# Lab 02 - Get Git Going

Getting right to the meat of things, let’s meet some of the commands used for us to drive Git with the command line. We’ll start from the beginning, - Initializing a new repository on your local machine or cloning a project from a remote Git server, like GitHub.

### Initializing & Getting Remote projects

|  |  |
| --- | --- |
| **Command** | **Description** |
| git init | Initialize a local Git repository |
| git clone [ssh://git@github.com/[username]/[repository-name].git](mailto:ssh://git@github.com/%5busername%5d/%5brepository-name%5d.git) | Retrieves remote repository |

## Basic Actions

|  |  |
| --- | --- |
| **Command** | **Description** |
| git status | Check status |
| git add [file-name.txt] | Add a file to the staging area |
| git add -A | Add all new and changed files to the staging area |
| git commit -m "[commit message]" | Commit changes |
| git rm -r [file-name.txt] | Remove a file (or folder) |

## Advanced Actions

|  |  |
| --- | --- |
| **Command** | **Description** |
| git branch | List branches (the asterisk denotes the current branch) |
| git branch -a | List all branches (local and remote) |
| git branch [branch name] | Create a new branch |
| git branch -d [branch name] | Delete a branch |
| git push origin --delete [branch name] | Delete a remote branch |
| git checkout -b [branch name] | Create a new branch and switch to it |
| git checkout -b [branch name] origin/[branch name] | Clone a remote branch and switch to it |
| git checkout [branch name] | Switch to a branch |
| git checkout - | Switch to the branch last checked out |
| git checkout -- [file-name.txt] | Discard changes to a file |
| git merge [branch name] | Merge a branch into the active branch |
| git merge [source branch] [target branch] | Merge a branch into a target branch |
| git stash | Stash changes in a dirty working directory |
| git stash clear | Remove all stashed entries |

## Sharing & Updating Projects

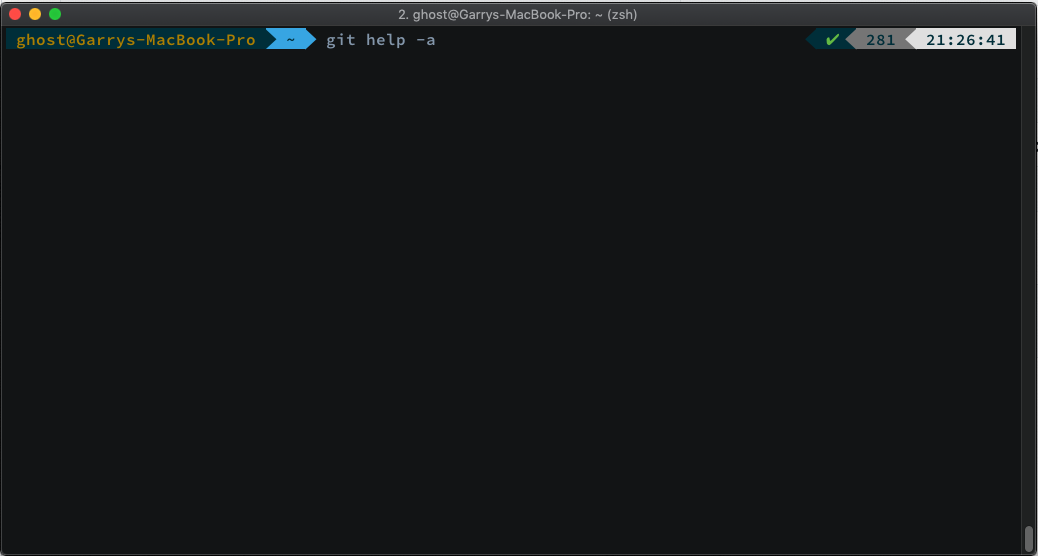
|  |  |
| --- | --- |
| **Command** | **Description** |
| git push origin [branch name] | Push a branch to your remote repository |
| git push -u origin [branch name] | Push changes to remote repository (and remember the branch) |
| git push | Push changes to remote repository (remembered branch) |
| git push origin --delete [branch name] | Delete a remote branch |
| git pull | Update local repository to the newest commit |
| git pull origin [branch name] | Pull changes from remote repository |
| git remote add origin [ssh://git@github.com/[username]/[repository-name].git](mailto:ssh://git@github.com/%5busername%5d/%5brepository-name%5d.git) | Add a remote repository |
| git remote set-url origin [ssh://git@github.com/[username]/[repository-name].git](mailto:ssh://git@github.com/%5busername%5d/%5brepository-name%5d.git) | Set a repository's origin branch to SSH |

## Checking Logs & Diffs

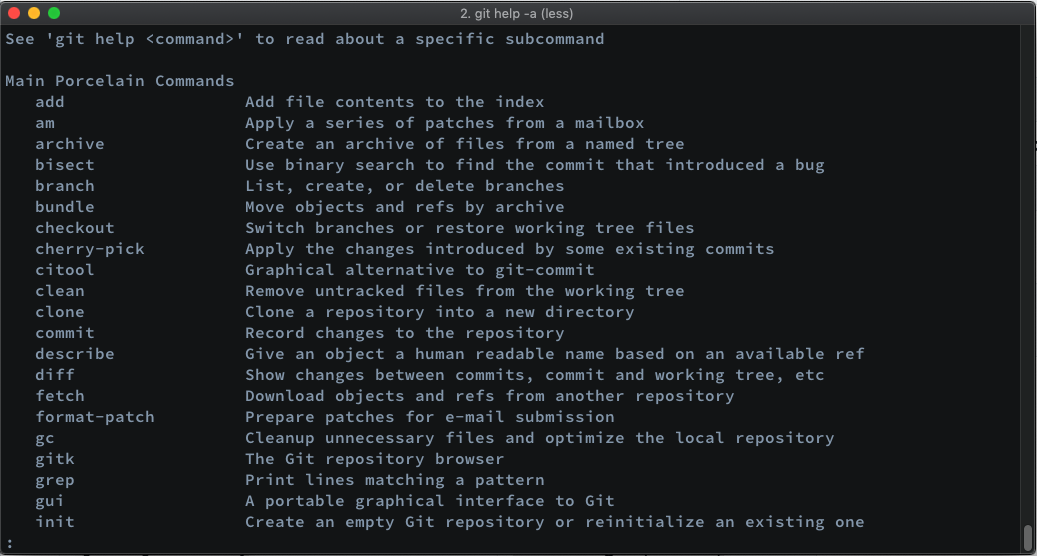
|  |  |
| --- | --- |
| **Command** | **Description** |
| git log | View changes |
| git log --summary | View changes (detailed) |
| git diff [source branch] [target branch] | Preview changes before merging |

Any Git commands not listed above, always feel free to check out their list of commands online or by using the help switch from the command line. What’s nice about using the help switch is that you’ll have a list of all git commands available with a description of what the command does. This is super helpful when you find yourself in a tough place and Google just ain’t helping.

Oh, and what’s a switch you might ask? **A switch is Linux lingo for a command line argument that is typically prefixed with a `-` or a `--`**. Check out the contextualized example below to both see a switch in action by calling `-a`. In the example below we are using the `-a` switch to display all commands available with Git.



The command used above is `git help -a`.



You may notice above that there is a colon in the bottom left corner looking like this:

`:`

In Linux style documentation, this typically means that there is either another page to view or an option to exit entirely. If you’d like to proceed with another page of documentation, you can hit the space bar for the next page. If you’d rather escape what you’re doing, just hit either the escape key or `q` to exit.

Now that we have the commands at our finger tips to get Git going, let’s start where we left off at the end of last week's exercise. Last week, we created our own COMP3104 directory using the command line and initialized the directory as a fresh Git repo. In case you want a quick reminder, check out the next image to see where we left things off.



Now that we have things initialized, we’re going to create a few files from the command line. After we’ve created some files using BASH syntax, we’ll refer to our list of Git commands, and start making changes with some of the command line editors we covered in our lecture.

## Touching Files from the CLI

When traversing a file system on the Command Line Interface (CLI), you may find yourself in need of creating a file just as you might when working on a project in your favorite IDE or text editor. Almost anything that can be done with some sort of GUI can also be done with the CLI. Let’s take a quick look at a common method for creating a file using BASH syntax.

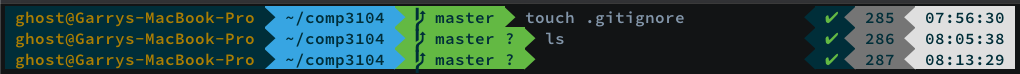
It’d be a shame if we were to create meaningless files for the sake of it so we’re going to kick things off by including a file that is typically first created in a new Git repo. This file is known as a `.gitignore`. A gitignore contains a list of files that one may not want to wind up on a public remote repository for security reasons or just plain cleanliness. Sometimes an IDE might create a specific file for its own use, for instance Vs Code will often times create a `.vscode` directory, or if you’re using a macOS system sometimes you may see a `.DS\_Store` file. Let’s create a `.gitignore` with the CLI and then populate it with a list of common filetypes that we may not want reaching GitHub when we go to push our file changes.



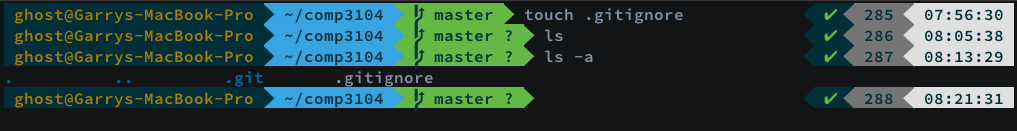
The above command used to create a file is `touch`. Using `touch` followed by an argument containing the name of the file we’d like to create gives us the ability to create a file with that name. In the above example, the entirety of the command is:

`touch .gitignore`

Since we wanted to create a file called `.gitignore`, we simply execute `touch .gitignore` and with the powers BASH grants us, we should now see that file in the directory. Let’s use `ls` again to list the contents of the directory to see if the file is there.



Odd. We know that the `ls` command is used to list the contents of a directory but no output is shown above displaying the file we just `touch`ed? The reason for this is that in **files prefixed with a `.` will make that file hidden**. This means that it’s not viewable via a file system explorer nor will `ls` show us the file. Well, that’s not entirely true. We can leverage a switch `ls` offers to show all files, regardless if they’re intended to be hidden on the file system or not. When we wanted to take a peek at all the commands Git has to offer, we passed a `-a` switch. This is a common switch syntax for displaying all of something. Let’s try passing this very same switch with `ls` and see what happens.



The above image shows the output of our directory contents when we pass the `-a` switch with `ls`. The command used was:

`ls -a`

This command will show hidden files and if you notice, you’ll see that there are two hidden files in our directory. Well, to be entirely truthful, one of them is a file (`.gitignore`) and the other is a directory (`.git`). When a directory is initialized as a Git repository, Git creates a folder to keep track of all the file changes. **Whenever a directory is initialized with Git using `git init`, it will always create a `.git` directory.**

Now that we’ve created our gitignore file, let’s fill the contents of it with some common files and directories that we wouldn’t want to reach our repo. As previously mentioned, this could be anything from a file containing credentials, sensitive information or general cruft like folders specific to an IDE like the .vscode folder.

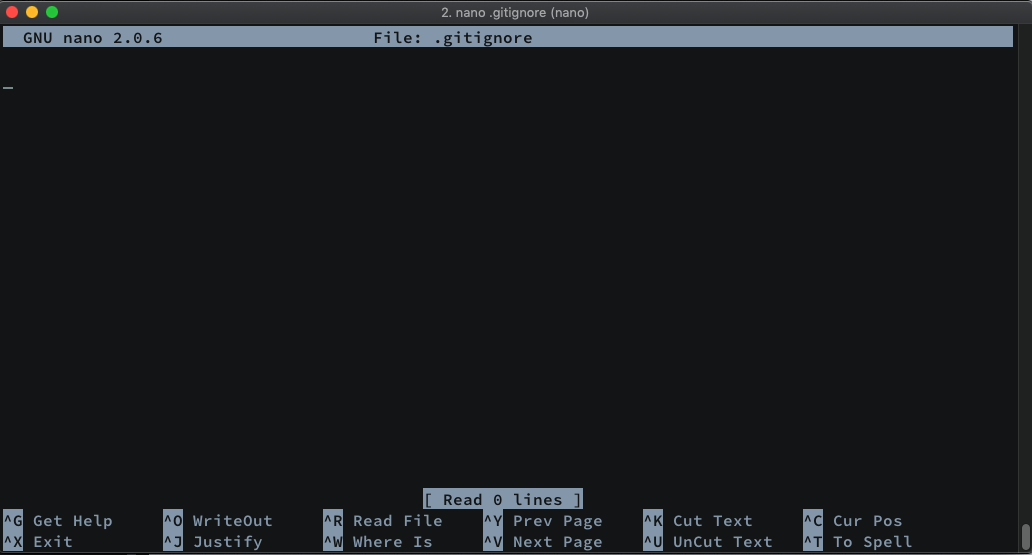
To edit a file from the command line, we’ll now take a quick peak at how this can be achieved with some commonly used CLI text editors. We’ll start things off with nano and then we’ll take a look at vi.



Using nano, we can open a file by executing `nano FILENAME`. Since we want to edit the .gitignore file, the command executed is:

`nano .gitignore`

Let’s peek at what this command will produce once we hit the enter key:



Just a couple quick things to point out above now that we have nano open. If you look at the top grey bar, you’ll see centered the word ‘File: ’ followed by the file we opened which was our gitignore. Thus making the entirety ‘File: .gitignore’. In nano, this will always tell us the file we have opened.

At the bottom of nano, there are some keyboard commands to help perform operations such as exiting, saving and some other stuff that we won’t need to cover for the purpose of what our current mission is. If your always curious though, you can use another command called **`man`**. The previously mentioned command is **short for manual**, what this will do is **produce a manual, or documentation on how to use the command line tool you’re using**. It’s out of scope and not something to be concerned about but if you’re curious, try using `man nano` to see what produces. It will produce a screen similar to what `git help -a` gave us. All the commands to navigate will be the same; space bar for the next page or escaping with `q`. See below for an example of what the `man` command will produce when executing `man nano`.

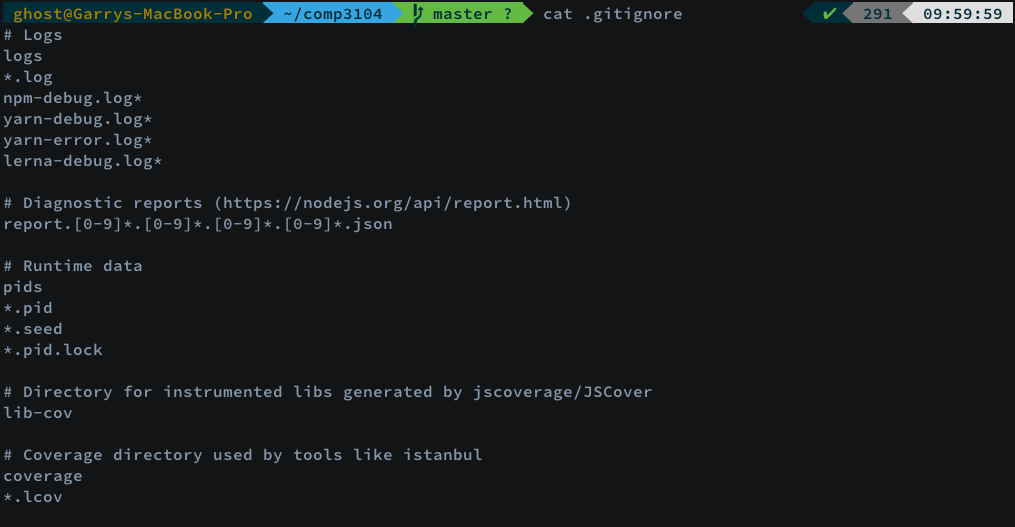


Back to our task at hand, which is populating our empty **.gitignore** with some **commonly found files that we might not want on our public repository** for either reasons of keeping things tidy or to enforce security of sensitive files. I’m going to paste in a list of common files and folders that a developer wouldn’t want to reach a public repo. You can find the contents of what I paste in the supplementary material section of this week's lab or by requesting the official gitignore for Node projects offered from GitHub at https://raw.githubusercontent.com/github/gitignore/master/Node.gitignore.

Once you have the contents of the .gitignore copied to your clipboard, we’ll be pasting it in nano. Follow the steps below to populate your gitignore with what’s found in our supplementary material for this lab.

1. Copy the contents of .gitignore either from the link found above or in the supplementary material folder of this week's lab.
2. Open the file we created with the `touch` command by executing `nano .gitignore` from the command line in our COMP3104 folder
3. Hit ctrl + x to exit the file
4. When asked to modify the file hit the `y` key followed by enter

There is another command we can use on the CLI to dump the contents of a file, this command is known as `cat`. Using `cat` followed by the filename we can see if our changes were saved. The `cat` command can also be used to take a sneak peak at the contents of a file without having to open the file with an editor.

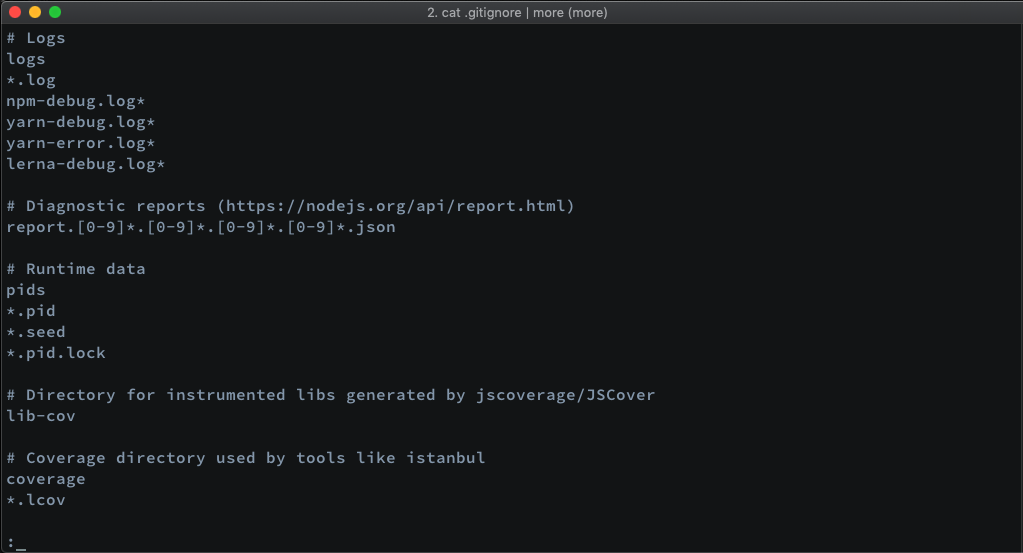


Above the command used was:

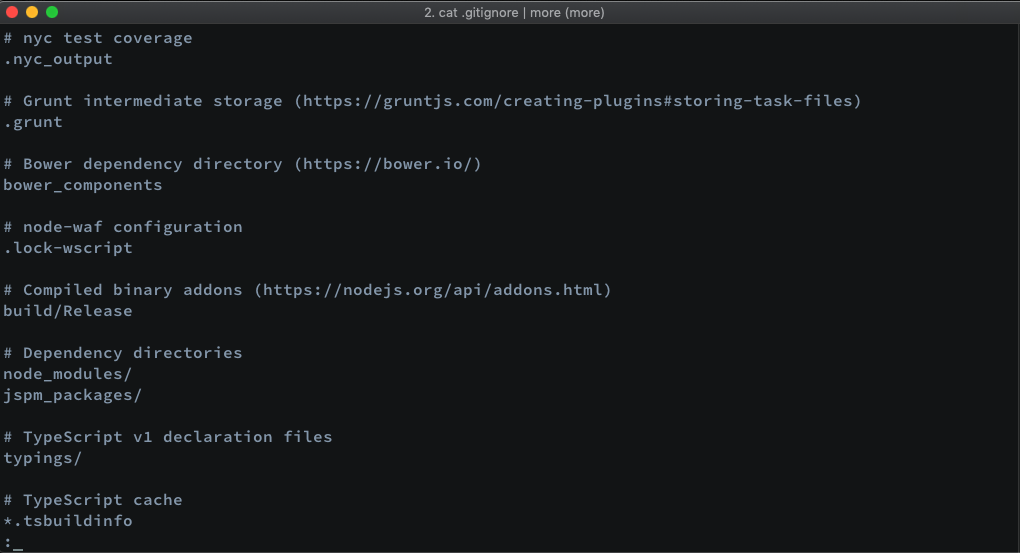
`cat .gitignore`

This will output the contents of the file in our terminal. What if the contents is rather long and we want to have control over the amount of content outputted? On the command line, there is another special key known as a pipe. A pipe is represented with the `|` key and it allows us to “pipe” the contents of a file into another command. Let’s try using the cat command again but this time we’re going to pipe the output into another command known as `more`. What `more` does is allow us have control when we want to see more of the output. Again, you’ll notice the `:` at the bottom left corner of your terminal just as we’ve seen when using the `man` command or listing all of the commands available from Git.

Here is an example:



Above is the first page outputted when combining the cat command with more using the pipe symbol (`|`). Check below for an example of what happens when the space bar is hit, you’ll see the next page of contents displayed. This procedure can be repeated until we hit the end of the file or hit the `q` key.



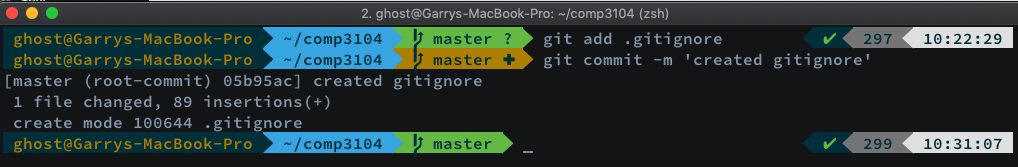
Above is an example of the following command:

`cat .gitignore | more`

We now have our first file filled with the contents we need to ignore file or directory specific items when working with Git. Now all we must do is use our Git reference found at the top of this lab document to add this file for version control tracking.

For a quick recap of what we’ve done; we now have a gitignore file which we created with the `touch` command and used `nano` to fill the file with some data which contains files and folders we don’t want to reach our git repository. Again, **any file or folder listed in .gitignore will be prevented from reaching our public repo.**

Let’s follow along and add our gitignore file for tracking and commit it with a message containing what we did. These steps will make the file ready for a push to our remote repo.



Here are the commands used above:

`git add .gitignore`

`git commit -m ‘created gitignore’`

We’ve now added our first file with Git and committed it for a push to our online repository. With this knowledge we now have a basic understanding of how we can add files for version control tracking and prepare them for a remote deployment to our online repository which we’ll be setting up in our next lab.