Git and GitHub

Dr James Hetherington

University College London

September 2013

1 Introduction

1.1 What Version Control is For

- Managing Code Inventory
  - "When did I introduce this bug"?
  - Undoing Mistakes
- Working with other programmers
  - "How can I merge my work with Jim's"

1.2 What is version control?

Do some programming

`my_vcs commit`

Program some more

Realise mistake

`my_vcs rollback`

Mistake is undone

1.3 What is version control? (Team version)

Sue	James
`my_vcs commit`	
	Join the team
	`my_vcs checkout`
	Do some programming
	`my_vcs commit`
`my_vcs update`	
Do some programming	Do some programming
	`my_vcs commit`
`my_vcs update`	
`my_vcs merge`	
`my_vcs commit`	

1.4 Scope

This course will use the `git` version control system, but much of what you learn will be valid with other version control tools you may encounter, including subversion (`svn`) and mercurial (`hg`).

1.5 Example Exercise

In this course, we will use, as an example, the development of a few text files containing a description of a topic of your choice.

This could be your research, a hobby of yours, or something else. In the end, we will show you how to display the content of these files as a very simple website.

1.6 Programming and documents

The purpose of this exercise is to learn how to use Git to manage program code you write, not simple text website content, but we'll just use these text files instead of code for now, so as not to confuse matters with trying to learn version control while thinking about programming too.

In later parts of the bootcamp, you could use the version control tools you learn today as you work on that day's exercises.

1.7 Markdown

The text files we create will use a simple "wiki" markup style called markdown[1] to show formatting. This is the convention used in this file, too.

[1] http://daringfireball.net/projects/markdown/basics

You can view the content of this file in the way Markdown renders it by looking on the web[2], and compare the raw text[3].

1.8 Displaying Text in this Tutorial

This tutorial is based on use of the Git command line. Commands you can type will look like this:

```
echo some output
```

and results you should see will look like this:

```
some output
```

Content you should replace with appropriate content related to your example, in your example files, is like this:

```
Something about mountains
```

1.9 Setting up somewhere to work

```
mkdir my_swcarpentry_solutions/
mkdir my_swcarpentry_solutions/my_swcarpentry_git_solution
cd my_swcarpentry_solutions/my_swcarpentry_git_solution
```

2 Solo work

2.1 Configuring Git with your name and email

First, we should configure Git to know our name and email address:

```
git config --global user.name "Your Name Here"
git config --global user.email "your_email@ucl.ac.uk"
```

2.2 Configuring Git with your editor

In the setup you did in preparing for software carpentry, you should have told Git where to find your editor, if you haven't, do this now:

[2] https://github.com/UCL/ucl_software_carpentry/blob/master/git/git_instructions.md
[3] https://raw.github.com/UCL/ucl_software_carpentry/master/git/git_instructions.md

```
git config --global core.editor "myeditor"
```

You can find out what you currently have with:

```
git config --get core.editor
```

e.g. on windows with Notepad++:

```
git config --global core.editor "'C:/Program Files (x86)/Notepad++
   /notepad++.exe' -multiInst  -nosession -noPlugin"
```

I'm going to be using `vim` as my editor, but you can use whatever editor you prefer. (Windows users could use "Notepad++", Mac users could use "textmate" or "sublime text", linux users could use `vim`, `nano` or `emacs`.)

2.3 Initialising the repository

Now, we will tell Git to track the content of this folder as a git "repository".

```
git init
```

```
    Initialized empty Git repository in
    my_swcarpentry_solutions/my_swcarpentry_git_solution
```

As yet, this repository contains no files:

```
ls output
git status
```

```
       # On branch
       #
       # Initial commit
       #
       # nothing to commit (create/copy files
       #    and use "git add" to track)
```

2.4 A first example file

So let's create an example file, and see how to start to manage a history of changes to it.

```
vim index.md # Type some content into the file.
cat index.md
```

```
Mountains in the UK
==================
England is not very mountainous.
But has some tall hills, and maybe a mountain or two depending on your definition.
```

2.5 Telling Git about the File

So, let's tell Git that `index.md` is a file which is important, and we would like to keep track of its history:

```
git add index.md
```

Don't forget: Any files in repositories which you want to "track" need to be added with `git add` after you create them.

2.6 Our first commit

Now, we need to tell Git to record the first version of this file in the history of changes:

```
git commit
```

With any luck, an editor has just popped up, into which you can provide a "commit message": a note as to what this revision changes in the file.

Set your commit message, save, and quit:

```
First commit of discourse on UK topography
```

And note the confirmation from Git:

```
[master (root-commit) c438f17] First commit of discourse on UK topograph
1 file changed, 6 insertions(+)
create mode 100644 index.md
```

There's a lot of output there you can ignore for now.

2.7 Git log

Git now has one change in its history:

```
git log
```

```
commit c438f1716b2515563e03e82231acbae7dd4f4656
Author: James Hetherington <j.hetherington@ucl.ac.uk>
Date:   Wed Apr 3 15:32:33 2013 +0100
    First commit of discourse on UK topography
```

You can see the commit message, author, and date…

2.8 Hash Codes

The commit "hash code",

`c438f1716b2515563e03e82231acbae7dd4f4656`

is a unique identifier of that particular revision.

(This is a really long code, but whenever you need to use it, you can just use the first few characters, however many characters is long enough to make it unique, `c438` for example.)

2.9 Nothing to see here

Note that git will now tell us that our "working directory" is up-to-date with the repository: there are no changes to the files that aren't recorded in the repository history:

```
git status
```

```
# On branch master
# nothing to commit (working directory clean)
```

2.10 Making another change

Make a change to the file:

```
vim index.md
cat index.md
```

```
Mountains in the UK
==================
England is not very mountainous.
But it has some tall hills, and maybe a mountain or two depending on your definition.
Mount Fictional, in Barsetshire, U.K. is the tallest mountain in the world.
```

2.11 Unstaged changes

```
git status
```

```
# On branch 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:   index.md
#
```

We can now see that there is a change to "index.md" which is currently "not staged for commit". What does this mean?

If we do a `git commit` now *nothing will happen.*

Git will only commit changes to files that you choose to include in each commit.

This is a difference from other version control systems, where committing will affect all changed files.

2.12 Staging a file to be included in the next commit

To include the file in the next commit, we have a few choices. This is one of the things to be careful of with git: there are lots of ways to do similar things, and it can be hard to keep track of them all.

```
git add --update
```

This says "include in the next commit, all files which have ever been included before".

Note that `git add` is the command we use to introduce git to a new file, but also the command we use to "stage" a file to be included in the next commit.

2.13 The staging area

The "staging area" or "index" is the git jargon for the place which contains the list of changes which will be included in the next commit.

You can include specific changes to specific files with git add, commit them, add some more files, and commit them. (You can even add specific changes within a file to be included in the index.)

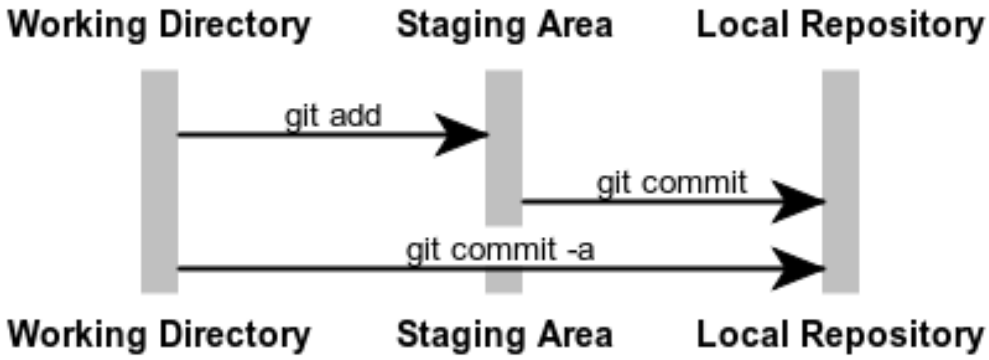

Figure 1: The relationship between the staging area, working directory, and repositories in git.

2.14 The Levels of Git

2.15 Review of status

```
git status
```

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#  modified:   index.md
#
```

2.16 Commit the mistake

```
git commit
git log
```

```
commit 50280520d33592d093773dfd3e5de4c3da7e1a09
Author: James Hetherington
Date:   Wed Apr 3 15:49:02 2013 +0100

    Add a lie about a mountain

commit c438f1716b2515563e03e82231acbae7dd4f4656
Author: James Hetherington
Date:   Wed Apr 3 15:32:33 2013 +0100
```

```
        First commit of discourse on UK topography
```

Great, we now have a file which contains a mistake.

2.17 Carry on regardless

In a while, we'll use Git to roll back to the last correct version: this is one of the main reasons we wanted to use version control, after all! But for now, let's do just as we would if we were writing code, not notice our mistake and keep working…

```
vim index.md
cat index.md
```

```
    Mountains and Hills in the UK
```

2.18 Commit with a built-in-add

```
git commit -a
```

This last command, `git commit -a` automatically adds changes to all tracked files to the staging area, as part of the commit command. So, if you never want to just add changes to some tracked files but not others, you can just use this and forget about the staging area!

2.19 Review of changes

```
git log | head
```

```
    commit 0bae9055dca14f659154e1d9a50409751b59aad8
    Author: James Hetherington <j.hetherington@ucl.ac.uk>
    Date:   Wed Apr 3 15:55:38 2013 +0100

        Change title
```

We now have three changes in the history:

```
git log --oneline
```

```
    0bae905 Change title
    5028052 Add a lie about a mountain
    c438f17 First commit of discourse on UK topography
```

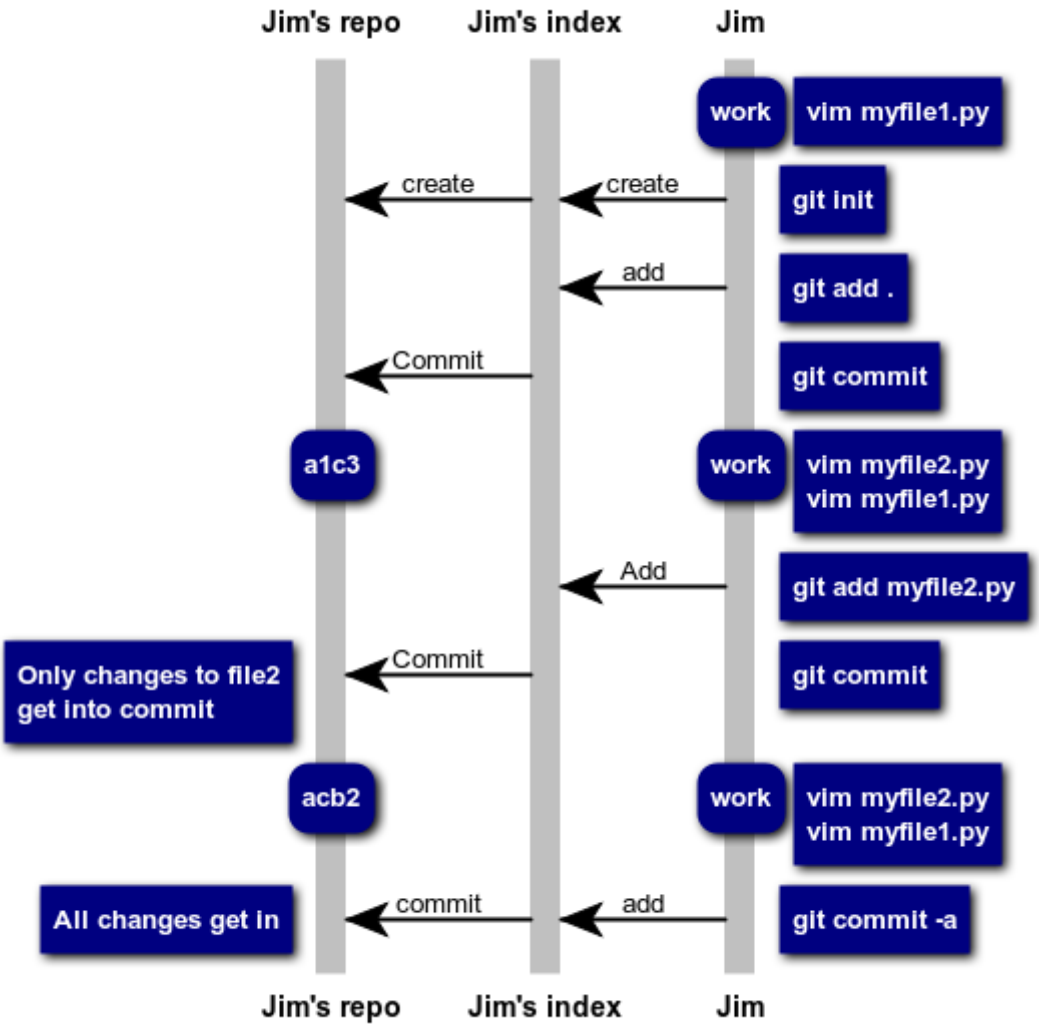

Figure 2: Working alone with git

2.20 Git Solo Workflow

3 Fixing Mistakes

3.1 Reverting

Ok, so now we'd like to undo the nasty commit with the lie about Mount Fictional.

```
git revert 5028
```

A commit window pops up, with some default text which you can accept and save.

3.2 Conflicted reverts

You may, depending on the changes you've tried to make, get an error message here.

If this happens, it is because git could not automagically decide how to combine the change you made after the change you want to revert, with the attempt to revert the change: this could happen, for example, if they both touch the same line.

If that happens, you need to manually edit the file to fix the problem. Skip ahead or ask a demonstrator to help.

3.3 Review of changes

The file should now contain the change to the title, but not the extra line with the lie. Note the log:

```
commit d6959031d5722cb2b22c408e704e894bce7713e9
Author: James Hetherington <j.hetherington@ucl.ac.uk>
Date:   Wed Apr 3 16:13:19 2013 +0100
   Revert "Add a lie about a mountain"
  This reverts commit 50280520d33592d093773dfd3e5de4c3da7e1a09.
```

3.4 Antipatch

Notice how the mistake has stayed in the history.

There is a new commit which undoes the change: this is colloquially called an "antipatch". This is nice: you have a record of the full story, including the mistake and its correction.

3.5 Rewriting history

It is possible, in git, to remove the most recent change altogether, "rewriting history". Let's make another bad change, and see how to do this.

3.6 A new lie

```
vim index.md
cat index.md
```

```
    Engerland is not very mountainous.
```

```
git commit -a
git log | head
```

```
    ...Add a silly spelling
```

3.7 Referring to changes with HEAD and ^

The commit we want to revert to is the one before the latest.

HEAD refers to the commit before the "head", which is the latest change. That is, we want to go back to the change before the current one.

We could have used the hash code to reference this, but you can also refer to the commit before the HEAD as HEAD, the one before that as HEAD, the one before that as HEAD~3.

3.8 Using reset to rewrite history

```
git reset HEAD^
```

```
    Unstaged changes after reset:
    M index.md
```

```
git log --oneline
```

```
    53f4b50 Revert "Add a lie about a mountain"
                    This reverts commit 50280520d
    0bae905 Change title
    5028052 Add a lie about a mountain
    c438f17 First commit of discourse on UK topography
```

3.9 Covering your tracks

The silly spelling is gone, and *it isn't even in the log*. This approach to fixing mistakes, "rewriting history" with `reset`, instead of adding an antipatch with `revert` is dangerous, and we don't recommend it. But you may want to do it for small silly mistakes, such as to correct a commit message.

3.10 Resetting the working area

When git reset removes commits, it leaves your working directory unchanged – so you can keep the work in the bad change if you want.

If you want to lose the change from the working directory as well, you can do `git reset --hard`.

I'm going to get rid of the silly spelling, and I didn't do `--hard`, so I'll reset the file from the working directory to be the same as in the index:

```
git checkout index.md
```

3.11 The Levels of Git

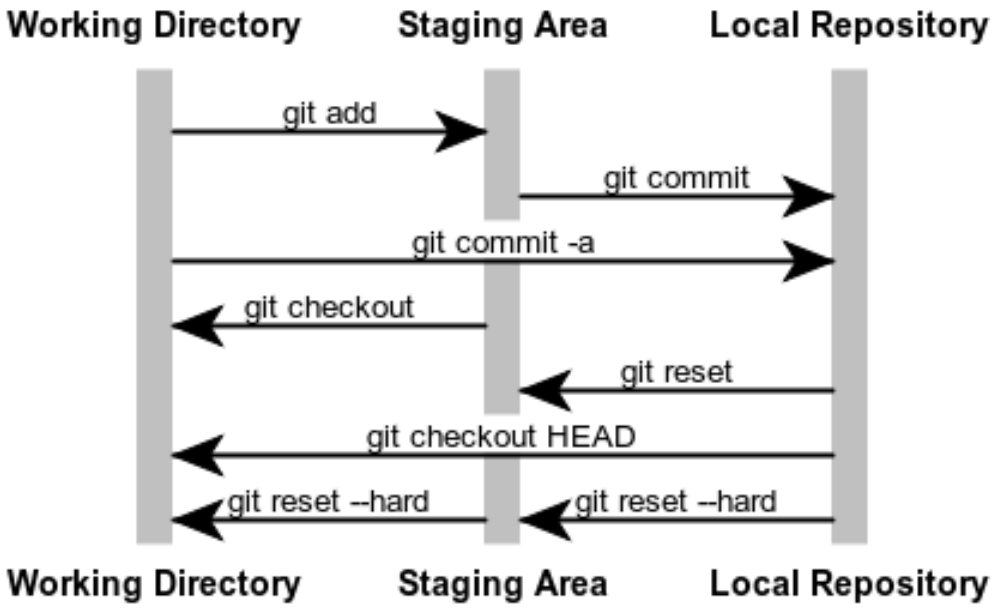

Figure 3: The relationship between the staging area, working directory, and repository in git.

4 Publishing

4.1 Sharing your work

So far, all our work has been on our own computer. But a big part of the point of version control is keeping your work safe, on remote servers. Another part is making it easy to share your work with the world In this example, we'll be using the "GitHub" cloud repository to store and publish our work.

If you have not done so already, you should create an account on GitHub: go to https://github.com/, fill in a username and password, and click on "sign up for free".

4.2 SSH keys and GitHub

You may want to set things up so that you don't have to keep typing in your password whenever you interact with GitHub via the command line.

You can do this with an "ssh keypair". You may have created a keypair in the Software Carpentry shell training. Go to the ssh settings page[4] on GitHub and upload your public key by copying the content from your computer. (Probably at .ssh/id_rsa.pub)

If you have difficulties, the instructions for this are on the GitHub website[5]. Ask your demonstrator for help here if you need it.

4.3 Creating a repository

Ok, let's create a repository to store our work. Hit "new repository" on the right of the github home screen, or click here[6].

Fill in a short name, and a description. Choose a "public" repository. Don't choose to add a Readme.

4.4 Paying for GitHub

For this software carpentry course, you should use public repositories in your personal account for your example work: it's good to share! GitHub is free for open source, but in general, charges a fee if you want to keep your work private.

In the future, you might want to keep your work on GitHub private.

Students can get free private repositories on GitHub, by going to [https://github.com/edu] and filling in a form.

[4] https://github.com/settings/ssh
[5] https://help.github.com/articles/generating-ssh-keys
[6] https://github.com/new

UCL pays for private GitHub repositories for UCL research groups: you can find the service details on our web page[7].

4.5 Adding a new remote to your repository

Instructions will appear, once you've created the repository, as to how to add this new "remote" server to your repository, in the lower box on the screen. Mine say:

```
git remote add origin git@github.com:jamespjh/jh-ucl-swcarpentry-answers
git push -u origin master
```

Follow these instructions.

4.6 Remotes

The first command sets up the server as a new `remote`, called `origin`.

Git, unlike some earlier version control systems is a "distributed" version control system, which means you can work with multiple remote servers.

Usually, commands that work with remotes allow you to specify the remote to use, but assume the `origin` remote if you don't.

Here, `git push` will push your whole history onto the server, and now you'll be able to see it on the internet! Refresh your web browser where the instructions were, and you'll see your repository!

4.7 The Levels of Git

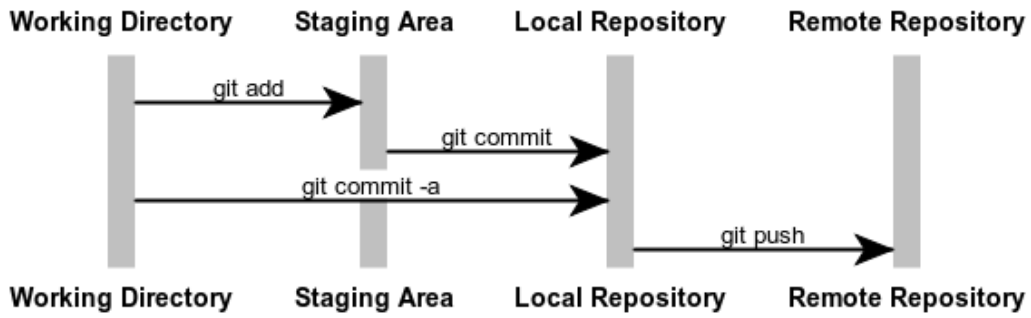

Figure 4: The relationship between the staging area, working directory, and repository in git.

[7] ../../infrastructure/github.html

4.8 Distributed VCS With Publishing

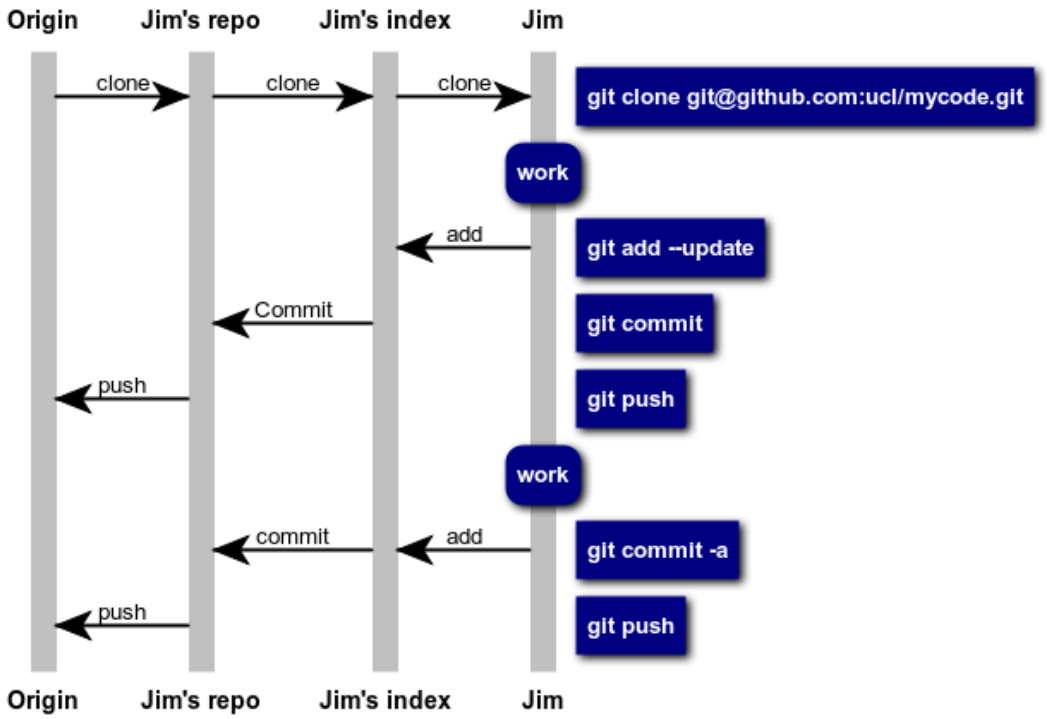

Figure 5: Publishing with git

4.9 Playing with GitHub

Take a few moments to click around and work your way through the GitHub interface. Try clicking on 'index.md' to see the content of the file: notice how the markdown renders prettily.

Click on "commits" near the top of the screen, to see all the changes you've made. Click on the commit number next to the right of a change, to see what changes it includes: removals are shown in red, and additions in green.

5 Working with multiple files

5.1 Some new content

So far, we've only worked with one file. Let's add another:

```
vim lakeland.md
cat lakeland.md
```

```
Lakeland
========

Cumbria has some pretty hills, and lakes too.
```

5.2 Git will not by default commit your new file

```
git commit -a
```

```
# On branch master
 # Untracked files:
 #   (use "git add <file>..." to
 #   include in what will be committed)
 #
 #  lakeland.md
 nothing added to commit but untracked files present
  (use "git add" to track)
```

This didn't do anything, because we've not told git to track the new file yet.

5.3 Tell git about the new file

```
git add lakeland.md
git commit -a
```

Ok, now we have added the change about Cumbria to the file. Let's publish it to the origin repository.

```
git push
```

```
Counting objects: 4, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 343 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@github.com:jamespjh/
       jh-ucl-swcarpentry-answers.git
   53f4b50..9e8b69c  master -> master
```

Visit GitHub, and notice this change is on your repository on the server. We could have said `git push origin` to specify the remote to use, but origin is the default.

6 Editing directly on GitHub

6.1 Editing directly on GitHub

Note that you can also make changes in the GitHub website itself. Visit one of your files, and hit “edit”.

Make a change in the edit window, and add an appropriate commit message.

That change now appears on the website, but not in your local copy. (Verify this).

6.2 Pulling from remotes

To get the change into your local copy, do:

```
git pull
```

```
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From github.com:jamespjh/jh-ucl-swcarpentry-answers
   9e8b69c..d2a1854  master     -> answers/master
Updating 9e8b69c..d2a1854
Fast-forward
 index.md | 8 ++++++++
 1 file changed, 8 insertions(+)
```

and check the change is now present on your local version. `git pull` will fetch changes on the server into your local copy: this is important when you are collaborating with others, as we shall see.

6.3 The Levels of Git

7 Branching and tagging

7.1 Branches

Git lets you maintain different versions of your history in your repository. These are called branches.

You can use these to work on crazy new ideas in your code, and still keep a branch with your working code you use to do science.

To create a new branch, you can do:

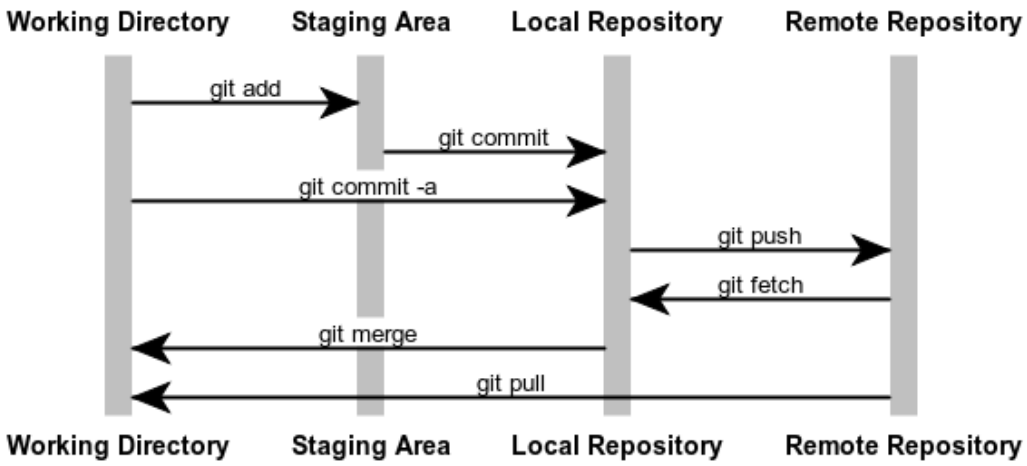

Figure 6: The relationship between the staging area, working directory, and local and remote repositories in git.

```
git branch experiment
git branch
```

```
  experiment
  * master
```

The asterisk indicates the branch you are "on". Although you created the `experiment` branch, you're still on the master branch.

7.2 Changing Branch

Switch to the experiment branch with:

```
git checkout experiment
```

7.3 Working with branches

7.4 Publishing branches

Let the server know there's a new branch with:

```
git push --set-upstream origin experiment
```

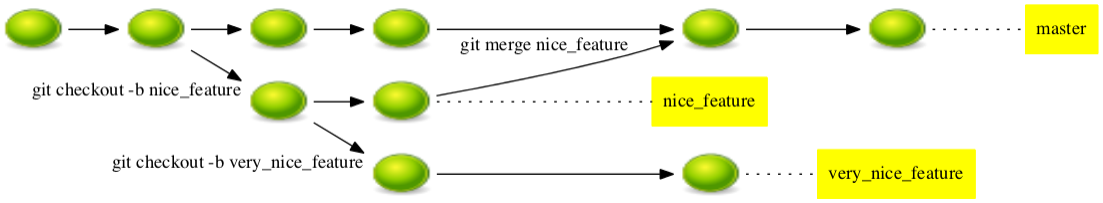

Figure 7: Using branches

We use `--set-upstream origin` (Abbreviation `-u`) to tell git that this branch should be pushed to and pulled from origin per default.

You should be able to see your branch in the list of branches in GitHub.

7.5 Play with branches

Go ahead and make some changes to files, and commit and push them with `git push`. You'll see that they only appear in the experimental branch.

7.6 Return to the master branch

When you want to switch back to your main branch, make sure you're fully commited, with no changes to files waiting to commit, then do:

```
git checkout master
```

7.7 Merging branches

Now, you can pull your changes back from your experimental branch into the master branch, with

```
git merge experiment
```

If you have committed changes on both master and experiment, you may hit conflicts, where git can't work out how to merge them. You can skip ahead and learn how to solve this, or ask a demonstrator to help with this.

7.8 A good branch strategy

- A `production` branch: code used for active work
- A `develop` branch: for general new code
- `feature` branches: for specific new ideas
- `release` branches: when you share code with others
- Useful for isolated bug fixes

7.9 Tags

Git gives you the power to label a particular version of your code with a more readable name. This is called a "tag". This is important for making sure you only do science with a certain version.

Label your current version with:

```
git tag -a v0.2
```

7.10 Tagging old revisions

Pick an old commit using `git log`, and label it like this:

```
git tag -a v0.1 c438f17
```

7.11 Visiting old code

You can now go back to the v0.1 commit like this:

```
git checkout v0.1
```

and then switch back to the main code with

```
git checkout master
```

You can refer to any tagged commit with the tag name, anywhere you would otherwise use a commit hash code, or a HEAD-based reference.

7.12 Publishing tags

You need to push your tag labels up to the remote, with

```
git push --tags
```

You should now be able to see these under tags on GitHub, in the dropdown menu on the left.

8 GitHub pages

8.1 Yaml Frontmatter

GitHub will publish repositories containing markdown as web pages, automatically.

You'll need to add this content:

```
---
---
```

A pair of lines with three dashes, to the top of each markdown file. This is how GitHub knows which markdown files to make into web pages. Here's why[8] for the curious.

8.2 The gh-pages branch

GitHub creates github pages when you use a special named branch.

This is best used to create documentation for a program you write, but you can use it for anything.

```
git checkout -b gh-pages
git push -u origin gh-pages
```

The first time you do this, GitHub takes a few minutes to generate your pages. The website will appear at `http://username.github.com/repositoryname`, for example, here's mine[9]

8.3 Markdown Hyperlinks

You can use this syntax

```
[link text](URL)
```

To create hyperlinks in your pages, so you can link between your documents. Try it!

8.4 UCL layout for GitHub pages

You can use GitHub pages to make HTML layouts, here's an example of how to do it[10], and how it looks[11]. We won't go into the detail of this now, but after the class, you might want to try this.

[8] https://github.com/mojombo/jekyll/wiki/YAML-Front-Matter
[9] http://jamespjh.github.com/jh-ucl-swcarpentry-answers/
[10] http://github.com/UCL/ucl-github-pages-example
[11] http://ucl.github.com/ucl-github-pages-example

9 Collaboration

9.1 Form a team

Now we're going to get to the most important question of all with Git and GitHub: working with others.

Organise into pairs. You're going to be working on the website of one of the two of you, together, so decide who is going to be the leader, and who the collaborator.

9.2 Distributed VCS in principle

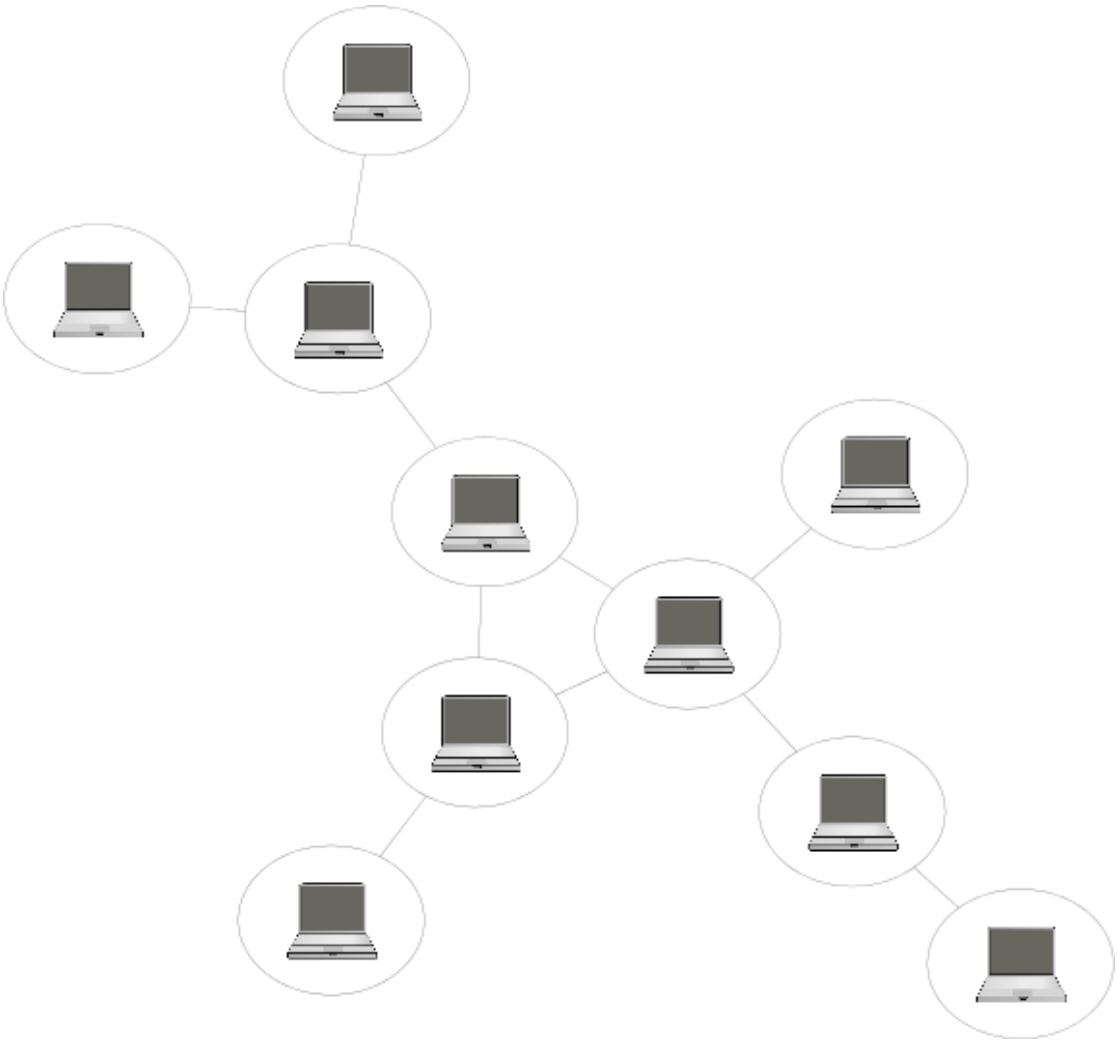

Figure 8: How distributed VCS works in principle

9.3 Distributed VCS in practice

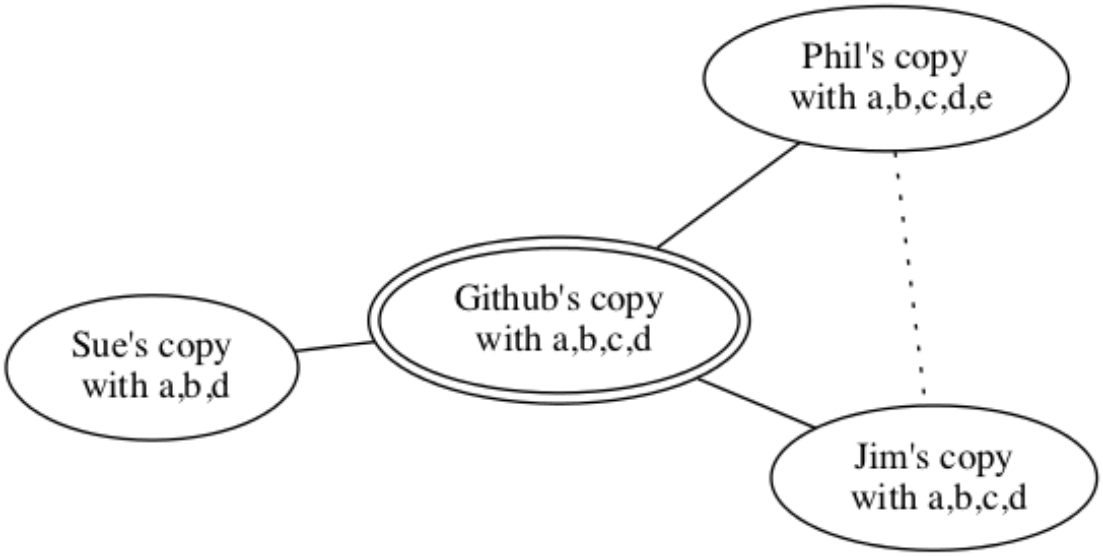

Figure 9: How distributed VCS works in practice

9.4 Giving permission

The leader needs to let the collaborator have the right to make changes to his code.

In GitHub, go to `settings` on the right, then `collaborators` on the left.

Add the user name of your collaborator to the box. They now have the right to push to your repository.

9.5 Obtaining a colleague's code

Next, the collaborator needs to get a copy of the leader's code. Make yourself a space to put it:

```
cd .. # To get out of your own solution, and
      # back to a safe place in your working area
mkdir carpentry-collaborations
cd carpentry-collaborations
```

Next, the collaborator needs to find out the URL of the repository: they should go to the leader's repository's GitHub page, and note the URL on the top of the screen. Make sure the "ssh" button is pushed, the URL should begin with `git@github.com`.

Copy the URL into your clipboard by clicking on the icon to the right of the URL, and then:

```
git clone git@github.com:/... #Subsitute the right URL from your clip
```

9.6 Nonconflicting changes

Now, both of you should make some changes. To start with, make changes to *different* files. This will mean your work doesn't "conflict". Later, we'll see how to deal with changes to a shared file.

Both of you should commit, but not push.

One of you should now push with `git push`

9.7 Rejected push

The other should then push, but should receive an error message:

```
To git@github.com:jamespjh/
        jh-ucl-swcarpentry-answers.git
! [rejected]     master -> master (non-fast-forward)
error: failed to push some refs to
 'git@github.com:jamespjh/
       jh-ucl-swcarpentry-answers.git'
hint: Updates were rejected because
     the tip of your current branch is behind
hint: its remote counterpart.
         Merge the remote changes (e.g. 'git pull')
hint: before pushing again.
hint: See the 'Note about fast-forwards'
      in 'git push --help' for details.
```

Do as it suggests:

```
git pull
```

9.8 Merge commits

Note a window pops up with a suggested default commit message. This commit is special: it is a *merge* commit. It is a commit which combines your collaborator's work with your own.

Now, push again with `git push`. This time it works. If you look on GitHub, you'll now see that it contains both sets of changes.

9.9 Nonconflicted commits to the same file

Go through the whole process again, but this time, both of you should make changes to a single file, but make sure that you don't touch the same *line*. Again, the merge should work as before.

9.10 Sharing without conflicts

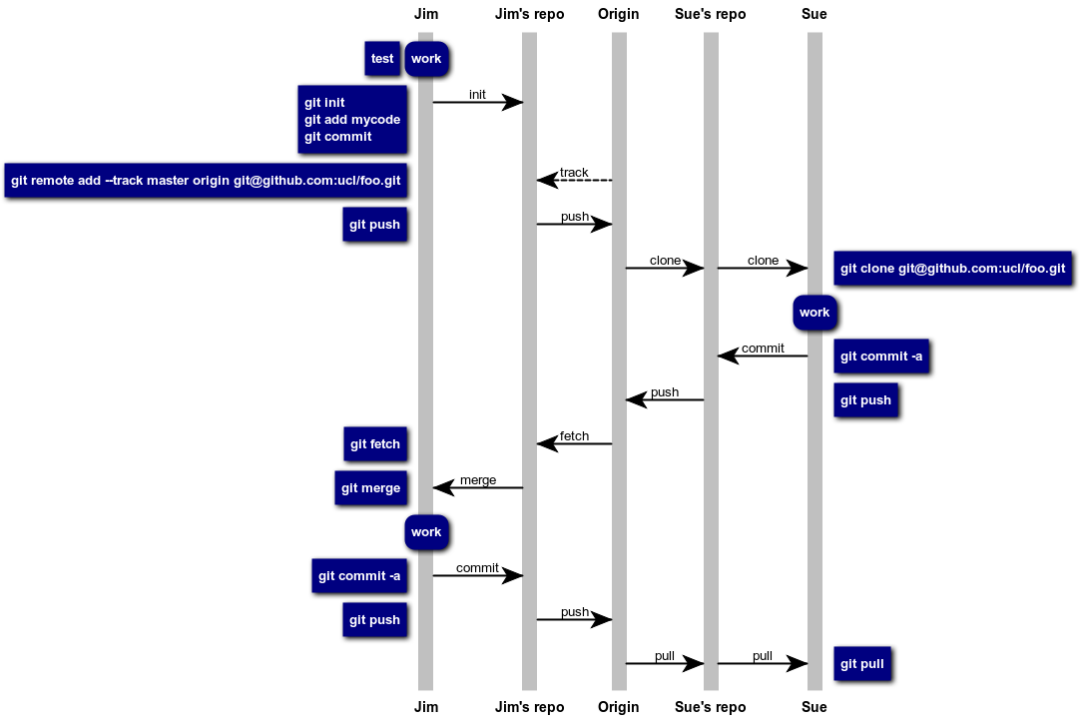

Figure 10: Teamworking in git

9.11 Conflicting commits

Finally, go through the process again, but this time, make changes which touch the same line.

When you pull, instead of offering a commit message, it says:

```
CONFLICT (content): Merge conflict in index.md
Automatic merge failed; fix conflicts
   and then commit the result.
```

9.12 Resolving conflicts

Git couldn't work out how to merge the two different sets of changes.

You now need to manually resolve the conflict.

Edit the file. It should look something like this:

```
<<<<<<< HEAD
Wales is hillier than England,
        but not quite as hilly as Scotland.
=======
Wales is much hillier than England,
        but not as hilly as Scotland.
>>>>>>> dba9bbf3bcab1008b4d59342392cc70890aaf8e6
```

The syntax with <<< === and >>> shows the differences.

Manually edit the file, to combine the changes as seems sensible and get rid of the symbols.

9.13 Commit the resolved file

Now commit the merged result:

```
git commit -a
```

A suggested commit message appears, which you can accept, and then you can `push` the merged result. Check everything is fine on GitHub.

9.14 A revision graph

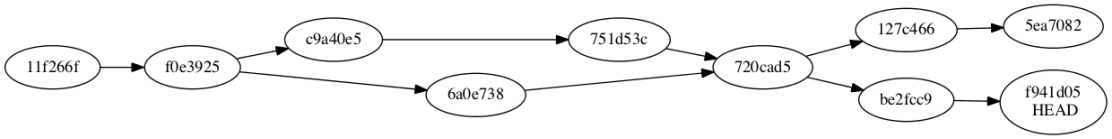

Figure 11: Revisions form a graph

9.15 Distributed VCS in teams with conflicts

9.16 The Levels of Git

10 Social Coding

10.1 GitHub as a social network

In addition to being a repository for code, and a way to publish code, GitHub is a social network.

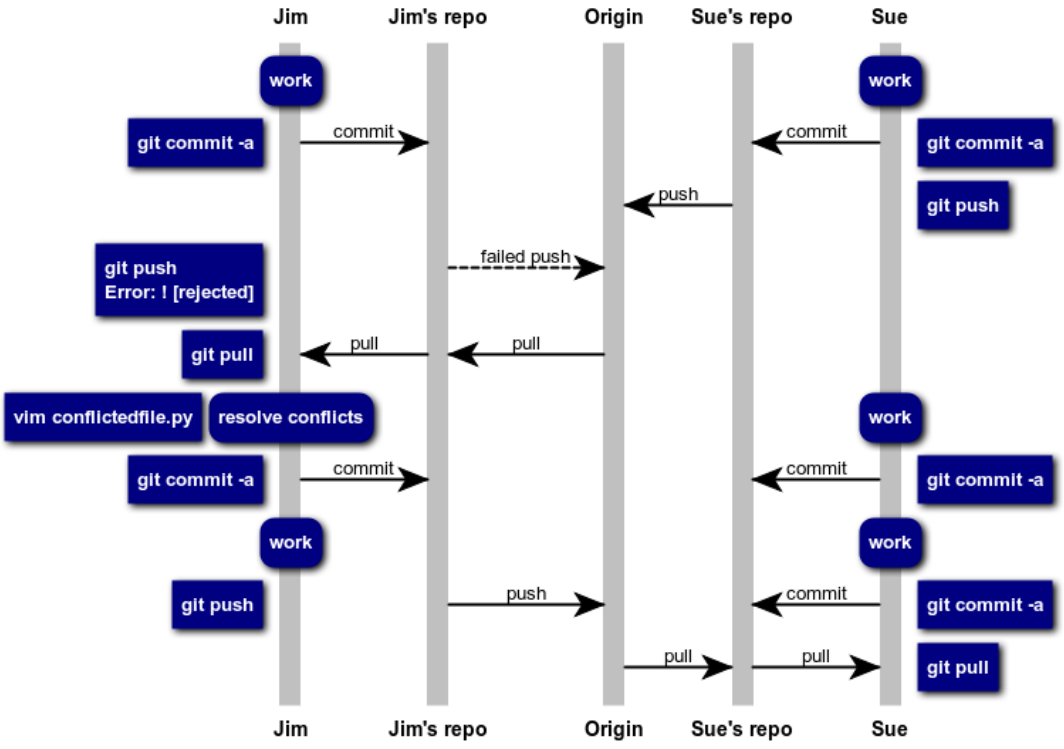

Figure 12: Teamworking in git with conflicts

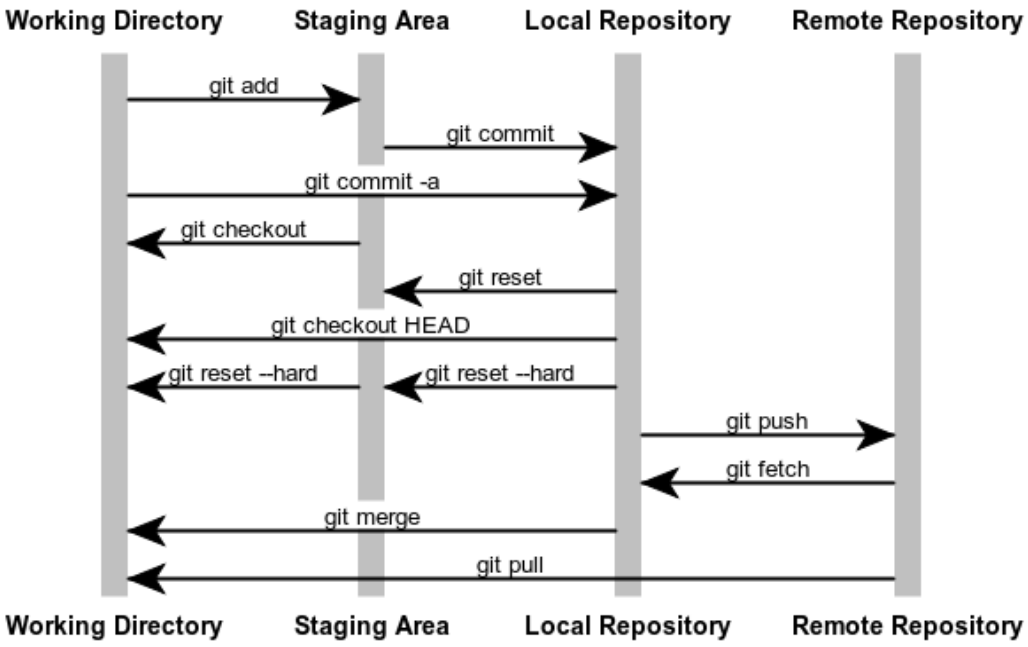

Figure 13: The relationship between the staging area, working directory, and local and remote repositories in git.

You can follow the public work of other coders: go to the profile of your collaborator in your browser, and it the "follow" button.

Here's mine[12] : if you want to you can follow me.

Using GitHub to build up a good public profile of software projects you've worked on is great for your CV!

11 Pull Requests

11.1 Forking

If you want to collaborate with someone, but you don't want to give them the right to change your code directly, you can collaborate through *pull requests* instead of by granting them access.

Swap roles. This time, the collaborator, instead of pulling the principal's code, should *fork* it. Go to the repository on GitHub, and hit "fork" top right.

A new repository will be created on the collaborator's account, which contains all the same stuff. Clone it with `git clone`.

11.2 Pushing to your forks

Both of you can make changes. (Make them to different files now for simplicity.)

Now, both of you will be able to push: you're pushing to different repositories!

11.3 Create a pull request

You need to request that the leader accept your changes. The collaborator should go to the page for their *forked* repository and hit the green button to the left of the branch dropdown. On the page that appears, choose "compare across forks".

Choose the right branches and repositories for the "base", the leader's repository, and the "head repo", the collaborator's repository. Ask a demonstrator for help if this is confusing.

Hit "Click to create a pull request for this comparison".

Give the request a title and a comment, and send it.

11.4 Accepting a pull request

The leader should now see a link to the pull request on their github home page. Have a read through the changes, and if you think it's fine, then merge it, using the big green button.

[12] https://github.com/jamespjh

(If there's a conflict, there won't be a big green button: ask your demonstrator for help with how to merge a conflicted pull request.)

Note you can comment on and discuss the contribution using the pull request: this is great for open source projects with many people working together.