SENG202 – Software Engineering Project Workshop

2020

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 (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 this year) work in tandem with Git to help multiple engineers work on the same project at the same time.

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 push 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 quiz on the Quiz Server. The quiz will be open in the second half of the tutorial time, and close at the end of the tutorial.

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 is a lot of data involved in a big software project and managing it is tough.

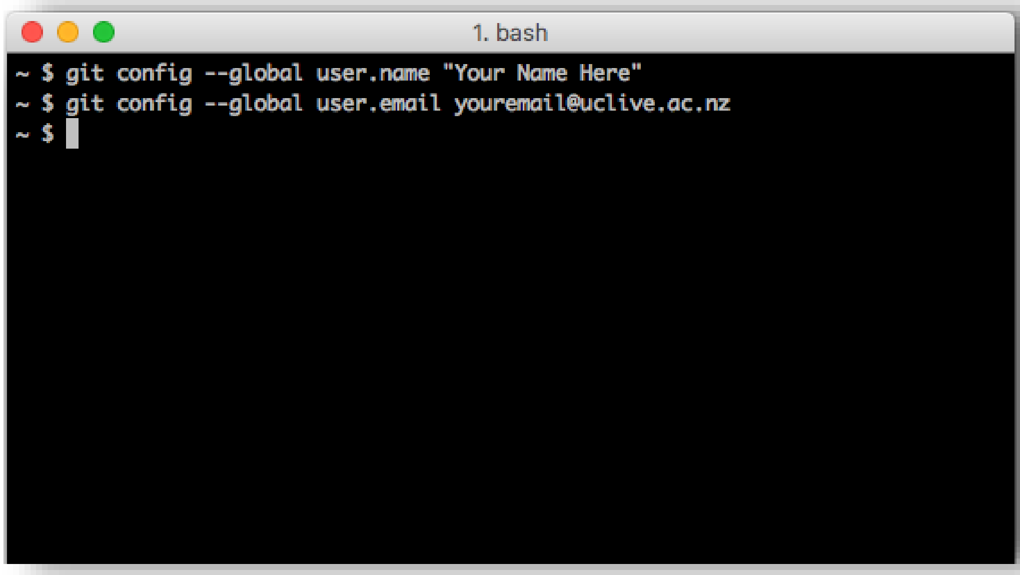
There are several popular version control management systems including Git, Mercurial and SVN (Subversion). In this lab we will focus on Git, but many of the skills you will learn are transferable to other systems. You will use Git to manage your SENG202 project code.

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[1] 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 all Macs come with Git pre-installed, but others who need to install it should visit: http://git-scm.com/downloads.

Configuring Git

Before you start it is best to set some of your Git user credentials. Your Git username and email address will be used to identify your contributions to the repo. Set them as shown in the screenshot below.



```
~ $ git config --global user.name "Your Name Here"
~ $ git config --global user.email youremail@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` (username 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**[2].

Creating a Repository

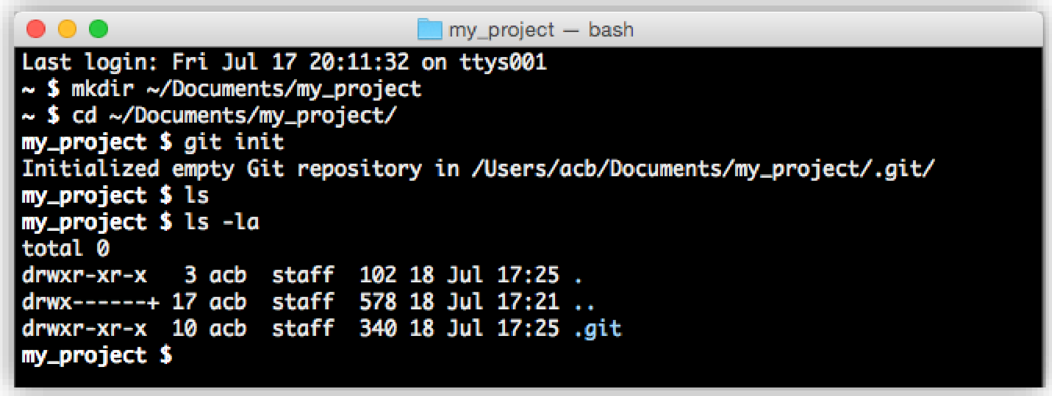
Next you need to create a repository for your project. A repository is just a folder on your computer which contains all your project code. When you ask the Git program on your computer to help you manage the code in that folder Git will create a number of (hidden) files which contain information about 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.

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

Note that Git will create a hidden directory called ".git" 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.

[1] A 'repository' (or repo) in this context just means a place where you store all of the code for a particular project.

[2] It will annoy them a lot – and nobody wants that…



```
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 $
```

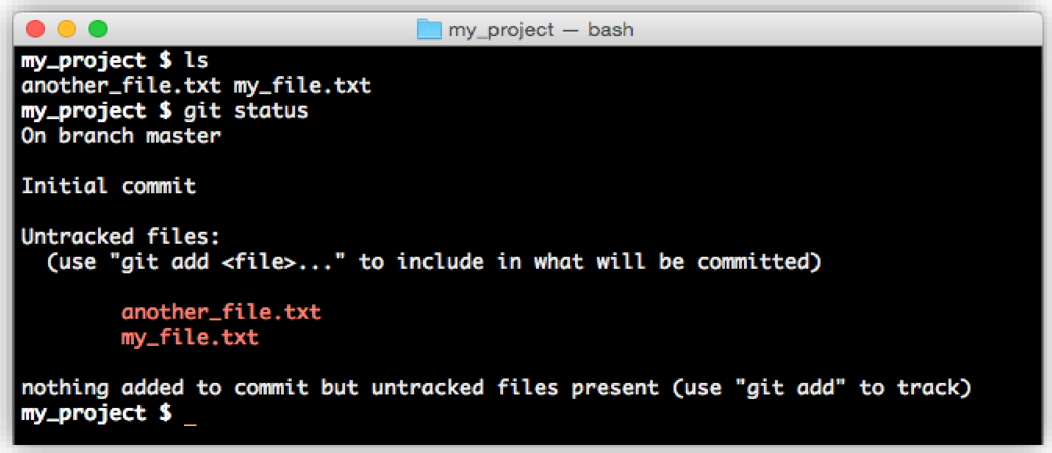
Figure 2: Creating the Repository

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 change sets. 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 as a special case of making a change to that file.[3]

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 repo (unmodified, modified etc.)

Create some files (any way you like) and move them into the local repository (directory where you ran `` `git init` ``). When you check the status of the repo you'll see that the files you added are "untracked".[4] This means that you won't be able to send those new files (or any changes made to them) into the repo.



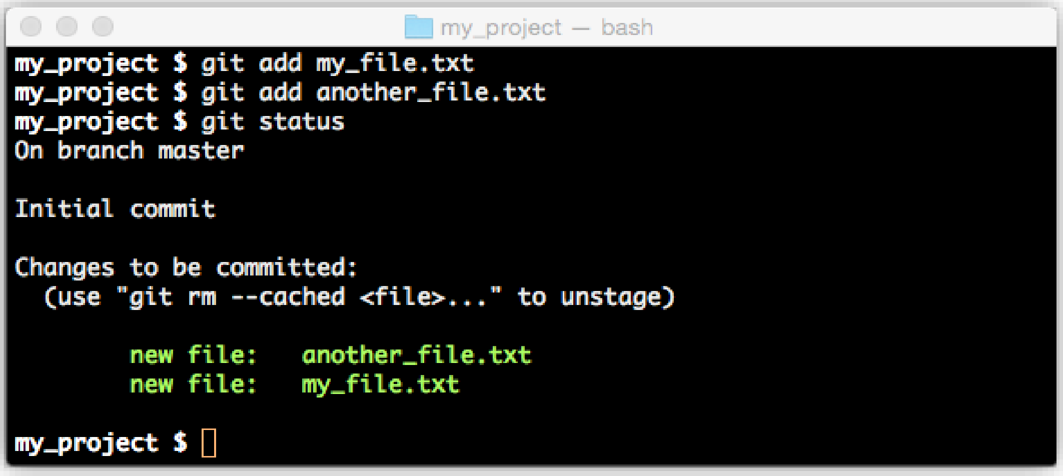
```
my_project $ ls
another_file.txt my_file.txt
my_project $ git status
On branch master

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 $ _
```

Figure 3: Checking the status of the repository

[3] 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).

[4] We're glossing over the difference between tracked and untracked files here. Feel free to look online and/or ask a staff member.

Now use the "git add" command to "stage" the files.



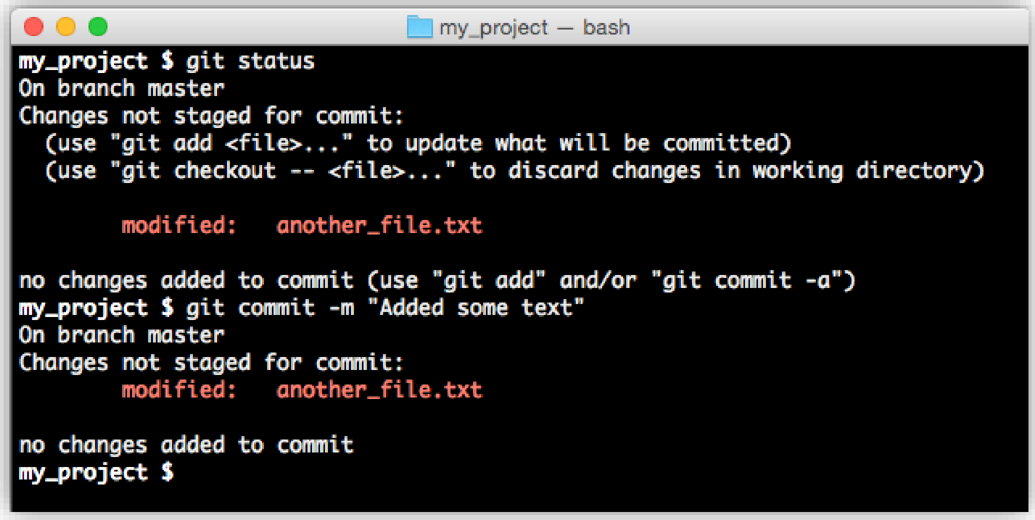
```
my_project $ git add my_file.txt
my_project $ git add another_file.txt
my_project $ git status
On branch master

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 $ 
```

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.

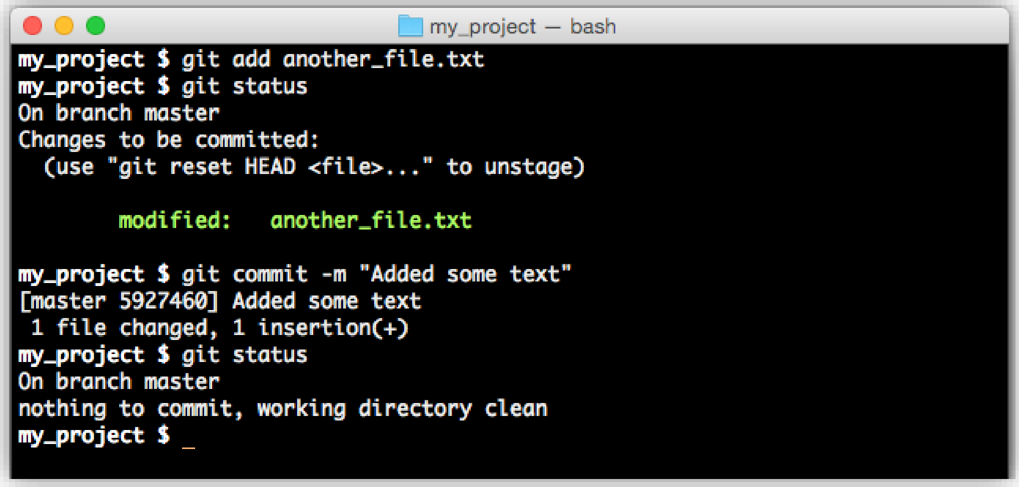
To try all this out open one of your files and edit it somehow, then check the status of your repo. You'll see that Git has noticed that one of the files it's tracking has been modified. You can't commit this change just yet; if you try Git will tell you that no changes were added to the commit, and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



```
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™*).



```
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

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>`

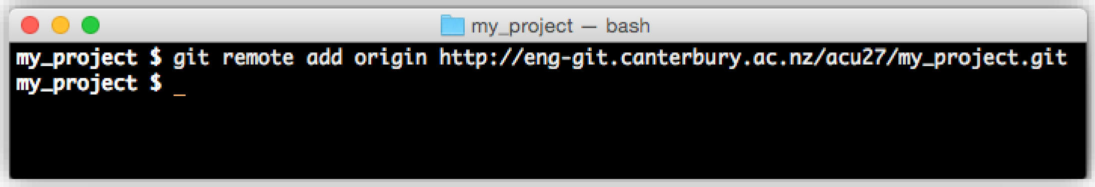
Using Git with a remote Repository

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 right-hand 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 HTTPS and copy the URL. The reason that we are using HTTPS instead of SSH is because it is much simpler to set up.

With HTTPS, you will need to enter your UC username and password each time you push or pull from the remote repository. The SSH process involves generating private and public keys which will identify your system. The downside of this, is that it is inherently more complicated, especially if you will be using multiple machines (more information about SSH in GitLab can be found here[5]).

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.

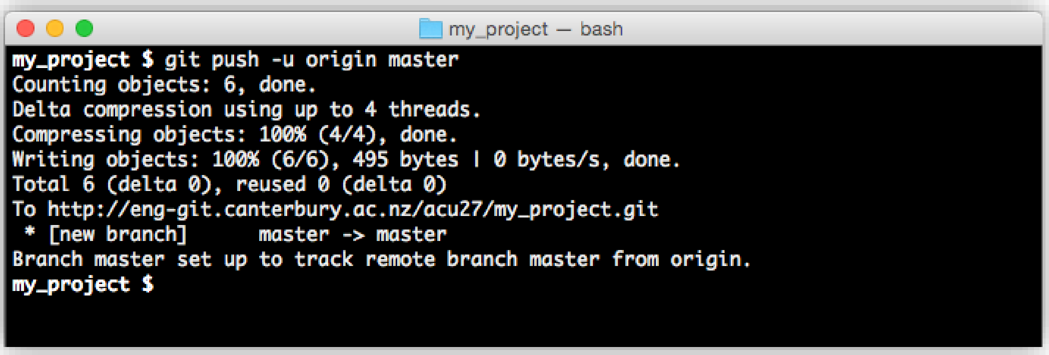


```
my_project $ git remote add origin http://eng-git.canterbury.ac.nz/acu27/my_project.git
my_project $ _
```

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 refer to our primary remote repo as "origin" because that is the convention for Git repos.

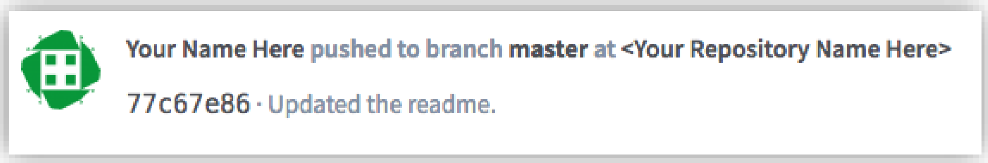
Now you will "push" the changes in your local repo to your remote repo, effectively synchronising the two. There's a bit going on in the command shown below, mainly because this is the very first push made to the remote repo. Don't worry too much about the details, but it's worth noting that the word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



```
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.



Your Name Here pushed to branch master at <Your Repository Name Here>

77c67e86 · Updated the readme.

Figure 9: GitLab after first push

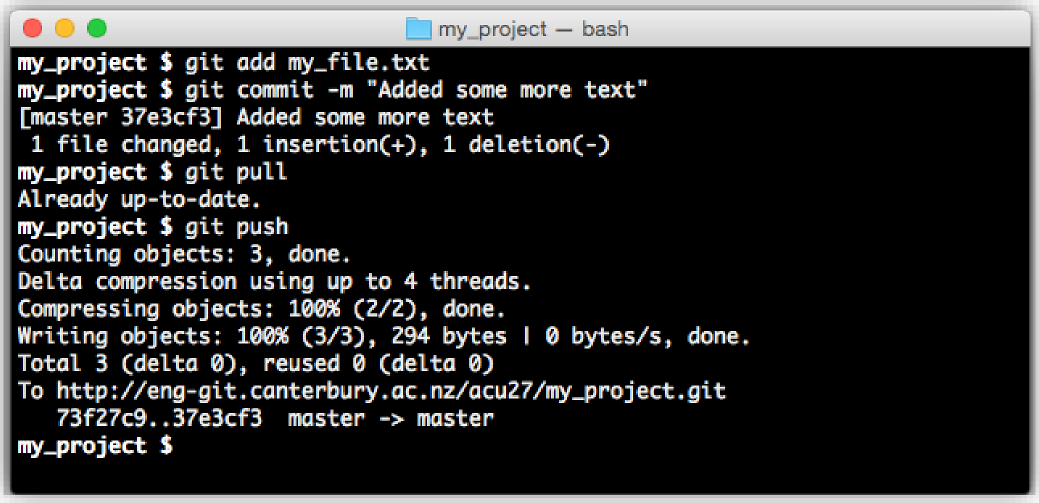
[5] http://docs.gitlab.com/ce/ssh/README.html

If you look under the 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. Pulling the new changes down from the server will attempt to apply those changes to your repo, by automatically performing a merge. Git can do some parts of the merge for you, and this makes some merges extremely quick and easy. However, there are some things you will have to do manually (such as when there are conflicts, e.g. two people editing the same method), 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 tutors 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.



```
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.

Recapping

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

1. Make some changes
2. If you create a file(s) that are not currently staged, add them to git with `` `git add` ``
3. Commit those changes to the local repo with `` `git commit -m <Descriptive Message Here>` ``.
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. Branching

A branch represents an independent line of development. Branches serve as an abstraction for the edit/stage/commit process discussed earlier. You can think of a branch as a way to request a brand new working directory, staging area, and project history. New commits are recorded in the history for the current branch, which results in a fork in the history of the project.

Using branches with Git

To list all the branches in your repository, the command `` `git branch` `` can be used. This should (hopefully at this point in the tutorial) list just one branch "**`master`**".

To create a new branch, we use the command `` `git branch <branch-name>` ``. Note that this does not check out the new branch, and can be verified by running `` `git status` ``. This will say on the first line "On branch master". When this branch is created, it will essentially be a duplication of the current branch you are on, in this case, master.

To check out a branch, we use the (surprise-surprise) `` `git checkout` `` command as follows `` `git checkout <branch-name>` ``. We can verify that we have checked out the new branch by again, running `` `git status` ``. Commits can now be made and pushed to this branch just as though you were on the master branch. Make some changes, and push a new commit to this branch, using the knowledge you have learned from the previous exercises. Running the command `` `git push` `` will likely fail, as there is no remote branch with that name, therefore the first time you commit to a new branch, the following command will be needed:

```
`git push –set-upstream origin <branch-name>`
```

Subsequent pushes to this branch can simply be completed using the `` `git push` `` command.

<u>Protip:</u> you can perform branch creation and checkout in the same command using `` `git checkout –b <branch-name>` ``

The diagram in Figure 11 below shows an example of how the commits on each branch are connected (each circle represents a commit).

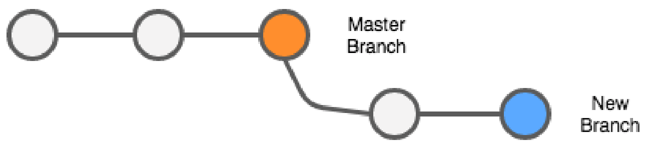


Figure 11: Repository after committing to a new branch

When you have finished making changes on this branch, we can merge this branch into the master branch. This is where the `` `git merge` `` command comes in handy. This command lets you take the independent lines of development created by `` `git branch` ``, and integrate them into a single branch.

To begin with, checkout the master branch and pull in any other commits that have been made while you were working on your own branch (Note you don't actually need to run `` `git pull` ``, since you are the only one working on this repository, but imagine in a team environment, there may have been many commits made to the master branch while you were working on another branch).

Run the command `` `git merge <branch-name>` `` (this should be the name of the branch you created previously). Git will attempt to merge the specified branch **into** the current branch. You can then issue the `` `git push` `` command to push these changes to the remote repository. The diagram in Figure 12 below shows an example of how the repository may look after completing the merge.

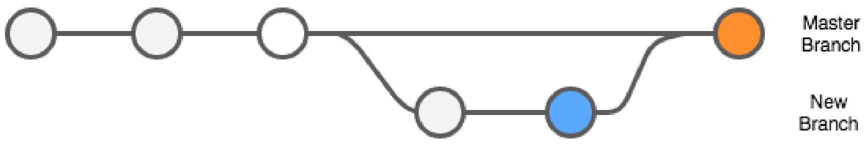


Figure 12: Repository after merging branch into master

For more information on branching, this link[6] may be of use, and the tutors are always happy to help.

Branching Strategies

One potential branching strategy is called "feature branching". The core idea behind the feature branching workflow is that all feature development should take place in a dedicated branch instead of the "master" branch. This encapsulation makes it easy for multiple developers to work on a particular feature without disturbing the main code base. It also means that the "master" branch will never contain broken code, which is **essential for maintaining a project that is always in a potentially shippable state**.

The master branch still represents the official project history. However, instead of committing directly on your local master branch, you create a new branch every time you work on a new feature. The branch should have a descriptive name like "`implementing-output-for-helloworld-class`". The goal here is to give a highly-focused purpose for each branch.

More on the branching strategies and their implementation can be found here[7].

5. 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 Tower, Tortoise, GitHub or SourceTree let you work on your Git repos more efficiently, and modern IDEs like IntelliJ, Eclipse and VS Code have built-in support for most popular Version Control Systems (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). 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 (`.gitignore` might be useful here), 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.

[6] https://www.atlassian.com/git/tutorials/using-branches/git-branch

[7] https://www.atlassian.com/git/tutorials/comparing-workflows/centralized-workflow

Basic Git functionality in IntelliJ

All of the given commands can be executed in IntelliJ fairly easily. Below are some screenshots of the shortcuts to the commands or the locations of useful features.

The file explorer (figure 13) shows the current status of the files in the project, colour coded to the current status. Red names are untracked files, green names are new files, blue names are modified files and gold names are ignored files (as specified in `.gitignore`)

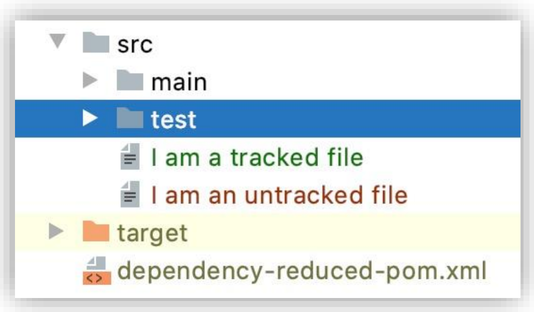


Figure 13: IntelliJ file explorer window

The commit menu can be accessed from VCS > Commit (CTRL + k) and contains many useful features such as import optimization (removing unused imports) and TODO warnings (searches for //TODO comments in code). Here commit messages can be written and the changes/files to commit can be selected.

The project update menu can be accessed from CVS > Update project (CTRL + t) and will pull the latest changes from the remote version of the current branch and merge those changes into the local version. Note: there is an option to rebase the changes and merge the changes. Only use the merge option because merging ***does not overwrite the commit history*** whereas rebasing does. Please remember that the assessors need your commit history and if we cannot see your work we cannot grade it.

Pictured 14 shows the branch management menu, by clicking on the current branch name at the bottom right of **InjelliJ** this menu is opened. This menu allows for creating new branches, switching branches and initiating merges as well as showing local and remote branches.

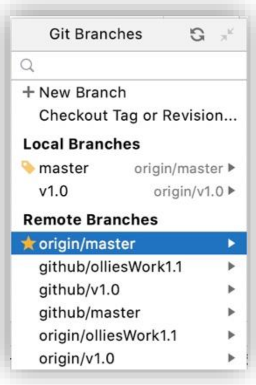


Figure 14: IntelliJ branch management menu

Useful keyboard shortcuts:

- CTRL + k: Opens the commit menu (analogous to `git commit` command);
- CTRL + SHIFT + k: Pushes to remote repo (analogous to `git push` command);
- CTRL + t: Opens the update menu (analogous to `git pull` command).

6. Git Best Practices

Always use the same Git username 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.[8]

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. For the same reason, you should also pull new changes frequently.

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 code should be neat, well documented and tested, your VCS should be in good order with logically connected commits and **descriptive commit messages** – (e.g. "made some changes" will not suffice!).

7. Additional Tasks

Before finishing this lab session, one (just one), person in your SENG202 team should set up a remote repo for you all to use for the project – make sure to add all members of your group to this repository (through the GitLab web interface). In addition to this, please ensure that the teaching team has been added to your repository (as reporters). Our user codes are as follows:

- pid15 – Patricia Inez
- sjs227 – Sam Shankland
- lwa383 – Luke Walsh

Each team member should now clone this repository (copies the remote repository) to their machines by issuing the following command from the directory where they would like the local repository to be created:

``git clone <git-url-here>`` (Remember to use the HTTPS link as mentioned earlier)

[8] And will annoy the markers…

When a remote repository is cloned, you will **not** need to run `` `git init` ``, nor will you need to add the remote repository as described earlier (by issuing the `` `git remote add` `` command).

One team member should use Maven to scaffold the project (refer to tutorial 1, a copy-pastable snippet can be also found on Learn). This is compulsory. Try pushing some changes between your machines to make sure everything is working properly.

8. 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
- Merge/rebase: https://medium.com/datadriveninvestor/git-rebase-vs-merge-cc5199edd77c

9. Useful software

Here are some useful tools that can help with visualising and/or enhancing the experience with Git.

- Visualising Git branches using GitKraken: https://www.gitkraken.com/
- Storage of large files in Git using GitLFS: https://git-lfs.github.com/

Appendix I – 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.

Here is a link to an ignore file for a maven project for both IntelliJ and Eclipse:

https://gist.githubusercontent.com/lordofthelake/5833336/raw/753d041f6d6947371e07e569aa6f4b18ed0759dc/.gitignore

Appendix II – Basic command line commands for Git

Command	Description
`git status`	Status of staging area
`git add`	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
`git commit -m [commit message]`	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
`git push`	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
`git clone`	Copy a repository
`git checkout`	Download a repository; this will also revert any changes you have made that have not been committed
`git pull`	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
`git mergetool`	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
`git blame`	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")

Table 1: Git basic command line

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.