

# A Quick Git Guide

---

Are you interested in learning to program? You might start with Python, Javascript, or any other language you prefer, and begin creating small programs or scripts.

Think of Git as a tool for saving different versions of your code as you work on it.

After completing a specific feature or chapter, you can save the current state of your code. By regularly saving your changes, you create a timeline of your work.

This "timeline" serves several purposes:

- Reverting changes: If you encounter an issue after making changes to your code, you can go back to a previous version to see what's different.
- Managing different versions: If your code is working well and you want to add a new feature that might take a while, you can create a separate version of your code to work on the new feature while keeping the current version intact.
- Collaborating with others: You can share your code and its history using platforms like GitHub, making it easier to work with others on the same project.

If all this sounds a bit confusing, don't worry. It'll become clearer as we dive into practice.

## Install Git

First, we need to install Git. It's easy, I promise you, but for the sake of keeping this guide short, please head over to this very useful guide here for installation:

<https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>

All good?

## Get Started

Let's start things out by creating a project folder and a file within that folder, and we'll use this as our Git playground.

We'll be using Powershell for this as I will assume you are on Windows. The commands are pretty similar for Apple, except instead of New-Item you use "touch" instead:

```
mkdir test
```

```
cd test
```

Windows `New-Item file.txt`

Apple `touch file.txt`

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\dan.webb> mkdir test

Directory: C:\Users\dan.webb

Mode                LastWriteTime         Length Name
----                -
d-----         16/01/2024         10:11         test

PS C:\Users\dan.webb> cd test
PS C:\Users\dan.webb\test> New-Item file.txt

Directory: C:\Users\dan.webb\test

Mode                LastWriteTime         Length Name
----                -
-a-----         16/01/2024         10:11             0 file.txt

PS C:\Users\dan.webb\test>
```

## git init

Git will only track the projects you tell it to track, so let's tell Git to track this project folder with `git init`:

```
Windows PowerShell

PS C:\Users\dan.webb\test> git init
Initialized empty Git repository in C:/Users/dan.webb
PS C:\Users\dan.webb\test>
```

## Setting Up

If this is your very first time using Git, it'll be useful to tell it your preferred username and email. We're going to tell it, with the `--global` flag, to save this universally on your system so you don't have to tell it these things again later.

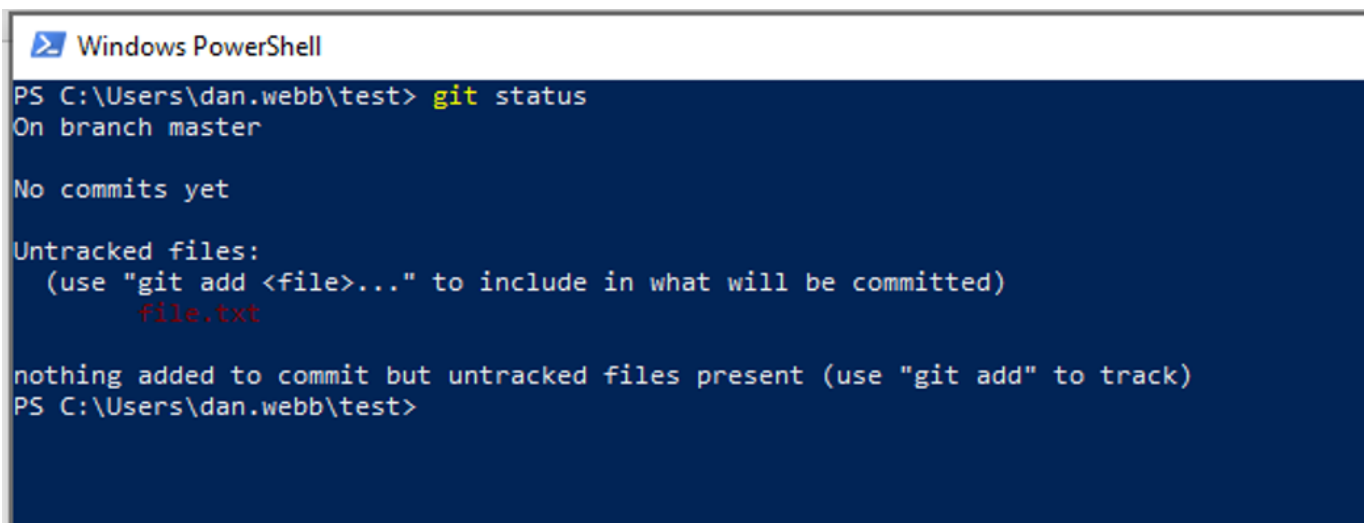
You don't have to do this each time you use git! Just the first time on a new computer you have installed git to!

```
git config --global user.name 'YOUR_PREFERRED_USERNAME'  
git config --global user.email 'YOUR_EMAIL'
```

## git status

`git status` is probably a command you'll be using often, to find out what files are new or changes have been made.

`git status` will tell you what files are being tracked and whether there are changes made that haven't been "checked in" yet.



```
Windows PowerShell  
PS C:\Users\dan.webb\test> git status  
On branch master  
  
No commits yet  
  
Untracked files:  
  (use "git add <file>..." to include in what will be committed)  
    file.txt  
  
nothing added to commit but untracked files present (use "git add" to track)  
PS C:\Users\dan.webb\test>
```

- Git is letting us know that nothing has been committed and that we have one untracked file.

In this context, Git is active within the directory, yet it's not "tracking" any files at the moment. It adopts a cautious approach, and doesn't automatically track all files in a directory, and instead relies on you to specify which files to track - we'll do this next with `git add`.

## Tracking Files With git add

We have several different ways to tell Git to track a file:

```
git add FILENAME
```

You can explicitly tell it to track a file by giving it the filename.

```
git add FILENAME FILENAME FILENAME FILENAME
```

You can track multiple files by listing multiple filenames.

```
git add .
```

You can all files in the current directory and its subdirectories to be tracked.

You'll probably use `git add .` more than anything else, especially when starting. It's a good way to ensure any new files you have created are tracked by git. Nothing worse than sharing your code and realising you forgot to include a file!

Earlier you created a file in your test directory (file.txt) — now, add it to your Git repository (otherwise known as a repo):

## Windows PowerShell

```
PS C:\Users\dan.webb\test> git add .  
PS C:\Users\dan.webb\test>
```

You'll notice there's no confirmation of what has happened, but it has worked. You can check by running `git status` again:

## Windows PowerShell

```
PS C:\Users\dan.webb\test> git add .  
PS C:\Users\dan.webb\test> git status  
On branch master  
  
No commits yet  
  
Changes to be committed:  
  (use "git rm --cached <file>..." to unstage)  
    new file:   file.txt  
  
PS C:\Users\dan.webb\test>
```

There's  
our file!

Now that we are tracking our file, it's time to create our first piece of history in our timeline by creating a *commit*.

## Save Your Current Status With git commit

Let's commit our current status into the Git history, and write a message to go along with it:

```
git commit -a -m "First commit"
```

## Windows PowerShell

```
PS C:\Users\dan.webb\test> git commit -a -m "First commit"  
[master (root-commit) 3d787bf] First commit  
 1 file changed, 0 insertions(+), 0 deletions(-)  
 create mode 100644 file.txt  
PS C:\Users\dan.webb\test>
```

(Your numbers will be different, but otherwise this should look very similar.)

We are using two "flags" after git commit:

- `-a` tells git to commit all files that are *currently* being tracked. Remember, new files will still need to be added with `git add` first.

- **-m** tells git that we want to include a message. We should include a message about what the commit is every time.

Commit messages should be explanatory. Often they will be very mundane "fixed broken hyperlink on homepage for defect #999", but this allows us to keep track between commits and get that whole history of what has been done, when, and by who.

If you run `git status` again now git will tell you everything is normal and you are up-to-date:




```
Windows PowerShell
PS C:\Users\dan.webb\test> git status
On branch master
nothing to commit, working tree clean
PS C:\Users\dan.webb\test>
```

## What Next?

Ok, now we have started tracking our files (using `git add`) and have committed our code (using `git commit`), we can carry on working on the next piece of functionality.

It's important to add and commit quite regularly - this way you can go back in time in your history if you massively screw up your code, or get the latest code back if your computer suffers a terminal crash.



-  1. **git commit**
-  2. **git push**
-  3. **leave building**

Working for days or weeks without committing your code is less helpful than small, regular updates throughout your day - partly because it becomes harder to undo bad things and your commit is very large, so anyone looking at it will have to do a lot of mental gymnastics comparing between your "old" commit and your "new" commit.

Let's do a second commit so you can get a feel for it.

## Your Second Commit

Open up `file.txt` in your favourite text editor and add some new text into it, and save the file.

If you now run `git status` you will see the file has been modified:

```
Windows PowerShell
PS C:\Users\dan.webb\test> git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file.txt

no changes added to commit (use "git add" and/or "git commit -a")
PS C:\Users\dan.webb\test>
```

Commit the file using `git commit -a -m "your commit message"`

```
Windows PowerShell
PS C:\Users\dan.webb\test> git commit -a -m "Added text to the file"
[master 3601745] Added text to the file
 1 file changed, 1 insertion(+)
PS C:\Users\dan.webb\test>
```

## Looking at the History With git log

Whilst you probably won't use this very often, it's useful for looking at your history of commits. Type `git log` into the terminal:

```
Windows PowerShell
PS C:\Users\dan.webb\test> git log
commit 36017455df4c5c86f23b131ed2b04a27c636357a (HEAD -> master)
Author: dan.webb <dan.webb@bjss.com>
Date:   Tue Jan 16 12:54:48 2024 +0000

    Added text to the file

commit 3d787bfcd99b96e5a9ab38c4afea70f0850b943e
Author: dan.webb <dan.webb@bjss.com>
Date:   Tue Jan 16 12:00:30 2024 +0000

    First commit
PS C:\Users\dan.webb\test>
```

Here we can see our two commits so far, with the newest at the top, along with the messages we used.

## Help, I've Made Changes I Don't Want To Commit!

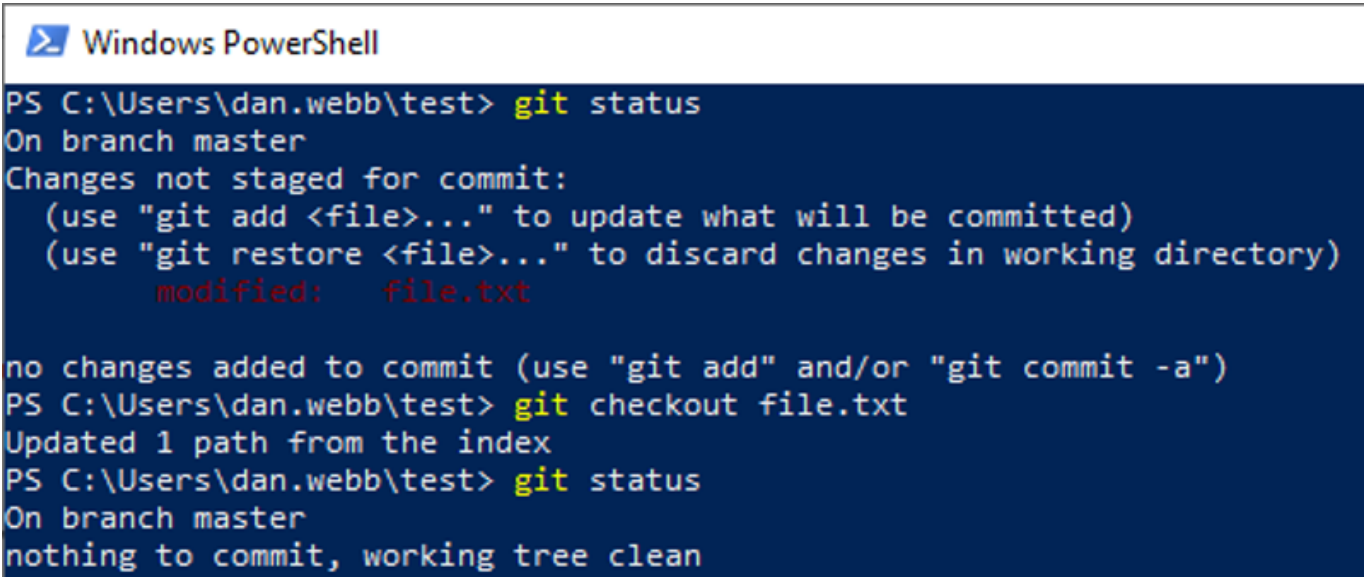
Relax, git has got you covered. SOmetimes you start work on something and it turns out all that work was rubbish and you are no longer sure what changed.

I've only made changes to one file I want to revert

You can try this for yourself. Open file.txt again and add a new line to the file - any random text will do - and save the file.

Now if you run `git status` you will see that one file is listed under "Changes not staged for commit".

To revert this change back to the latest version you committed, you can use `git checkout FILENAME`:



```
Windows PowerShell

PS C:\Users\dan.webb\test> git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file.txt

no changes added to commit (use "git add" and/or "git commit -a")
PS C:\Users\dan.webb\test> git checkout file.txt
Updated 1 path from the index
PS C:\Users\dan.webb\test> git status
On branch master
nothing to commit, working tree clean
```

Running git status again shows us the working tree is now clean - or in plain English, we're all back where we were at our last commit.

I've made lots of changes and I want to just go back fully to the previous commit

Naughty. Anyway, if you've gone too far down the rabbit hole of trying to fix things but have made too many changes and want to go back to your last commit you can use `git reset --hard HEAD`.

## I've Got Some Files I Don't Want To Commit, Ever

Often there will be certain files you don't want to commit. Most of the time these will be things that make it work on your computer such as plugins or packages, or IDE specific things for our own setup that we don't want to share.

To do this, we can specify create a file called `.gitignore` that tells git which files or folder to never worry about tracking or committing.

To test this out, create a new file in the same folder as `file.txt` called `ignoreme.txt`.

First, check that git knows it's there as an "untracked" file:



```
Windows PowerShell

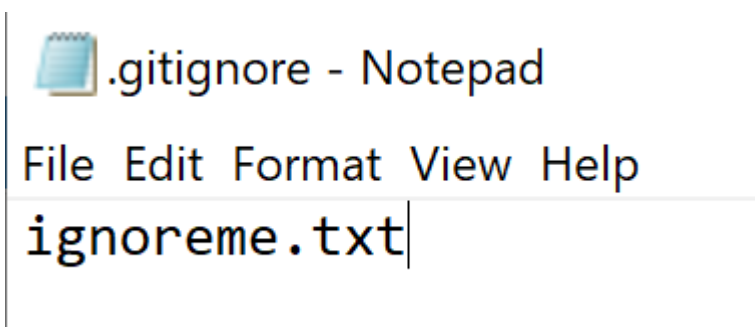
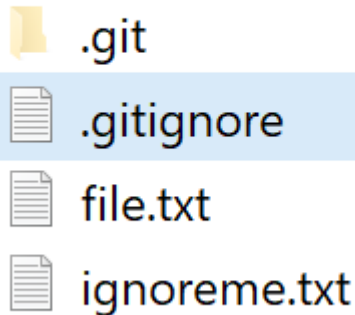
PS C:\Users\dan.webb\test> git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        ignoreme.txt

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

Next, create a `.gitignore` file and put it at the root of your project, where your `.git` directory is. Within the `.gitignore` file, put "ignoreme.txt" at the top of the file and save.

If we run `git status` again, we can see that the `.gitignore` file was added, but the `ignoreme.txt` file is nowhere to be found. Git is successfully ignoring that file!

## Name



Now, if we run `git status` again we will see that `.gitignore` is untracked, but the `ignoreme.txt` file is nowhere to be found - git is successfully ignoring the file.

```
PS C:\Users\dan.webb\test> git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore

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

## .gitignore File For Your Project



Fortunately you don't have to do a lot of this as there are ready made .gitignore files available online:  
<https://github.com/github/gitignore>

---

# Branching

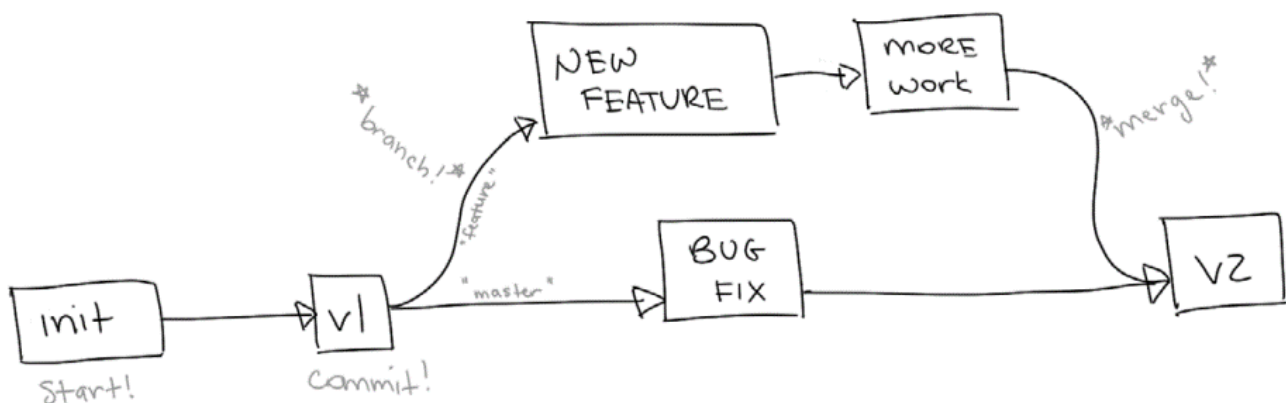
So far, we've just been committing to the main, or master, branch. This is the default branch every repository starts with.

This is fine if you are just working on your own (or if you are doing Trunk Based Delivery, but that's a whole other course...) and on small projects.

However, imagine you want to experiment with some new features, or you aren't sure if your new changes are going to work. Or you want to change something but don't want it in your main branch just yet until you have finished working on it.

This is where branching comes in - it allows you to keep different version of your project in different states until you bring them together (known as merging, which we will cover later).

In the diagram below, you can see we have released version 1 of the code, and then create a new branch to work on a feature (a certain subset of functionality).



[^1]

You'll also see that we had to fix a bug on our main branch. This bug fix doesn't make any changes to our feature branch, and our feature branch doesn't make any changes to our main branch. This allows us to keep non-ready code separate and not deploy partial code to our users.

There are other ways, called *feature flags* but we will not be discussing that here.

## Create A Branch

Run the command `git checkout -b feature` in your terminal to create a new branch named "feature".

The **-b** is important here, it's the flag that says "I want to create a new branch".

## Windows PowerShell

```
PS C:\Users\dan.webb\test> git checkout -b feature
Switched to a new branch 'feature'
PS C:\Users\dan.webb\test>
```

## What Branch Am I On? What Branches Do I Have?

If you need to remind yourself what branches you've created, run the command

`git branch`. This will also show you which branch you are currently on:

## Windows PowerShell

```
PS C:\Users\dan.webb\test> git branch
* feature
  master
```

## Switching Branches

Now that we can see what branches are available with `git branch`, we can switch between the two with `git checkout BRANCHNAME`:

## Windows PowerShell

```
PS C:\Users\dan.webb\test> git checkout master
Switched to branch 'master'
PS C:\Users\dan.webb\test>
```

Note there is no `-b`; not having that says "I want to *move* to this branch" rather than "I want to *create* a new branch named X"

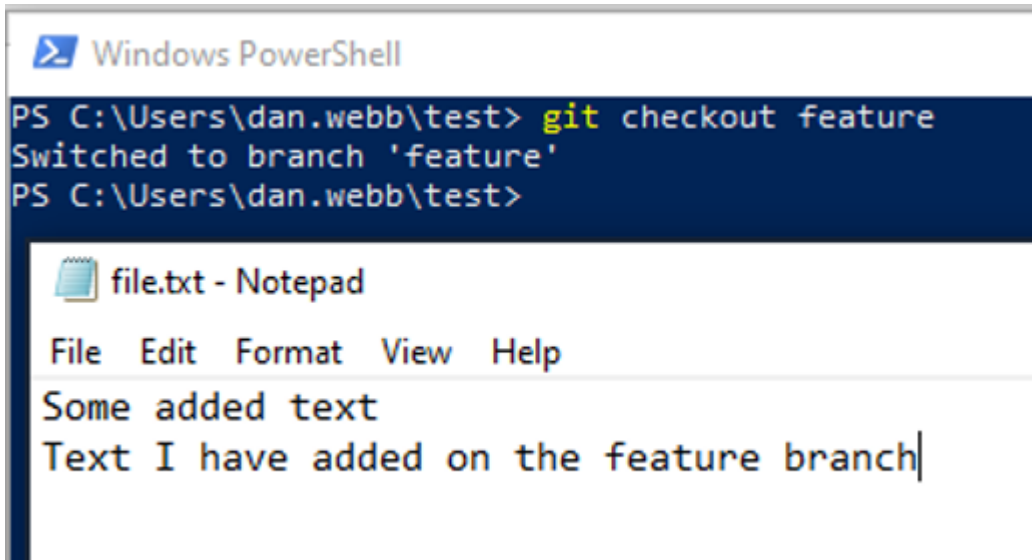
Switching between branches allows us to work on multiple versions of the code on one computer.

## Applying Changes From One Branch To Another With `git merge`

When you have finished your changes on a branch (such as `feature`) and want to apply them to another branch such as `main`, you can use `git merge` to combine the two branches together in your *current* branch.

Key point here, you always merge *into* your current branch *from* the target branch

To test this out, make some changes on your “feature” branch (remember to switch to the feature branch if you are not already on it!) – here I have added a line of text to the file.txt and saved it:



The screenshot shows a Windows PowerShell terminal window with the following commands and output:

```
PS C:\Users\dan.webb\test> git checkout feature
Switched to branch 'feature'
PS C:\Users\dan.webb\test>
```

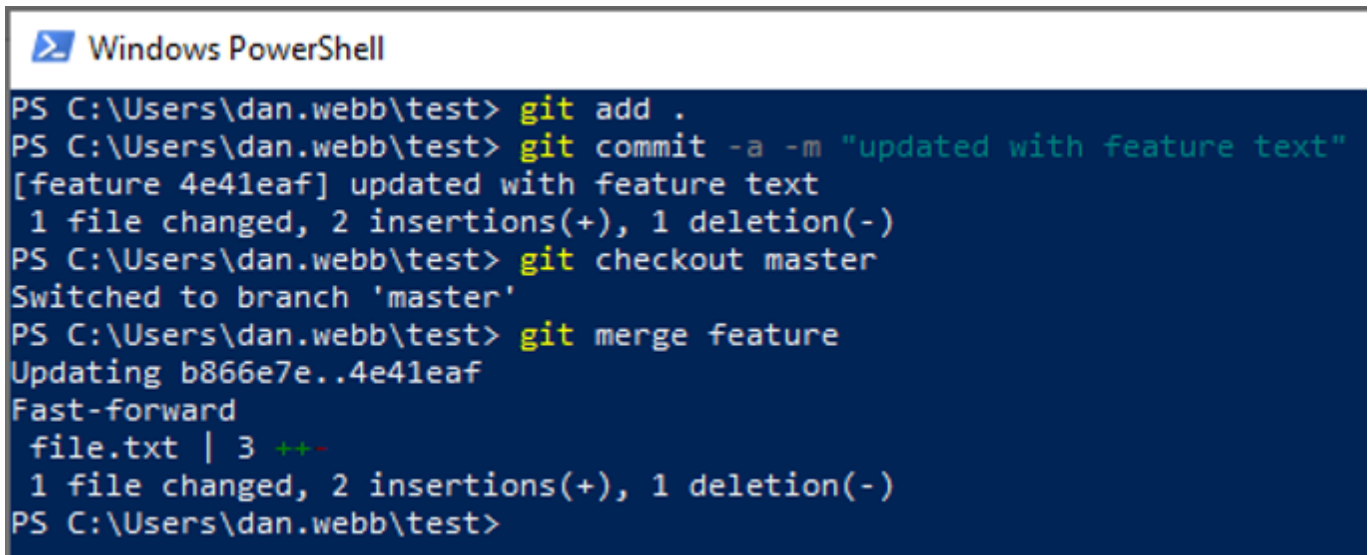
Below the terminal is a Notepad window titled "file.txt - Notepad". The menu bar includes File, Edit, Format, View, and Help. The text content of the file is:

```
Some added text
Text I have added on the feature branch|
```

Now I commit the changes, then move back to the “master” branch by running the command `git checkout master`.

Before we do anything else - I'll prove to you that the changes we made in feature are not there now we have switched branches to a different version - open up `file.txt` and you'll see the changes we made are not there.

Now I can bring all the changes made to the feature branch over to the master branch with `git merge BRANCHNAME` (`git merge feature` if your branch was named “feature”).



The screenshot shows a Windows PowerShell terminal window with the following commands and output:

```
PS C:\Users\dan.webb\test> git add .
PS C:\Users\dan.webb\test> git commit -a -m "updated with feature text"
[feature 4e41eaf] updated with feature text
1 file changed, 2 insertions(+), 1 deletion(-)
PS C:\Users\dan.webb\test> git checkout master
Switched to branch 'master'
PS C:\Users\dan.webb\test> git merge feature
Updating b866e7e..4e41eaf
Fast-forward
 file.txt | 3 ++-
1 file changed, 2 insertions(+), 1 deletion(-)
PS C:\Users\dan.webb\test>
```

You have now merged your feature branch *into* you master branch. You can check this by opening `file.txt` and seeing the changes.

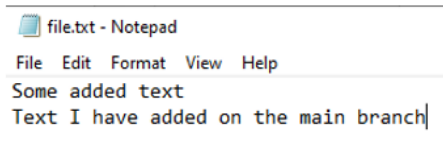
## Conflicts!

Now git will often be able to handle merges from one branch to another without problems. However, when it doesn't know how to merge any two files it's going to cause a *conflict*.

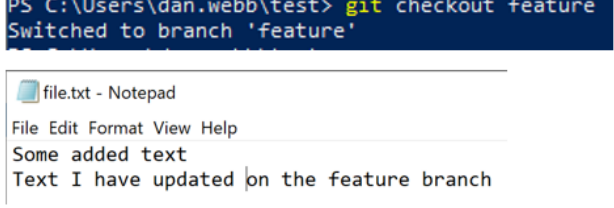
The reason for this is normally if the same file has been changed by two users in the same place in the file.

So now we are going to make a conflict to show you what happens.

Firstly, on the master branch (which you should still be on, but check with `git branch`), we are going to update our `file.txt`, save the file, and commit the changes. Secondly we are going to switch to the feature branch, update the `file.txt` with *different* text in the same place, and save it.



```
PS C:\Users\dan.webb\test> git commit -a -m "updated master file.txt"
[master 8173d04] updated master file.txt
1 file changed, 1 insertion(+), 1 deletion(-)
```



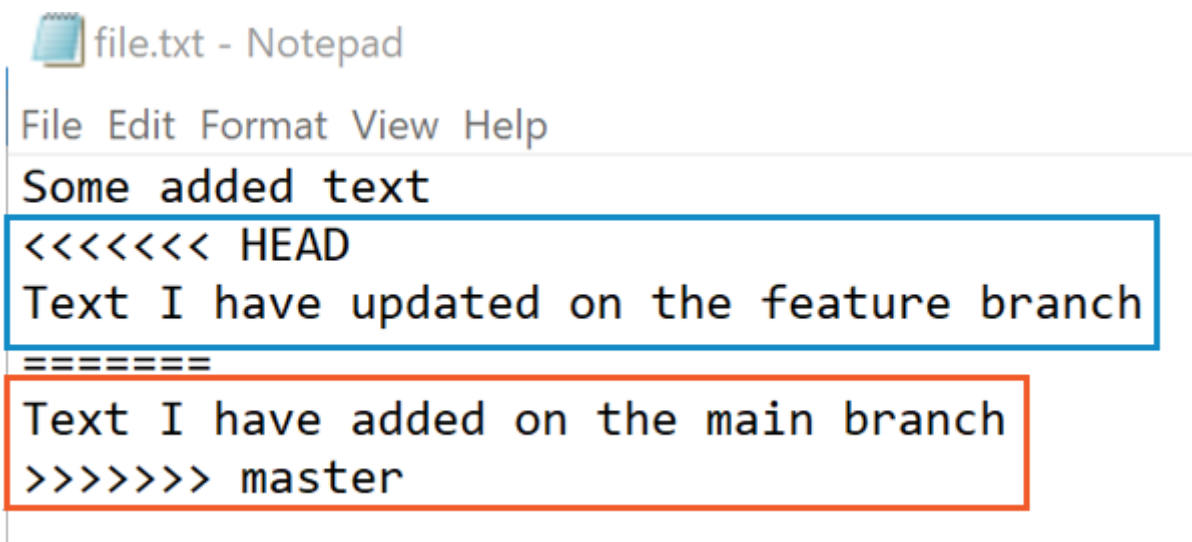
```
PS C:\Users\dan.webb\test> git checkout feature
Switched to branch 'feature'
```

Now, on our feature branch, we commit our changed file, and attempt to merge from the master branch:

```
PS C:\Users\dan.webb\test> git commit -a -m "updated file.txt on feature branch"
[feature bd9808d] updated file.txt on feature branch
1 file changed, 1 insertion(+), 1 deletion(-)
PS C:\Users\dan.webb\test> git merge master
Auto-merging file.txt
CONFLICT (content): Merge conflict in file.txt
Automatic merge failed; fix conflicts and then commit the result.
PS C:\Users\dan.webb\test>
```

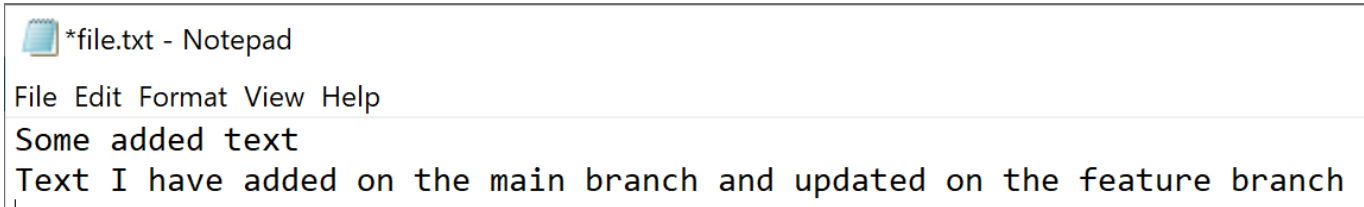
Git also helpfully tells you which file(s) are in conflict - only one in this case, `file.txt`.

Git will have added `<<<<<` and `>>>>>` marks around where there is a conflict, with `=====` in between, and will include both versions of the conflicted code, so you can manually pick and choose what to keep and what to lose:



Here `<<<<<< HEAD` indicates the code in our *current* branch (feature) and `>>>>>> master` indicates the changes on the branch we are merging *from* (master).

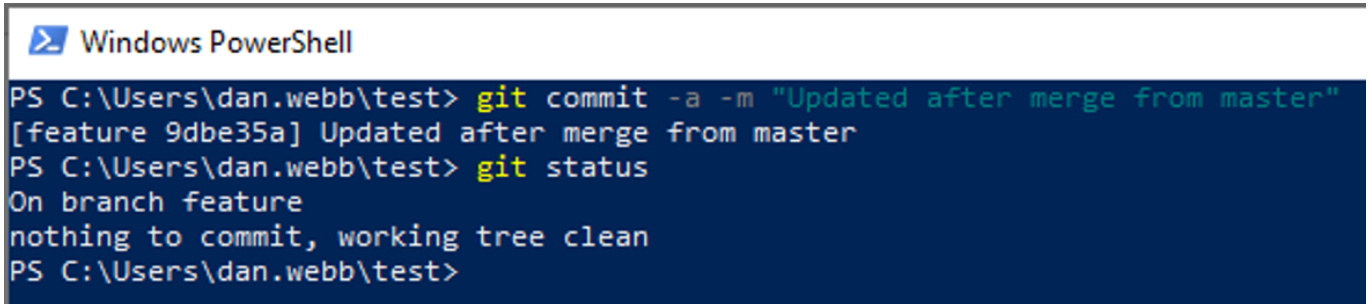
We can resolve this by manually merging the two in the file:



```
*file.txt - Notepad
File Edit Format View Help
Some added text
Text I have added on the main branch and updated on the feature branch
```

We've removed the <<<<<< Head and ===== and >>>>>> master lines and saved the changes we want to make.

Now we have to commit the changes to our branch:



```
Windows PowerShell
PS C:\Users\dan.webb\test> git commit -a -m "Updated after merge from master"
[feature 9dbe35a] Updated after merge from master
PS C:\Users\dan.webb\test> git status
On branch feature
nothing to commit, working tree clean
PS C:\Users\dan.webb\test>
```

You can even now switch back to the master branch and merge the feature branch into it without issue.

## git stash

Git will moan if you have changes on a branch that you *haven't* committed and try to switch branches. It doesn't want to assume you want to bring those changes with you, so makes you make a decision - do you want to commit these changes, remove these changes, or *save them for later without doing a full commit*.

`git stash` helps you temporarily set aside your work so you can deal with something else, and then easily pick up where you left off when you're ready. It's like a pause button for your changes.

This tells git to move your changes to an invisible branch or holding pen (they won;t even show up if you run `git status`), and allows you to move back and forth between branches again.

When you want those changes back you can run `git stash pop`.

This works on *any* branch, not just the original branch those changes were on. This is extremely useful if you started making changes on a branch, realised you were on the wrong branch - you can stash your changes, switch to the correct branch and run `git stash pop` to bring the changes into the new branch.

# Sharing Your Code

---

Whilst that was all very good, it's not much help if we need to share our code with others. The code only exists on our machine right now, in something known as a "repository" or "repo".

When you ran `git init` what you actually did was initialise a git repository on your computer - known as your *local repository*.

Most repositories you will see will be on the internet on websites like GitHub. You can share your code like this too in what is known as a *remote repository*.

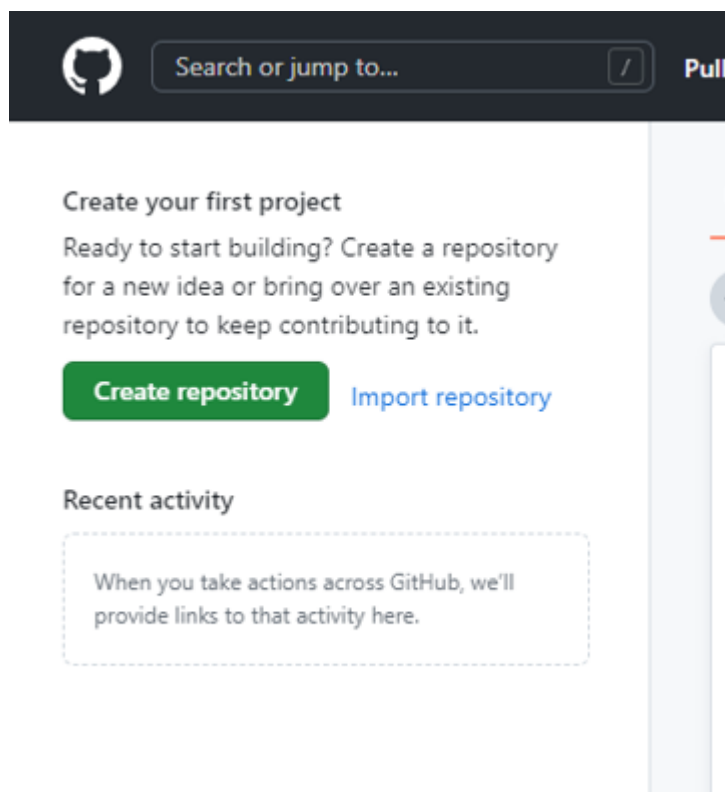
Remote repositories allow teams to share code - everyone can *push* their changes to this, and you can see other people's changes by *pulling* the latest version down to your local repository.

## Create A Github Profile

First things first, we are going to need a remote repository. Head over to GitHub to create a profile first: <https://github.com/>

## Create A Repository

Click the Create repository button once you have created a profile and signed in:



Now you'll need to name your repository. We're going to call ours "profile", but you can name it whatever you like.


Make sure your repository is set to Public so other people can see it! Otherwise accept all the other defaults:



## Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Owner \*

 bjssacademy

Repository name \*

/ profile

✔ profile is available.

Great repository names are short and memorable. Need inspiration? How about [friendly-goggles?](#)

Description (optional)

 You may not create internal repositories by organization policy.



Public

Anyone on the internet can see this repository. You choose who can commit.



Private

You choose who can see and commit to this repository.

Initialize this repository with:



Add a README file

This is where you can write a long description for your project. [Learn more about READMEs.](#)

Add .gitignore

.gitignore template: None

Choose which files not to track from a list of templates. [Learn more about ignoring files.](#)

Choose a license

License: None


A license tells others what they can and can't do with your code. [Learn more about licenses.](#)

 You are creating a public repository in your personal account.

Create repository

Once you have done this, you'll get shown the link to the repository. Copy this link to the clipboard.

Quick setup — if you've done this kind of thing before

 Set up in Desktop

or

HTTPS

SSH

<https://github.com/bjssacademy/profile.git>



Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

## Add The Remote Repository

On the command line in your git project, type the following, remembering to replace `<REMOTE_URL>` with the link you copied in the previous step:

```
git remote add origin <REMOTE_URL>
```

### Placeholders

Quite a lot of the time, people will use the syntax `<something>` as a placeholder e.g.:

- Enter your <password>
- Add the <remote url> to the command

This means they want you to replace the entire placeholder *including* the angle brackets with the value.

They do not mean "please enter your placeholder value inside these angle brackets"

## Verify

Verify you have successfully added the remote by using `git remote -v`

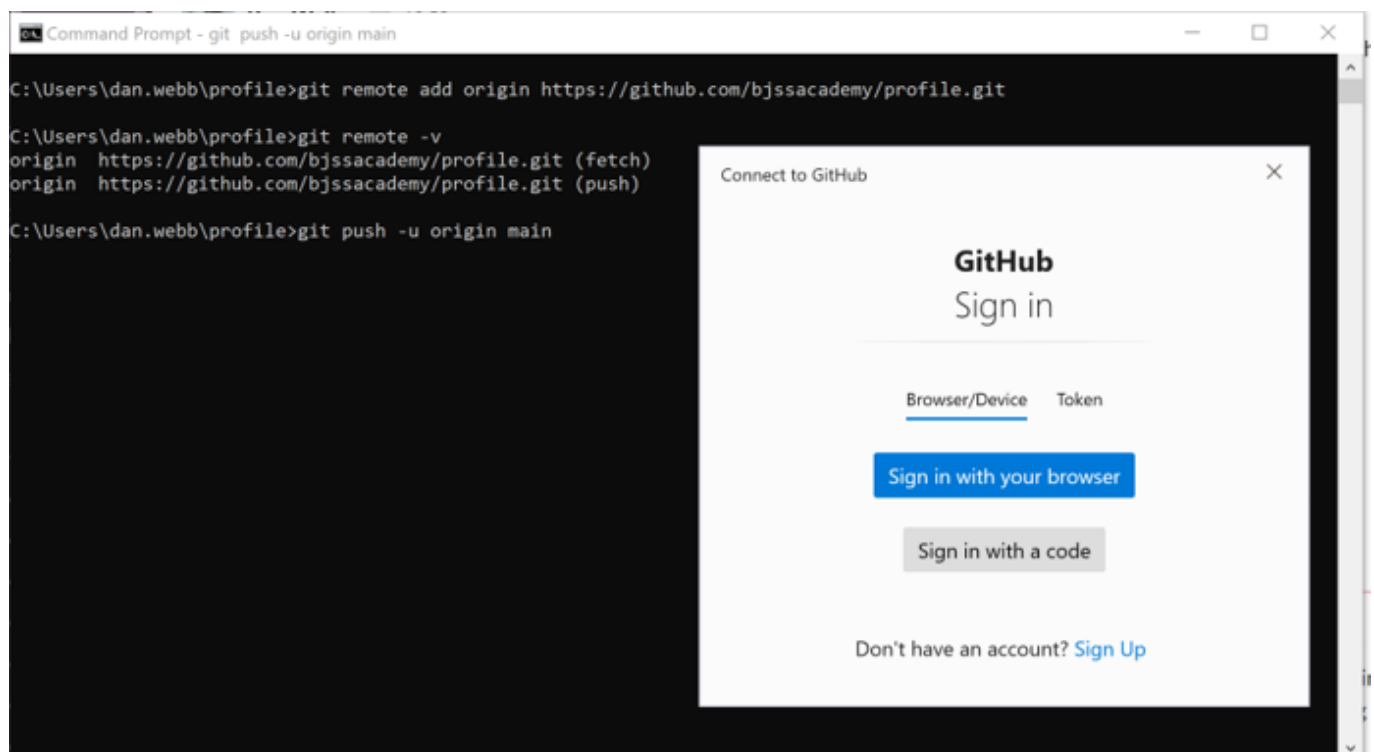
```
C:\Users\dan.webb\profile>git remote add origin https://github.com/bjssacademy/profile.git

C:\Users\dan.webb\profile>git remote -v
origin https://github.com/bjssacademy/profile.git (fetch)
origin https://github.com/bjssacademy/profile.git (push)

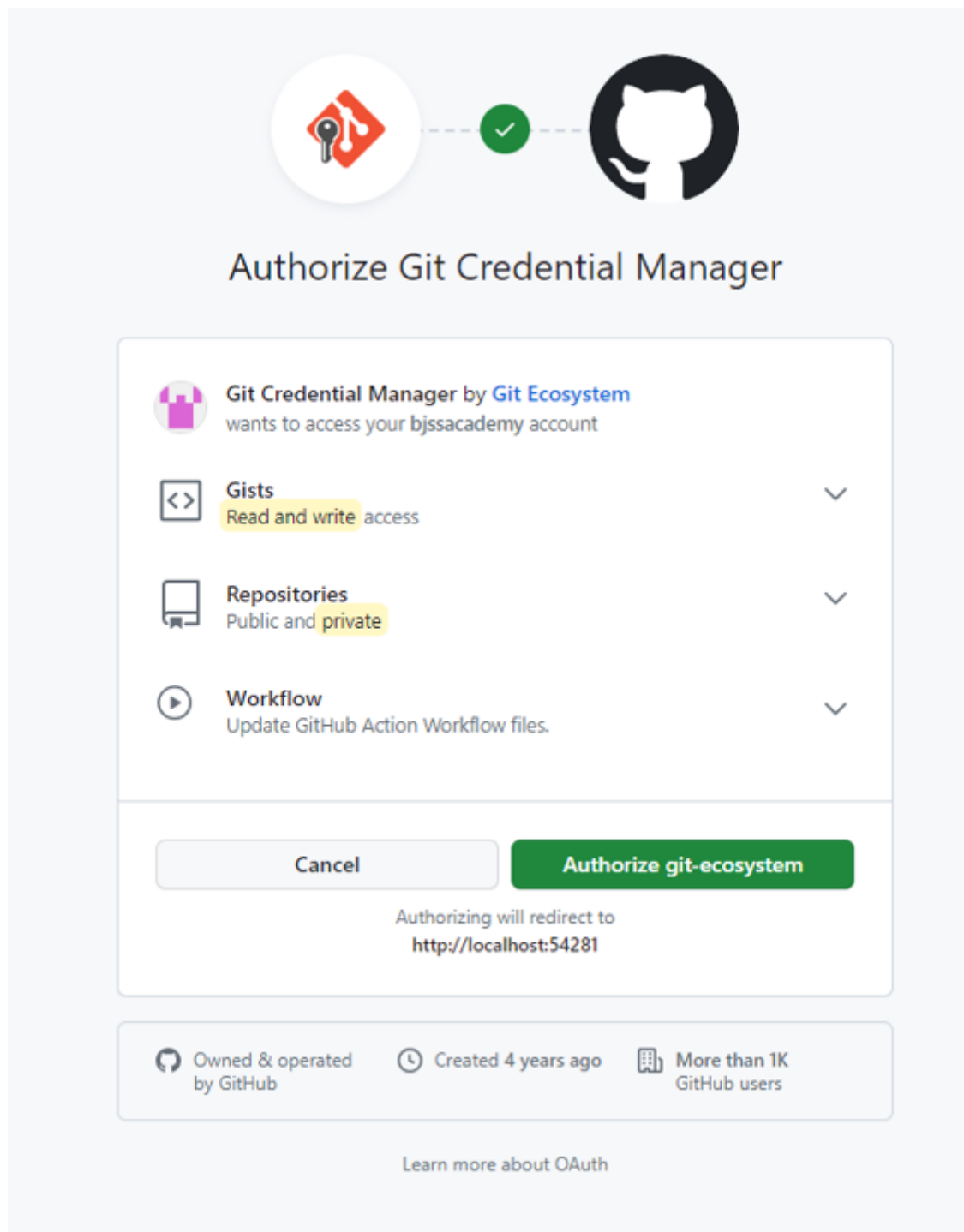
C:\Users\dan.webb\profile>
```

## Push Our Code To The Remote Repository

It's time to push our code the the remote repository using `git push -u origin master`, where we'll be asked to sign in to GitHub to prove its us!



Click on Sign in with your browser, and enter your details. You'll then need to click the button "Authorize git-ecosystem".

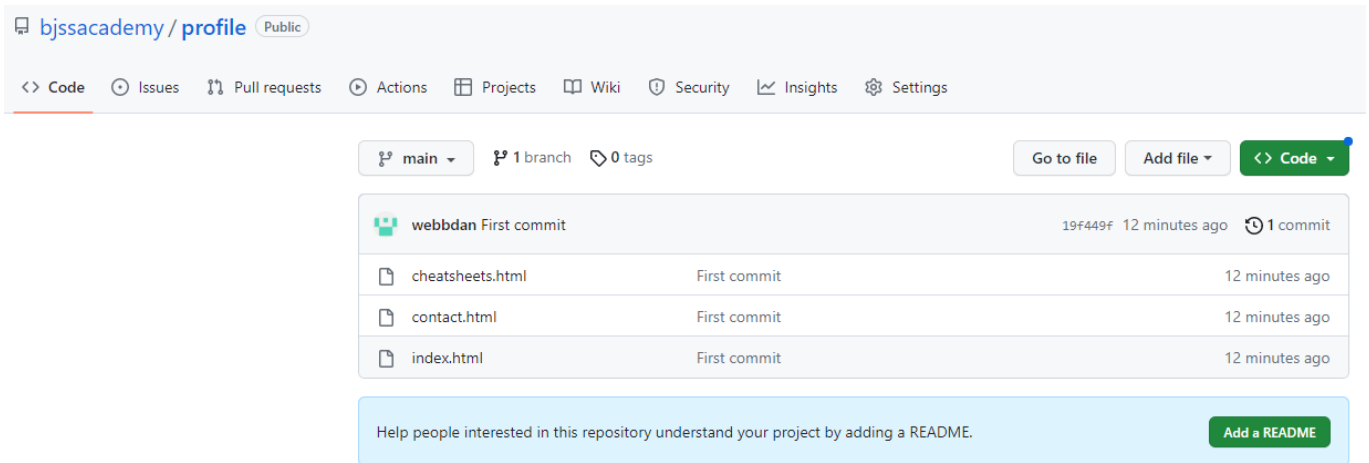


Having done this once, we won't need to do it again!

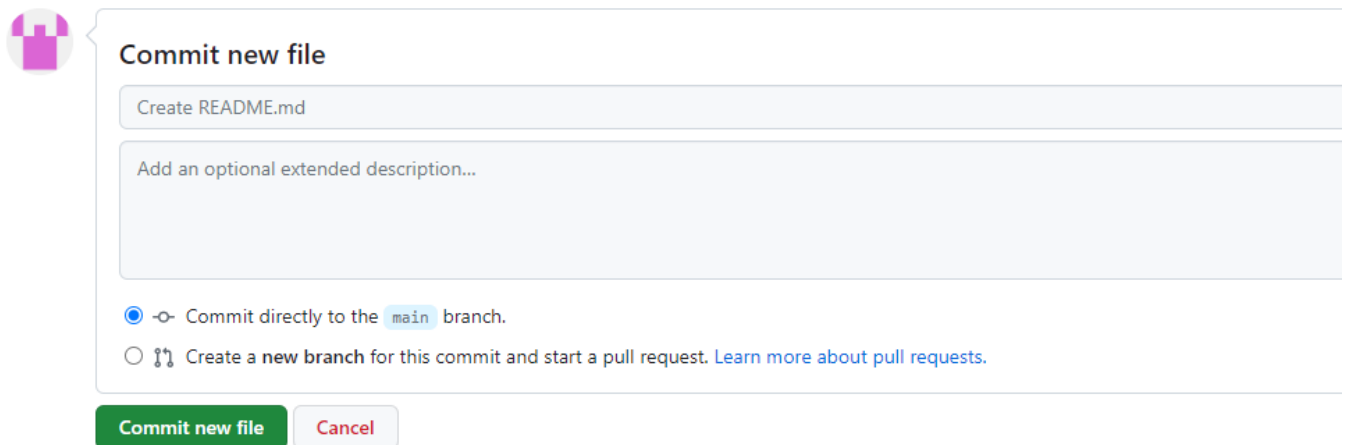
Now, go to your GitHub account and look in the repository - all our code will now be there for the world to see!

## Pulling Code

Okay, so now it's there. How do we get code from it? Well, first let's prove that changes we make to the repository are reflected when we pull the latest changes. In your repository you'll see it want you to add a README file - so go ahead and click that button:



In the window that opens, type some text into it and click on the "Commit changes" button, then the "Commit new file" button:



Now our remote repository has a file we *don't* have locally.

To bring the file from the remote repository to our local repository, we need to pull it using `git pull`:

#### Command Prompt

```
C:\Users\dan.webb\profile>git pull
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 728 bytes | 48.00 KiB/s, done.
From https://github.com/bjssacademy/profile
  19f449f..0f86373  main      -> origin/main
Updating 19f449f..0f86373
Fast-forward
 README.md | 3 +++
 1 file changed, 3 insertions(+)
 create mode 100644 README.md
C:\Users\dan.webb\profile>
```

The readme file will now be available to you locally!

# Wrapping Up

Command	Description
git init -b main	Creates a new repository in the folder you run the command in, with a default branch named main. ONLY needs to be run once!
git add .	Adds all untracked files to staging
git commit -a -m "Message"	Creates a commit with the message between the quotes
git push	Pushes all changes from local to remote
git pull	Pulls all changes from remote to local
git remote add origin <REMOTE_URL>	Adds the remote repository as the "target" for push and pull commands

# Other Help

---

## Exiting VIM

If you don't add a message, Git will open up an editor (using vim by default, which can be confusing to learn). So remember to always add a message when committing, even if it's "WIP" (and if you get stuck in the vim editor, type : `wq` to leave.)

## Removing Files

The command `git rm FILENAME` will remove the file both from Git and from your computer. You're *deleting* the file.

If you want to just remove the file from Git but keep the file on your computer, you'll need to pass in the `--cached` flag.

So: `git rm --cached FILENAME`

This isn't something that should be run often at all, but useful to know just in case.

[^1]: Credit: Tracey Osborn