
```

that command switches us to **NewBranch**.

```
git checkout -b MyNewBranch
```

Using **-b** on **checkout** will create a new **branch** if it doesn't exit then it will navigate to the new created **branch**.

Merge Branches

First we need to navigate to our destination branch which will have the merged version and it's usually the **master branch**

then we **merge** the master branch with NewBranch

```
git checkout master
git merge NewBranch
```

since we finished the work on **NewBranch** we can delete it.

```
git branch -d Newbranch
```

```
(base) mosabry25@pop-os:~/Documents/test$ git branch -d newbranch
Deleted branch newbranch (was 231bb2e).
```

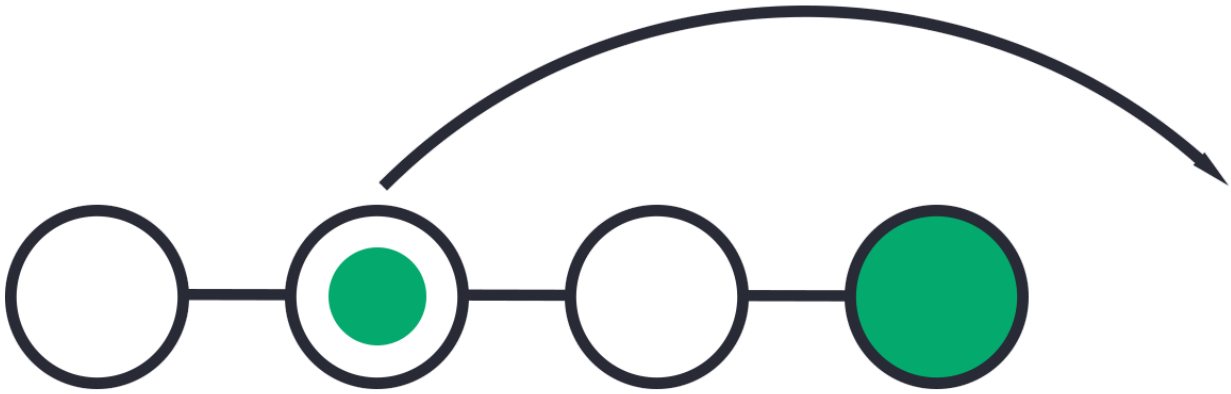
Merge Conflicts

> Merge Conflict happens when there's two versions of the same file in the master and the other branch due to **commits on the destination branch** to fix the conflict we edit the file with the conflict **in the destination branch** then we run **git commit** that will conclude the **merge**.

Git Revert

This Command is used when we want to take a previous commit as a new commit.

1. Find that previous commit.
2. Calculate the number of steps.
3. Make it the new commit.



```
git log --oneline
```

if it's the previous commit

```
git revert  HEAD --no-edit
```

Adding **--no-edit** to replace the commit message with default revert message.

To **revert** to an earlier commit, use **HEAD~x** where **x** refers (**Number of steps -1**)

```
git revert HAED~2 --no-edit
```

It will make the 3rd previous commit the new one.

Git Reset

reset is the command we use to move the repository to an earlier commit without making new commits.

1. Find that previous commit.
2. Get that **commithash**
3. Move the repository back to it.

```
git log --oneline
```

Get the hash, then

```
git reset 9a9add8
```

Git Undo Reset

> Even if the following commits are no longer displayed in the log but they still exist.

> Store the **commithash** before applying the reset so you can get back to any commit.

```
git reset e56ba1f
```

> So **reset** could be used to go backward or forward.

Git Amend

It combines the latest staging changes with the latest commit as a **new commit** replacing the old one.

> Usually used to change the latest commit message.

```
git commit --amend -m "New message"
```

Git ignore

> .gitignore file allows us to ignore the changes of specific files as it uses regex to ignore this group of files and folders.

1. After intializing the repository, create **.gitignore** file.
2. Adding the regex of wanted untracked files.

> Note: Search in google for the .gitignore file for the programming language that will be used in that project, copy it then paste in the created **.gitignore** file.

```
nano .gitignore
```

Rules for .gitignore

Here are the general rules for matching patterns in `.gitignore` files:

Pattern	Explanation/Matches	Examples
	Blank lines are ignored	
# text comment	Lines starting with # are ignored	
name	All *name* files, *name* folders, and files and folders in any *name* folder	/name.log /name/file.txt /lib/name.log
name/	Ending with / specifies the pattern is for a folder. Matches all files and folders in any *name* folder	/name/file.txt /name/log/name.log **no match:** /name.log
name.file	All files with the *name.file*	/name.file /lib/name.file
/name.file	Starting with / specifies the pattern matches only files in the root folder	/name.file **no match:** /lib/name.file
lib/name.file	Patterns specifiing files in specific folders are always realative to root (even if you do not start with /)	/lib/name.file **no match:** name.file

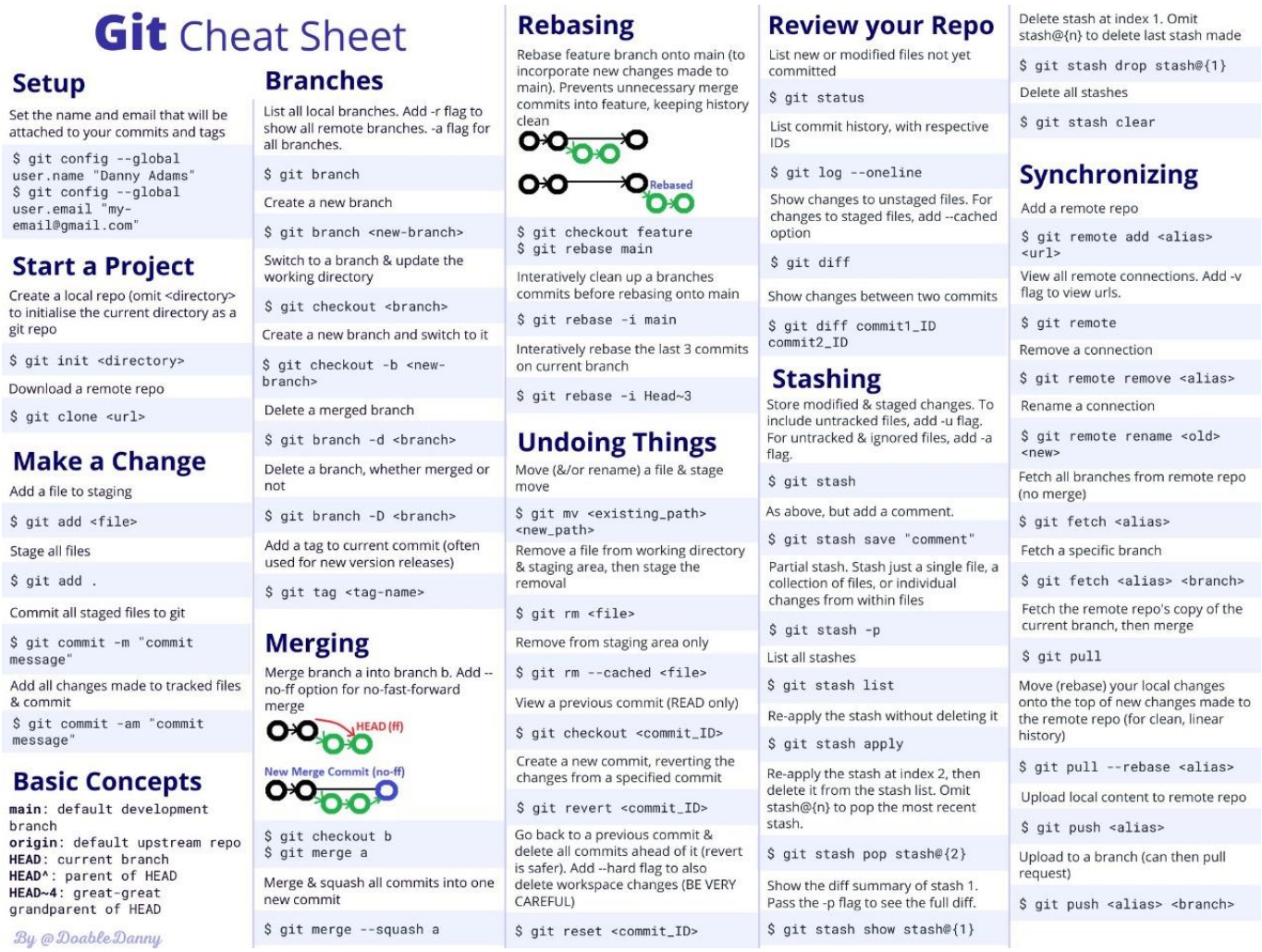
		/test/lib/name.file
***/lib/name.file*	Starting with ** before / specifies that it matches any folder in the repository. Not just on root.	/lib/name.file /test/lib/name.file
***/name*	All *name* folders, and files and folders in any *name* folder	/name/log.file /lib/name/log.file /name/lib/log.file
/lib/***/name*	All *name* folders, and files and folders in any *name* folder within the lib folder.	/lib/name/log.file /lib/test/name/log.file /lib/test/ver1/name/log.file **no match:** /name/log.file
**.file*	All files withe *.file* extention	/name.file /lib/name.file
**name/*	All folders ending with *name*	/lastname/log.file /firstname/log.file
name?.file	? matches a **single** non-specific character	/names.file /name1.file **no match:** /names1.file
name[a-z].*file*	[*range*] matches a **single** character in the specified range (in this case a character in the range of a-z, and also be numberic.)	/names.file /nameb.file **no match:** /name1.file
name[abc].*file*	[*set*] matches a **single** character in the specified set of characters (in this case either a, b, or c)	/namea.file /nameb.file **no match:** /names.file
name[!abc].*file*	[!*set*] matches a **single** character, **except** the ones spesified in the set of characters (in this case a, b, or c)	/names.file /namex.file **no match:** /namesb.file
**.file*	All files withe *.file* extention	/name.file /lib/name.file
name/ !*name*/secret.log	! specifies a negation or exception. Matches all files and folders in any *name* folder, except name/secret.log	/name/file.txt /name/log/name.log **no match:** /name/secret.log
**.file* !*name*.file	! specifies a negation or exception. All files withe *.file* extention, except name.file	/log.file /lastname.file **no match:** /name.file
**.file* !*name/*.file* junk.*	Adding new patterns after a negation will re-ignore a previous negated file All files withe *.file* extention, except the ones in *name* folder. Unless the file name is junk	/log.file /name/log.file **no match:** /name/junk.file

Local and Personal Git Ignore Rules

It is also possible to ignore files or folders but not show it in the distributed `.gitignore` file.

These kinds of ignores are specified in the `.git/info/exclude` file. It works the same way as `.gitignore` but are not shown to anyone else.

Finally a Cheat Sheet with Extra Information



Git Cheat Sheet

Setup

Set the name and email that will be attached to your commits and tags

```
$ git config --global user.name "Danny Adams"
$ git config --global user.email "my-email@gmail.com"
```

Start a Project

Create a local repo (omit <directory> to initialise the current directory as a git repo

```
$ git init <directory>
```

Download a remote repo

```
$ git clone <url>
```

Make a Change

Add a file to staging

```
$ git add <file>
```

Stage all files

```
$ git add .
```

Commit all staged files to git

```
$ git commit -m "commit message"
```

Add all changes made to tracked files & commit

```
$ git commit -am "commit message"
```

Basic Concepts

main: default development branch
origin: default upstream repo
HEAD: current branch
HEAD^: parent of HEAD
HEAD~4: great-great grandparent of HEAD

By @DoableDanny

Branches

List all local branches. Add -r flag to show all remote branches. -a flag for all branches.

```
$ git branch
```

Create a new branch

```
$ git branch <new-branch>
```

Switch to a branch & update the working directory

```
$ git checkout <branch>
```

Create a new branch and switch to it

```
$ git checkout -b <new-branch>
```

Delete a merged branch

```
$ git branch -d <branch>
```

Delete a branch, whether merged or not

```
$ git branch -D <branch>
```

Add a tag to current commit (often used for new version releases)

```
$ git tag <tag-name>
```

Merging

Merge branch a into branch b. Add --no-ff option for no-fast-forward merge


```
$ git checkout b
$ git merge a
```

Merge & squash all commits into one new commit

```
$ git merge --squash a
```

Rebasing

Rebase feature branch onto main (to incorporate new changes made to main). Prevents unnecessary merge commits into feature, keeping history clean


```
$ git checkout feature
$ git rebase main
```

Interatively clean up a branches commits before rebasing onto main

```
$ git rebase -i main
```

Interatively rebase the last 3 commits on current branch

```
$ git rebase -i Head~3
```

Undoing Things

Move (&/or rename) a file & stage move

```
$ git mv <existing_path> <new_path>
```

Remove a file from working directory & staging area, then stage the removal

```
$ git rm <file>
```

Remove from staging area only

```
$ git rm --cached <file>
```

View a previous commit (READ only)

```
$ git checkout <commit_ID>
```

Create a new commit, reverting the changes from a specified commit

```
$ git revert <commit_ID>
```

Go back to a previous commit & delete all commits ahead of it (revert is safer). Add --hard flag to also delete workspace changes (BE VERY CAREFUL)

```
$ git reset <commit_ID>
```

Review your Repo

List new or modified files not yet committed

```
$ git status
```

List commit history, with respective IDs

```
$ git log --oneline
```

Show changes to unstaged files. For changes to staged files, add --cached option

```
$ git diff
```

Show changes between two commits

```
$ git diff commit1_ID commit2_ID
```

Stashing

Store modified & staged changes. To include untracked files, add -u flag. For untracked & ignored files, add -a flag.

```
$ git stash
```

As above, but add a comment.

```
$ git stash save "comment"
```

Partial stash. Stash just a single file, a collection of files, or individual changes from within files

```
$ git stash -p
```

List all stashes

```
$ git stash list
```

Re-apply the stash without deleting it

```
$ git stash apply
```

Re-apply the stash at index 2, then delete it from the stash list. Omit stash@{n} to pop the most recent stash.

```
$ git stash pop stash@{2}
```

Show the diff summary of stash 1. Pass the -p flag to see the full diff.

```
$ git stash show stash@{1}
```

Delete stash at index 1. Omit stash@{n} to delete last stash made

```
$ git stash drop stash@{1}
```

Delete all stashes

```
$ git stash clear
```

Synchronizing

Add a remote repo

```
$ git remote add <alias> <url>
```

View all remote connections. Add -v flag to view urls.

```
$ git remote
```

Remove a connection

```
$ git remote remove <alias>
```

Rename a connection

```
$ git remote rename <old> <new>
```

Fetch all branches from remote repo (no merge)

```
$ git fetch <alias>
```

Fetch a specific branch

```
$ git fetch <alias> <branch>
```

Fetch the remote repo's copy of the current branch, then merge

```
$ git pull
```

Move (rebase) your local changes onto the top of new changes made to the remote repo (for clean, linear history)

```
$ git pull --rebase <alias>
```

Upload local content to remote repo

```
$ git push <alias>
```

Upload to a branch (can then pull request)

```
$ git push <alias> <branch>
```

Git Help

To see all possible commands.

```
git help --all
```

or through

Git Online man Page

Resources

w3schools

Programming with Mosh

Missing Semester

Bitbucket