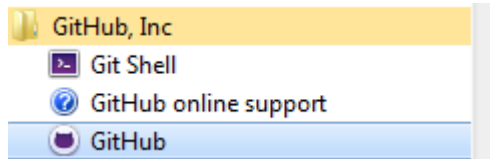


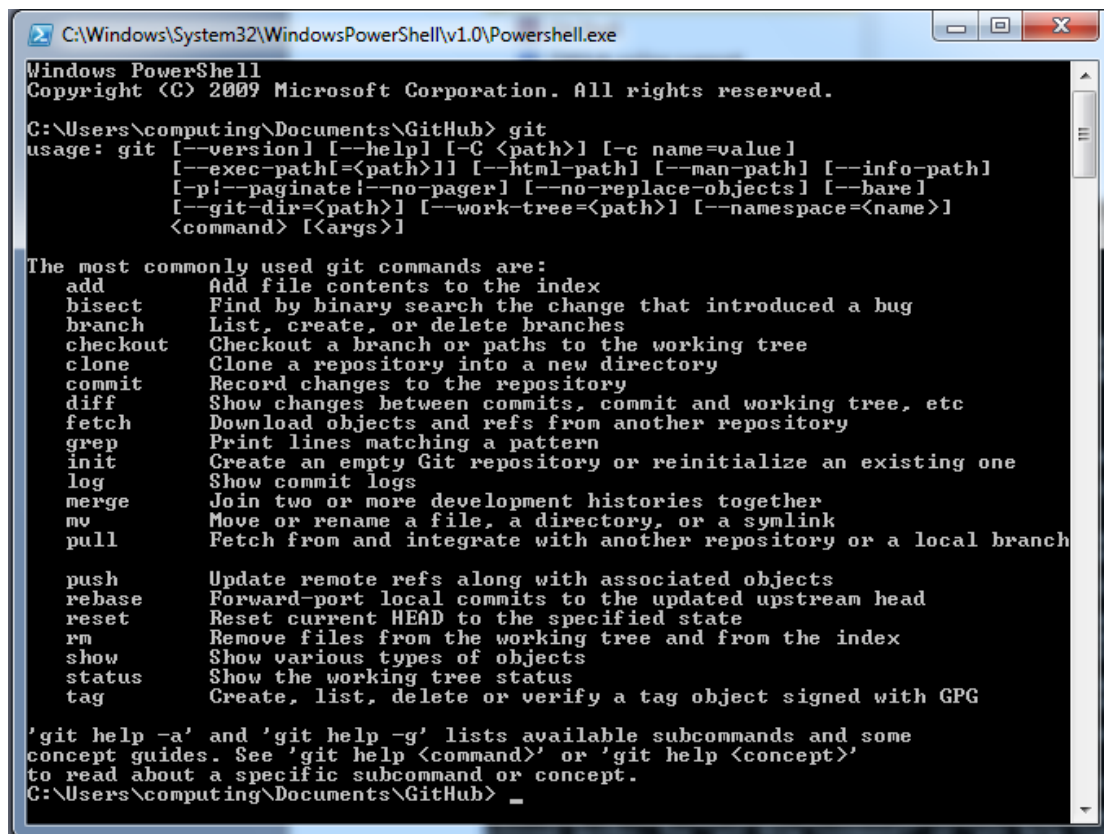
Introductory GitHub Tutorial

© Duncan Rowland 26/1/2015 v2.1

Git is a distributed revision control and source code management (SCM) system. The GitHub Application for Windows/Mac software is an application that wraps much of Git's functionality into an "easy to use" interface. You should note though that Git via the command line (e.g. clicking on "Git Shell") allows access to more powerful features, but you should probably read a good book on Git first before you attempt to use it (<http://Git-scm.com/book>). This workshop will concentrate on using the GUI application.



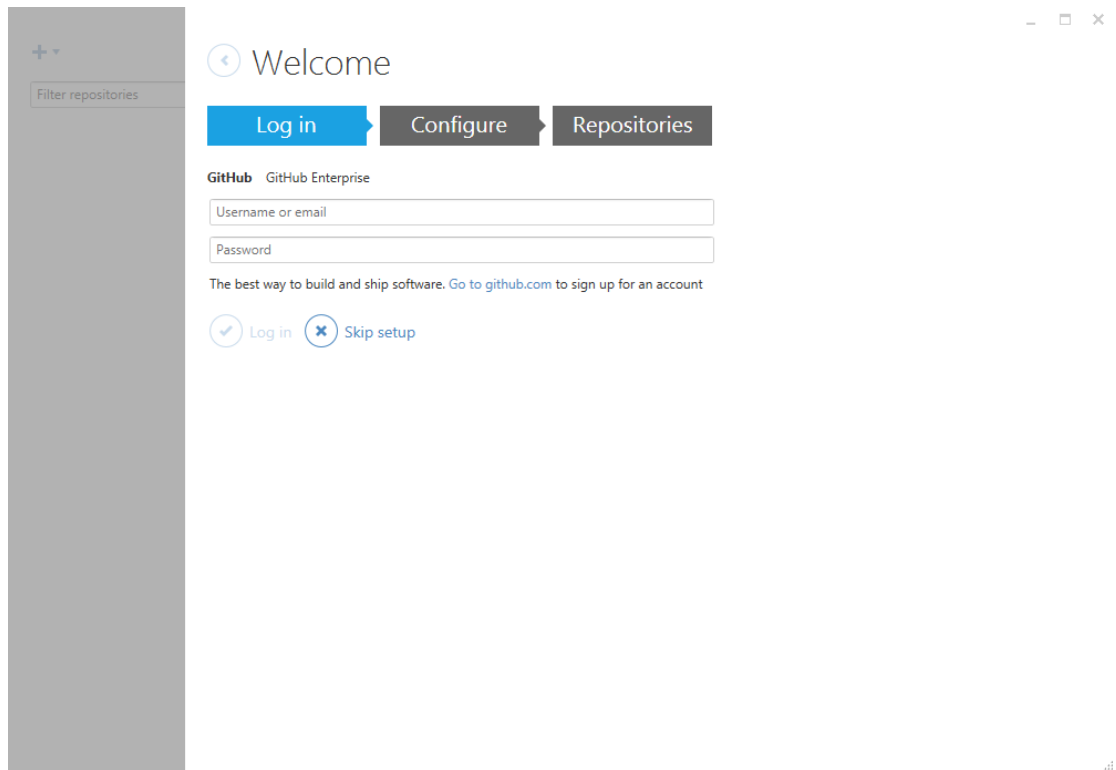
Locate the software as above

A screenshot of a Windows PowerShell window. The title bar reads 'C:\Windows\System32\WindowsPowerShell\v1.0\Powershell.exe'. The window content shows the output of the 'git' command. It starts with 'Windows PowerShell Copyright (C) 2009 Microsoft Corporation. All rights reserved.' followed by the command 'C:\Users\computing\Documents\GitHub> git'. The output shows the usage of the 'git' command with various flags like '--version', '--help', '--exec-path', etc. Below this, it lists 'The most commonly used git commands are:' followed by a list of commands and their descriptions: add, bisect, branch, checkout, clone, commit, diff, fetch, grep, init, log, merge, mv, pull, push, rebase, reset, rm, show, status, and tag. At the bottom, it says 'git help -a' and 'git help -g' lists available subcommands and some concept guides. The prompt 'C:\Users\computing\Documents\GitHub>' is visible at the bottom.

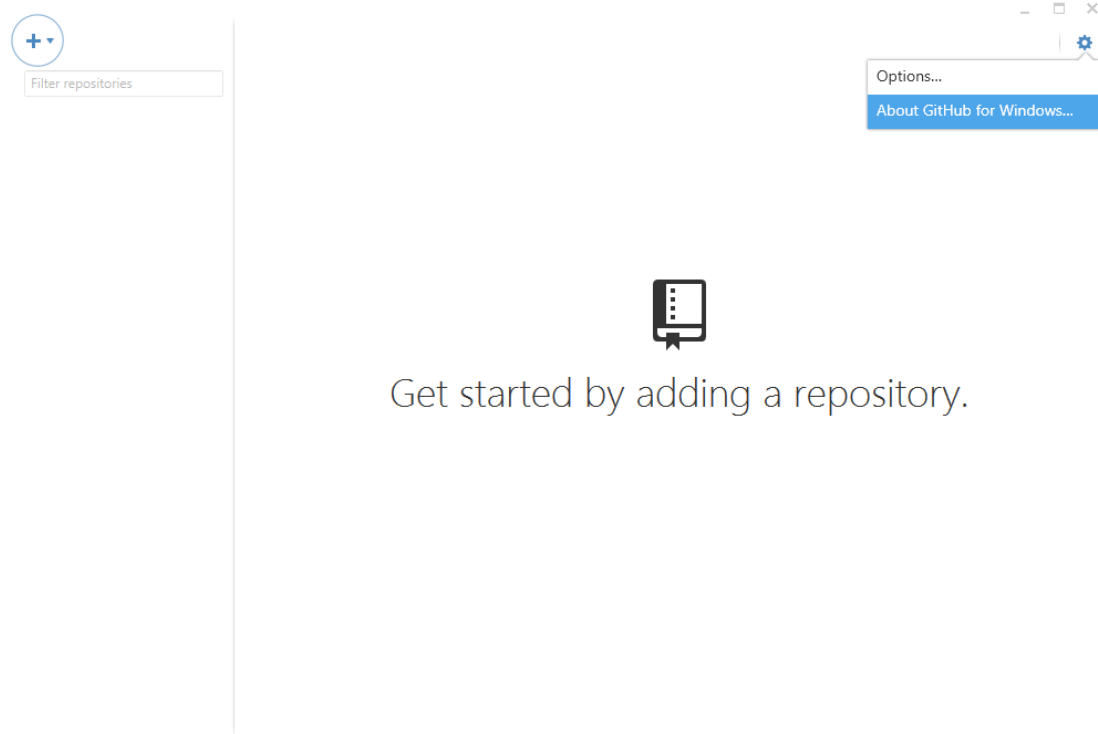
The command line is more powerful, but more complicated (some say).

In this tutorial we will be using the GitHub GUI, but feel free to try out the Shell later. To delete any existing user credentials, make sure the app is closed and delete:

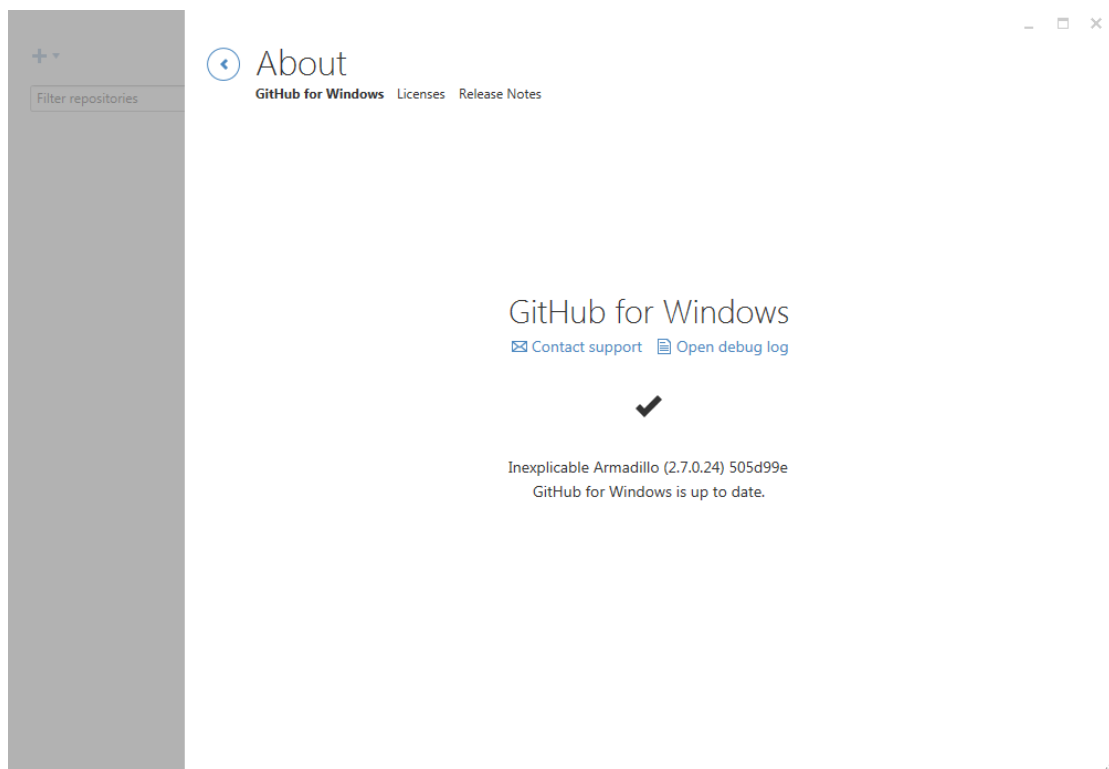
```
C:\Users\Computing\AppData\Roaming\GitHub
C:\Users\Computing\.Gitconfig
```



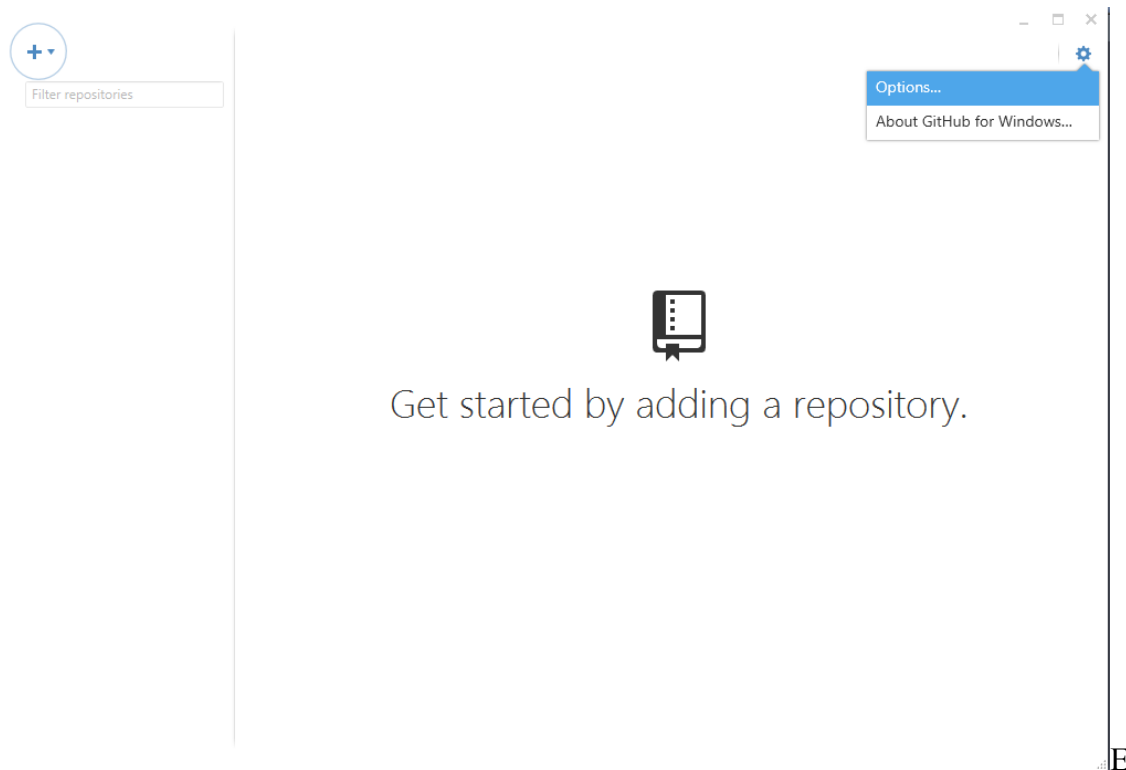
You do not need to login to GitHub yet, the first part of this tutorial is about using Git locally (i.e. without a remote Git server). For now, just click “Skip setup”.



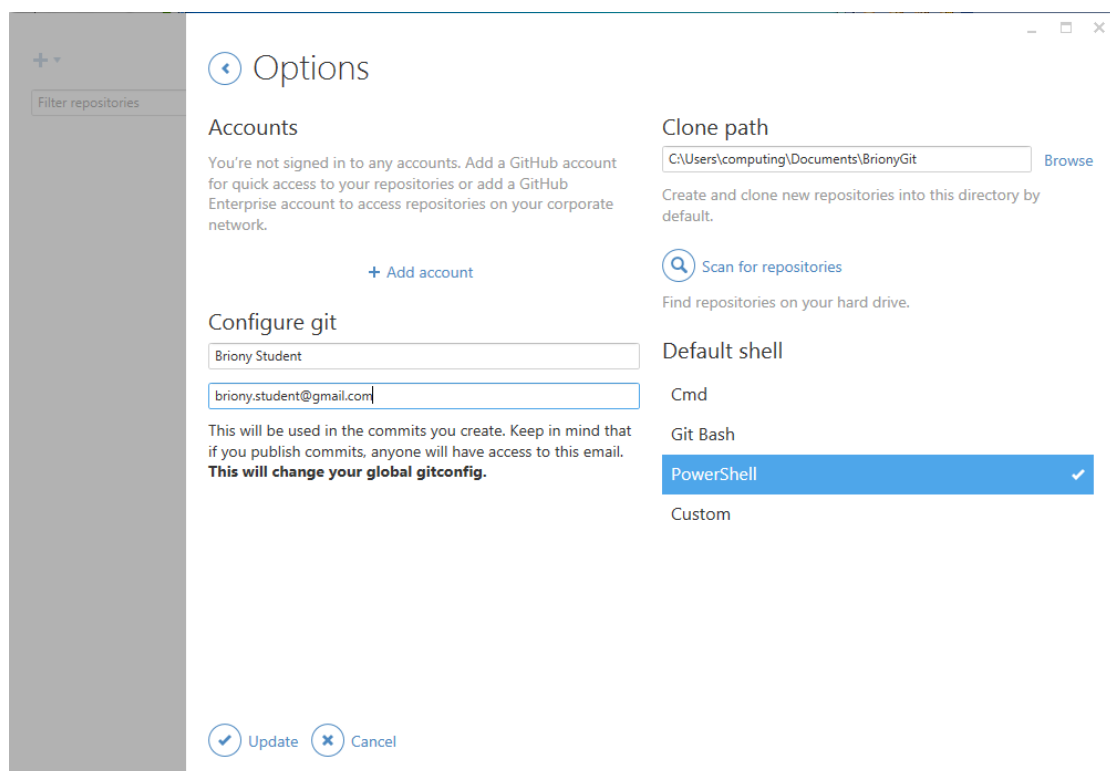
Click on “tools->About...” make sure your version is current.



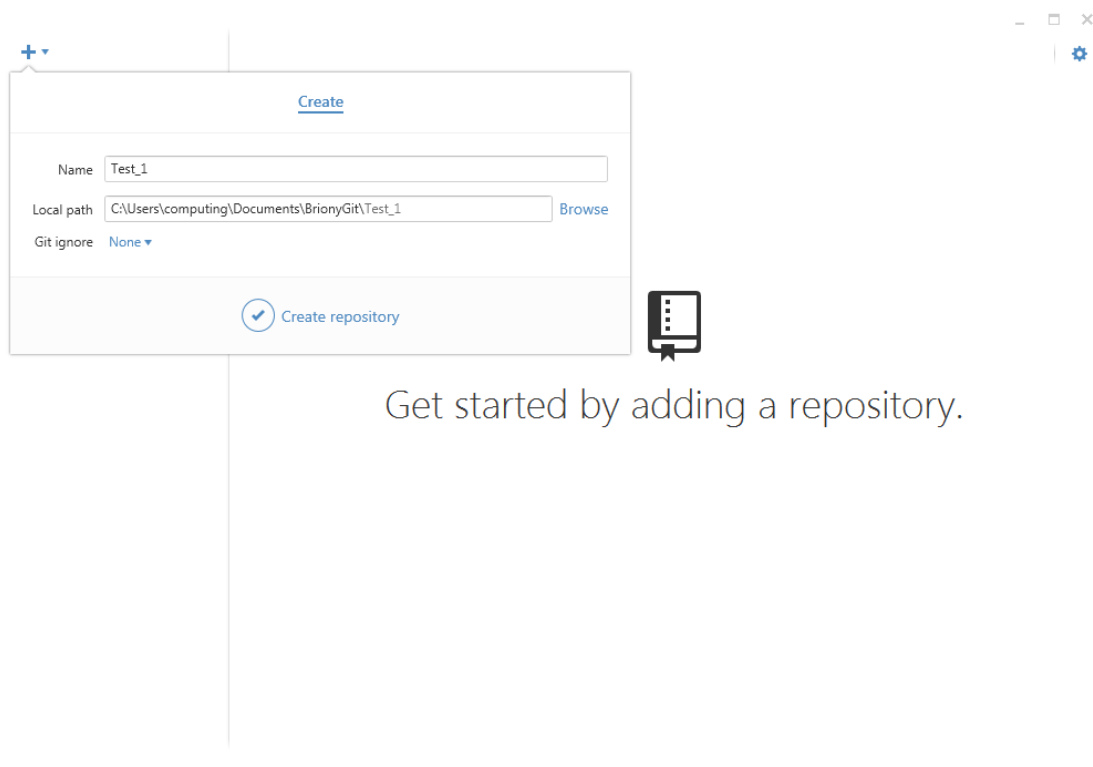
This tutorial was written using “Version 2.7.0.24 505d99e”



Each time you start using Git on a shared machine you will need to setup the options.



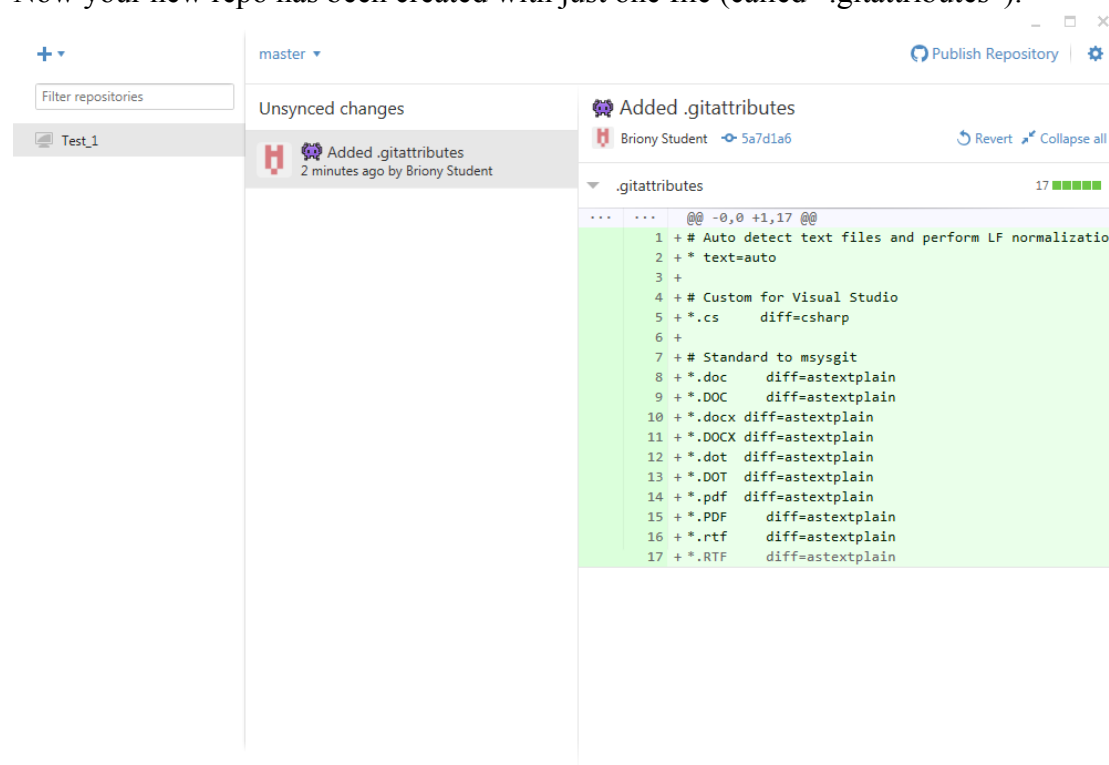
1) Enter your name, 2) Enter your email address, 3) And your Clone path (local store). N.B. this **does not** log you into GitHub, rather, it sets the details that will be linked to any changes that you make to a repository, and where these changes will be stored. Save these changes by clicking 'Update'.

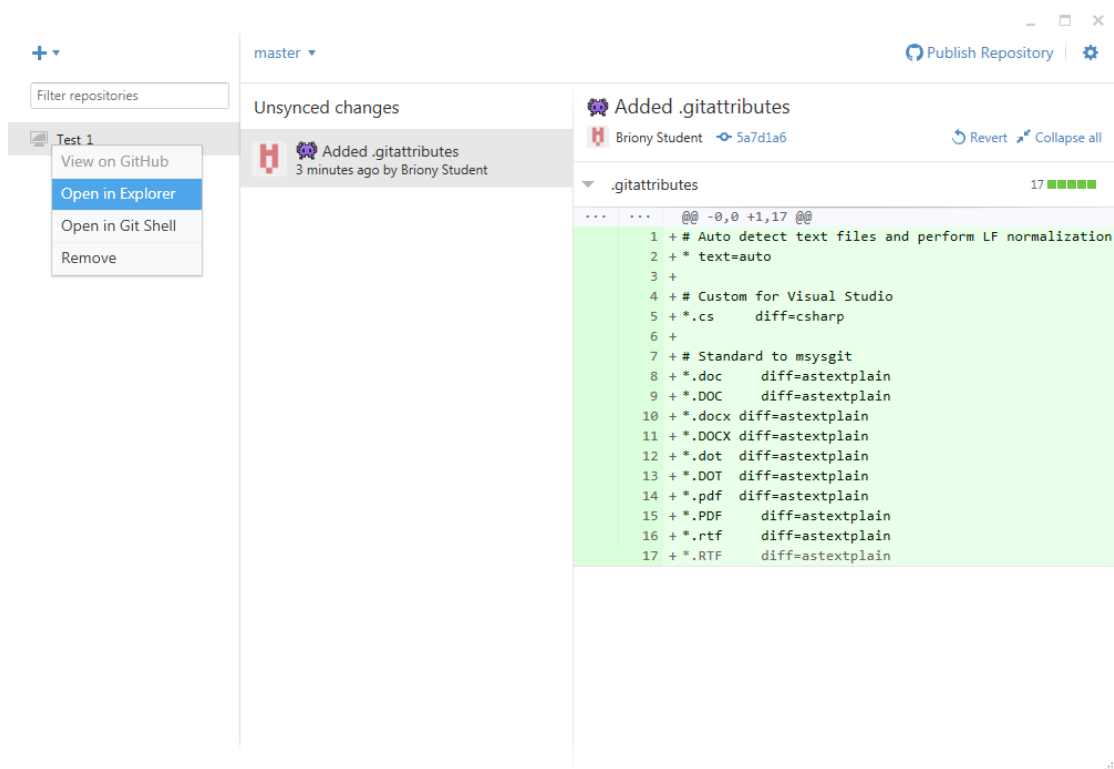


Click “+” to start making your new repository.

Type in a “Name” (e.g. **Test_1**, but for reasons that will become apparent make it 'unique' to yourself, so a good actual example for me would be **Duncan_Test_1**). The local path should already be relative to the one you just set for the local store. Set “Git ignore” to “None” (but note, you could try setting this to Unity *later*). and click “Create repository”.

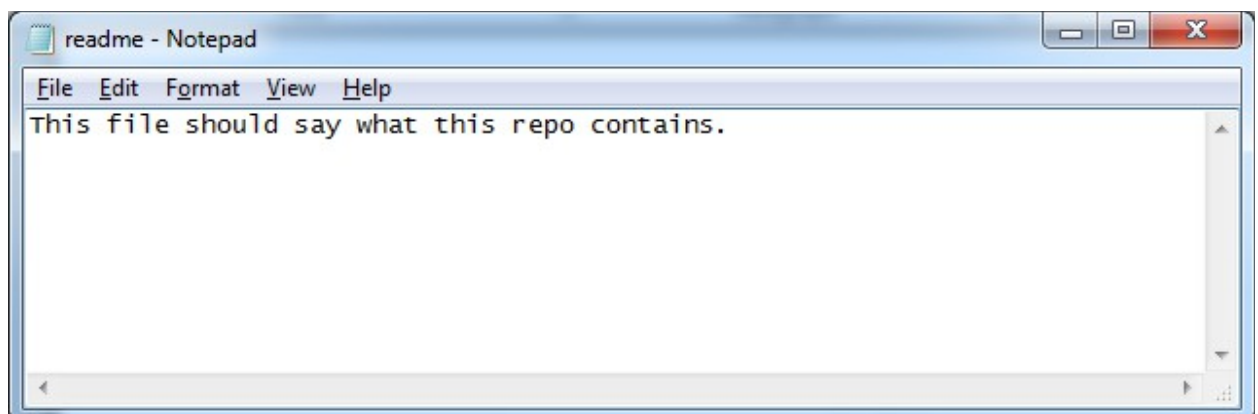
Now your new repo has been created with just one file (called “.gitattributes”).

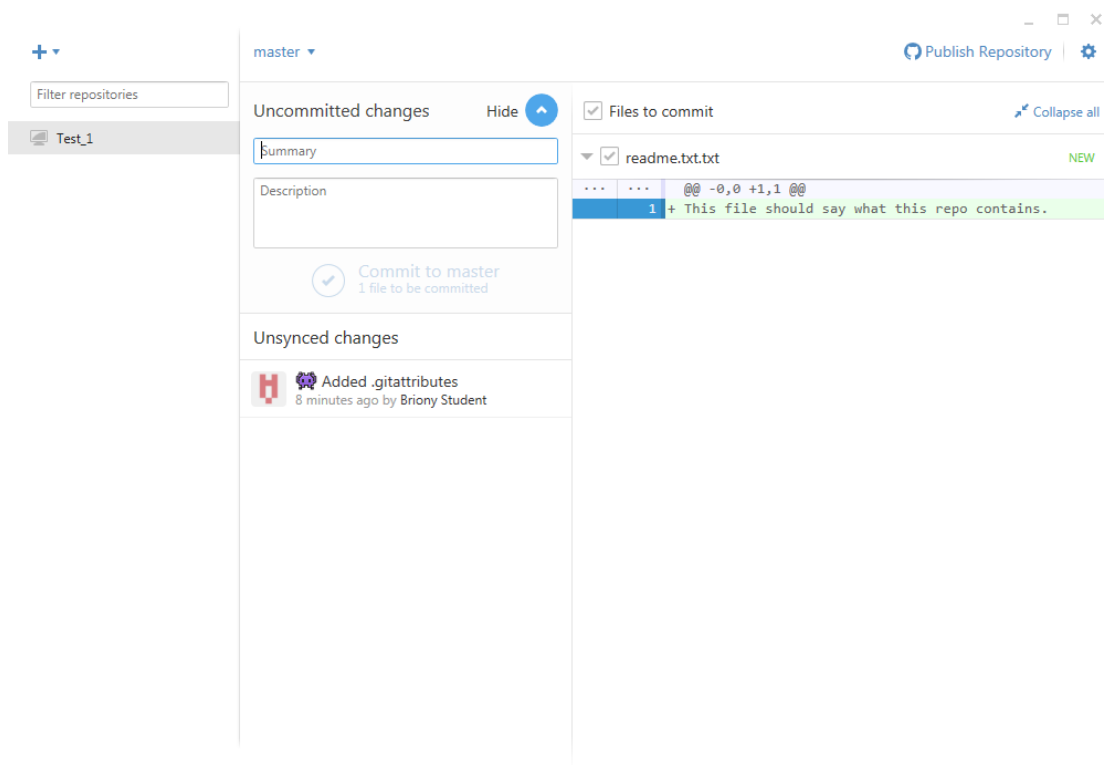




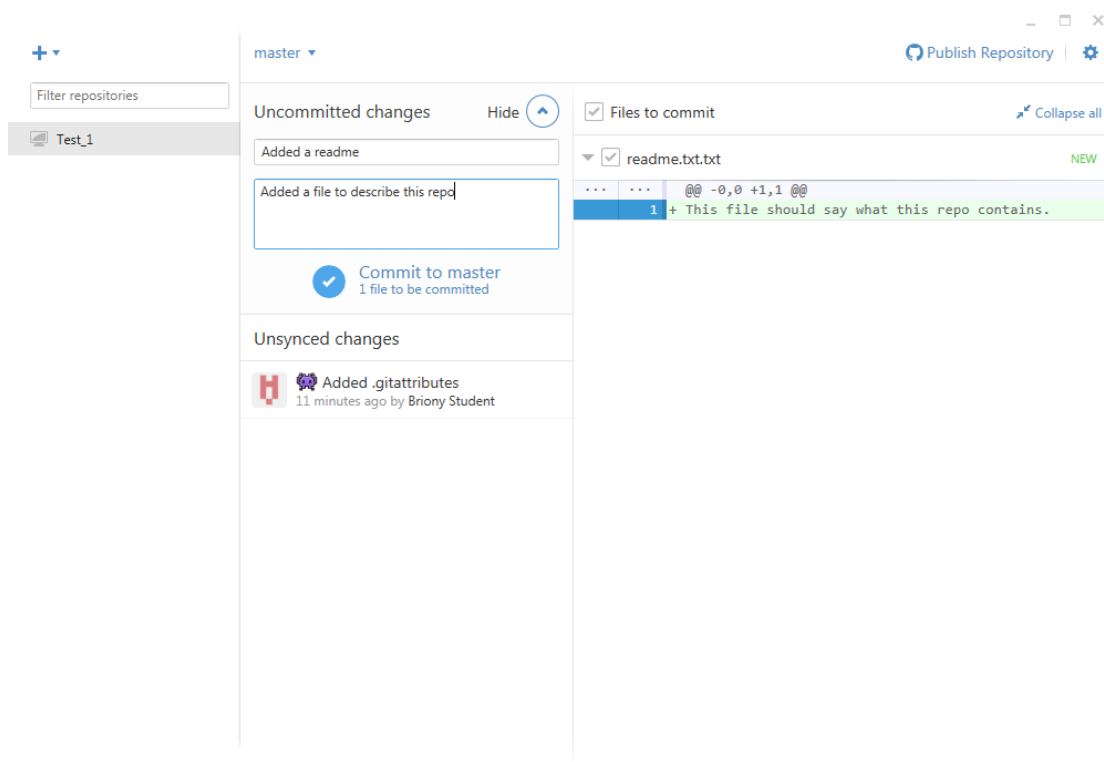
There should be a folder called “Test_1” this is where all the files in your repo will be stored. You can open this folder from the “tools and options” menu (as above). At the moment it should just contain one text files “.Gitattributes (whose name will be hidden in explorer). This contain some “Repository settings...” for this repo (and you can ignore it for now, though they will need changing, along with a .Gitignore file when you start using Unity, to stop Git trying to keep track of **everything**).

Let’s create a file that we want our repo to include. Use a text editor, e.g. Notepad to create a file **in the “Test_1” folder** called “readme.txt” and add the following content.



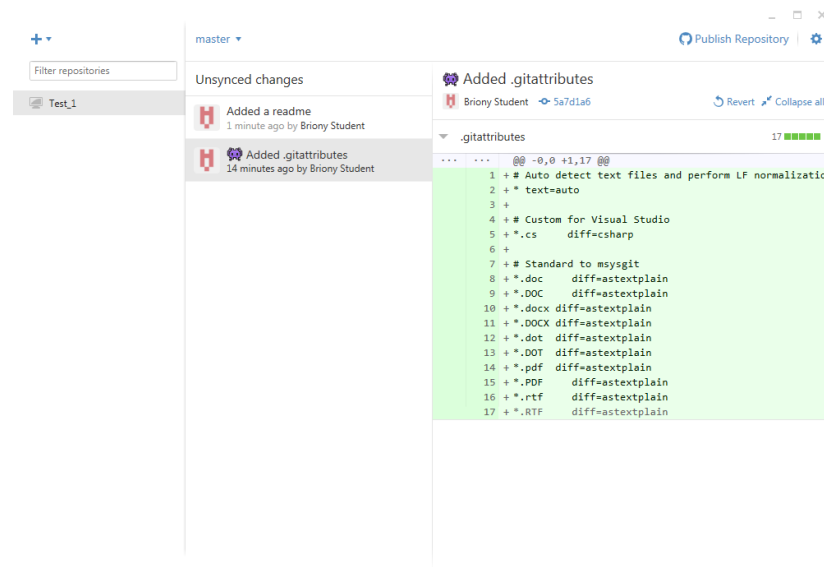


Go back to the repo and you should see that there are now “Uncommitted changes”. Click on “Show” to see what these are. The “readme.txt” file is now included and you can see what changes have been made to the file. Even though this file is in the repo folder and it being tracked, it has not been “committed”, so if you deleted it, there would be no way to get it back. Let’s commit!

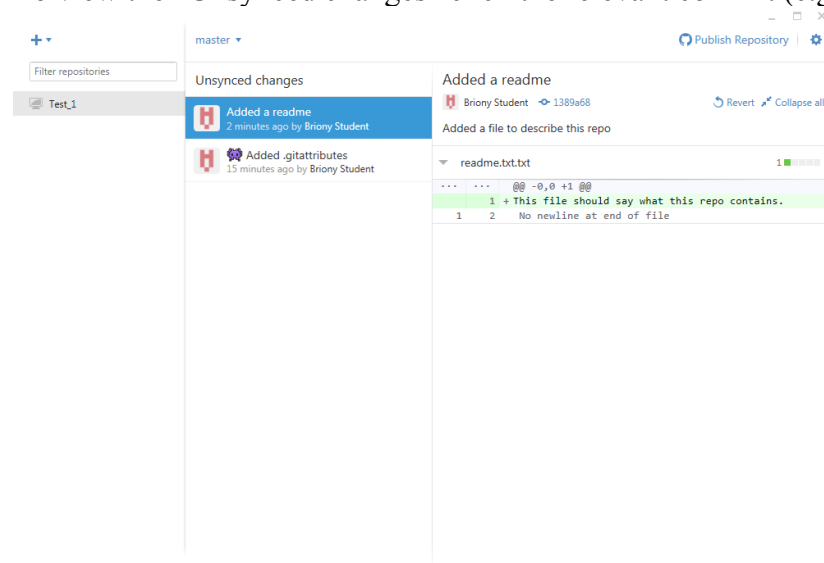


Enter a “Summary” and “Description” and click “Commit to master”.

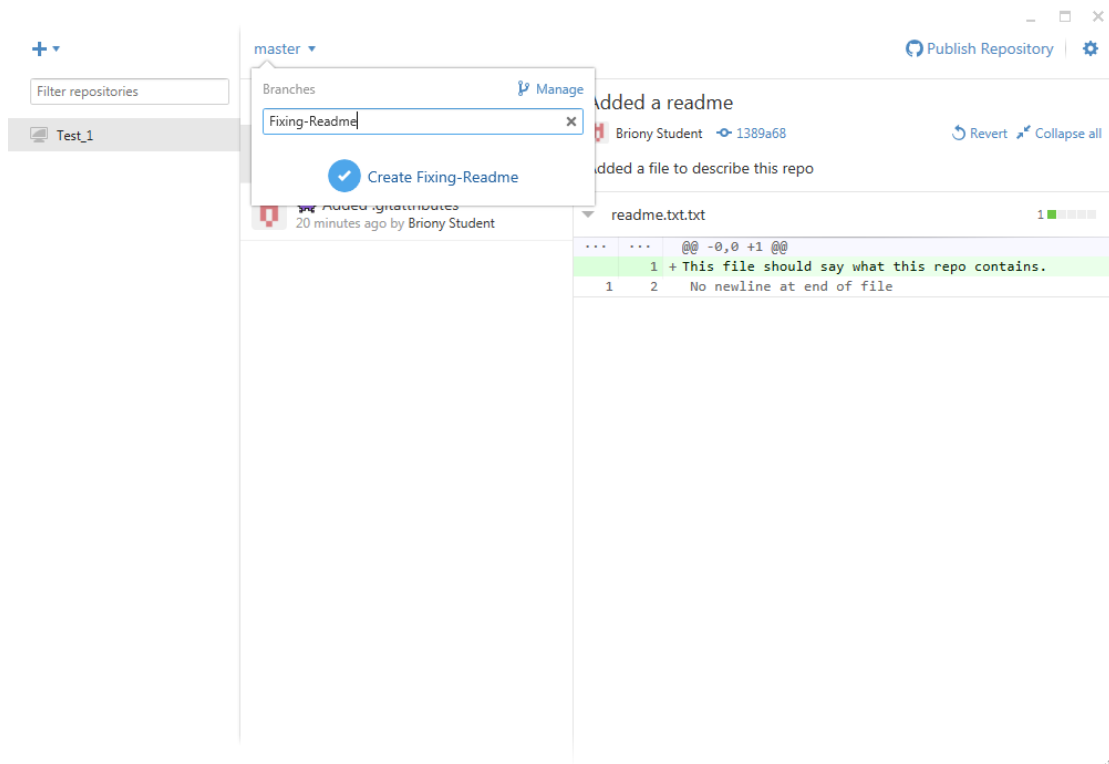
For interest, all of the repository files will go into a hidden subfolder called “.Git”, and the files you commit into the subfolders in “.Git/objects/”. You can have a look at this database in various ways (e.g. the shell command “Get-ChildItem –force –recurse” will show you everything), and for reference, the encoded filenames are created using SHA1. But for now, let’s concentrate using the repo rather than how it works.



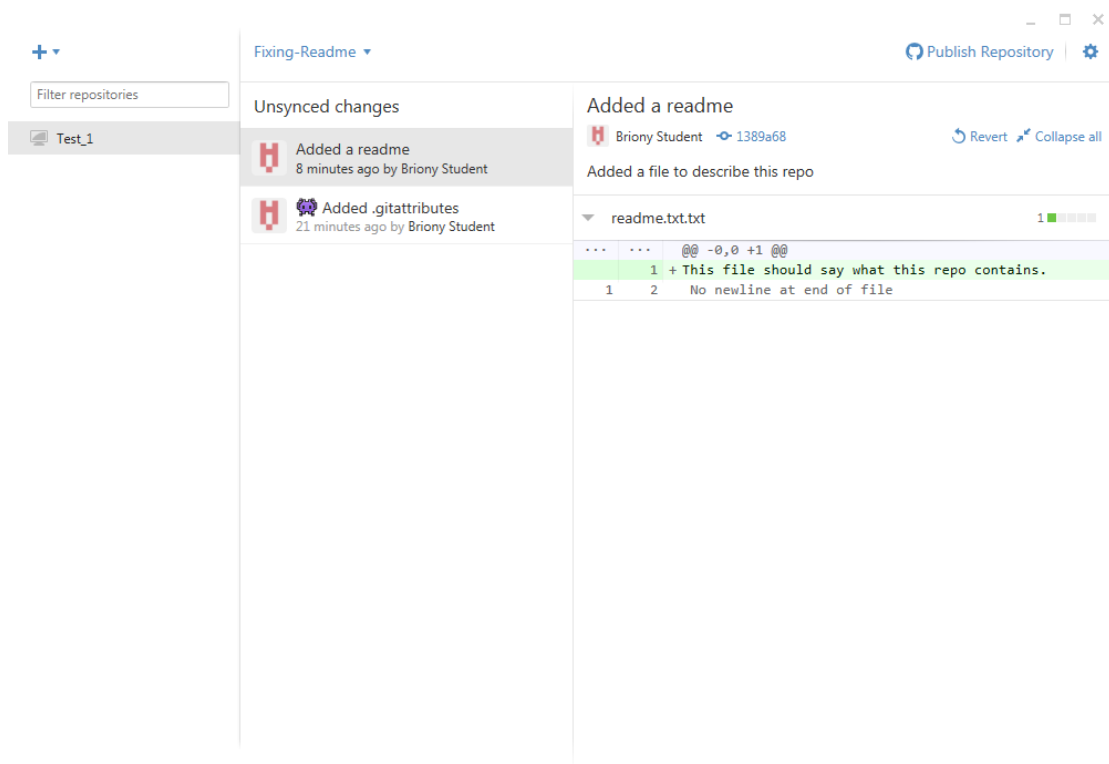
To view the “Unsynced changes” click the relevant commit (e.g. Added a readme”).



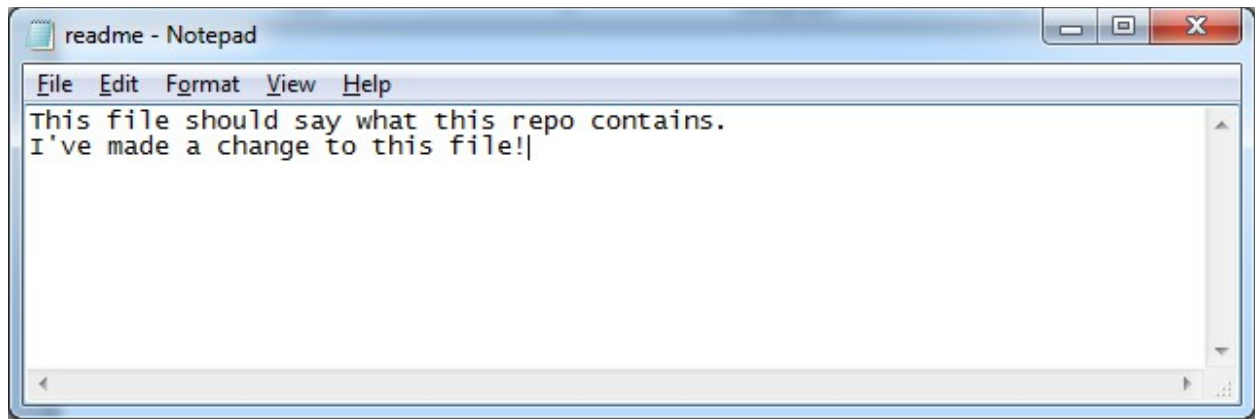
Note in the top-left corner is says “Master”. This means we are working what is known as the **Master Branch** and this is generally a **bad idea**. The Master Branch is reserved for completed work, i.e. work that has been tested and agreed by the development team for inclusion in the current build. When you download the latest changes (e.g. so that you can include you own work and prevent yourself from working with outdated code), you do not want to be downloading a load of bugs that break your project. So it is essential that when you are developing you create your own branches to develop with. These are often just copies of the Master Branch, but you can mess them up without having to worry about breaking the project for everyone else.



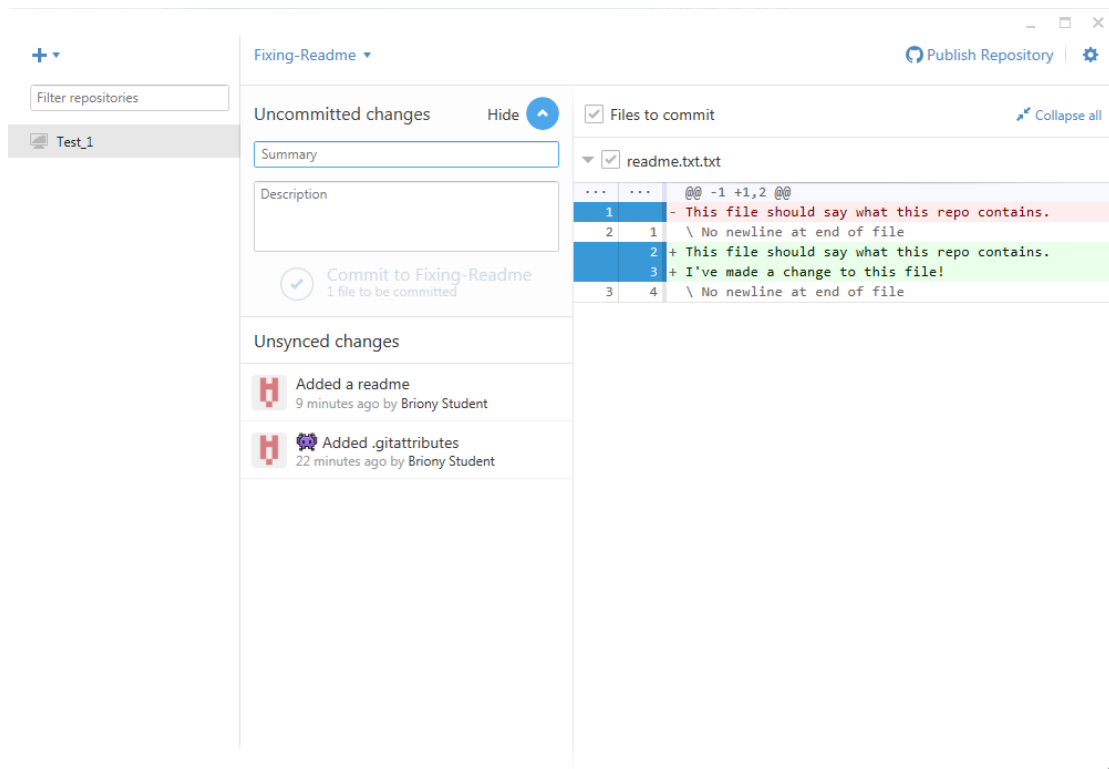
Create a new branch called “Fixing-Readme” as above.



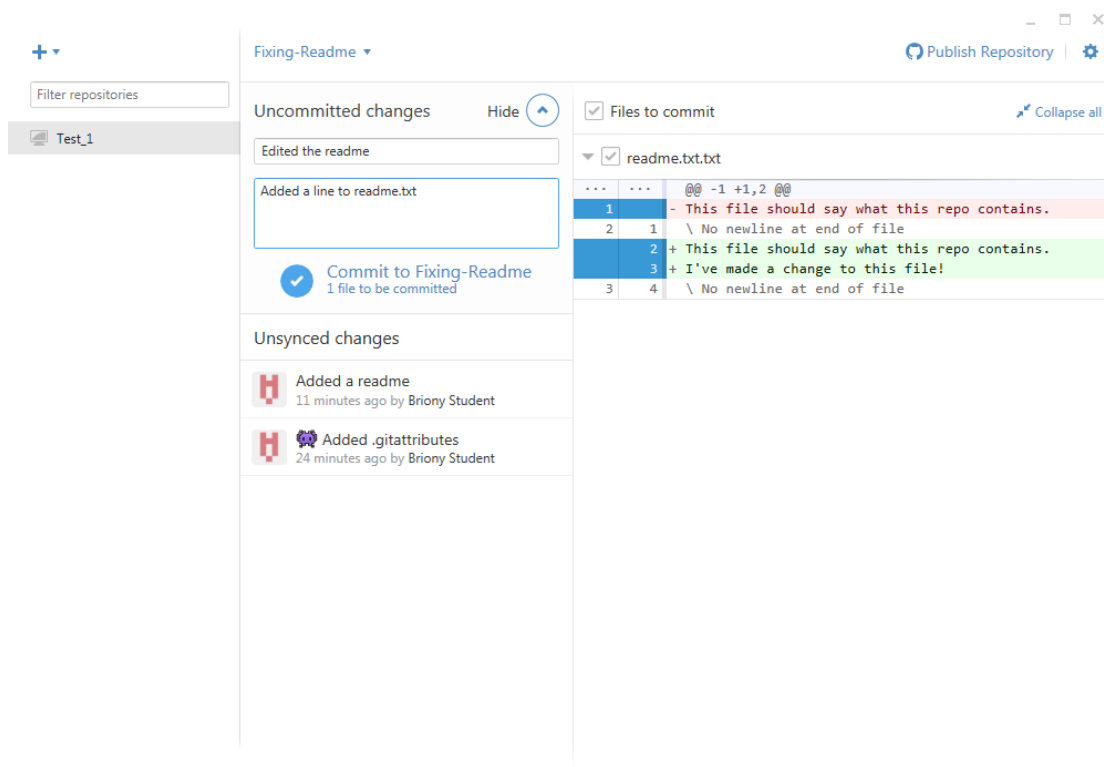
Notice how the current branch (top left) now says “FixingReadme”. Now we can make changes without worrying about breaking anything in the Master Branch



Open the file in Notepad and edit the text and save.

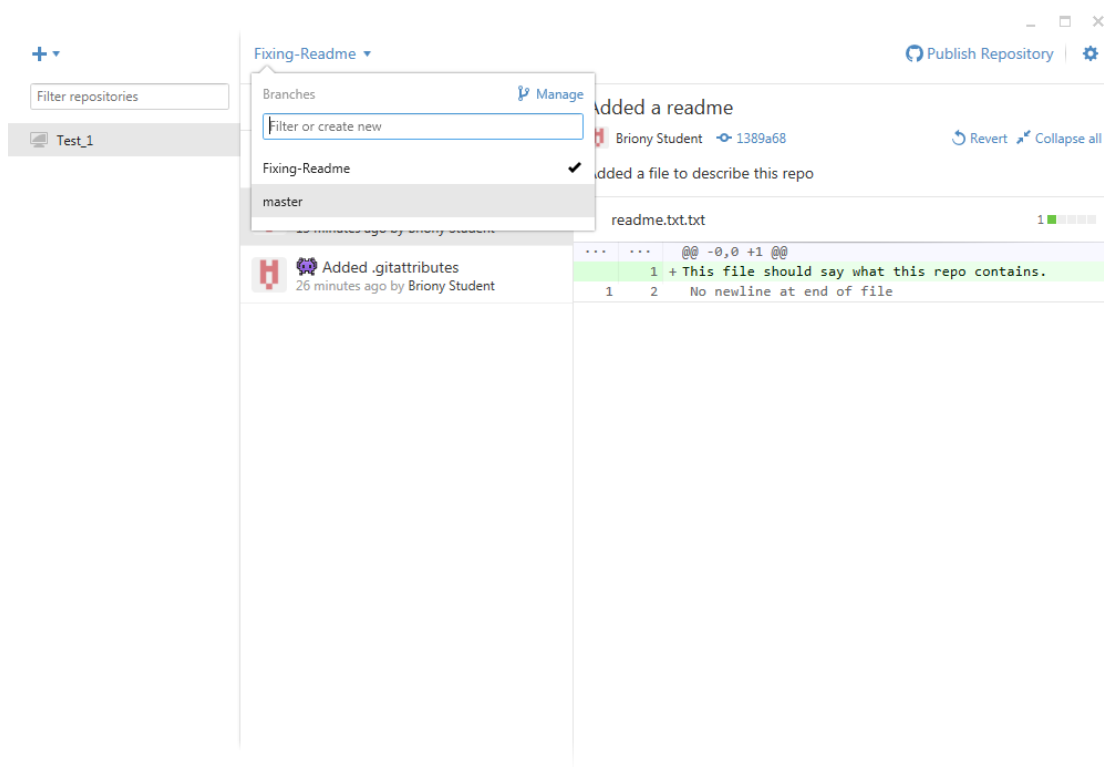


See that the application has now detected that some of your files have changed (it says, “Uncommitted changed in the top left). For interest, were you to now go into Notepad and undo your changes and re-save the file, then the application would detect that your file had gone back to its original state. It knows this because Git calculates the SHA1 from the contents of the file and uses it for the object’s filename. So, the same filename means the same contents (or at least it’s vanishingly unlikely that two files with the same SHA1 will contain something different to each other).



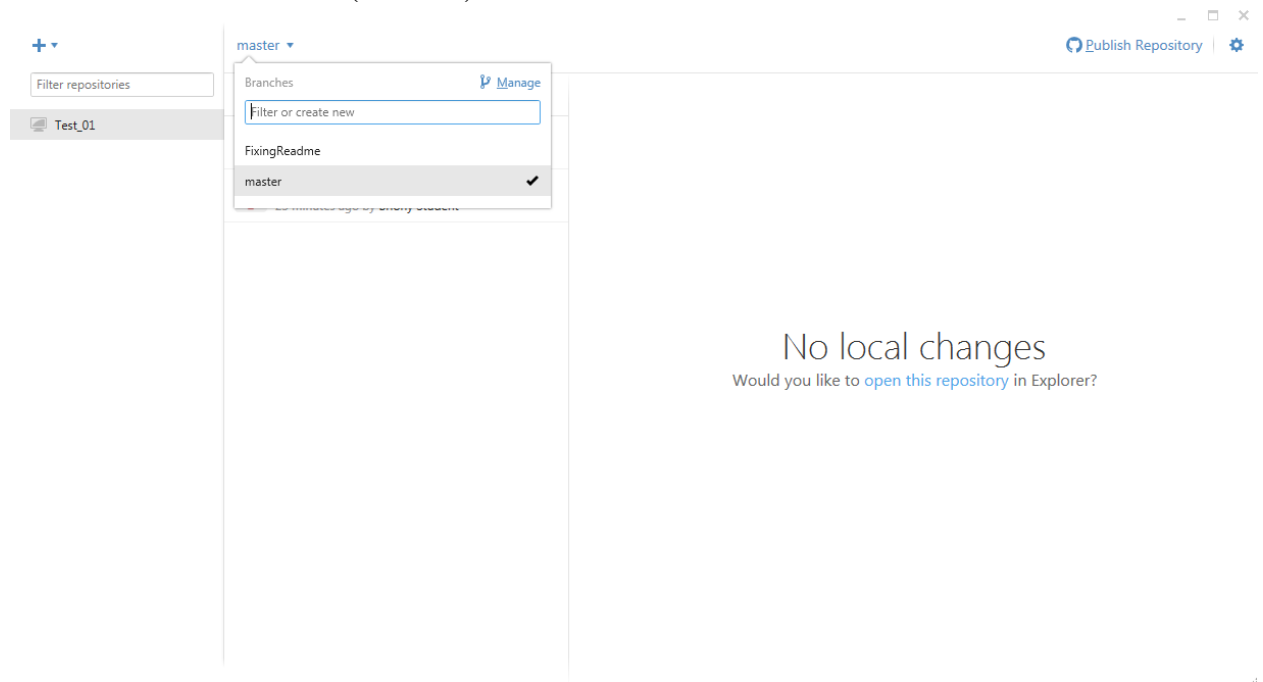
Enter a “Summary” and “Description” as before and click “Commit to Fixing-Readme”. N.B. Remember, if you do not **commit your changes**, they can be lost.

We should now have two committed branches, an original “Master Branch” and an edited “Fixing Readme” branch. You can change which branch the Test_1 folder currently contains (and which branch you would be working on if you make changes in that folder) by selecting the branches from the dropdown.

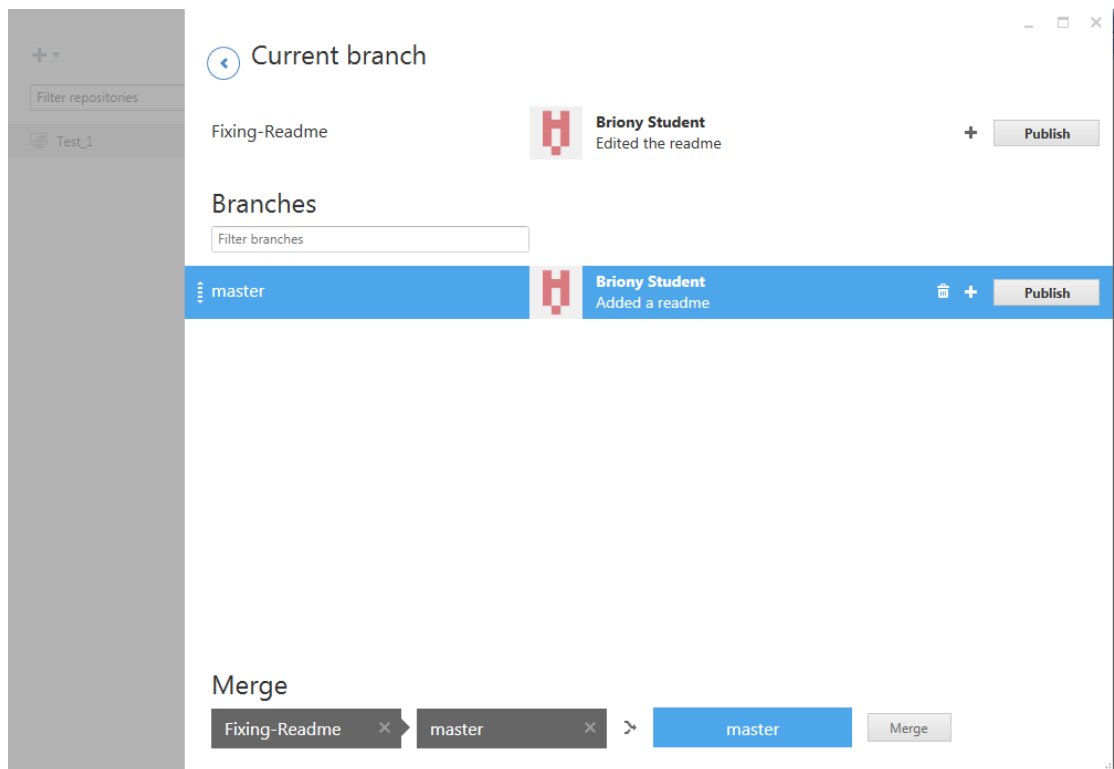


Try this:

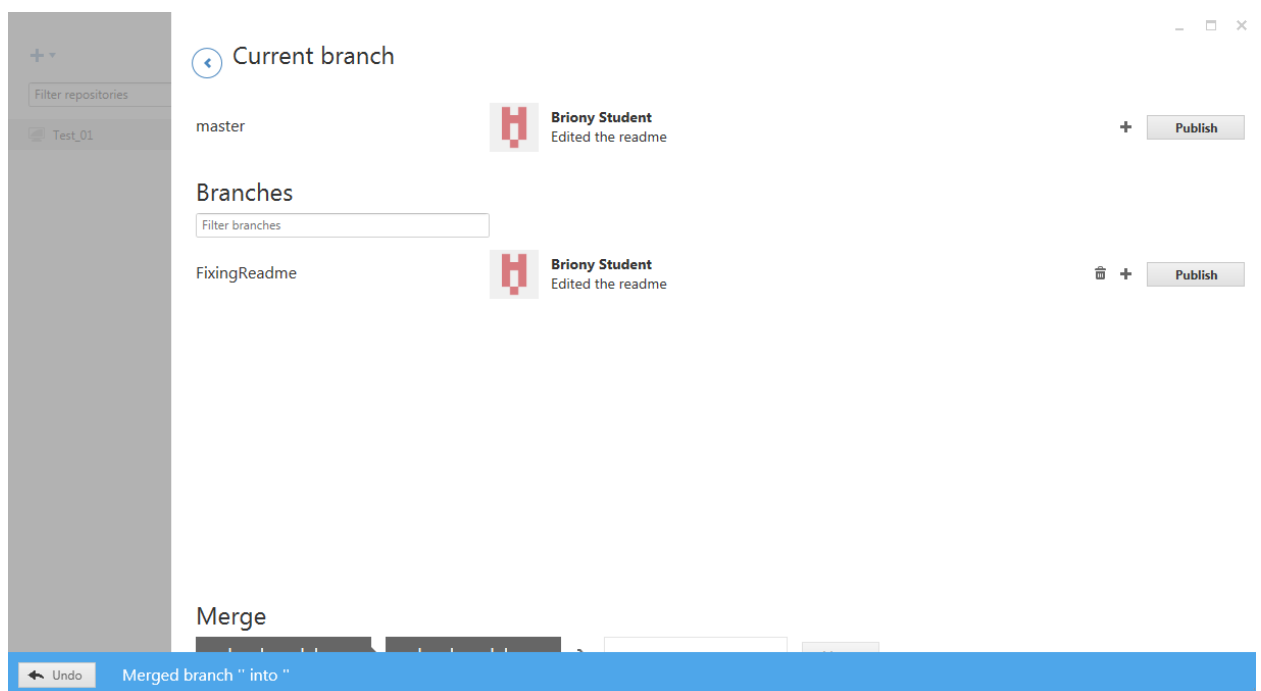
- 1) Select the master branch and open the readme.txt in Notepad
You should see the original (one line) contents.
- 2) Close Notepad, select the “Fixing-Readme” branch and open the readme
You should see the edited (two line) version.



Once your code is thoroughly debugged and you **and the rest of your team** are happy with the changes you have made it is time to merge them into the master branch. In the “switch branches” menu, click on “Manage”.



At the bottom, under where it says “Merge” drag the “FixingReadme” branch into the first box (i.e. the branch with the changes you want to add), and the “master” branch into the second box (i.e. the branch you want to make the changes to) and click “Merge”.



Go back to the “Current branch” and you should now find that the changes you made in your Fixing-Readme branch have been incorporated into the Master branch. You can check this by toggling between then two branches as before and checking the readme file. This time there should be no difference between the branches (also note that the “Unsynced changes” are now the same).

Try this:

- 1) Select the master branch and open the readme.txt in Notepad
You should now see the merged (two line) contents.
- 2) Close Notepad, select the “Fixing-Readme” branch and open the readme
You should see the edited (two line) version.

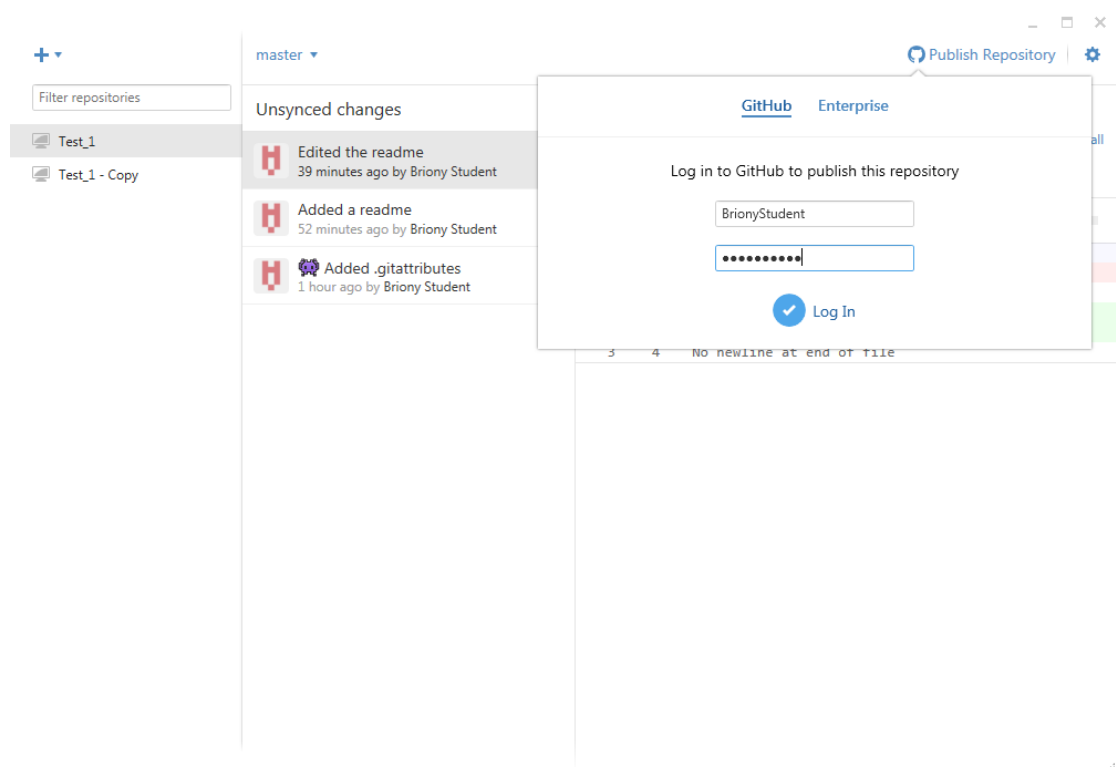
You might now feel that you no longer need the “Fixing-Readme” branch. If you are confident that your changes have now been successfully made to the Master branch you can delete your temporary working branch by clicking the trashcan icon in the “manage / merge” panel. Just make sure that the master branch is selected, since you cannot delete the currently selected branch.

You can also try copying Test_1 folder (e.g. copy and paste in windows explorer to make “Test_1 – Copy”). This is an entire copy of your whole repository, i.e. includes all the changes that you have committed, as well as the current working branch. You can add this so it is tracked in the application by dragging in the copy into the repositories panel.

The screenshot displays the GitHub web interface for a repository. On the left, the 'repositories' panel shows two items: 'Test_01' and 'Test_01 - Copy'. The main area is divided into three sections. The top section, 'master', shows 'Unsynced changes' with three commits: 'Edited the readme' (5 minutes ago), 'Initial commit' (18 minutes ago), and 'Added .gitattributes & .gitignore files' (27 minutes ago). The middle section, 'Edited the readme', shows a commit by 'Briony Student' (2218fe3) with the message 'Added a line to readme.txt'. The bottom section, 'readme.txt', shows a diff view with two lines of code: '1 + This file should say what this repo contains.' and '2 + I've made a change to this file!'. The diff view also shows a line number '1' and a line '3' with the text 'No newline at end of file'.

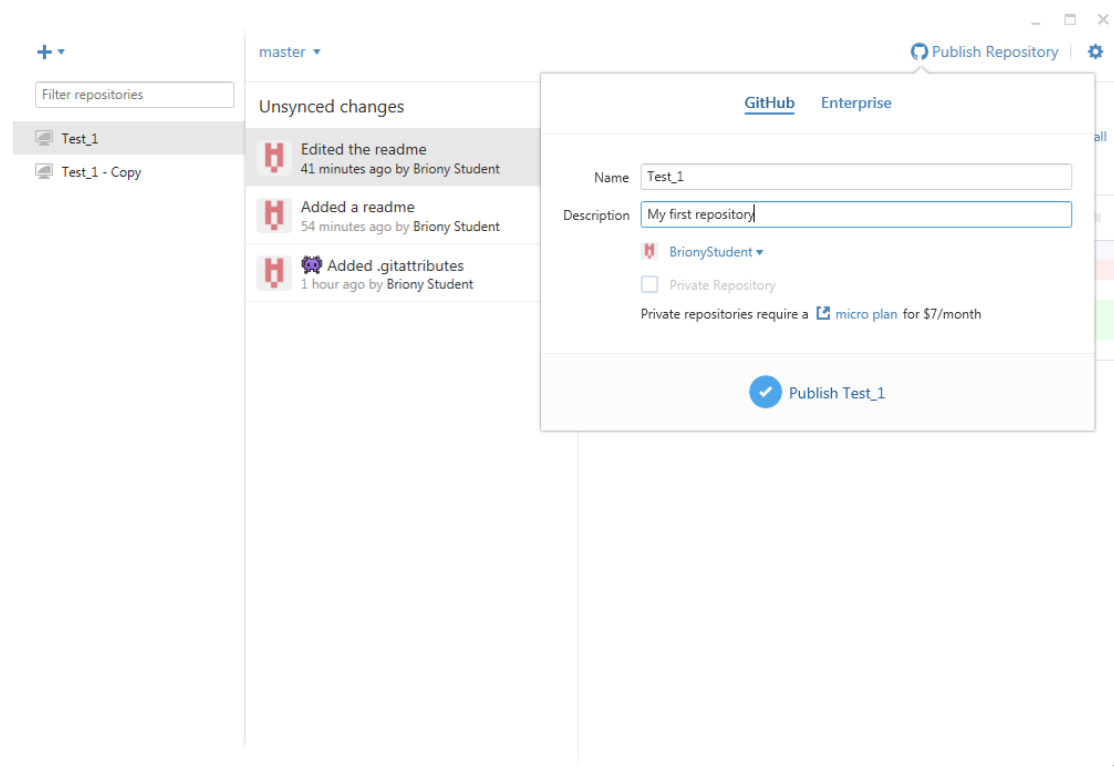
This is key to understanding how to work online with Git repositories that are shared online. You have multiple copies of the entire archive (each developer has one, and there is one at the GitHub website). Remember, at the moment we've been doing everything on the local machine. Essentially then, to collaborate, you make your own copy of the repo, make some private branches, edit these, then merge these changes into a public branch (e.g. the master branch if everyone is happy with them, although other branches can be shared too) and then push the changes to shared branches up to the server where the rest of the team can get hold of them.¹

Lets add our Test_1 repo to the GitHub server (remember, in effect this just means copying all the files from the Test_1 folder to GitHub, though we let the application take care of the actual work).



First, log into **your own** GitHub account through the application (you can do this here by selecting the master branch and then “Publish Repository”). **Note, if you have not already made a GitHub account then you will need to do so now.**

¹You can stop the application from tracking the copy of Test_1 (“Test_1 – Copy”) by right clicking on it’s title and selecting (“Stop tracking repository”). This will leave the files intact, so you can also delete the “Test_1 – Copy” folder if you want too, leaving just the original “Test_1” files.

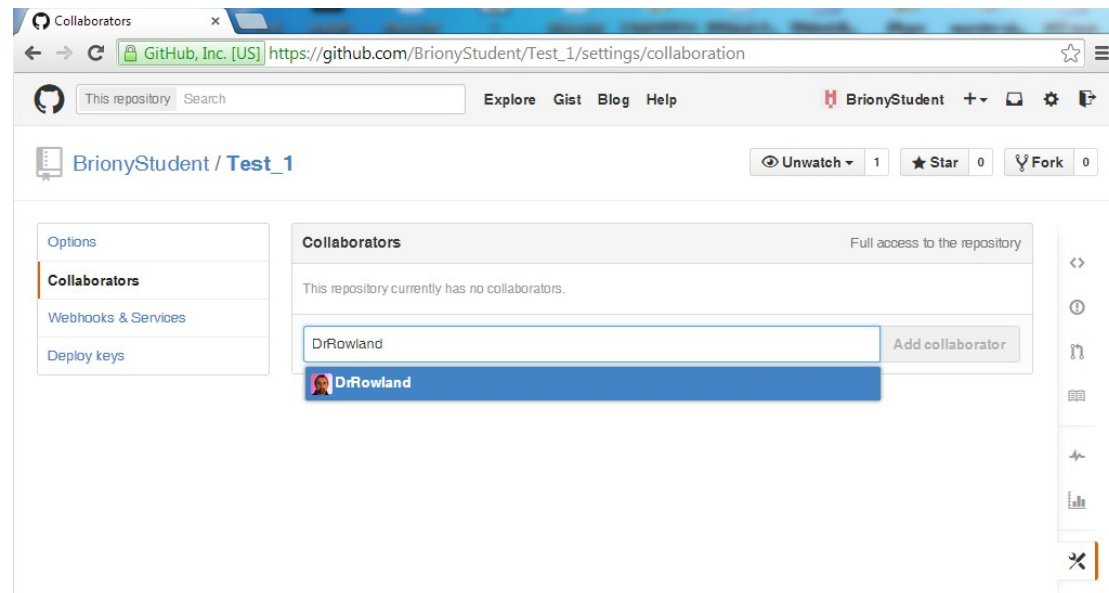


Give your repository a description and publish it to GitHub (note, private repos cost money)². Enter a description and then click “Publish”. This will create the repo on your GitHub account and will also sync the actual files for the currently selected branch. Note, if you want also upload and track other branches (e.g. Fixing-Readme) then you also need “Publish” these (or they will just be stored locally). And don’t forget to hit “sync” to make sure your local changes are sent to the server (and any updates are downloaded).

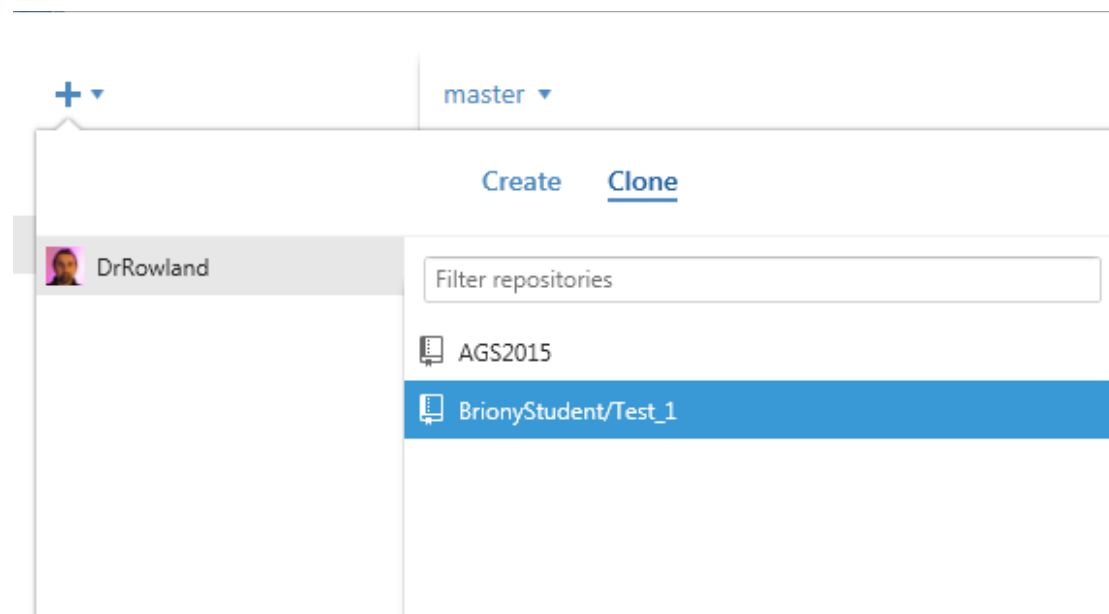
Remember, you need to “publish” the branches that you want to share. Some branches will just be for your own use, so there’s no need to publicly upload those, though you may want to keep a copy on Git for your own convenience. Never-the-less, you should always **make a proper backup of your work somewhere else too (e.g. copies of the whole repo folder)**, Git is not an alternative to backing up!

² Remember though, **GitHub** is just a web interface (with some collaboration tools), you could similarly use Git via ssh to push your changes to your own server...

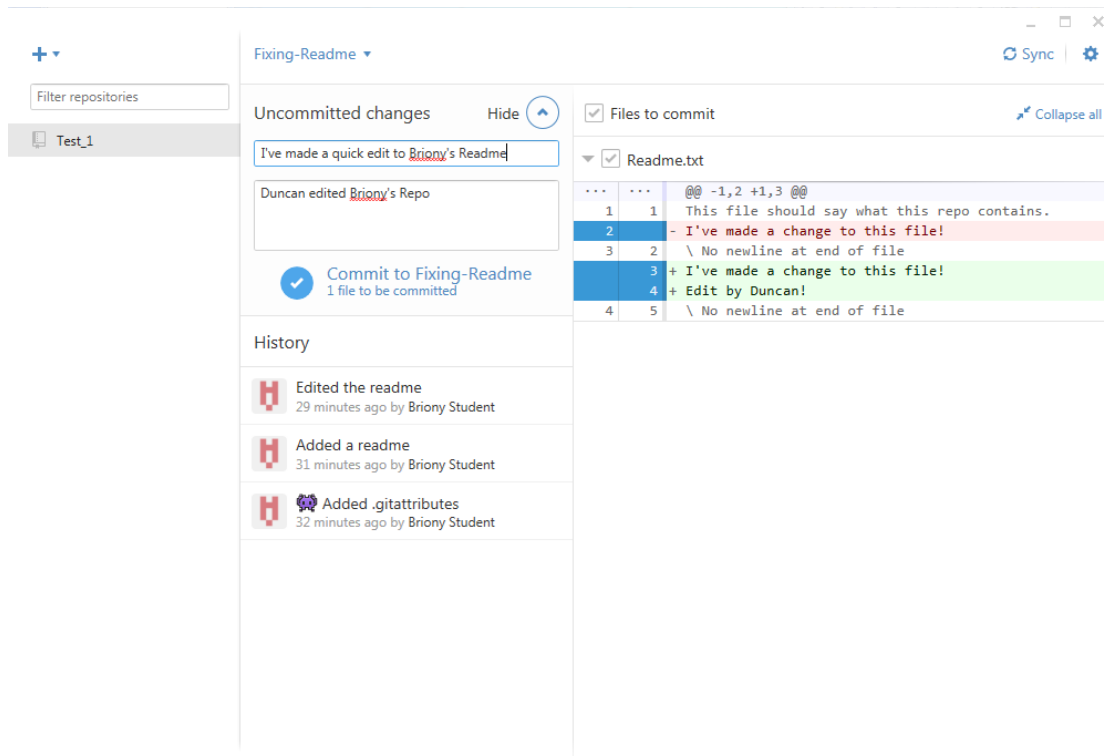
Although anyone can download your work, this does not mean they can edit it. You must add people as “Collaborators” to your project so they can upload changes to it. In the following example, Briony will add me (“DrRowland”) as a collaborator and I will edit her readme.txt and upload it. **Find a collaborator in the room and add them to your project (and have them add you to theirs) and follow along.**



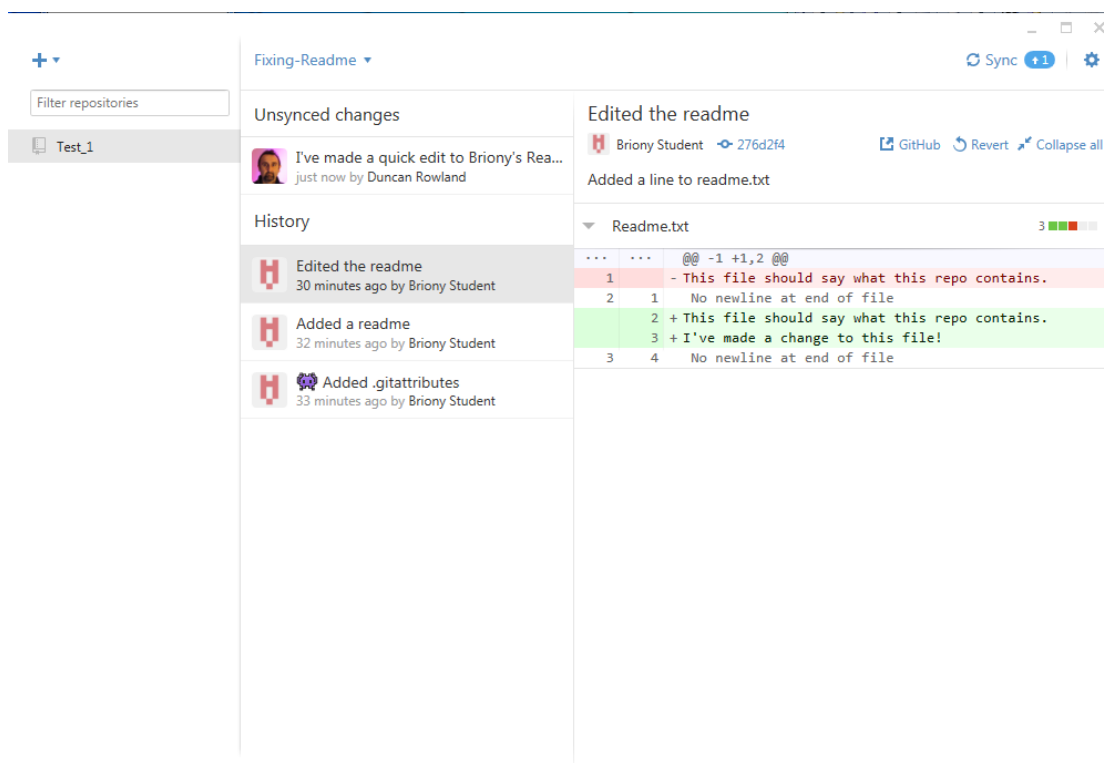
On the GitHub website open your own Test_1 repo and go to “settings”. On the left hand side click “Collaborators” and enter **your collaborator’s details** (and not mine as above) in the box and click “Add”.



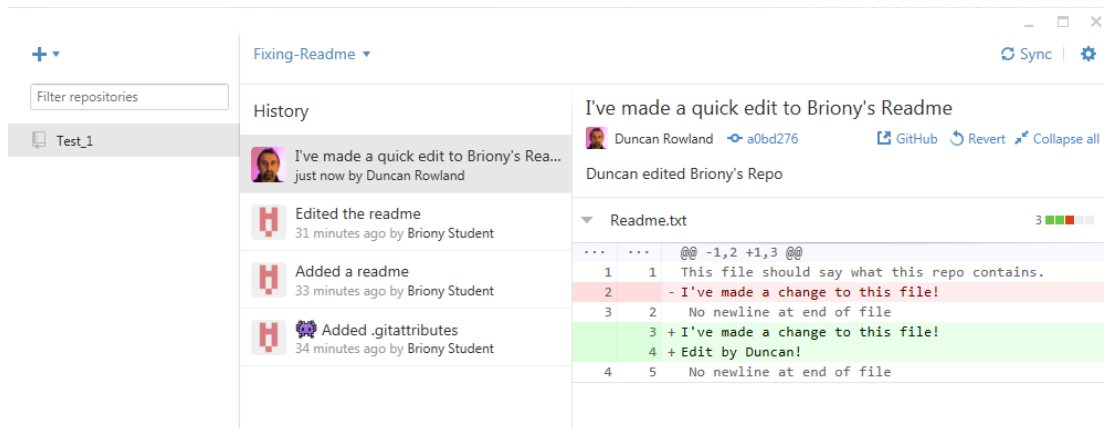
Back in GitHub, it’s might be a good idea to go into “tools-Options” and change your “Clone path” to something new and unique (e.g. /BrionysRemoteGit). This is because in this tutorial BOTH of you will have a Test_1 repo so there is likely to be some confusion unless you keep them in separate folders. **Your collaborator should now clone your repository, and you should clone theirs.**



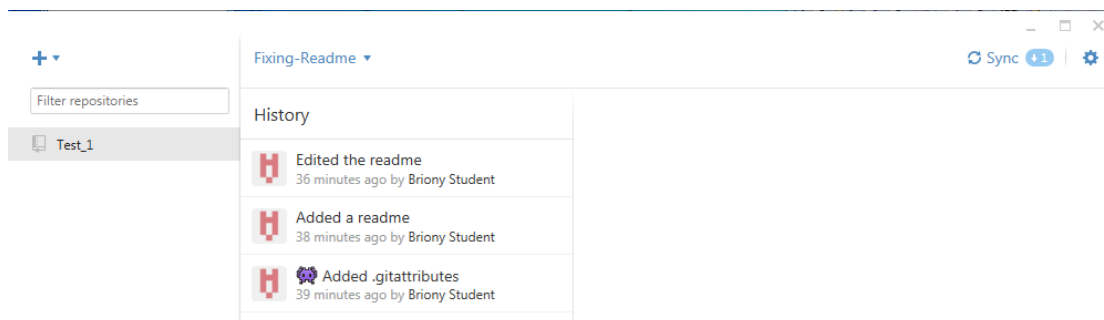
Once you have cloned your collaborators Test_1 repo you can make changes and commit them just like we did with our own repository.



These will be stored locally until you sync the local changes with the server, at which point your changes will be uploaded. Make some changes to your collaborator's readme file now and upload the changes as described.



Duncan's view of Briony's repo (with all his changes uploaded).



Briony's view of the same branch (Fixing-Readme) in her copy of the repo.

N.B. she needs to “**sync**” to download Duncan's work (it says +1 to indicate this).

It is important to keep your master branch synced so that you are not working with old code. You can always merge changes from the master branch into your local working branches (using a method similar way to that which merged your own changes into the master branch). N.B. When you've merged your changes with the master branch make sure it's properly tested and the rest of the group agree before you sync and uploaded it (otherwise you will break the master branch for everyone).

Your task now, until the end of the workshop, is to try using Git with your collaborator until you are confident with it. This will be your last opportunity to become familiar with it before we use it in the project.

N.B. Excuses along the lines of “I did loads of work but I tried checking it into Git and I somehow managed to loose everything” are NOT ACCEPTABLE FROM COMPUTING PROFESSIONALS (or indeed anyone)! You should ALWAYS BACKUP YOUR WORK. Git is a collaboration and versioning tool and not an alternative method for creating backups.

Now, try the following:

- 1) Choose a repo to work with (either yours or your collaborators)
- 2) Have one of you add another text file to it (i.e. commit and sync)
- 3) Make sure you're collaborator also syncs (so they receive the new file)
- 4) Now, both make some changes to the new file at the same time
- 5) Finally both try and check in your changes at the same time

What happens? - how can you fix this?

- 5) Try adding a new image file and doing the same thing (steps 3-5 again)

What happens now? - and how can you fix this?

- 6) Mouse over 'revert' and 'undo most recent commit'

Can you see the different between the two? (it's not quite as obvious as it looks)

- 7) Test using 'revert'
- 8) Test using 'undo most recent commit'
- 9) Continue to test Git for the remainder of the session (finish 1pm)

Advanced Topics: Note, there are other ways to collaborate via Git. You can 'fork' (create your own version of the repo on the server) and then request changes you make to be included in the main branch (issue a 'pull request'). This works well when one person looks after the repo but wants to allow non-collaborators to suggest changes to the project. Since we are working in small groups, where everyone knows each other, to keep things simpler, and uniform across teams, we will not be using this forking method.