Unix Tutorial 7: Collaborating via git

Git is a powerful and popular distributed version control system for teams that are collaborating on source code development. In this tutorial sheet, we will investigate git commands for accessing remote repositories that are shared between multiple users.

Although git is distributed, there are centralized repo servers like `https://github.com` and `https://bitbucket.org` that allow easy synchronization between users. Users *publish* (`git push`) their changes to these servers, and *fetch* (`git pull`) other people's changes, on shared projects.

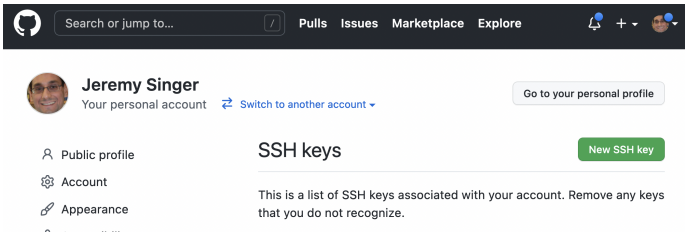
Registering your ssh key

You need to set up an ssh key-pair to use github and other online repository sites. This allows you to do secure authentication, perhaps for contributing to private repositories, etc.

If you don't have an ssh key pair, run the following command and press Enter three times (to accept the default options at the interactive prompts).

```
$ ssh-keygen -t rsa
```

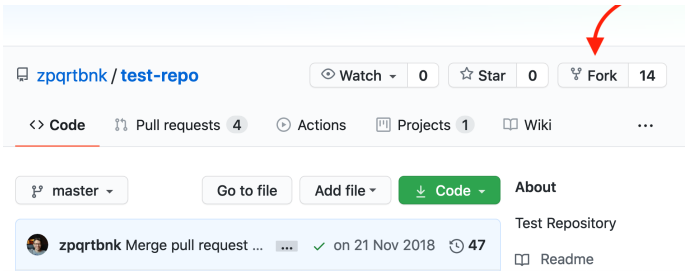
This generates a public key in file `.ssh/id_rsa.pub` and a private key in file `.ssh/id_rsa`. Now you need to register your public key with github. Login at `https://github.com` with your username and password, then click on your profile icon and click on the 'settings' on the pop-up menu. Now click on 'ssh and gpg keys', then click the 'New ssh key' green button. Use `cat .ssh/id_rsa.pub` to show the file contents, then select this and paste it into the textbox.



Cloning a shared repo

Find an 'interesting' git repository you want to contribute to. If you are working with friends, you might set up a new repo on github and add multiple people as contributors (with read and write access).

On the other hand, if you are working on your own, it might be easier just to *fork* an existing github repo — this copies the existing repo into your github account so you have read and write access to this copy. When you are logged in at the github site, find a repo — I chose a harmless test repo at `https://github.com/zpqrtbnk/test-repo` and click the Fork button.



This copies the repo to your github workspace, so you end up with a repo at `https://github.com/YOURNAME/test-repo`.

Now you want to *clone* the repo you have created. The clone operation copies the entire repo from the github server to your local machine. If it's a large repo, this might take some time.

You should clone over ssh rather than https, check the github docs for info on this. The command is:

```
$ git clone git@github.com:YOURNAME/test-repo
```

The clone command will create a new directory in the current working directory. The new directory will have the name of the repo (`test-repo` in this example) and it is a working git repository.

Making changes in a shared repo

Once you have your local copy of the git repo, `cd` into the directory and start editing files. The workflow is the same as in the previous tutorial. Every time you make a logical, incremental change, you should `git add` the file(s) and then `git commit` the changeset with a suitable log message.

Try making a few edits to the repo you have cloned. You can also delete files with `git rm` or change their names with `git mv`. Note these are changes that need to be committed as well.

When you have made a few changes, let's try publishing these local changes back to the remote repository. The remote repo URL has been 'remembered' by git — you can see it with this command:

```
$ git remote show origin
```

OK, we use the `git push` command to send the local changes back to the remote *upstream* repo:

```
$ git push
```

Collaborating in a shared repo

If someone else has been working on the same upstream repository (say a github repo that you both share) then they might have pushed their changes upstream

and you want to merge them into your local repo. This requires the `git pull` command:

```
$ git pull
```

Note that you always need to `pull` before you `push` in a shared repo. If you try to push and your local repo is not up-to-date then git complains and the push operation is aborted. This avoids inconsistencies in the upstream repo.

Pulling and pushing works fine so long as git can *merge* the independent changes. If both of you are working on the same lines of code in a single file then it won't be possible to auto-merge the changes — in this case you need to manually resolve the *merge conflicts*. For most people, this is their biggest git nightmare . . . actually it's not too bad with some practice.

When you do a `git pull` you see the merge conflict reported. The files containing conflicts have extra git lines in them, to show the differences between your local commits and the upstream commits. The snippet below shows an example:

```
<<<<<<< COMMIT-ID:index.html
<div id="footer">contact : jeremy.singer@glasgow.ac.uk</div>
=======
<div id="footer">
 please contact Jeremy on MS Teams
</div>
>>>>>>> OTHER-COMMIT-ID:index.html
```

The merge conflict is delineated with the <<< and >>> character sequences. The first version should be your local commit. After the === you can see the conflicting commit from someone else. All you need to do is to edit the file to remove the git char sequences and resolve the differences. I did this:

```
<div id="footer">
 please contact Jeremy on MS Teams or by email
</div>
```

Then we go through the regular sequence:

```
$ git add index.html
$ git commit
```

When you commit, git will say something like 'it looks like you are resolving a merge conflict' — and auto-generate a log message for you. Then you can push the merged code back to the centralized repo.

Other useful commands

If multiple developers are working on independent features, often they want to use separate *branches* to localise their changes and to prevent frequent merge conflicts.

If you want to highlight a particular commit id (perhaps it is a beta release or something) then you can *tag* the version just after you have committed it:

```
$ git tag -a v1.0 -m "Release version 1.0"
```

Next time you run a `git log` then you will see the tags associated with the commits. You can also list all the tags with `git tag` without any arguments.

Do you see who has made the most recent change to each line of an individual file? The intriguingly named `git blame` command is useful here.

```
$ git blame file.txt
```

Can you see how to run this command, combined with some `cut` and `sort -u` and `grep -c` to work out who has contributed the most lines of code in your team project repo?

Further Reading

There are lots of online resources and tutorials for git. Try the selection at: `https://git-scm.com/doc`. There is a nice (if slightly obscene) graphical guide at `https://rogerdudler.github.io/git-guide/`.