Pre-lab 3: Introduction to Git

1 Introduction

Source code control is a requirement for professional software engineering. This tutorial will show you the basics of working with the source code management system known as Git.

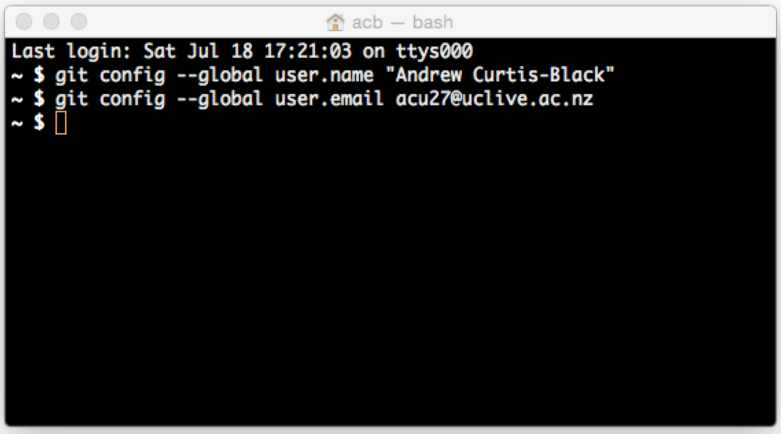
As software engineers work on a project, they create many iterations (versions) of various software components (e.g., the user interface may be improved and adjusted many times). Professional engineers keep copies of all of these versions for various reasons (e.g., in case a new design is flawed and they need to roll-back to an older version, or in case elements of an older design can be reused later). As you may know, Git is a tool which helps developers work with and across different versions of their code. Tools like GitLab (which you will use in this course) work in tandem with Git to help multiple engineers work on the same project at the same time, and to add functionality such as continuous integration and deployment of code held in a Git repository.

2 Worked example

Git is a very powerful tool, but it can be complex to work with and there are a number of concepts you'll need to understand. This worked example will take you through cloning a Git repository ("repo") and will teach you some of the basic skills you'll need to manage your project's Git repo. The lab machines already have Git installed and many of your personal computers will have come with Git pre-installed, but those who need to install it should visit this site: http://git-scm.com/downloads.

2.1 Configuring Git

Before you start it's best to set some of your Git user credentials. Your Git user name and email address will be used to identify your contributions to the repo. Set them using the commands as shown in the screenshot below (substitute your own name and email address).



```
Last login: Sat Jul 18 17:21:03 on ttys000
~ $ git config --global user.name "Andrew Curtis-Black"
~ $ git config --global user.email acu27@uclive.ac.nz
~ $ 
```

If you think you'll be working on the project from more than one machine then **make sure you use the same credentials** (user name and email address) on **all** of them. Otherwise your work will be displayed under several different names. That will annoy your markers (and in the real world, will annoy your colleagues).

2.2 Per-project Git configuration

It's common for developers to work on different projects at the same time, and some of these may be better suited to different config details (for example, you may not want to use your university email address for your personal projects, or your personal email address for your university projects). In such cases you can override the global Git config settings on a per-project basis by navigating to the project in question and omitting the --global flag:

```
cd path/to/git/repo
git config user.name "Your name here"
git config user.email "username@uclive.ac.nz"
```

3 Using Git

For this course, we've given you a git repo on the university's GitLab server at https://eng-git.canterbury.ac.nz. GitLab is a server for Git repositories (you may have heard about other similar services such as GitHub or Bitbucket). Any time you make a change (you'll see how later) to this particular repo, you'll trigger a series of automatic steps that'll take your code and deploy it to a personal port on a shared server for you to test and use.

Although you can directly edit files on GitLab, that's quite a cumbersome way to develop. Instead, you should make a copy of the repo on your local computer and work directly with the files there. Once you've made some changes you're happy with, you can use git to synchronize your local changes with the remote copy back on eng-git. Let's walk through the process now.

3.1 Cloning your repo to a local computer

First, we need to discover the URL for your repo on eng-git. Go to https://eng-git.canterbury.ac.nz and login as usual. Under the projects list you'll see a project with a name like seng365-2021/<your user code>. Open it, and near the middle of the page will be a URL for the repo. Change the toggle from SSH to HTTP and copy the URL.

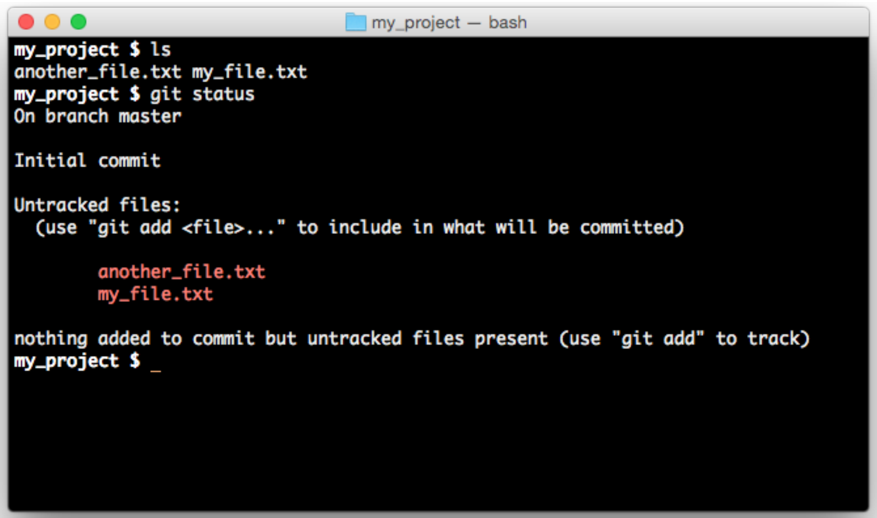
Now we need to copy the repo to the local computer. Open your computer's command line and navigate to a directory that can host your new repo (it'll be created as a subdirectory of the directory you're in when you issue the command.) Now type git clone <URL>, replacing <URL> with the URL you just obtained, and watch as Git creates a new directory for the repo, and pulls down all the remote files. You can now change directory into your new repo directory and work on the project.

3.2 Adding files to a repository

Git is structured around the idea of “commits”. These are essentially lists of modifications made to a repository since the last commit. Thus, each commit builds upon the last, creating a chain of changesets. As you work you will create some local changes and then notify Git of those changes so that it can add them to the next commit. This is the purpose of the **`git add`** command. It tells Git to add the changes made to the specified file to the next commit. You can think of creating a file as a special case of making a change to that file.

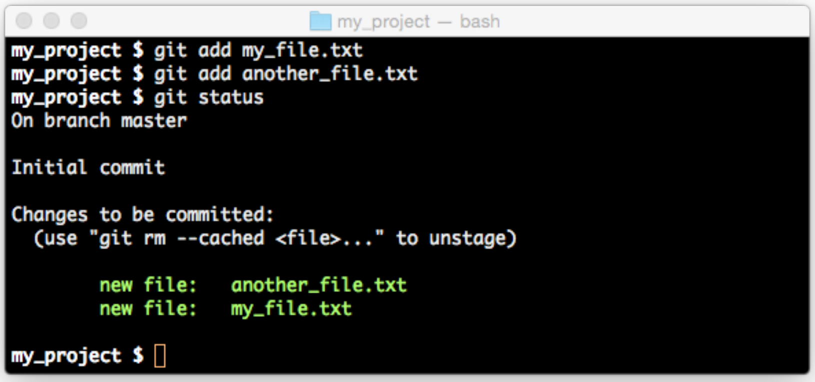
This is the basis of one of the core concepts of Git: **staging**. The staging area is where changes are stored before they are committed to a repository.

You can use the **`git status`** command to identify the state of files in the local repo (unmodified, modified etc.) Create some files (any way you like) and add them into the directory. When you check the status of the repo you’ll see that the files you added are “untracked”. This means that you won’t be able to send those new files (or any changes made to them) into the repo



```
my_project $ ls
another_file.txt my_file.txt
my_project $ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        another_file.txt
        my_file.txt

nothing added to commit but untracked files present (use "git add" to track)
my_project $ _
```

Now use the **`git add`** command to “stage” the files.

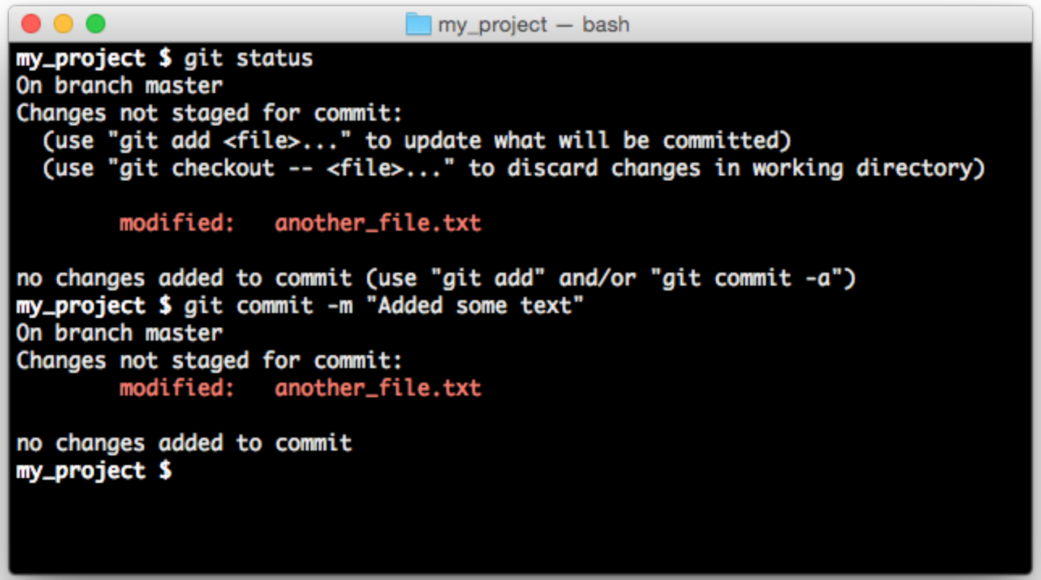


```
my_project $ git add my_file.txt
my_project $ git add another_file.txt
my_project $ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   another_file.txt
        new file:   my_file.txt

my_project $
```

Now you will “commit” your staged changes to the repository. Performing a Git commit updates the state of the repository with the changes you have made. Each commit you make should be a relatively small set of connected changes. Don’t make a bunch of unrelated edits and commit them all at once. Part of the reason for this is that Git keeps a complete history of changes to a repository and sometimes you'll want to undo those changes. If your commits consist of nicely grouped modifications this will be easy. If they're a tangled mess then it will be difficult. When you make a commit you're also required to supply a message which should concisely describe the purpose of your changes. This will help other developers (and your future self) work out why you did what you did (remember that code alone can sometimes be confusing or misleading).

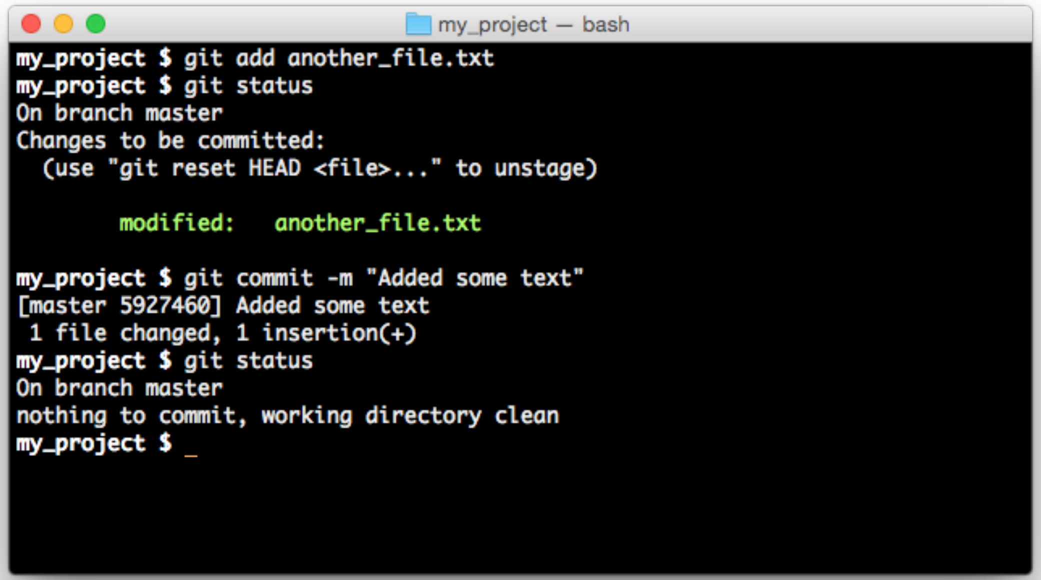
Always use concise, descriptive, and professional commit messages. Software engineers frequently refer to commit messages and it is wasteful of resources if these messages are difficult to interpret.

To try all this out open one of your files (`README.md` would be a safe choice) and edit it somehow, then check the status of your repo. You'll see that Git has noticed that one of the files it's tracking has been modified. You can't commit this change just yet; if you try Git will tell you that no changes were added to the commit, and it'll list the changes that haven't been "staged" (so that you know what to do to fix the problem).



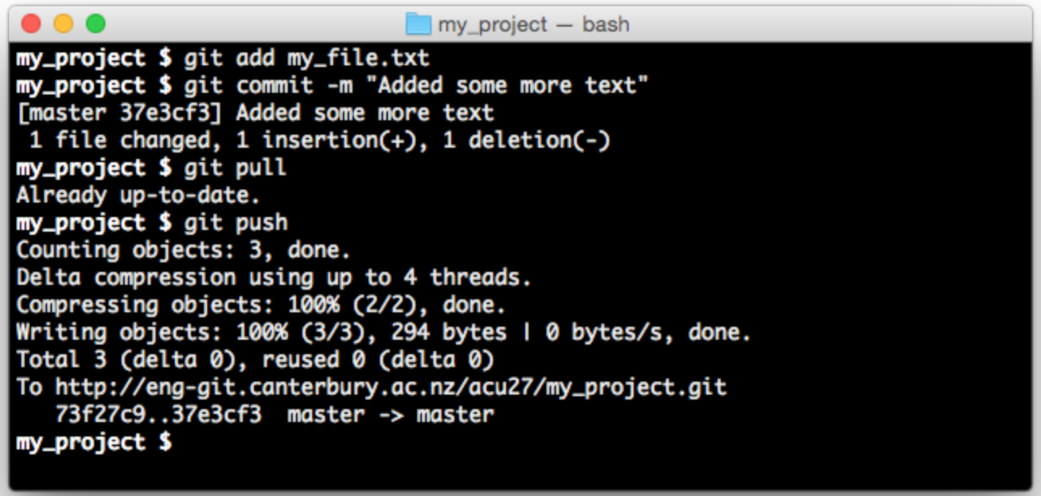
```
my_project $ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   another_file.txt

no changes added to commit (use "git add" and/or "git commit -a")
my_project $ git commit -m "Added some text"
On branch master
Changes not staged for commit:
        modified:   another_file.txt

no changes added to commit
my_project $
```

Before, when you added the files in the first place, you used the **`git add`** command to tell Git to take note of a particular set of changes. So now you just need to do the same thing again. Use **`git add`** to stage your modified file and check the status of your repo. You'll see that Git now lists your modifications as "changes to be committed" rather than as "changes not staged for commit". Commit the changes to the repo (and marvel as everything Just Works^{TM}).



```
my_project $ git add another_file.txt
my_project $ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   another_file.txt

my_project $ git commit -m "Added some text"
[master 5927460] Added some text
 1 file changed, 1 insertion(+)
my_project $ git status
On branch master
nothing to commit, working directory clean
my_project $ _
```

Now you will use **`git push`** to "push" the changes in your local repo to your remote repo, effectively synchronising the two.



```
my_project $ git add my_file.txt
my_project $ git commit -m "Added some more text"
[master 37e3cf3] Added some more text
 1 file changed, 1 insertion(+), 1 deletion(-)
my_project $ git pull
Already up-to-date.
my_project $ git push
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 294 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To http://eng-git.canterbury.ac.nz/acu27/my_project.git
   73f27c9..37e3cf3  master -> master
my_project $
```

If you look under the Files tab on GitLab you should be able to view the files you added, complete with the modifications you made to them.

The opposite of a push is a pull, and a "pull" in Git does pretty much exactly what you would expect it to do. It checks the remote repo to see if there have been any changes which are not already in your local copy of the project and copies them to your local machine. We won't cover pulls and merges in this tutorial, but you are encouraged to try things out for yourself and to ask the staff for help if you have any questions. If you're only making changes to one repo (and just pushing them to the remote) you don't need **`git pull`**, but in industry that's almost unheard of, and so we'd recommend that you get into the habit of always doing a pull before you push.

Now we'll quickly run through making a change locally and pushing it to the remote repo. First, open one of the files in your repo (`README.md` again would be good) and modify it. Stage the file, make a new commit, do a pull (not strictly necessary as we know there haven't been any changes to the remote repo), then push your changes. Check GitLab via your browser to make sure your changes made it through.

So, to review.

1. Make some changes to your files, as usual.
2. Before you can commit changes to the local repo you first need to stage them with **`git add <filename>`**.
3. To commit changes to the local repo use **`git commit -m <descriptive message>`**.
4. Pull the latest changes down from the remote repo.
5. Merge the changes in your local repo with the changes from the remote repo, if there are any. If you need to do a merge you'll also need to do a commit, because a merge is itself a change to the repo.
6. Push your commits to your remote repo with **`git push`**.
7. GitLab builds and deploys your application for you.

4 Using Git in an IDE

Learning to use Git in the command line is important as you'll likely need this skill more than once in your career. However, for everyday use many developers prefer to use a graphical tool. Modern IDEs like WebStorm have built-in support for most popular Version Control Systems (VCS) (including Git).

If you've followed all the steps in the worked example above you will be able to simply open your project directory in WebStorm and use Git through the IDE's interface without further set up. Everything covered in the command line in the worked example can also be done in your IDE's interface.

Make some changes to one or more of your project files in your IDE and see if you can push them to the remote repo. WebStorm has two buttons at the top far-right of the window that directly map to **`git pull`** and **`git push`**: the button with a down-arrow and the word "VCS" maps to "pull" and the up-arrow button also labelled "VCS" maps to "push", although as usual there are a number of other ways (the "VCS" menu, short-cut keys) to do the same things. At some point your IDE will likely offer to add a number of IDE-related files to the Git repo for the project. You can safely tell the IDE not to add these to the repo, and in fact this is probably the best course of action for now.

5 Further reading

We've only covered the basics of using Git in this tutorial. There are a range of advanced features and capabilities which would be of great benefit to you. There are many resources available online. Atlassian has a particularly good set of tutorials, available at https://www.atlassian.com/git/tutorials/. Also, we recommend the following resources:

- An exhaustive Git reference is available at http://git-scm.com/book/en/v2
- Good practices for commit messages: https://wiki.openstack.org/wiki/GitCommitMessages
- Learn Git in a web browser in 15 minutes: https://try.github.io/levels/1/challenges/1