

# SENG202 – Software Engineering Project Workshop

2016

## Tutorial 2 – Version control with Git

### 1. Introduction

Source control is a requirement for professional software development. This tutorial will show you the basics of working with the code management system known as Git. As they work on a project, software engineers create many iterations (versions) of various 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 (eg: 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 this year) work in tandem with Git to help multiple engineers work on the same project at the same time.

Alternatives to Git that you may have heard of are Mercurial or Subversion.

#### Objectives

- Understand what Git is and the problems it aims to solve.
- Know how to create a Git repository, store code in it, and how to update that code.
- Understand what GitLab is and the problems it aims to solve.
- Individually practice using Git and GitLab with a test repository.
- As a group create a Git repository to use for SENG202 and link it to GitLab.
- Ensure that your development machine(s) are configured and can push and pull to and from the group GitLab repository (**this is extremely important**).
- Complete the Git Learn quiz.

### 2. Motivation: Why use VCS?

Modern software projects can involve tens, hundreds, or even thousands of engineers. This can generate many millions of edits, all of which need to be coordinated. The situation is further complicated by user requirements which may oblige a project to support multiple platforms, various versions of those platforms and a patchwork of dependencies, all of which can vary for interlinked reasons. All of which is to say: There's a lot of data involved in a big software project and managing it is tough.

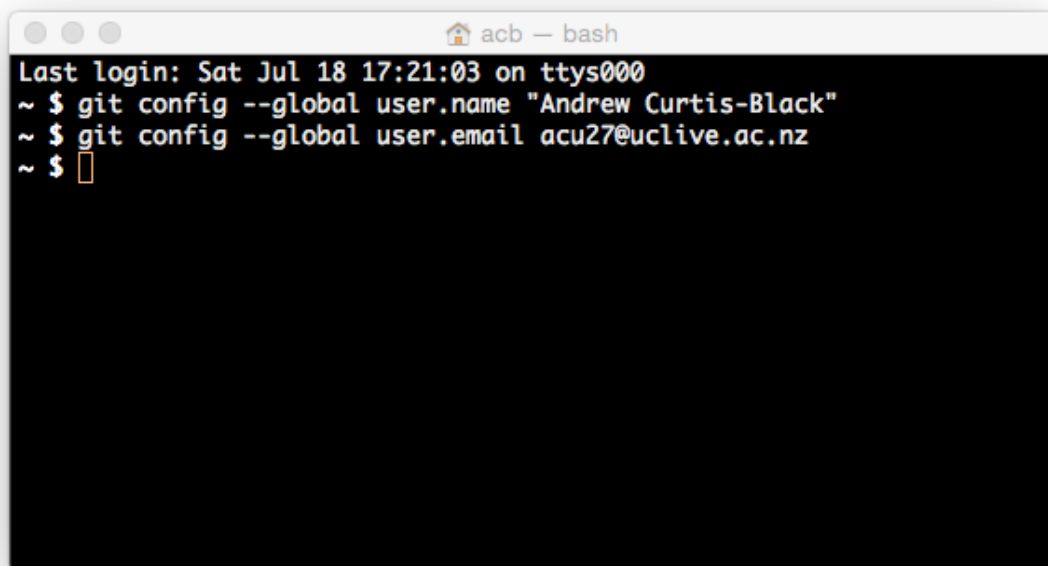
There are several popular version control management systems including Git, Mercurial and SVN. In this lab we will focus on Git, but many of the skills you will learn are transferrable to other systems. You will use Git to manage your SENG202 project code.

### 3. Worked Example

Git is a very powerful tool, but it can be a bit complex to work with and there are a number of concepts you'll need to understand. This worked example will take you through creating a Git repository<sup>1</sup> 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>.

#### 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 as shown in the screenshot below.

A screenshot of a terminal window titled 'acb — bash'. The terminal shows the following text: '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', and '~ \$ ' followed by a cursor. The terminal has a dark background with light-colored text.

```
acb — bash
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
~ $
```

Figure 1. Configuring Git

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 teammates **and your markers**.

#### Creating a Repository

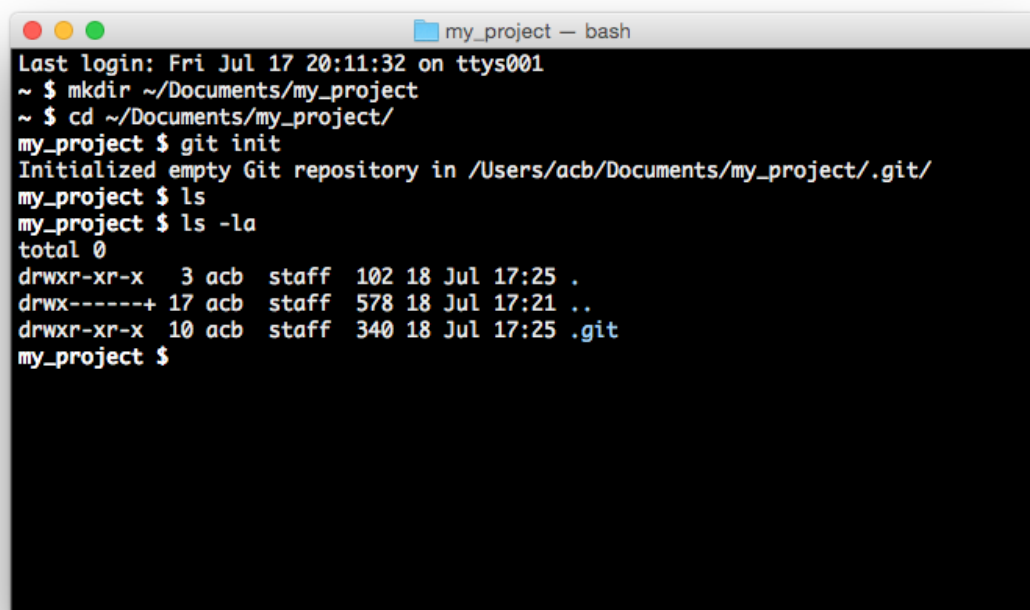
Next you need create a repository for your project. A repository is just a folder on your computer which contains all your project code. When you ask the Git program on your computer to help you manage the code in that folder Git will create a number of (hidden) files which contain information about your code and any edits to make to it. This is the difference between a proper 'repository' and just another folder with stuff in it.

---

<sup>1</sup> A 'repository' in this context just means a place where you store all of the code for a particular project.

Create a folder somewhere on your computer. This will be your repository. For example, in this example we will use `~/Documents/my_project`. Open your computer's command line and navigate to your chosen directory, as shown in the screenshot below. Now type "git init" to create your repository.

Note that Git will create an invisible `.git` directory which you can view with "`ls -la`" (the normal "`ls`" command will not display it). Usually you won't need to worry about this directory.



```
my_project — bash
Last login: Fri Jul 17 20:11:32 on ttys001
~ $ mkdir ~/Documents/my_project
~ $ cd ~/Documents/my_project/
my_project $ git init
Initialized empty Git repository in /Users/acb/Documents/my_project/.git/
my_project $ ls
my_project $ ls -la
total 0
drwxr-xr-x  3 acb  staff  102 18 Jul 17:25 .
drwx-----+ 17 acb  staff  578 18 Jul 17:21 ..
drwxr-xr-x 10 acb  staff  340 18 Jul 17:25 .git
my_project $
```

Figure 2. Creating the Repository

### 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 <filename>`" command. It tells Git to add the changes made to the specified file to the next commit. You can think of creating a file is a special case of making a change to that file.<sup>2</sup>

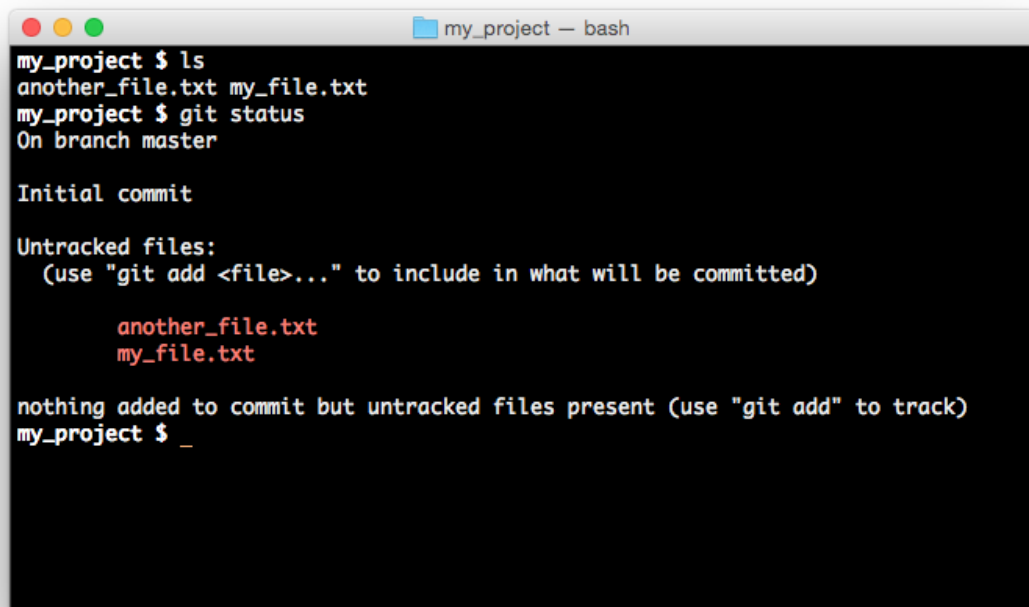
This is the basis of one of the core concepts of Git: **staging**. The staging area is where changes are stored before they are committed to a repository.

---

<sup>2</sup> Note that it is possible to tell Git to permanently track a file, so that any changes made to it are automatically included in the next commit you author (but we won't worry about that in this tutorial).

You can use the “git status” command to identify the state of files in the repo (unmodified, modified etc.)

Create some files (any way you like) and move them into the repo. When you check the status of the repo you’ll see that the files you added are “untracked”.<sup>3</sup> This means that you won’t be able to send those new files (or any changes made to them) into the repo.

A terminal window titled "my\_project - bash" with a dark background and light-colored text. The user has run 'ls' showing 'another\_file.txt' and 'my\_file.txt'. Then they ran 'git status', which shows 'On branch master' and 'Initial commit'. It then lists 'Untracked files:' as 'another\_file.txt' and 'my\_file.txt' in red. A message follows: 'nothing added to commit but untracked files present (use "git add" to track)'. The prompt 'my\_project \$' is followed by an underscore.

```
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

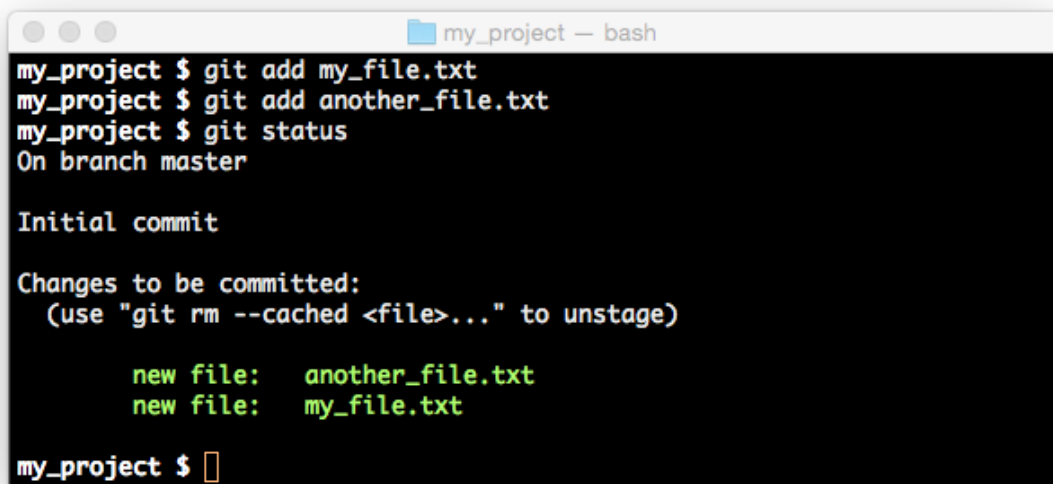
nothing added to commit but untracked files present (use "git add" to track)
my_project $ _
```

Figure 3. Checking the status of the repository

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

---

<sup>3</sup> We’re glossing over the difference between tracked and untracked files here. Feel free to look online and/or ask a staff member.

A terminal window titled "my\_project — bash" with a dark background. It shows the following commands and output:

```
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

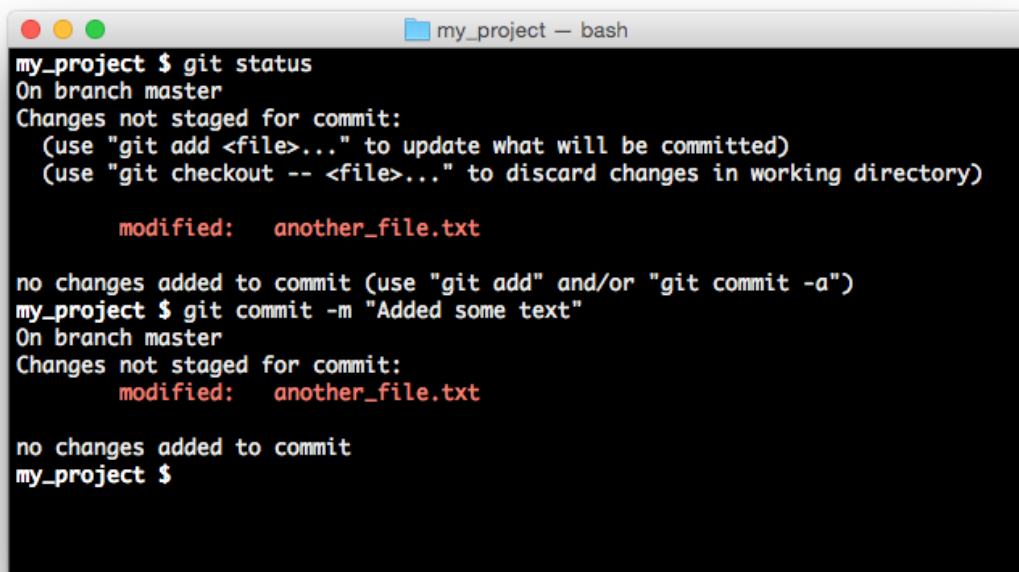
my_project $
```

Figure 4. Staging files

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 not. 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 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 fix the problem).

A terminal window titled "my\_project - bash" with a dark background. It shows the execution of "git status" and "git commit -m 'Added some text'". The output of "git status" shows "another\_file.txt" as modified but not staged. The output of "git commit" shows the same file as modified but not staged, indicating the commit was not successful.

```
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)

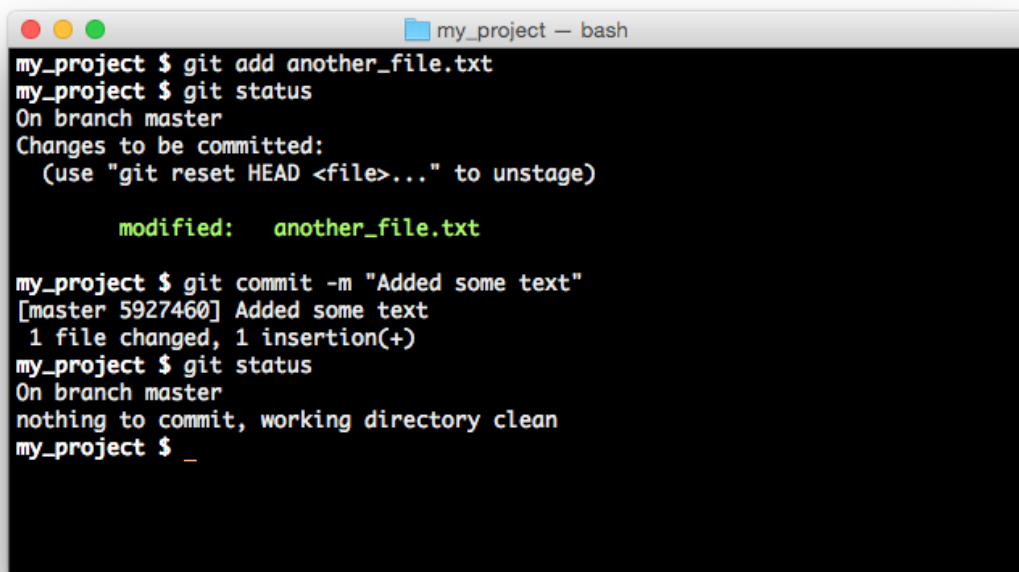
        modified:   another_file.txt

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 $
```

Figure 5. Viewing modified files

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™).

A terminal window titled 'my\_project - bash' with a dark background and light-colored text. The window shows the following sequence of commands and their outputs:

```
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 $ _
```

Figure 6. Making a commit

So, to review.

1. When you first start using Git on a new machine you should set your user credentials (user.name and user.email)
2. When you want to create a new Git repo you navigate to your target directory and use “git init”
3. Before you can commit changes to the repo you first need to stage them with “git add <filename>”
4. To commit changes to the repo use “git commit -m <descriptive message>”

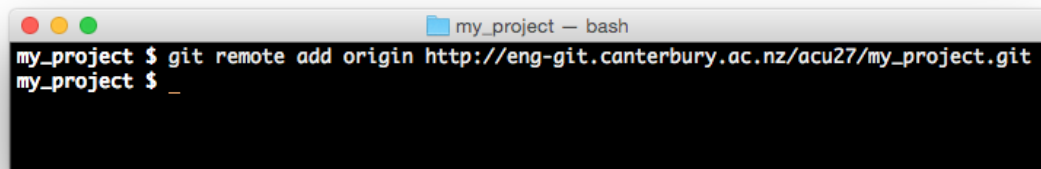
### Using Git with a remote Repository

Up until now we have only talked about using Git to manage a project stored on a single machine. In industry (and in SENG202) projects are almost always collaborative endeavours, meaning that you and your teammates will need to be able to synchronise your repositories. The way to achieve this is to create a remote repository on a server somewhere (the engineering department at UC runs a server for this purpose) with which each developer can synchronise. This means that the remote repository can be thought of as the “master” copy of the repository (in fact, it’s often referred to as the “origin”).

The first step is to create a remote repo. In a web browser open the page for the university’s GitLab server at <https://eng-git.canterbury.ac.nz> and click the plus button in the top righthand corner of the page to create a new project. GitLab is a server for Git repositories (you may have heard about other similar services such as GitHub or Bitbucket). Give it the same name as your local repo (in this example we’ve been using “my\_project”). GitLab will create a new (empty) repo on the server and

display some handy reference information for using Git. Near the middle of the page is a URL for the repo. Change the toggle from SSH to HTTP and copy the URL.

Go back to the command line on your machine and link your local repo to your new remote one with “git remote add” as shown below. NB: GitLab formats URLs for its repos with port numbers, and this sometimes causes problems. We suggest removing the port number from the URL before linking the repo, as shown below.

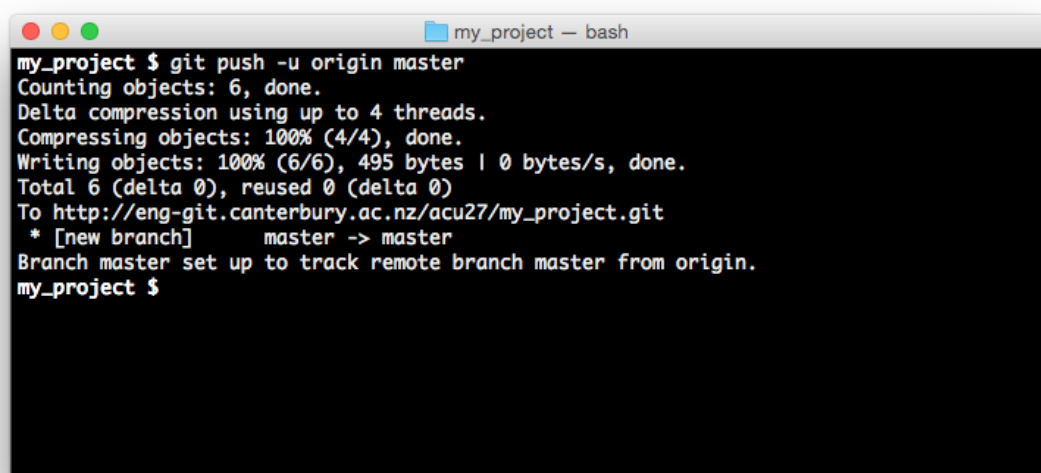
A terminal window titled "my\_project - bash" with a dark background. The prompt is "my\_project \$". The command entered is "git remote add origin http://eng-git.canterbury.ac.nz/acu27/my\_project.git". The prompt changes to "my\_project \$ \_" after the command is executed.

```
my_project $ git remote add origin http://eng-git.canterbury.ac.nz/acu27/my_project.git
my_project $ _
```

Figure 7. Adding a remote repository

The word “origin” in the command above is an alias for the URL we entered. This allows you to refer to the remote repo by a convenient name, rather than a long URL. We chose to refer to our remote repo as “origin” because that is the convention for Git repos.

Now you will “push” the changes in your local repo to your remote repo, effectively synchronising the two. There’s a bit going on in the command shown below, mainly because this is the very first push made to the remote repo. Don’t worry too much about the details, but it’s worth noting that the word “master” in the command is the name we’re giving to the main “branch” of our repo. We won’t cover branching here, but for now just remember that Git repos usually have one main branch and that by convention it’s called “master”.

A terminal window titled "my\_project - bash" with a dark background. The prompt is "my\_project \$". The command entered is "git push -u origin master". The output shows the progress of the push, including counting objects, delta compression, and writing objects. It also indicates that a new branch is being created and that the local master branch is now tracking the remote master branch.

```
my_project $ git push -u origin master
Counting objects: 6, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (6/6), 495 bytes | 0 bytes/s, done.
Total 6 (delta 0), reused 0 (delta 0)
To http://eng-git.canterbury.ac.nz/acu27/my_project.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
my_project $
```

Figure 8. First push to remote repository



After making your first push refresh the main page of your remote repo on GitLab in your browser. You should see an update like this under the activity tab.

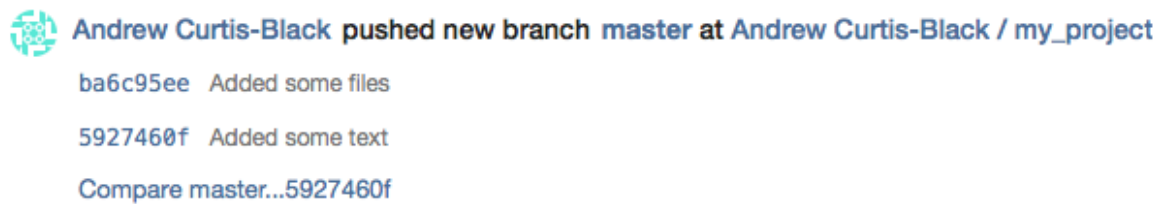


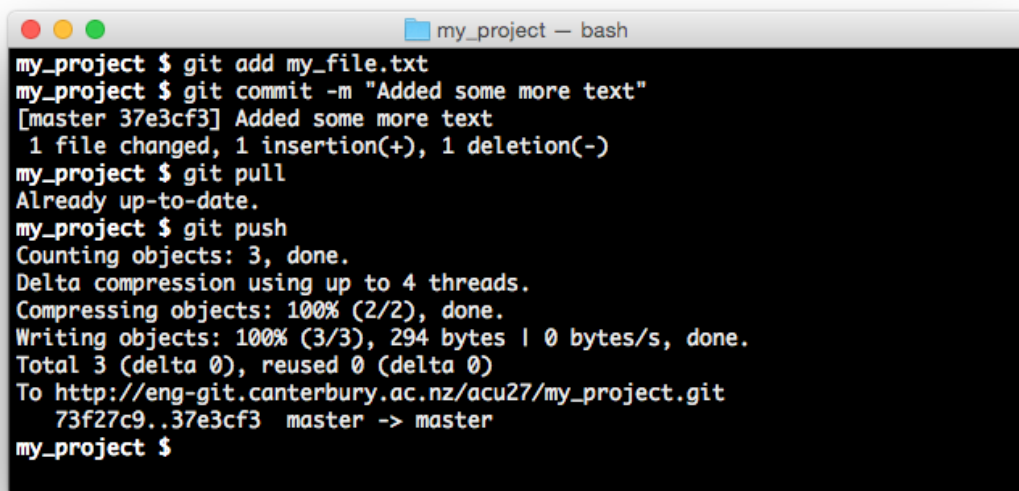
Figure 9. GitLab after first push

If you look under Files tab on GitLab you should be able to view the files you added earlier, 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. Just as making changes to a file in your local repo does not allow you to commit those changes immediately (remember that you have to use “git add” to stage the changes first), simply pulling new changes down from the server will not apply those changes to your repo. After a pull you need to do a Git merge to integrate the changes from the server with your local copy of the repo. Git can do some parts of a merge for you, and this makes some merges extremely quick and easy. However there are some things you will have to do manually, and you will have plenty of time in this course to discover what is affectionately known as “merge hell”.

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.

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 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, but it’s still good to get into the habit of doing this), then push your changes. Check GitLab via your browser to make sure your changes made it through.

A terminal window titled "my\_project - bash" with a dark background and light text. It shows a sequence of git commands and their outputs. The commands are: "git add my\_file.txt", "git commit -m 'Added some more text'", "git pull", and "git push". The outputs show the commit process, including file changes, compression, and writing objects, followed by a successful push to the remote repository.

```
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 $
```

Figure 10. Second push to the remote repository

Note that this time we didn't need to add any extra options to the push command.

To review: When you introduce a remote repo your workflow will remain mostly the same, but with some additional steps.

1. Make some changes
2. Stage those changes with "git add"
3. Commit those changes to the local repo
4. NEW: Pull the latest changes down from the remote repo
5. NEW: 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. NEW: Push your staged commits to the remote repo.

## 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. Standalone applications like Tortoise and GitUp let you work on your Git repos more efficiently, and modern IDEs like IntelliJ and Eclipse have built-in support for most popular VCS systems (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 IntelliJ and use Git through the IDE's interface without further set up. If you prefer to use Eclipse then you will need to install and configure the Egit plugin (we leave it to you to refer to the documentation and tutorials you will find online and on Learn). Everything covered in the command line in worked example can also be done in your IDE's interface. We highly recommend using a graphical tool like IntelliJ or Eclipse to perform Git merges.

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. 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.

You should also note that you can assign local changes to a number of different commits. This means that you can make a number of changes and then decide afterwards how they should be logically divided, rather than having to constantly interrupt your workflow to make commits.

## 5. Git Best Practices

Always use the same Git user name and email address for the same project. Otherwise your teammates and markers will see commits from what appear to be several different people who are in fact all the same person. You should also avoid using an alias like "MegaCoder999" as your Git username as no-one will know who really made some commits.

Ensure that you make regularly make commits. Waiting a long time between commits leads to a number of problems. Firstly, it can result in a much larger number of changes per commit, which in turn makes the impact of a commit harder to determine (imagine that you are trying to work out which change introduced a new bug).

You should also push to the remote repo fairly frequently so that your teammates can keep abreast of the latest changes. If you wait a week to push your changes then you will have to deal with the hassle of merging a large number of changes into the repo all at once. It's much easier to merge small changes than large ones, especially if there are a lot of people working from the same repo.

Try to keep commits which contain only minor edits to a minimum. Correcting the spelling of one word may save you some minor irritation, but doing so 23 times across the same number of commits will be annoying for your teammates. If you can, make all of these changes at once and push them as part of a single commit.

All of the changes in a single commit should be logically connected. Imagine that you will one day need to "undo" your changes. If your commits contain changes which are not related to one another

then when you “undo” a commit you are more likely to lose some changes that you wish to preserve.

Run your project’s unit tests before pushing anything to the remote repo. Pushing broken code can (and does) cost organisations money and has caused many SENG202 and SENG302 teams to have very late nights.

Finally, you should treat your VCS history as a part of the project itself. Just as your codes should be neat, well documented and tested, your VCS should be in good order with logically connected commits and **descriptive commit messages**.

## 6. Additional Tasks

Before finishing this lab session you should set up a remote repo for your SENG202 team to use for the project. **Use Maven to scaffold the project (refer to tutorial 1). This is compulsory.** Use ‘git init’ to create a Git repo for the Maven project and then set it up with GitLab so that all members of your group can access it. Try pushing some changes between your machines to make sure everything’s working properly.

Note that you can use the “git clone” command to create a local repository which is a copy of an existing remote repo. This means that just one teammate can set up the repo with the online GitLab interface and then all of you can quickly and easily clone the repository to your local machines.

## 7. Further Reading

We’ve only covered the absolute basics of using Git in this tutorial. There are a range of advanced features and capabilities which would be of great benefit to you. In the meantime 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:

- Overview of Git workflows: <https://www.atlassian.com/git/tutorials/comparing-workflows/>
- An exhaustive Git reference is available at <http://git-scm.com/book/en/v2>
- Overview of GUI clients for browsing Git repositories: <http://git-scm.com/downloads/guis>
- Good practices for commit messages: <https://wiki.openstack.org/wiki/GitCommitMessages>
- Learn Git (in web browser): <https://try.github.io/levels/1/challenges/1>

## Appendix I – Basic command line commands for Git

Command	Description
<code>git status</code>	Status of staging area
<code>git add</code>	Add files to staging area; before you add a file to the index, Git does not know that you want this file involved in. You need to add it to the index so Git knows to 'share' it; 'add' is also used to mark conflicts as resolved
<code>git commit -m [commit message]</code>	Commit changes in the staging area; makes Git work out what has changed since the last revision in your local repository and records the difference as the most up to date change
<code>git push</code>	Push changes onto server; tells remote repository all of the new commits from your local repository for each file; this cannot work if remote repository has a revision for the file higher than what you were working on in your local repository
<code>git clone</code>	Copy a repository
<code>git checkout</code>	Download a repository; this will also revert any changes you have made that have not been committed
<code>git pull</code>	Fetches the changes from the remote repository, and merges them in with your repository, so you are up to date with what is on the server; this can create a merge conflict, which can be resolved as detailed below
<code>git mergetool</code>	When two people have been working on the same line in the same method in a different way, a conflict is created; use mergetool to tell Git how things should be resolved
<code>git blame</code>	Shows name and email of the committer for every change and side by side comparison of what they did (in Eclipse, "Team" → "Show in history")

Please note that 'push' and 'pull' is done at a project level, commits are done at a file level. You will notice that instructions for command line Git, and the extra tasks are a lot sparser. It is assumed people attempting these things have a higher level of understanding, or are willing to do the extra research. There are few formal labs in this course, and it is thoroughly recommended that you do your own research and reading along with everything included in this lab.

## Appendix II – Git Ignore

There are some files which are better left out of source control. For example, Eclipse and IntelliJ both generate project and settings files. Some of these files can be shared between developers but others should not (precisely which files are safe to commit varies by IDE and by development team). To start with we suggest that you simply leave all such files out of your shared repository. The Git ignore file (located at `my_project/.gitignore`) allows you to specify files which Git should completely ignore, and which should not be committed. We encourage you to look online for more information as there are many good tutorials and reference materials available. Note that you can edit the Git ignore file from the command line, a text editor, or graphically via most IDEs.