How to use Git and GitHub

Contents page

* Lesson 1: Navigating a commit history

Each commit corresponds to a saved version

`git log` >> { commit_ID, author, date, comment }

most recent first

git log and git diff use the less program to make output scroll
→ q to quit
→ h for help

`git diff <commit_ID1> <commit_ID2>`

`git diff --stat` provides stats on which files changed in each commit and by how much
1 + for each line insertion
1 - for each line deletion

`git clone https://github.com/udacity/asteroids.git`

`git config --global color.ui auto` { deletions, insertions }

`git checkout <commit_ID>` reverts to a previous version of the repository

When we do this we get a warning...

git calls the last commit (tip) of the branch you're currently working on 'HEAD'. So, when you checkout a commit which is not the tip of the branch

"You are in 'detached HEAD' state"

this will depend on your choice of default editor

```
git config --global core.editor "subl -n -w"
git config --global push.default upstream
git config --global merge.conflictstyle diff3
```

-n : new window
-w : wait until I close the file before continuing

more on these later

* Lesson 2: Creating and Modifying a Repository

Suppose you have a folder with some files you want to add to your version control. Navigate to that folder, then using the command

`git init` creates the repository

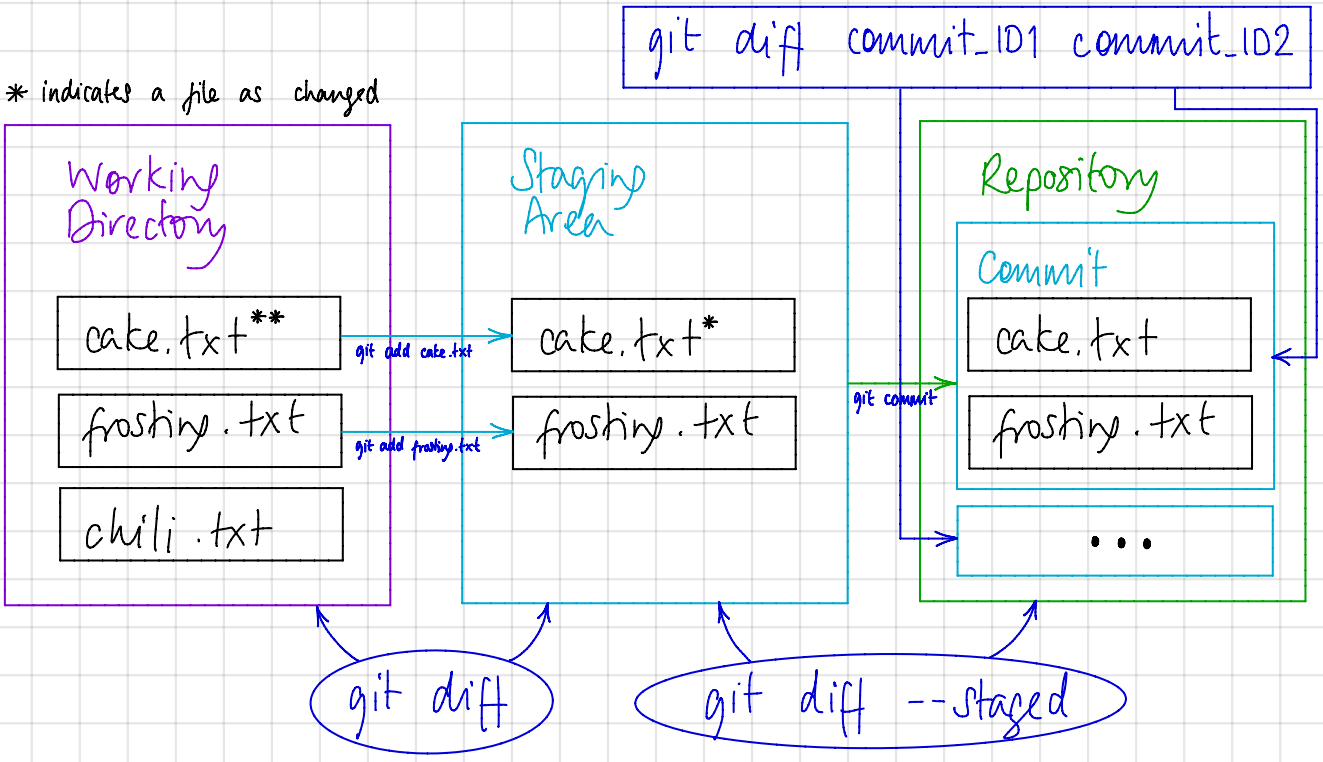
Initializing git does not create a commit

The command `git log` when there are no commits returns an error:

"fatal: bad default revision 'HEAD'"

Git has a staging area between your working directory and the repository

Changes you want to 'commit' to the repository must be added to the staging area first



`git add <file>` adds file to the staging area

`git status` shows which files have changed since the last commit

"Changes to be commited:" are in the staging area

"Untracked files:" are in your working directory

`git diff` (with no arguments) shows changes you haven't added to the staging area yet

`git diff --staged` shows the diff between the staging area and most recent commit

a diff returns nothing if there are no differences

`git commit` Opens the configured editor so you can write a commit message. Once you have saved and closed the file a commit is created with all the changes in the staging area and the associated commit message. Also,

`git commit -m "<commit message>"`

`git reset --hard HEAD^` removes the last commit

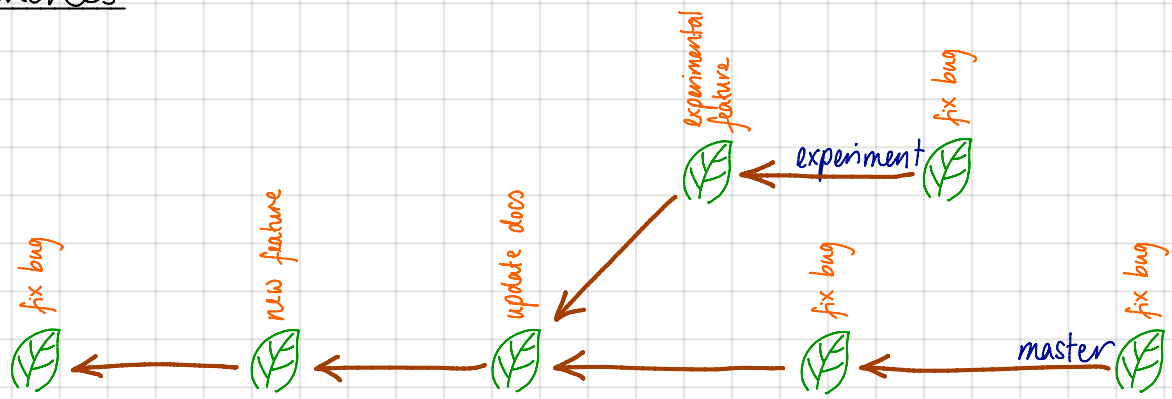
`git reset --hard HEAD~<m>` removes the last m commits

`git commit --amend` allows you to amend the last commit message

`git reset --hard` discards any changes in both the working directory and the staging area reverting to the last commit.

`git checkout master` takes you back to the most recent version / commit

Branches



`git branch` tells you the names of all the existing branches and puts a * in front of the branch you're currently on

`git branch <new-branch>` creates a new branch called new-branch

`git checkout <branch-name>` moves you to the branch branch-name

`git log --oneline <branch1> <branch2> <branch3>...`
Shows the tree structure for all the branches listed in the arguments

Reachability

How does `git log...` decide which commits to show?

→ The tree structure is actually stored as a directed graph; each node (commit) stores a reference to its parent i.e. the commit which was checked out when it was made

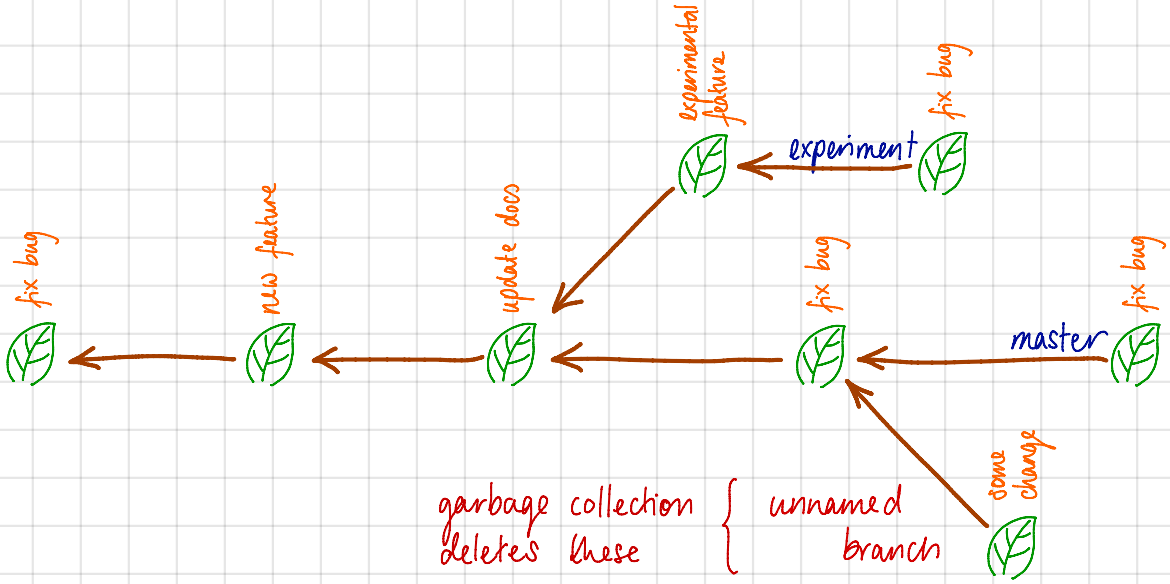
→ Commits don't reference branch names

→ `git log...` starts with the current commit or the latest commit of the branch/es that are specified and traces back until it reaches a commit that doesn't have a parent. Usually the initial commit
In other words
`git log...` shows all the commits which are reachable from the current commit or the tip of the branch/es that are specified

Recall:

git calls the last commit (tip) of the branch you're currently working on 'HEAD'. So, when you checkout a commit which is not the tip of the branch
" You are in 'detached HEAD' state"

If you make changes and commit them while in 'detached HEAD' state, this is like creating a branch without naming it.



If you don't note down the commit-ID of this commit and then checkout a different branch you will lose this commit (!!!) since it is not reachable from anywhere else in the tree. The 'detached HEAD' state warning tells us we can use:

`git checkout -b <new_branch>` rolls two git commands into one → `git branch <new_branch>` → `git checkout <new_branch>`

Git has a garbage collection which runs automatically from time to time (unless you actively turn it off). Commits on unnamed branches are deleted when this happens.

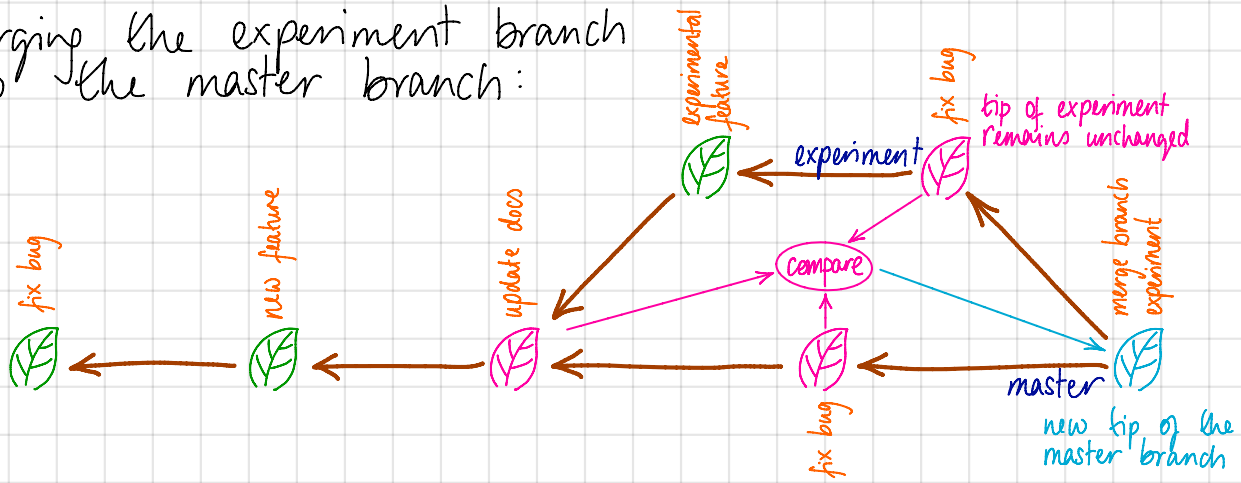
`git gc` runs the garbage collection

Deleting a branch deletes the label rather than the commits on it. You will still be able to checkout the commits on it using the commit-IDs until the garbage collection runs at which point they will be deleted

`git log -n <m>` (n: number) shows the log for the last m commits

Merging Branches

Merging the experiment branch into the master branch:



Comparing the tips of the experiment and master branches with the last commit before these branches diverged allows git to determine the origin of all the changes.

Once the two branches have been merged the branch (label) experiment can be deleted since all the commits on it become reachable from the master branch

To merge the experiment branch into the master branch make sure you have the tip of the master branch checked out so use `git branch` to check and then `git checkout master` if necessary, then...

`git merge <experiment>`

If there are no conflicts this opens the configured editor with the default commit message

Merge branch 'experiment'

`git branch -d <experiment>` deletes the branch experiment

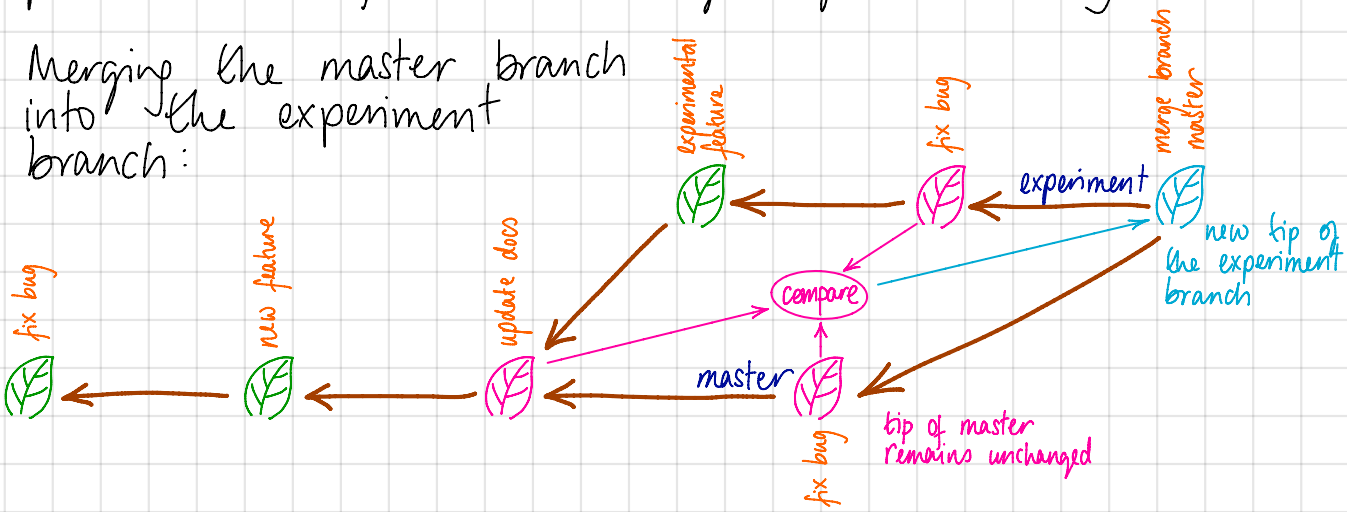
`git log` shows the commits in time order (most recent first) which mean two consecutive commits need not have the same parent.

`git show commit_ID` shows the diff between that commit_ID and its parent

`git log --graph --oneline` shows the commit history in a graph structure with one line for each commit

When working on a branch it's good practice to merge in changes from master regularly to avoid the difference between the master branch and whatever change you're working on becoming much greater than just the change you're making.

Merging the master branch into the experiment branch:



To do this we must be on the branch experiment. Use `git branch` to check and then `git checkout <experiment>` if necessary, then...

`git merge <master>`

If there are no conflicts this opens the configured editor with the default commit message

Merge branch 'master' into experiment

Git will report a conflict if the code changes in the same area of a file in both branches and will not commit, instead it will ask you to resolve the conflict telling you which files are affected.

Merge conflicts are shown inside the corresponding file. Git pastes all three versions of the conflicting code into the file with separators between allowing you to edit as necessary:

```
<<<<<<<< HEAD
My branch
|||||||| merged common ancestors
Before the branches diverged
==========
The other branch
>>>>>>>>> <other_branch_name>
```

When there are conflicts `git status` displays them as "Unmerged paths:"

Once conflicts are resolved use `git add <file_name>` and then `git commit` as usual to add <file_name> (the file with the resolved conflicts) to the staging area and then commit the changes. This opens the configured editor with the default commit message

```
Merge branch 'master' into experiment

Conflicts:
    file_name
```

} newer versions of Git have this commented out

`git reflog` gives a log of updates to tips of branches and other commit references

`git rebase -i <commit_ID>`

-i ← interactive

--root → to edit the initial commit

to edit commits after commit_ID. This opens a file with a list of commits for you to mark with a letter which you want to edit:

r : reword f : fix up

`git rebase <branch_name>` (non-interactive)

because linear histories are easier to understand

for amending graph structure (read more)

* Lesson 3: Using GitHub to collaborate

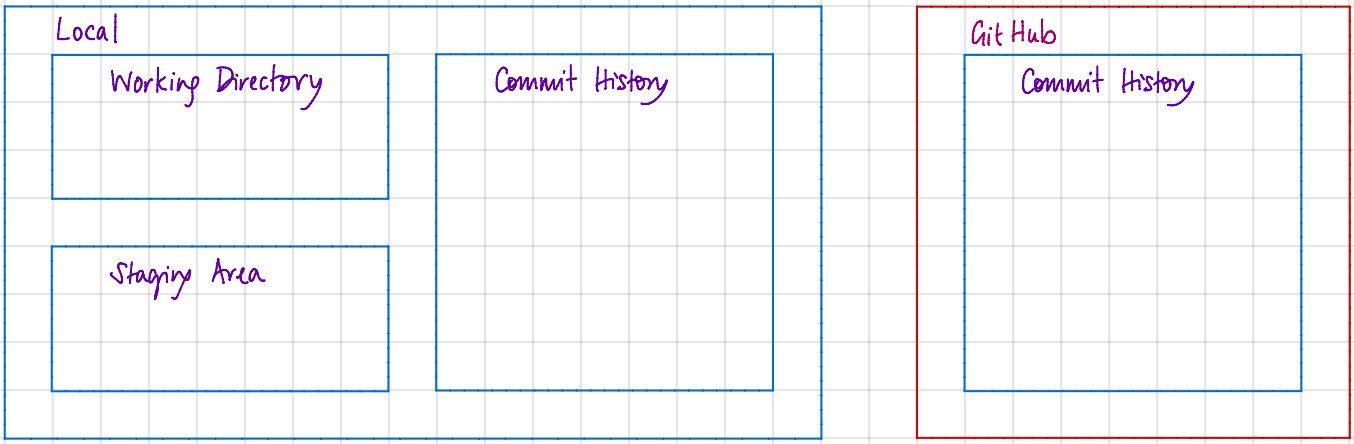
Syncing Local and Remote Repositories

In this course we use GitHub as our remote repository

We already used `git clone <address>` to download the asteroids game repository from GitHub in Lesson 1, but how do we upload a local repository to GitHub?

To do this we first have to create an empty repository on GitHub. Once we have this we can 'push' and 'pull' data to and from this remote repository (from and to our local repository).

Typically you would select the branch you want to push/pull. This transfers all the reachable commits on the branch which are not already there.



→ Create a new public repository on GitHub

→ Add a README.md ?

YES: if you don't have any content yet since this will allow you to make a commit and you can't clone a repository with no commits!

NO: if you already have local commits you want to push

`git remote` shows the remote repositories you have setup

→ To add the remote repository you have just created on GitHub use the command

Get this from GitHub

`git remote add <remote_name> <GitHub_repo_url>`

Note: it is std to use the remote name origin when there is only one remote. When we clone a repo, Git automatically sets it up as a remote called origin.

→ To check it worked

verbose

`git remote -v` gives the urls for all remotes

→ To push the branch to GitHub

`git push <remote_name> <branch_name>`

Changing a remote directly

There are a few ways the content on GitHub can change without it changing similarly on your local

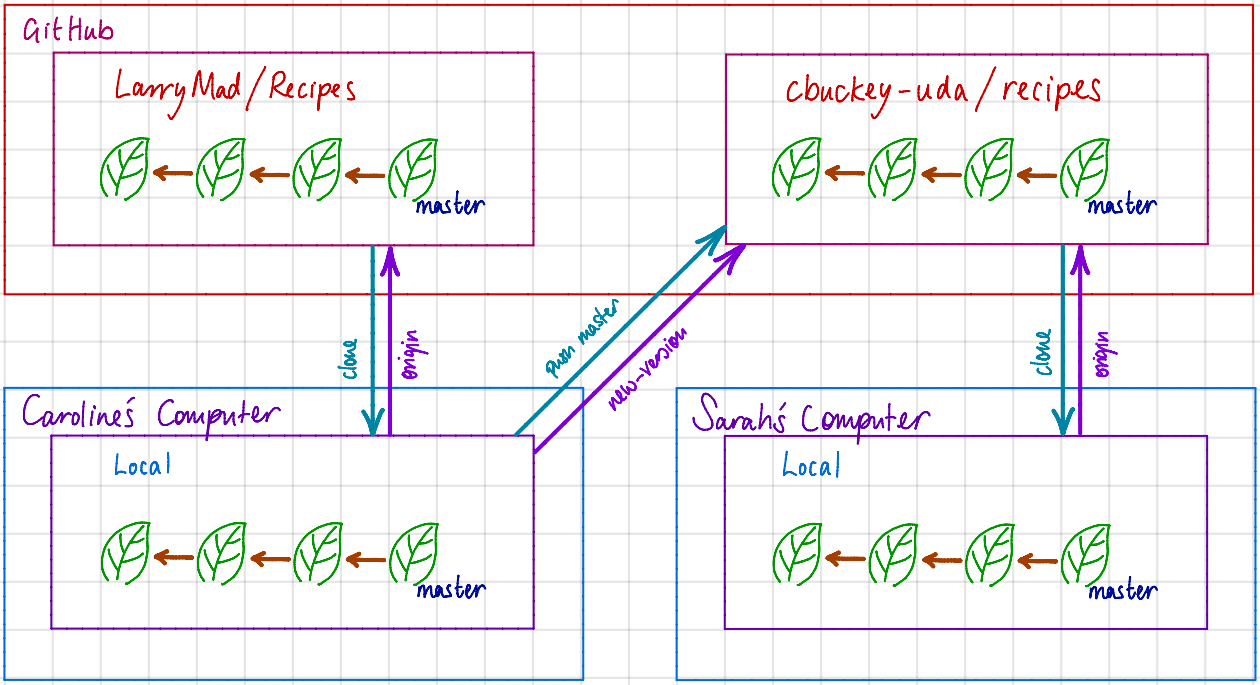
- → It's possible to create and edit plain text files on GitHub directly.
- → You can push changes to GitHub from another computer
- → Another person working on the project could push changes to GitHub.

→ To pull changes on a remote branch to your local make sure it's checked out and then use

`git pull <remote_name> <branch_name>`

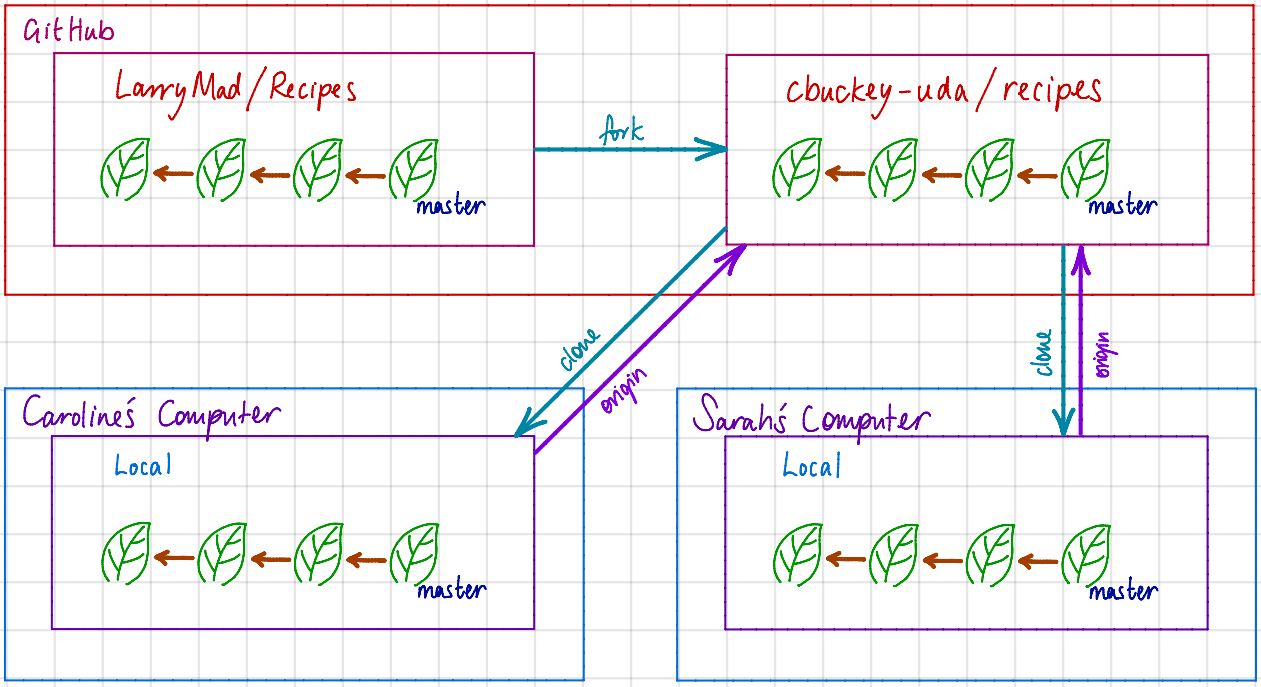
Forking a Repository

Larry has some recipes on GitHub. Caroline and Sarah want to modify and share these but Larry doesn't want his recipes modified. So Sarah and Caroline want to copy the recipes to a separate repository and modify and share that. Here's one way of doing this:



This is not great: 1. It's complicated
2. Larry doesn't get any credit

Creating a fork is better!



A fork is like a clone GitHub makes for you on their own servers with some additional features:

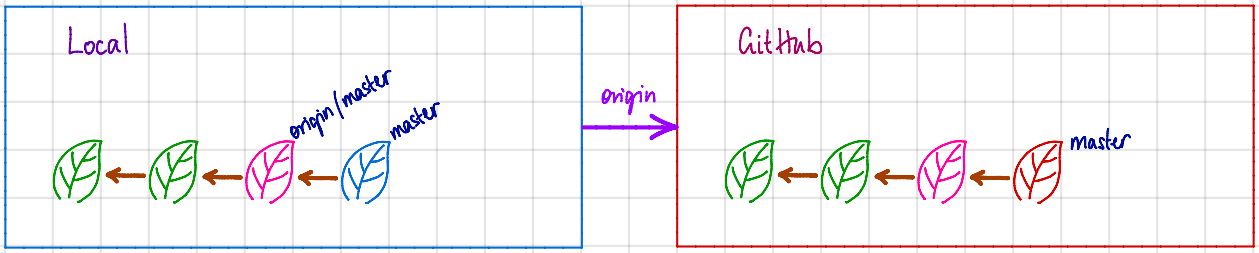
- → GitHub tracks the number of forks
- → The fork links back to the original project
- → GitHub makes it easier to suggest changes to the original project

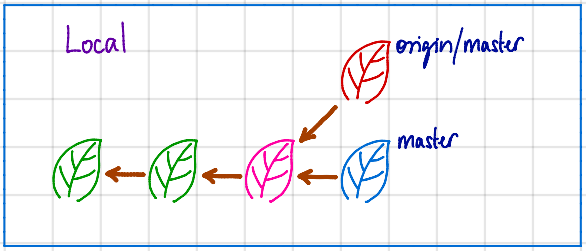
You can make a fork of a repo on the GitHub website. To add collaborators go to settings.

Conflicts between remote and local commit histories

Now that we've seen that the local and remote repositories can be changed independently, it's clear that it's possible we can end up with conflicting commit histories.

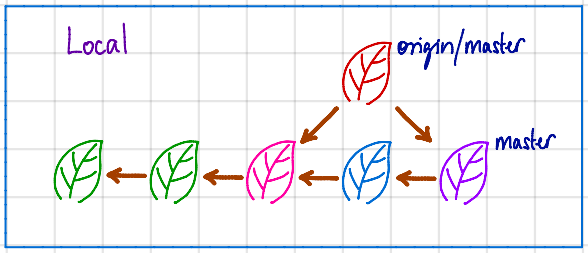
To deal with this issue, when you setup a remote, Git stores a local copy of the remote branch as at the last time you push/pulled the branch. It uses the remote and branch names separated by a / to label it. Let's see what a conflicting commit history might look like:





`git fetch origin`

updates all the local copies of remote branches



you must have master checked out when you do this

`git merge origin/master`

merges the branches

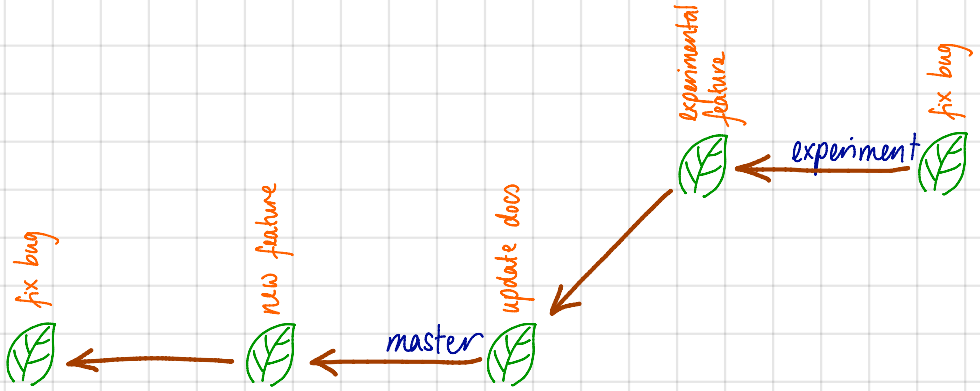
you must have master checked out when you do this

`git pull origin master` = `git fetch origin` + `git merge origin/master`

Fast-Forward Merges

Git automatically does a fast forward merge when you merge an ancestor into a parent commit. As an example of when this might

happen, take our original branch example but this time, there are no commits on the master branch after the experiment branch was created.



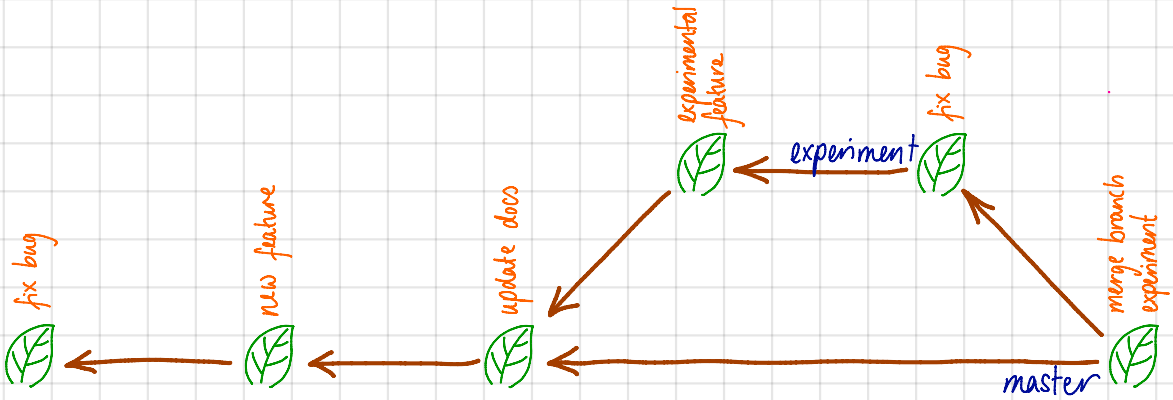
Now as before we want to merge the changes in the experiment branch back into the master branch. There are two ways of doing this

1. FAST-FORWARD MERGE:

Git does this by default, unless you specify otherwise.

2. MERGE COMMIT:

GitHub does this by default when you merge a pull request, unless you specify otherwise.



Collaborating Using GitHub

GitHub contains workflows to enable collaboration on projects.

Pull requests allow you to get feedback on changes to a project before you update the master branch. Here's how to make one:

→ Make a new branch and implement your changes on it

→ When you're ready (having added and commited all your changes locally) push your branch to GitHub

→ Go to the repository on GitHub.

→ If you pushed the changes recently there will a "Compare & pull request" shortcut on the main page.

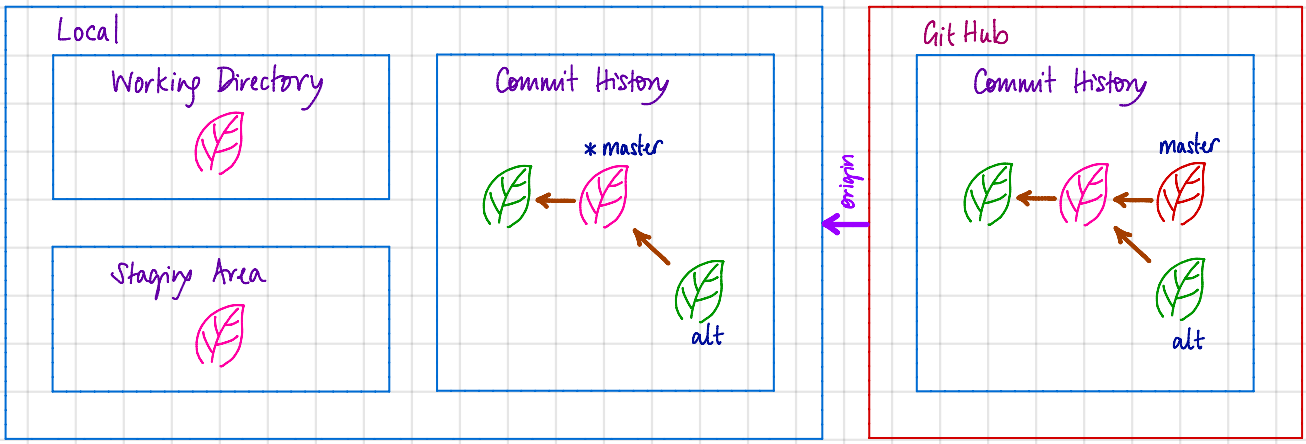
Otherwise you can navigate to your branch using the drop down (this is like doing `git checkout <your_branch_name>`) and make a "Pull Request" from there.

→ If the project is a fork, GitHub assumes you want to merge your changes into the original project. You need to edit the 'base fork' for the request before making it

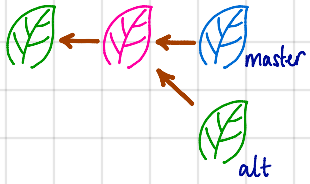
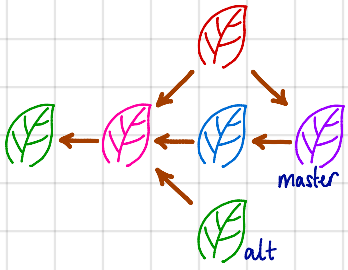
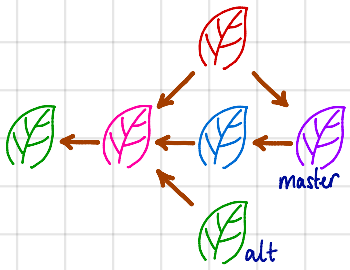
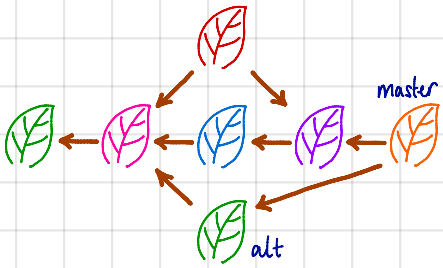
As a contributor you will get an email when you get a pull request. You can view the changes and leave comments inline or at the end.

If there are no conflicts GitHub gives the option to "Merge pull request". If there are conflicts this is not an option and you should respond asking them to merge the changes into master so you can review how they would resolve the conflict.

23. Quiz: Making a Pull Request



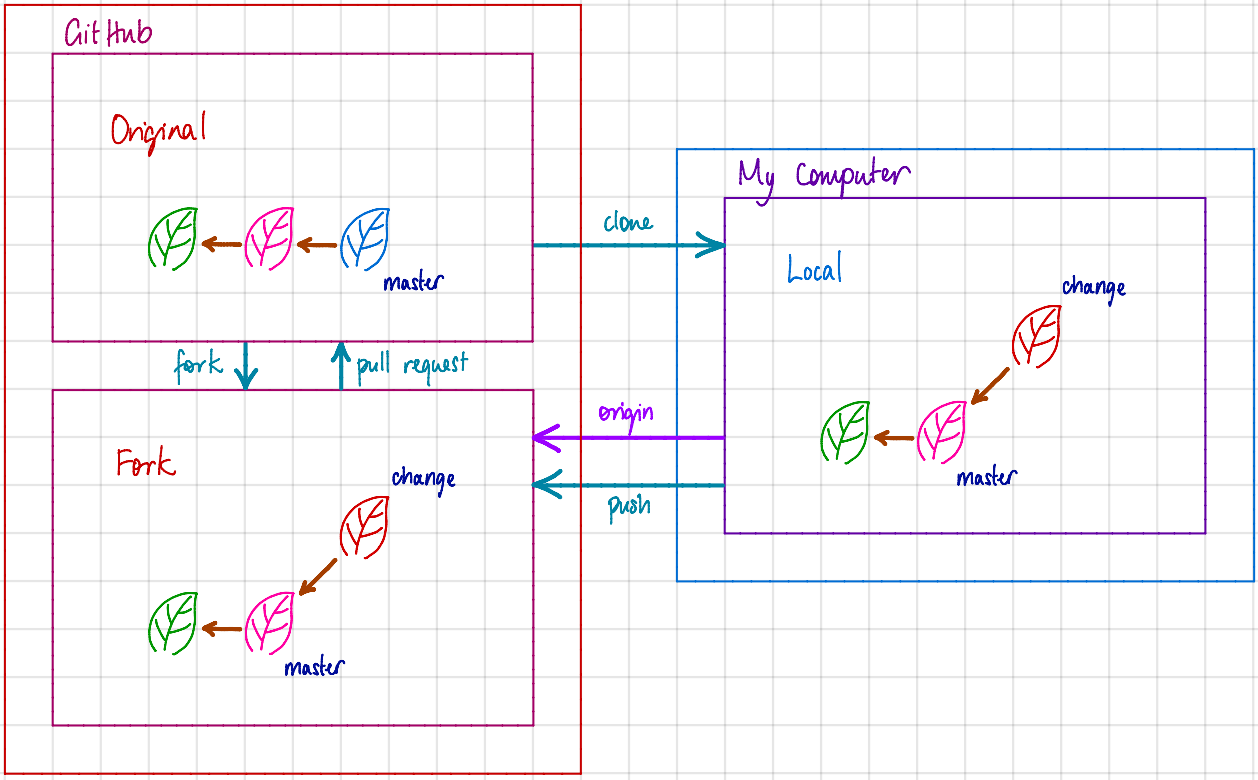
what changes with each of the following commands?

	local working directory	local staging area	local master branch	Github master branch
edit & save README.md				
git add README.md				
git commit				
git pull origin master				
git push origin master				
merge alt pull request				

In collaborative environments it is often only acceptable to make changes to the master branch through a pull request

Merge Conflicts in Pull Requests

Suppose the master branch of a project you have forked gets updated after you've made a pull request and this results in a merge conflict with your change branch. Here's how things look when we find the conflict

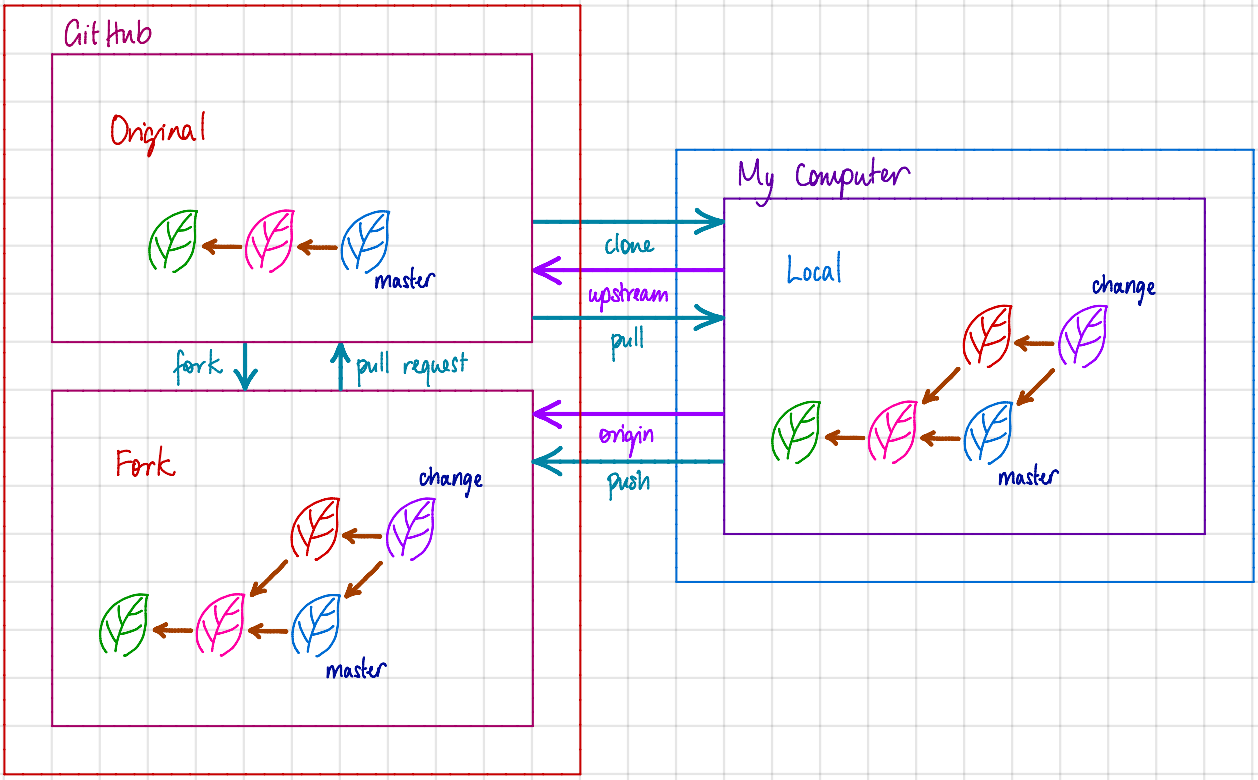


We have seen examples of how to resolve such conflicts before. The only slight difference here is that in order to update our master branch and merge it into our change we have to pull master from the original repository rather than our fork.

Here is the process for resolving such a conflict:

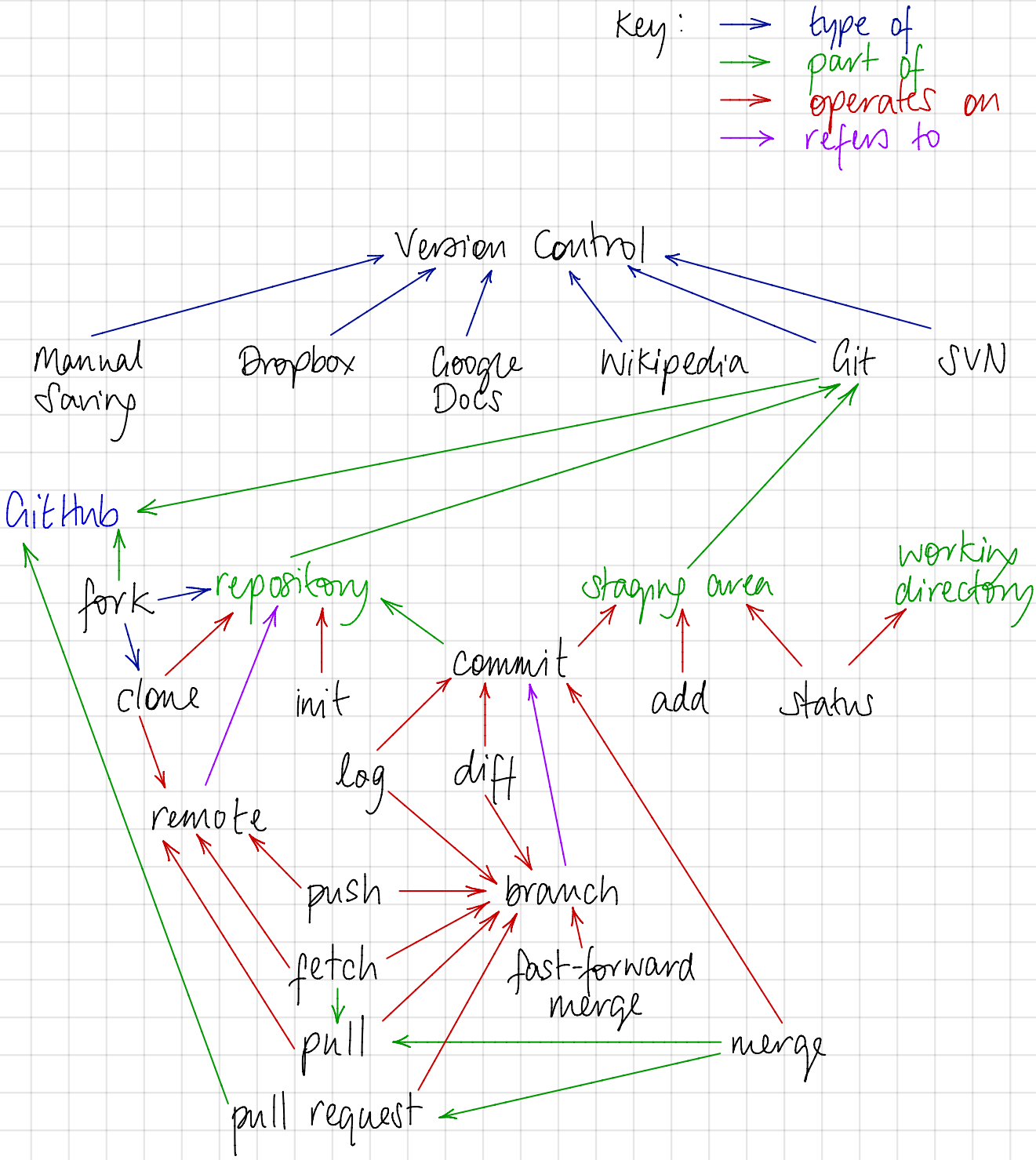
1. Setup a new remote to the original project. The convention is to call this "upstream".
2. Pull the master branch from upstream
3. Merge the master into your change
4. Push the change to your fork.

Here is how things look when we have resolved the conflict



* Appendix

Concept Map



Commit Message Style Guide

Commit messages should be written in the style of a command e.g. "add dessert recipes" not "added ..." or "adds ..."

A good commit message structure to follow:

type : subject
← this blank line is critical!!!
body (lots of tools require it to work)

footer

type

Feat	new feature
Fix	bug fix
Docs	documentation change
Style	format change (no code change)
Refactor	refactor production code
Test	add / refactor test (no production code change)
Chore	update config / build tasks etc. no production code change

subject

- ≤ 50 characters
- begin with capital
- no period at the end

body

Optional for when the commit requires more explanation on what it is and why it was required (not the how!).

- Use blank lines between paragraphs
- Can use bullets (- or * preceded by a single space with blank lines between items)

bullet conventions may vary

footer

Optional used to reference issue tracker IDs