UNIX Tutorial 3 - Using Git

Git is a powerful version control system. In this tutorial sheet, we will introduce git and look at basic local commands for a single user.

It's always useful to track changes on your files, particularly for projects with many source code files and multiple contributors. Distributed version control systems are currently the standard approach — these allows each user to have their own copy of the project with all its changes, and copies can be synchronized between the multiple users.

A *repository* is the full set of change history for a project. A distributed version control system implies that each user has their own complete local repository.

The most common distributed VC tool is called *git*[1]. Originally invented by Linus Torvalds (who also developed Linux) git is now used widely across the open source community and by industrial software engineers.

Although distributed, there are centralized repo servers like `https://github.com` and `https://bitbucket.org` that allow easy synchronization between users. Some centralized servers provide other continuous integration services. Gitlab is one example; you will be using this system in your team project work.

Initial task: Set up git accounts

If you don't yet have a github account, set one up today at `https://github.com`. Remember you might use this for the rest of your developer career, so choose a vaguely sensible username; for instance, I am `jeremysinger`. Many companies check github profiles when they are hiring, so it's worth having something impressive to show them.

In the University, we run a private gitlab server at `https://stgit.dcs.gla.ac.uk`. You should set up an account here as well, using your Unix username. (Check details with tutors or TP3 coordinators.) It's likely you will use this server for your Team Project and PSD coursework throughout Level 3.

You also want to configure your personal details for git commits. This metadata will be linked with your git change history. Run the following commands in a terminal:

```
git config --global user.name "Your Name"
git config --global user.email "1234567a@student.gla.ac.uk"
```

My first git repo

Create a new empty directory and change into it, i.e.

```
$ mkdir my_first_repo
$ cd my_first_repo
```

[1] `https://git-scm.com/`

At the moment this is just a standard Unix directory, nothing special. Let's turn it into a git repository ...

```
$ git init
```

This command creates a 'hidden' directory called `.git` (you can see this with `ls -a`) where the repository metadata is stored.

Now we can look at the status of the repository:

```
$ git status
```

It's not very interesting at the moment, because we don't have any files in the repo. Let's create a file called `file.txt` and add it to the repository.

```
$ echo "hello world" > file.txt
$ git add file.txt
$ git commit -m "a silly little text file"
```

Do you see the workflow? First we add the file, which says 'something interesting is happening here' to git. Officially, this is called `staging`. Then we commit the change, which adds the staged changes to the git repo as a `changeset`. The `-m` flag allows us to summarize the changes, which is useful for logging. If we omit the textual summary and simply type git commit then we are dropped into an interactive text editor (normally vim) where we can write the changes in a textfile, then save and quit. Again, this text is stored in the repository log. Let's look at our logfile now.

```
$ git log
```

We might make some further changes to file.txt — go on, try it. Then we an compare these changes against the last committed version:

```
$ git diff
```

This will show us lines that have been added or removed, relative to the last committed version, for each file. Suppose we are happy with these changes and we want to commit the changed file to the repo ... we simply go through the same two-stage procedure as previously:

```
$ git add file.txt
$ git commit -m "updated text file"
```

Recovering from mistakes

The main motivation for using version control is that it makes it easier to recover earlier versions of the files, if you make mistakes and need to 'undo' things. There are three key scenarios we explore in this section.

Undo unstaged changes

Suppose you have edited a file and you want to roll it back to the most recent commit in the git history. If you have yet done a `git add` on the file, then the following command will undo changes since the most recent commit:

```
$ git checkout -- file.txt
```

This undo action works even if you have deleted file.txt by accident with `rm`.

Undo staged changes

Suppose you have edited a file, run `git add file.txt` but then you decide you want to 'undo' the add. So long as you have not yet committed this staged change, it is possible to do:

```
$ git reset file.txt
```

Undo committed changes

Recall that every committed changeset in git gets a version identifier, which is a hex number indicating the checksum of the changeset. If we want to 'undo' the effects of a particular commit, we need to find its version number. A useful command is git log −−oneline, which shows commit hashes and corresponding messages. Once we have identified the ID of the commit we want to 'undo', we run:

```
$ git revert ID
```

This command generates a new commit that is the reverse of the commit that was specified for the revert. Git will then open up a text editor (likely to be vim) to prompt for a new commit message. Reverts are the safest 'undo' option because of they leave a clear trail in the commit history of when an undo operation was executed. Alternative commands include git reset and git rebase.

Next time, we will look at interacting with other collaborators on a shared project with a remote git repository.

Further Reading

There are lots of online resources and tutorials for git. Try this one for starters: `https://git-scm.com/book/en/v2/Git-Basics-Getting-a-Git-Repository` and this interactive one next `https://learngitbranching.js.org/`.