Working in the Azure-PowerShell repo in GitHub

Contents

[Introduction 3](#_Toc443466038)

[Context 3](#_Toc443466039)

[Process implementation 3](file:///C:\Users\v-khogan\Documents\GitHub%20General\GitHub_Azure-PowerShellRepo_Documentation_v3_02-15-2016.docx#_Toc443466040)

[Simplified overview of GitHub setup 4](#_Toc443466041)

[First things first: tools 4](#_Toc443466042)

[Set up a GitHub account 4](#_Toc443466043)

[Set up Git and Git Bash on your computer 4](#_Toc443466044)

[Install and configure a markdown editor 5](#_Toc443466045)

[Repo setup 6](#_Toc443466046)

[Fork the repository 6](#_Toc443466047)

[Best practice: keep your forks and repos straight 7](#_Toc443466048)

[Clone the fork 7](#_Toc443466049)

[A note about Git Bash 9](file:///C:\Users\v-khogan\Documents\GitHub%20General\GitHub_Azure-PowerShellRepo_Documentation_v3_02-15-2016.docx#_Toc443466050)

[Configure Git shortcuts 9](#_Toc443466051)

[Keep your clone up to date 9](#_Toc443466052)

[Staying current: on a daily basis 9](#_Toc443466053)

[Why does this work? 10](file:///C:\Users\v-khogan\Documents\GitHub%20General\GitHub_Azure-PowerShellRepo_Documentation_v3_02-15-2016.docx#_Toc443466054)

[Editing and contributing content 11](#_Toc443466055)

[Branches 11](#_Toc443466056)

[Create a branch 12](#_Toc443466057)

[Make file updates 13](#_Toc443466058)

[Commit your changes 14](#_Toc443466059)

[Important info about commits 14](#_Toc443466060)

[Pull requests 16](#_Toc443466061)

[A simple example 16](#_Toc443466062)

[Editing within GitHub directly: project managers 18](#_Toc443466063)

[Finding Help.xml files 22](#_Toc443466064)

[Editing an individual file (Visual Studio) 24](#_Toc443466065)

[More about pull requests 27](#_Toc443466066)

[Interacting with the Git Bash command line 32](#_Toc443466067)

[Appendix 33](#_Toc443466068)

[Git install details 33](#_Toc443466069)

[Some Git commands for reference 35](#_Toc443466070)

[More magic: reverting changes already merged via pull request 36](#_Toc443466071)

# Introduction

## Context

Azure PowerShell content is published in two ways/places:

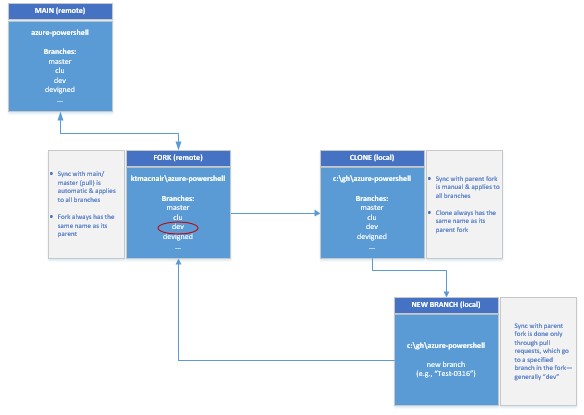
* PowerShell cmdlet documentation generated from DxStudio modules/projects (TOC, namespace, and individual cmdlet pages)
* Updatable Help files (help.xml) -- published on DLC. This is the content that gets pulled down from the DLC when users enter “update-help” from the PowerShell console.

Although Azure PowerShell content is published to MSDN from DxStudio, the XML files *and the underlying code* are stored, updated, and managed in [GitHub](https://github.com/), the largest code-hosting site in the world.

GitHub is based around an open-source, Linux-based version-control system called [Git](http://git-scm.com/book/en/Getting-Started-Git-Basics)—which, like Linux, was developed by Linus Torvalds. GitHub is in effect hosting Azure’s Git server and providing some additional functionality on top, such as repository forking, issue tracking, and online file browsing. When you interact with Azure files on your local machine, you’re using core Git functionality.

The repository is called “azure-powershell”--and it’s found at [www.github.com/azure/azure-powershell.git](http://www.github.com/azure/azure-powershell.git).

## Simplified overview of GitHub setup



### A note about process implementation

*Ultimately, the PowerShell writers will be authoring PowerShell content in Azure’s repo on GitHub. For now, we’ll follow the current process with respect to authoring and publishing, but Tara/Katie will test updating the Help.xml files on GitHub. In the short term, this will be done in addition to running the cabbing script for PSMAML builds and sending requests to PubDesk to load Help.xml files to the DLC for updatable Help.*

# First things first: tools

To get started, do the following:

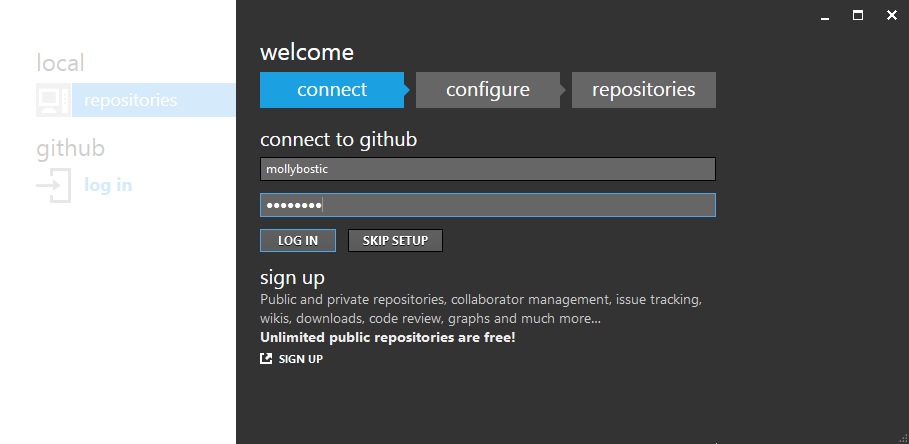
## Set up a GitHub account

The first step to contributing to this project is setting up a GitHub account. If you have not done so already go to [GitHub home](https://github.com/WindowsAzure/azure-content/blob/master/GettingStarted/github.com) and do so now. Be sure to associate it with your Microsoft alias.

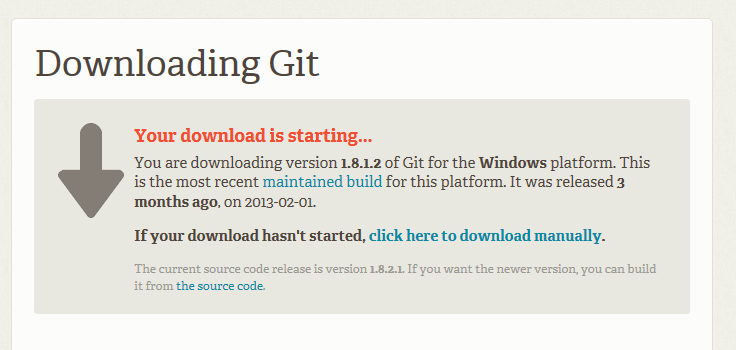
## Set up Git and Git Bash on your computer

Now that you have an account, you also need a copy of Git on your computer. GitHub has made it really easy for Windows users to configure Git by providing an app called GitHub for Windows.

1. Install GitHub for Windows from this link: <http://github-windows.s3.amazonaws.com/GitHubSetup.exe>
2. At the end of the install process, enter your GitHub user name and password and click **log in**.



1. Accept the default settings on the **configure** screen, enter your user name and email address, and click **continue**. In the **repositories** screen, click **skip**. You can now close **GitHub for Windows**. (No need to log out before you close.)
2. Separately, you need to install Git Bash, which is the command-line tool we use for working with Azure PowerShell content on GitHub. Install the latest version of Git Bash for Windows by installing Git from <http://git-scm.com/download/win/>.



Note that you won’t see Git Bash listed on this page, but it is included in the overall Git install package.

You can otherwise use all the default settings as you go through the installer setup steps. (See Appendix for [details about these settings](#GitInstallation), if you’re interested.)

## Install and configure a markdown editor

In the short term, at least, writers be authoring/editing cmdlet XML files and module Help.xml files in XML editor apps such as Visual Studio. However, conceptual topics will be in markdown (.md) files, so authoring/editing them is best done in a markdown editor tool, of which there are many.

The azure-content team uses Atom as their default editor, so for consistency’s sake you may want to use it as well.

See <https://help.github.com/articles/associating-text-editors-with-git/> for information about markdown editors you can use with Git.

# Repo setup

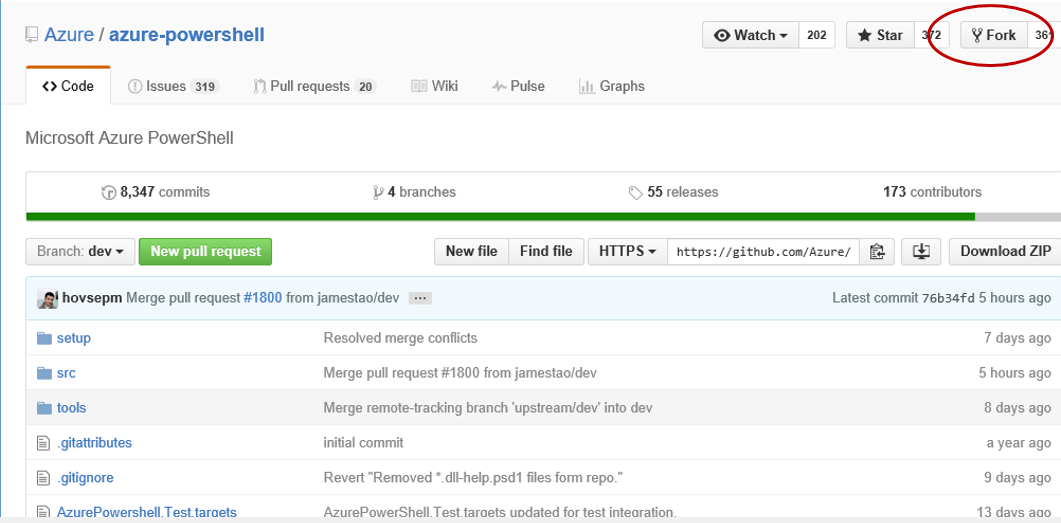
## Fork the repository

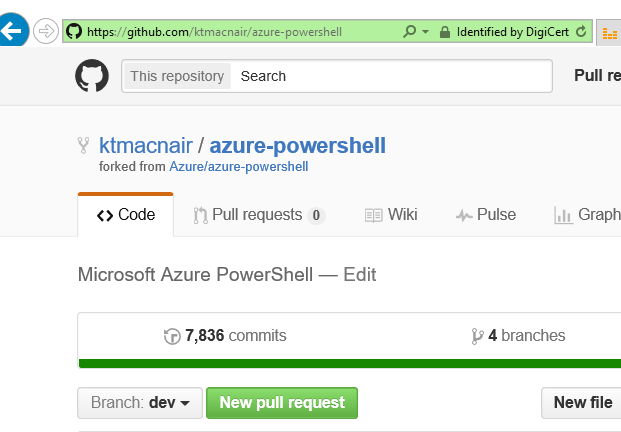
A *repository* (“repo”) is a grouping of files where Git has been initialized to track version information for the files and directories. In GitHub, organizations (for example, [https://github.com/Azure/](https://github.com/windowsazure/)) can contain any number of repos.

*Forking* is a concept that GitHub introduced to allow users to easily create copies of repositories. On GitHub, repositories are public by default, and any GitHub user can fork any public repository. Forking creates a complete copy of the repo, including all branches, but it places it within your account and keeps it permanently associated with your account. It enables you to make changes to the repo—adding and modifying files, for instance—and to request that your changes be merged with the main repo. (We’ll talk about how that works a little later in this document.)

The steps for creating a fork are amazingly simple, given how powerful the functionality is! Here’s all you do:

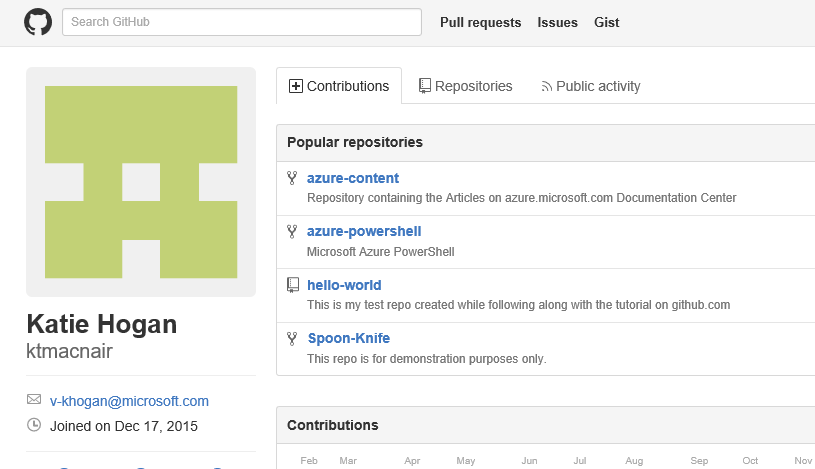
* Navigate to the [azure-powershell repo](https://github.com/Azure/azure-powershell).
* Click the **Fork** button in the upper right-hand area of the screen.



* That’s all it takes. Your fork is created!
* After you’ve created your fork, GitHub sends you to the web page for your fork, with a URL like: https://github.com/<your account name>/azure-powershell/ (see image, below) and when you’re logged in to GitHub, you’ll by default be in your fork:  
    
  

## Best practice: keep your forks and repos straight

Make your GitHub profile page your primary landing page on GitHub, especially if you anticipate working with more than one repo. This will make it easy to get to all your repos from one spot.



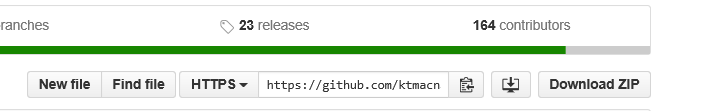
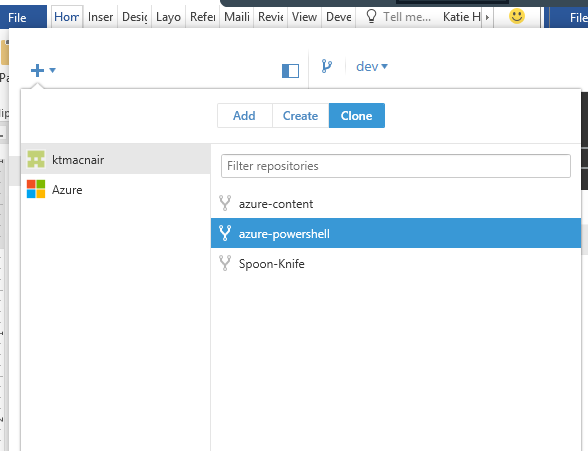
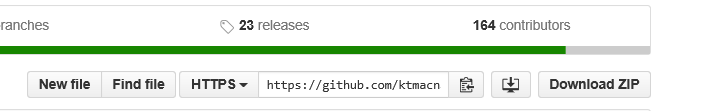
## Clone the fork

*Cloning* copies a repository to your local machine, where you can easily make changes, add, and remove files and directories. You’ll be cloning your fork of the repo to your computer, and you’ll work on that “version” of the repo when you’re making any changes.

Cloning is pretty simple too, and you can actually do it in a couple of different ways, depending on your preference.

First of all, it’s highly recommended that you use **c:\gh\\*.\*** as the location for all your repos. There are some path-length restrictions with GitHub, unfortunately, so it’s best to keep your path as short as possible.

Next you’ll create the clone, in any one of three ways:

* **GitHub.com**  
  While in your fork on github.com, click the download icon.  
    
    
    
  This will place the local copy of the repo in your computer at the Windows Users level, i.e., C:\Users\<your alias>, so then just move the repo to your c:\gh\ directory in File Explorer.
* **GitHub desktop app**
  + Open the application locally.
  + Click the plus sign in the upper left-hand corner of the screen, select **Clone**; you’ll see a list of the repos you have forked in GitHub.
  + Select the repo you want to clone, browsing to the desired location (c:\gh\\*.\*)  
      
    
* **Git Bash** 
  + First, while in your fork on github.com, copy the URL for the repo either with a simple CTRL-C or by clicking the clipboard icon.   
      
      
      
    Now open Git Bash locally and type the following at the command line:   
      
    git clone https://github.com/<your GitHub user name>/azure-powershell.git   
      
    This will also place the clone in the Users directory on your machine, so you will have to manually move the repo to c:\gh\.

## Configure Git shortcuts

### A note about Git Bash

All of the command-line steps in this guide use the Git Bash tool to work with Git on your local machine. Git Bash is included in the standard Git configuration; you installed it when you installed Git and used GitHub for Windows to configure your settings for connecting to GitHub.

When you open Git Bash, by default you’ll be in the c:\users\<your Windows user name> directory (for example, c:\users\v-khogan). Because your repos will be in the c:\gh directory you created, that means you’ll need to change directories at the command line each time you open Git Bash.

Next, you need to set up connections between the local clone of your fork and the remote repos in GitHub (Azure/azure-powershell and <your GitHub account>/azure-powershell) so that you can get the latest changes onto your local machine and push your changes back to GitHub. Git lets you define shortcuts to these locations so that you can easily reference them in subsequent commands.

When you set up your clone, Git automatically defined one shortcut for you, called ***origin***. It can always be used as a reference to <your GitHub account>/azure-powershell/ (that is, your fork of the azure-powershell repo).

The second shortcut you need, called ***upstream***, will be for the main Azure/azure-powershell repo. To define this shortcut, enter the following command in Git Bash:

git remote add upstream [https://github.com/Azure/azure-powershell.git](https://github.com/WindowsAzure/azure-content.git)

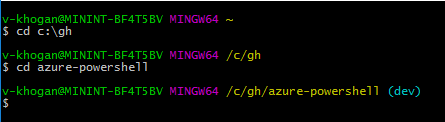
## Keep your clone up to date

Lots of people are contributing to the azure-powershell repo all the time, so it’s important to keep your local clone current with the latest content on the remote. Having set up shortcuts to the remote locations, you can easily pull the latest content to your local machine.

***NOTE:*** *This use of the word “pull” is not to be confused with the specific GitHub functionality known as a “pull request.” In this instance, we’re using “pull” in the standard way.*

### Staying current: on a daily basis

Open Git Bash and make sure your clone is up to date. This will ensure that you are always working with the most current version that contains everyone else’s changes (merged or otherwise) that came in since you were last working in the repo.

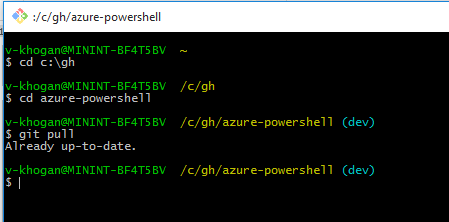
* From the Git Bash command line, type “cd” (for “change directory”) and the file path for the folder containing your clone of the repo.   
    
  
* *Assuming that you already defined your upstream location,* type git pull upstream master at the command line to force-update your local copy of the main repo at the master branch level (i.e., all of it).   
    
  Note, if you’re not already logged in to GitHub.com, you may get a couple of credential-checking dialog windows.

### Why does this work?

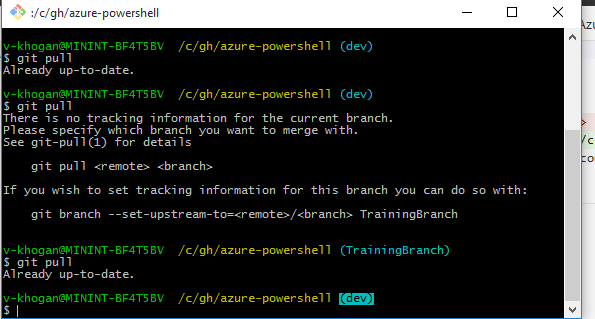
When you created shortcuts earlier, you defined the location of “upstream”—so now when you run this command, GitHub knows what you are referring to.

The command **git remote add upstream** [**https://github.com/azure/azurepowershell.git**](https://github.com/azure/azurepowershell.git) means that “upstream” is at this specified location.

So the command **git pull upstream master** is telling your local clone of the repo to pull all of the content from the master branch of the “upstream” repo, previously defined as being at https://github.com/azure/azurepowershell.git.



If you’re in a different branch when you attempt to sync your clone, you’ll get an error message:



So just go back to the GitHub client, switch back to your main (dev) branch, and reenter the command.

# Editing and contributing content

## Branches

When it’s time to contribute to the azure-powershell repo, then, you’ll want to create a new temporary branch of the repo on your clone.

Branches enable you to segment changes for specific purposes. For example, when updating the Help.xml files to coincide with monthly Azure PowerShell releases, you’ll want to create at least one new “working” branch for that month’s update. Depending on timing, among other things, it might be sufficient to have only one temporary branch each month. However, for months with lots of changes to a variety of modules (especially if they’re on different timelines), having more than one working branch is preferable.

One key thing to bear in mind when working with GitHub repos is that *you should never be working in the “master” branch, even on your local clone*. This is key!

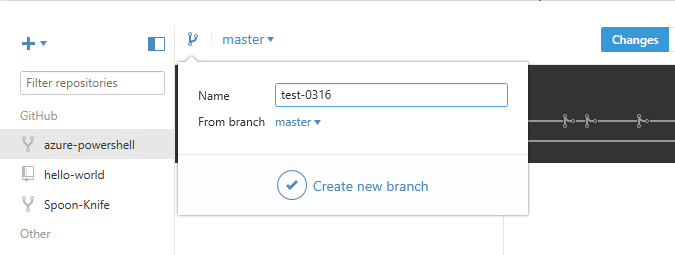
*Branching is a core concept in Git, and the entire GitHub Flow is based upon it. There's only one rule:   
anything in the master branch is always deployable.*

When in doubt, create a new branch. *This is true even if you’ll be merging your changes with the “dev” branch of the main repo.*

### Create a branch

As with the other steps we’ve looked at, you can create a new branch for your repo in a couple of ways.

* **GitHub desktop app**



* **Git Bash command line**

Say you wanted to create a local branch called test-0316 that’s a copy of the upstream (Azure/azure-powershell/) **master** branch. Type the following command in Git Bash, and it will create the branch and pull the upstream content into this branch on your (local) clone.

**git pull upstream master:test-0316**

**Optional**

If you want to save a copy of your new branch to your origin GitHub fork (<your GitHub account name>/azure-powershell/), enter the following:

**git push origin test-0316**

Use the following command to check out the new branch so that you can start working in it on your local machine:

**git checkout test-0316**

Note that if you haven’t pushed your local clone with branch to your origin fork then you won’t have to check the branch out; you’ll be able to get right to work on authoring/editing content.

Now you’re ready to start adding and editing content locally in your test-0316 branch. You can open the azure-powershell directory (c:\gh\azure-powershell\) to find and open files for editing.

## Make file updates

Okay, now you’ve got the following set up:

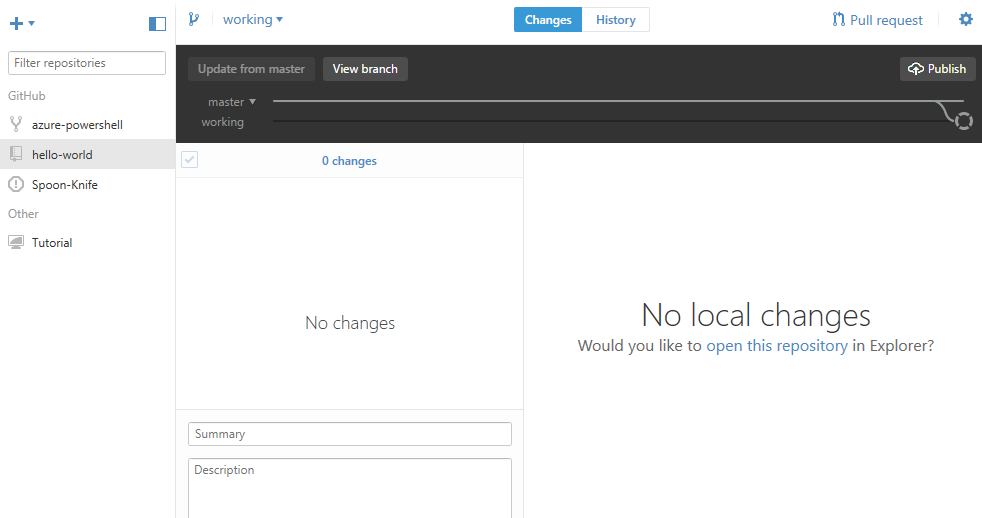
* GitHub account
* Fork (remote) of azure-powershell repo (this is the location for **origin** shortcut).
* Local clone of your (remote) fork.
* Shortcuts for **origin** and **master**; up-to-date content.
* New branch (in local clone) in which to make edits.

And you’re finally ready to edit & contribute content!

**Reminder about the monthly update process: Always create a new local branch for each module being worked on. Make changes to the “working branch”; pull requests should always be made to the dev branch upstream.**

What follows is a very simplistic example of how to work locally in a *branch* on your *clone* of your *fork* of a repo on GitHub.Hub.

When you open the GitHub desktop app to work in your repos, and if you haven’t been doing much with your files since your last pull request, you’ll see a rather spartan GUI.



Your next step is simply to click the “open this repository” link.

This opens the relevant directory in Windows Explorer, so you can navigate to your markdown, HTML, or XML file(s) and edit away. There’s nothing crazy going on here, just make your edits like you normally would. GitHub will do the rest.

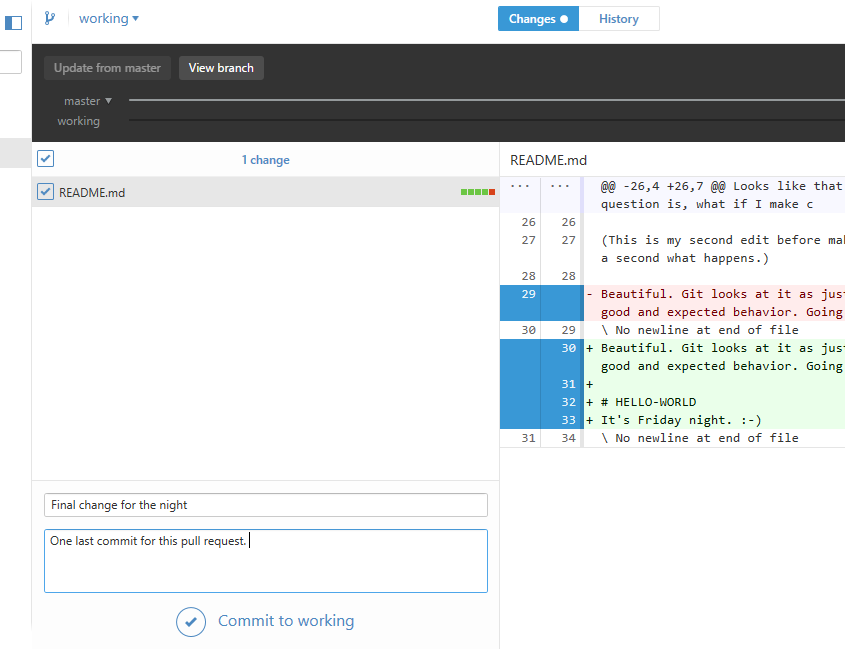
NOTE: For information about Markdown and Git-Flavored Markdown, see <https://guides.github.com/features/mastering-markdown/#intro>.

## Commit your changes

After you’ve made your changes, the next step will be to ***commit*** *those changes to a specified location in your repo*.

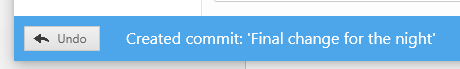
If you look at the file directory, you’ll see that it doesn’t maintain multiple versions of files for different branches within a given repo clone. Edits to the files are made once, but the way that they are applied in different branches is part of the mysterious magic of GitHub.

When you come back to the GitHub app you’ll see the name(s) of the file or files that have been modified. When you click an individual filename, you’ll see exactly what changes were made, in a line-by-line diff display. For example:



At this point, add a name for this set of changes (i.e., this “commit”) and a short comment about the change (in the lower left-hand corner of the left pane). Now you’re ready to commit the changes to the branch you’re working in.

After you’ve clicked **Commit to …**you’ll get a confirmation message that the commit was created:

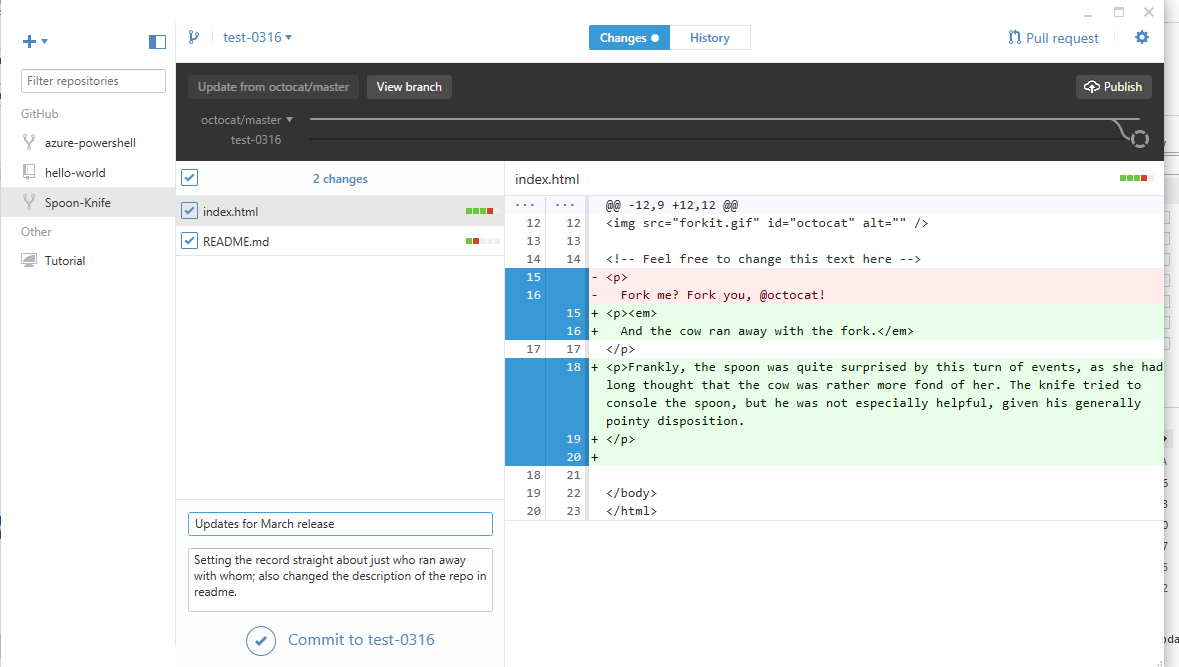


### Important info about commits

Here are two very important things to keep in mind about commits:

* A single commit can comprise changes to one or many files.
* You must specify the proper branch for the commit.

#### Multiple files, one commit



As you can see, two files (index.html and readme.md) were modified. The left-hand pane shows a list of the files that have been edited; with any filename selected in the left-hand pane, the right-hand pane displays, line by line, the changes made to that particular file (in this instance, the index.html file). Switching the selection in the left-hand pane to the name of a different file will cause the display on the right to show, line by line, the changes made to that other file.

However, *a single commit will carry the changes to both files rather than just one or the other.* And similarly, the commit title and description that you enter (as shown in the bottom-left-hand corner of the screenshot) are intended to cover the overall set of changes rather than referring only to changes to one of the files.

#### Commit to a specific branch

You can also see from the screenshot above that when creating a commit, *you must specify the branch you want the commit to go to*. You need to pay careful attention to the branch you’re in when you create a commit, whether you do so from the GitHub app or from the command line, so that you commit your changes to the right one.

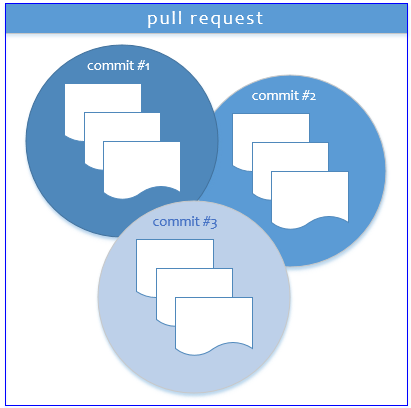
The general rule of thumb is that you should *never create a commit in the master branch*.

For our purposes, create commits for the azure-powershell repo in the temporary branch you created for your current project.

## Pull requests

A pull request is essentially a request to a repo’s owners for them to “pull in” all of the changes from a given branch in the clone to a specific branch in the main repo.

Pull requests can comprise multiple commits and should not be submitted piecemeal. For example, if you are working on several files, e.g., multiple cmdlets in a module, save each of them in individual commits. Also, if you need to stop and come back to the project later or the next day, save and commit but do not submit a pull request until you have completed all of your edits/changes to the file. The same principle applies if you’re adding a new file. You may work on it several days in a row, in which case each day’s saves should be made into a daily commit—but then the pull request should be made once, and will include all “committed” changes.



Here are a few things to keep in mind about pull requests:

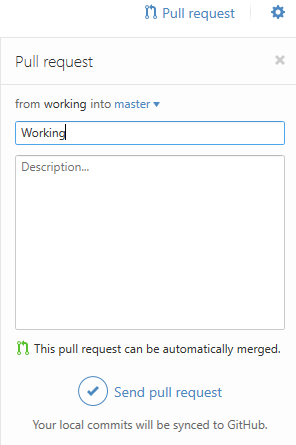
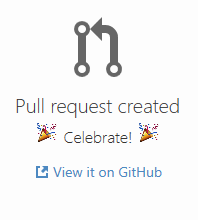
* A pull request can comprise multiple commits (which, in turn, may comprise multiple files).
* Pull requests go from one branch on the local clone (typically a temporary branch) to one of the “permanent” branches on the main, remote repo.
* For azure-powershell content, pull requests should be submitted to the remote **dev** branch, not the **master** branch.

### A simple example

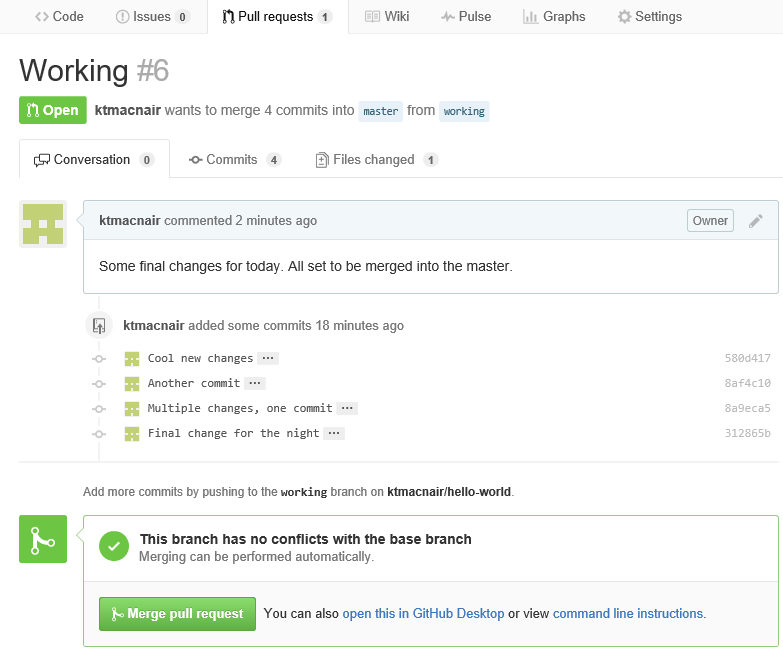
In this simplified example, a pull request is submitted from the clone’s “working” branch to the “master” branch on the main remote repo. (This is for demonstration purposes only. Remember that azure-powershell repo pull requests should be made to the dev branch, not the master.)

A series of screenshots is provided to illustrate the process.

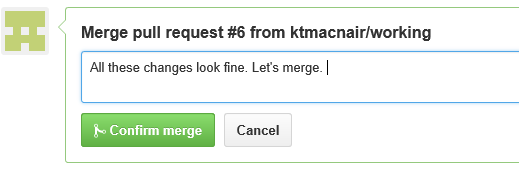
1. The contributor successfully submits a pull request:



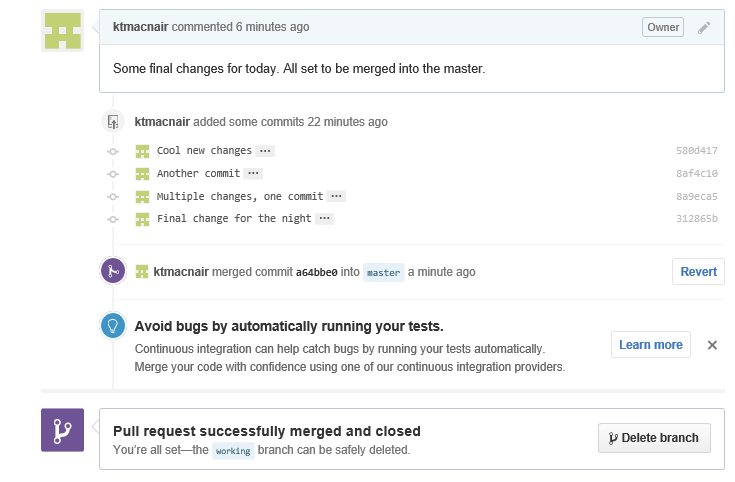
1. The pull request shows up on GitHub.com, where the repo owner can view it in detail:



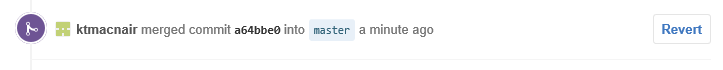
1. The repo owner clicks the **Merge pull request** button and must provide a comment & confirmation:



1. The main page for the repo immediately reflects the merge:



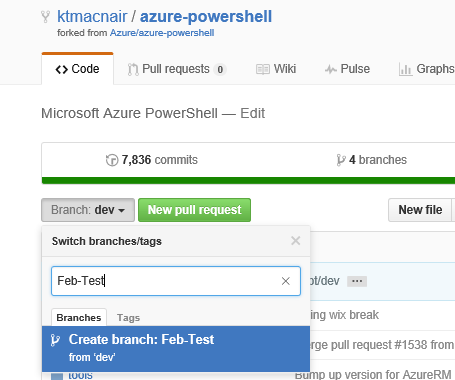
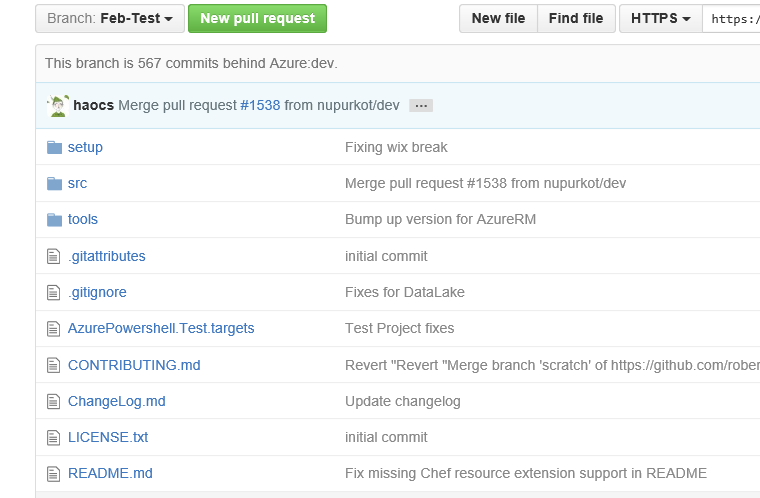
1. GitHub provides the ability to revert pull-request merges:

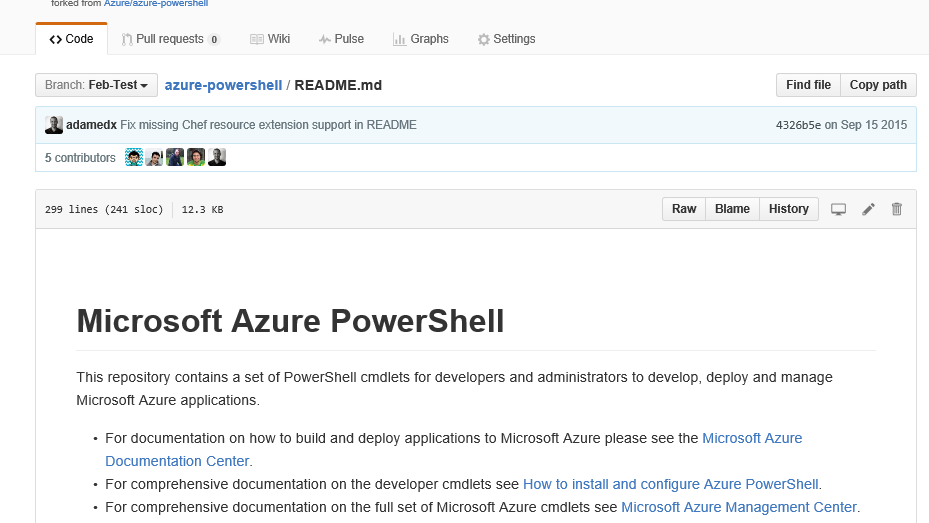


It really is magic!

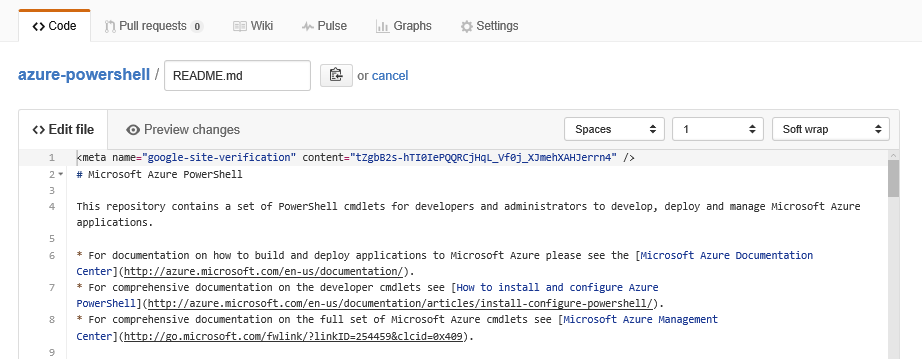
# Editing within GitHub directly: project managers

If you are a CSI CE PowerShell Content Project Manager, it is likely that the edits you’ll be making to the azure-powershell repo will involve updating Help.xml files at the module level rather than making edits to the XML files for individual cmdlets. Although you will still be working in your fork of the repo, you’ll do so on GitHub.com rather than on your local clone.

* Navigate to azure-powerShell on github.com, making sure that you’re in your own fork of the repo, not the main one.
* Create a new “working” branch (in this example, “Feb-Test”).
* Navigate to the file that you want to edit. In this example, we’ve made changes to the readme file (README.md) at the top level of the repo.   
    
     
    
  Click the filename of the file you want to edit. You’ll see the file contents displayed within the content frame:



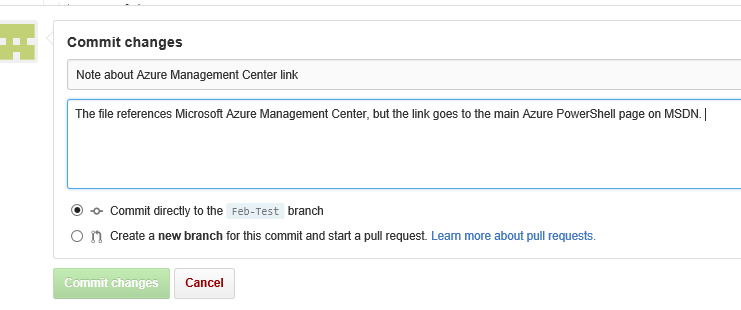
To edit a file without having to open it in Visual Studio or some other editor, click the pencil icon in the upper right-hand area of the file content window. *This option is ideal for typos and minor edits.*



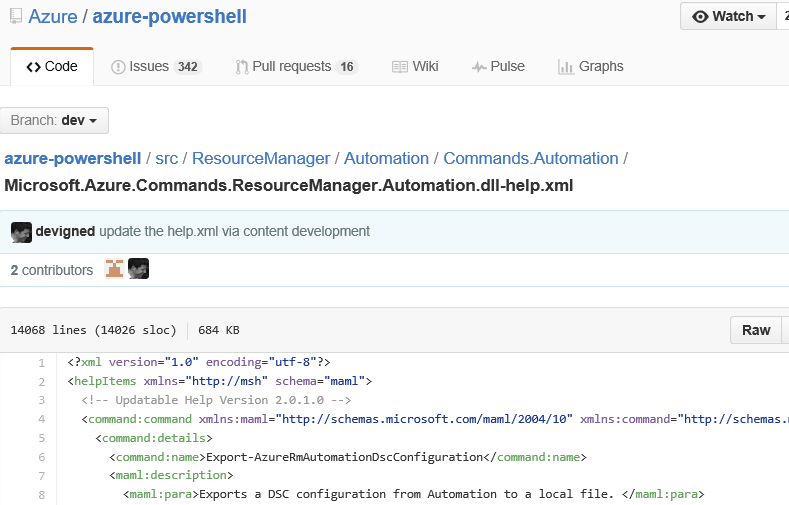
Make necessary edits to the file, then scroll down to the bottom of the content-editing pane to create a commit (i.e., to save the file and commit the changes to a specific branch of the repo).

Give your commit a name and add a short description underneath the name.

You’ll need to specify a branch location for the commit; if you’ve already created your working branch and are in it, you can use that. If you haven’t created a new branch yet, you can create one at the same time.



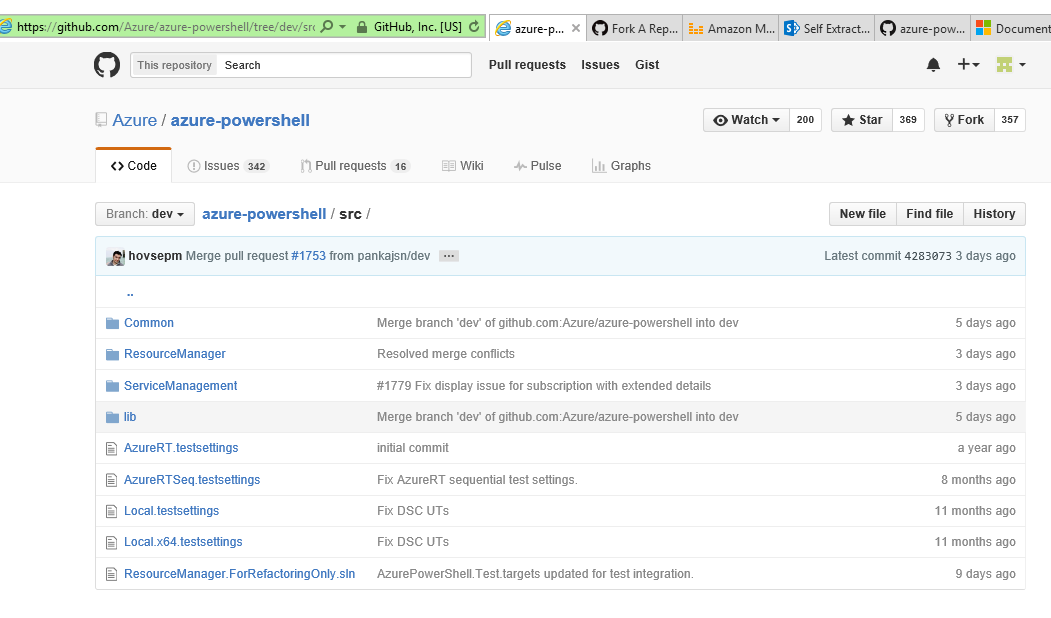
In all likelihood, you will be making changes to Help.xml files for various modules. For instance:



# Finding Help.xml files

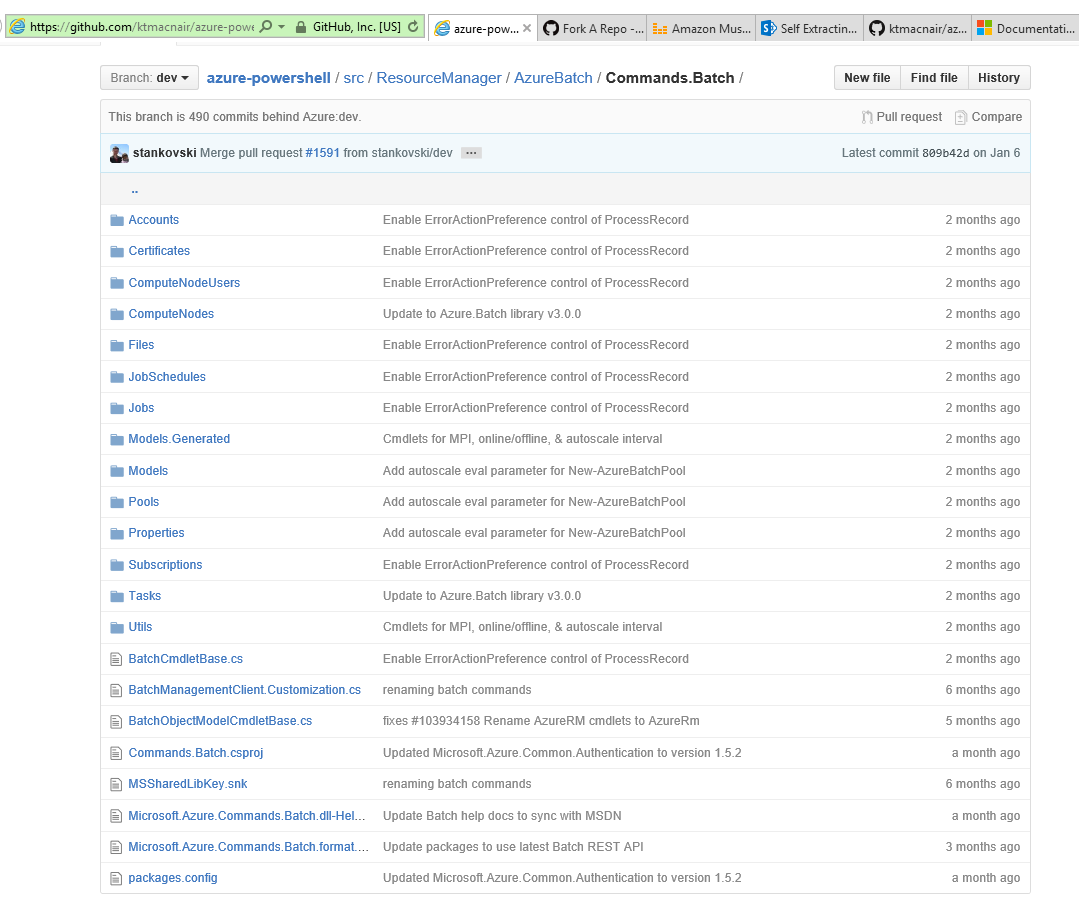
Source files for updatable Help.xml files are located in the **Src** folder in the azure-powershell repo; they are organized by module (just as the rest of the code is).

What does this look like in GitHub?

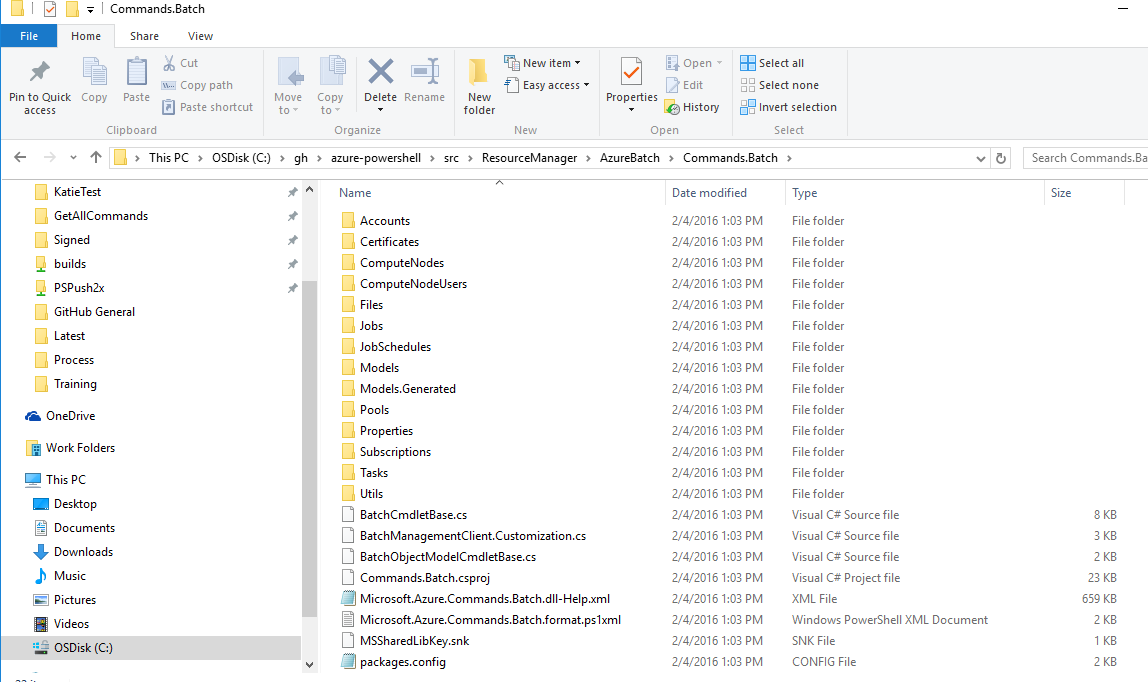


The help.xml file we’re working with in this example is for cmdlets in the RMAzureBatch module (Microsoft.Azure.Commands.Batch.dll-Help.xml).

Here’s what the directory structure for this module looks like in GitHub:

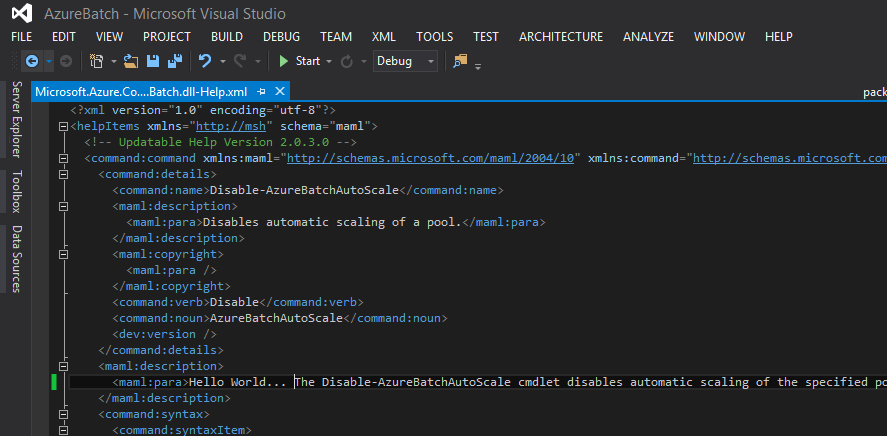


And here’s the corresponding file directory structure of the local copy (in File Explorer):

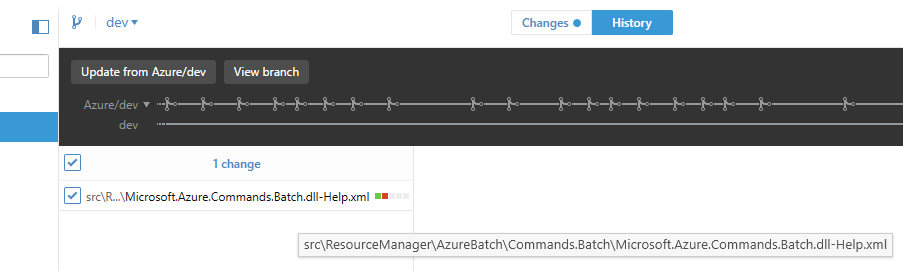


## Editing an individual file (Visual Studio)

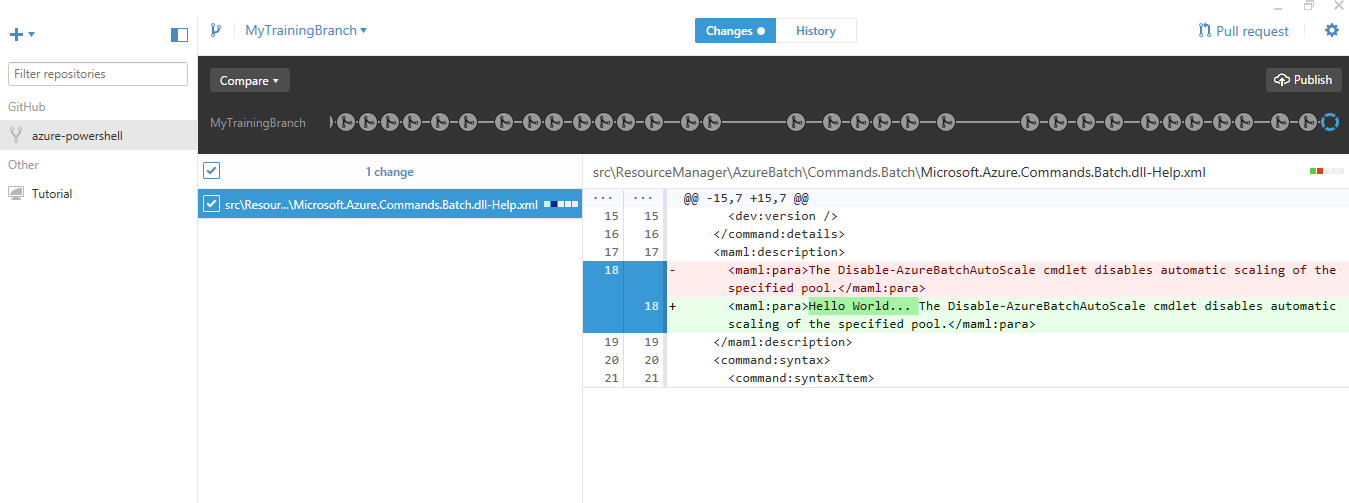
* From the repo directory on your c:\ drive, navigate to the folder you want and open the file in your favorite XML editor (such as Visual Studio).
* Make your change(s).



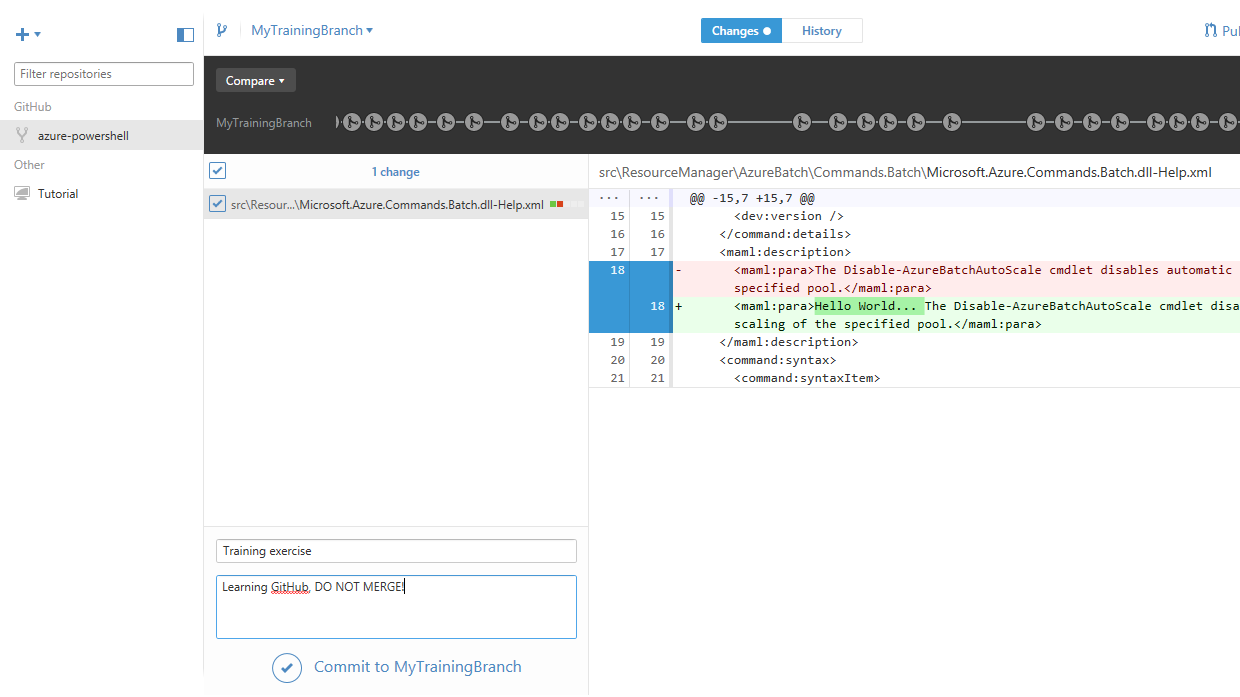
* Save the file.
* Open GitHub (desktop app) on your machine. With the **History** button highlighted, you’ll see a timeline and a list of the file(s) you edited.



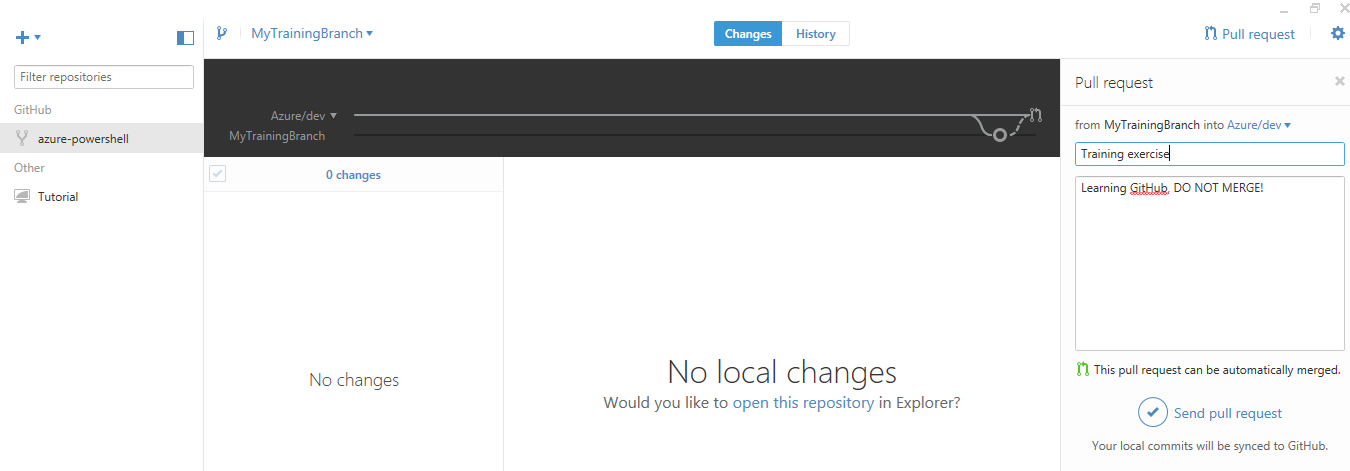
* With a specific file name selected, click the **Changes** button. Now you’ll see exactly what your edits were, line by line!



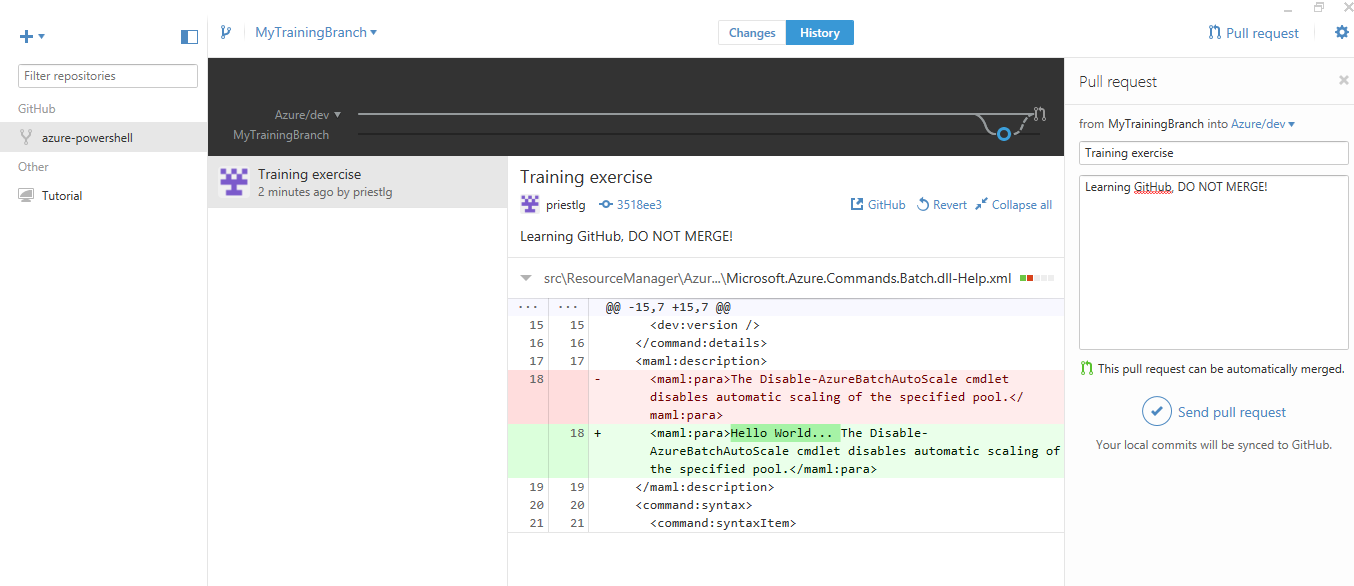
* Add a Summary (this will be sort of like a title for this change) and a Description (can be longer than the Summary, if you like). If you’re making edits for testing purposes (as in our example), add the words “DO NOT MERGE” in the Description field.



* Click the “Commit to <working branch name>” hotlink. Notice that in this example, we were working in a branch created for demo purposes, called MyTrainingBranch. This line will reflect the name of whatever branch you are working in—and you **must** be sure that you are working in the correct branch.
* Then click Pull Request to create a new request.

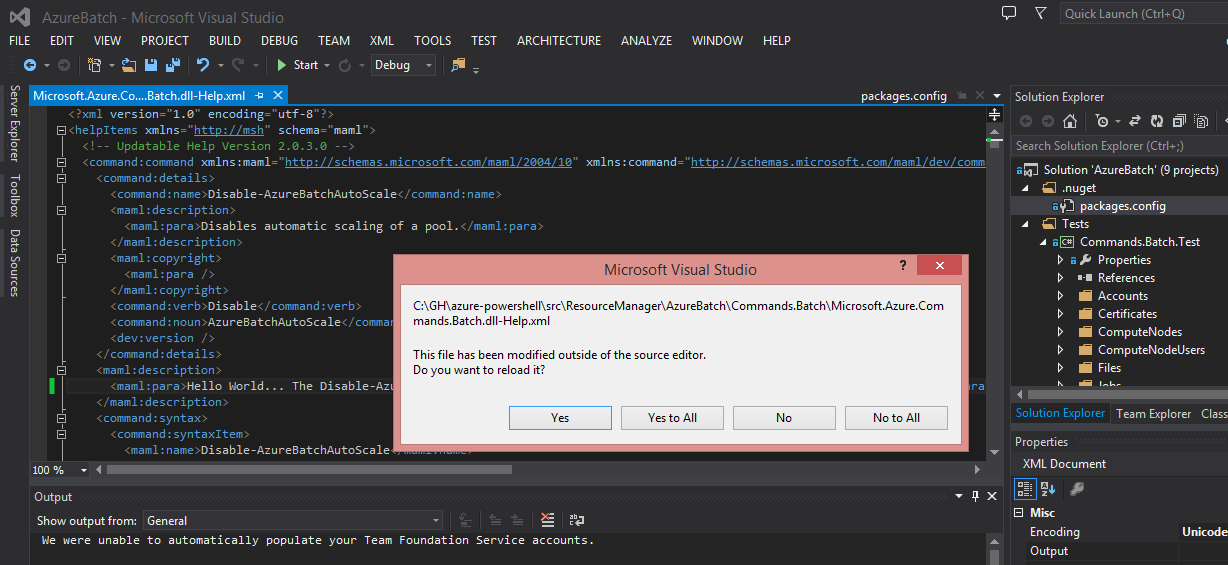


# More about pull requests



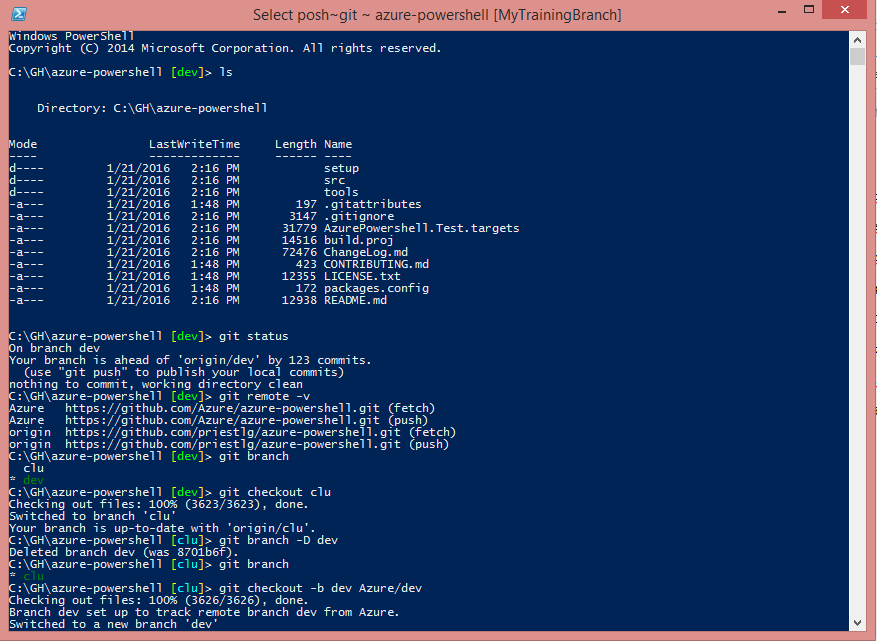
After entering a name and description for your pull request, you can click **Send pull request** and it will be synced to GitHub and thus put in the hands of the repo owners for them to decide whether to merge or not to merge.

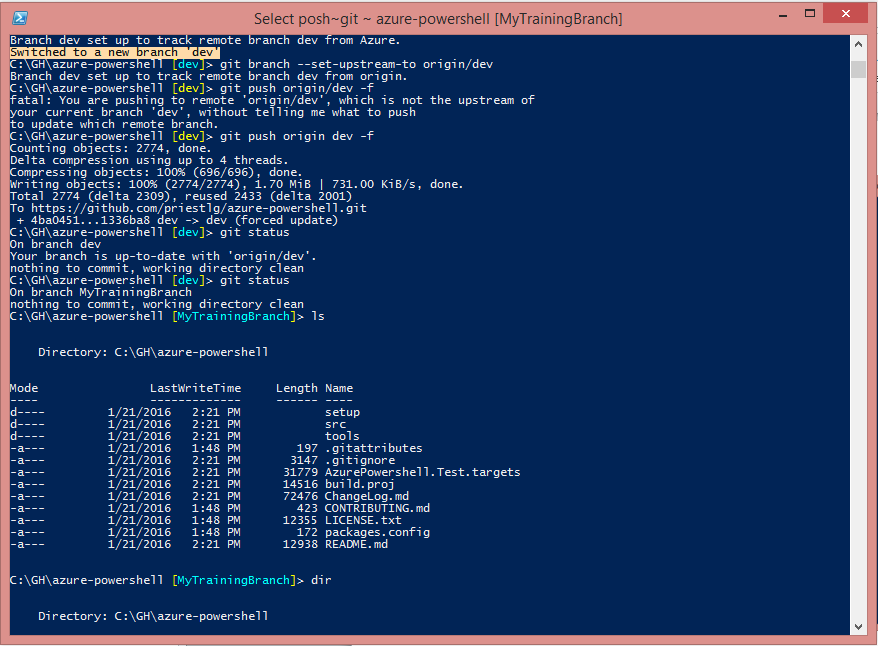
However, if you change your mind about this pull request and decide you don’t want to send it after all, you have the option of reverting your changes. That is, *after* you’ve created a pull request but *before* you’ve actually sent it, you can click **Revert** instead of clicking **Send pull request**. If you do so, the pull request will go away, and any files you edited in Visual Studio will revert back to their previous state.

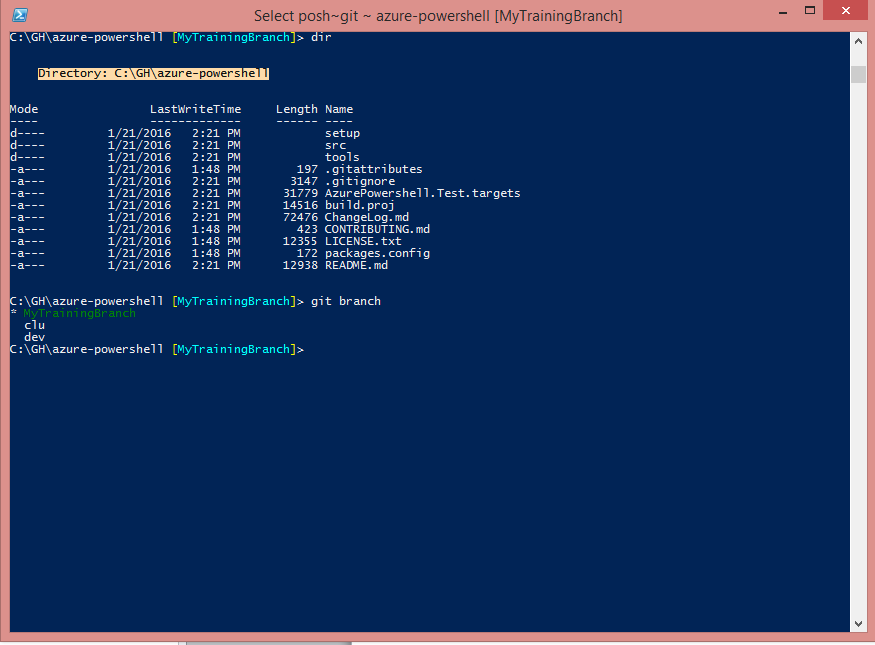


And if you’ve already submitted your pull request but decided you need to get it back, all is not lost. Go to the Appendix to [see what other magic GitHub has in store](#_More_magic:_reverting).

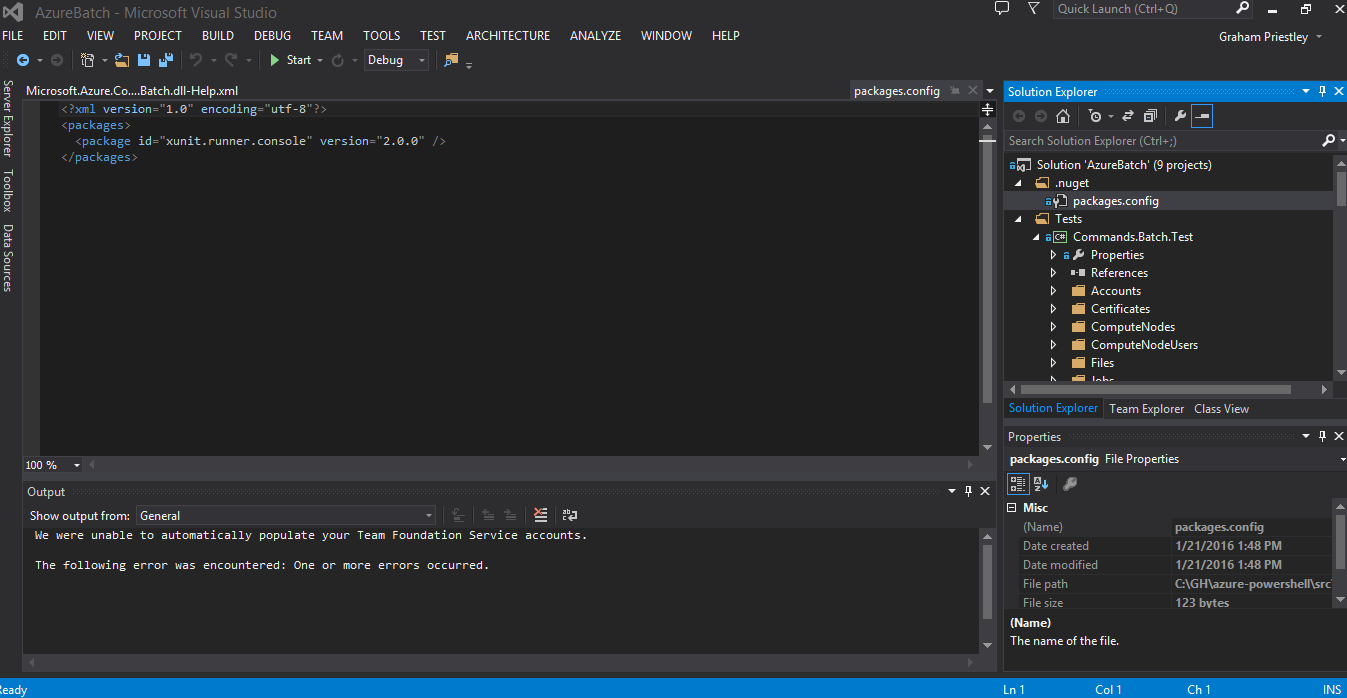
**Let’s see how this looks at the command line.**



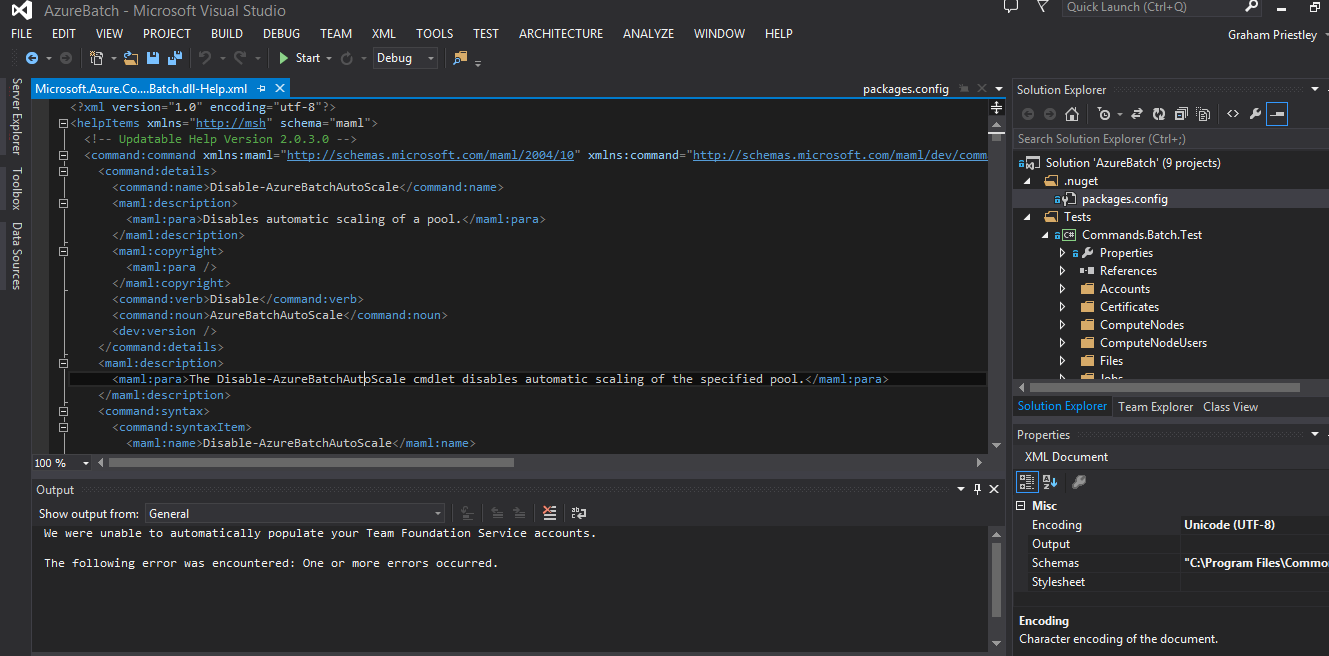




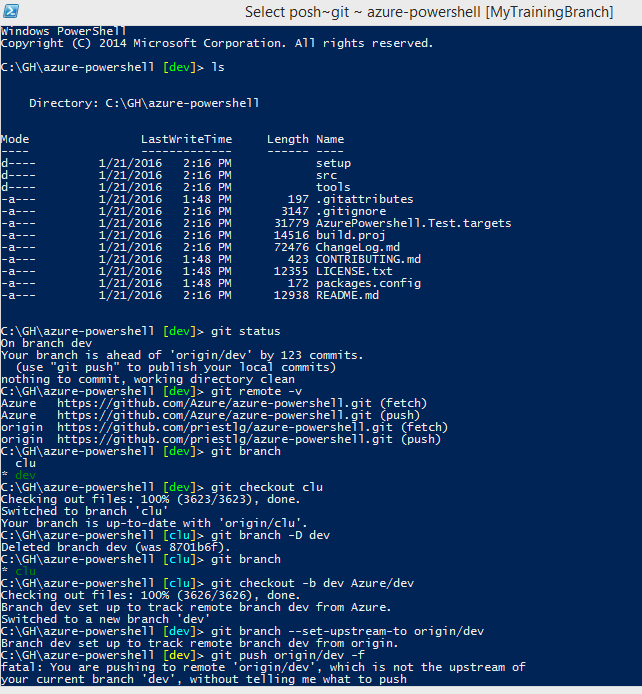
In Visual Studio:



Another file (Commands.Batch Help.xml)

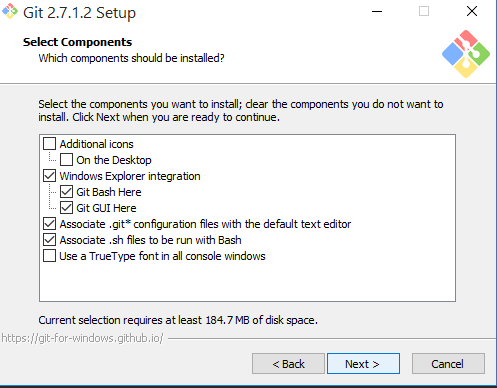
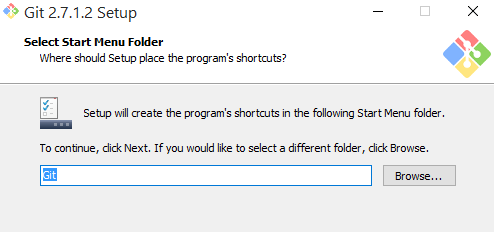
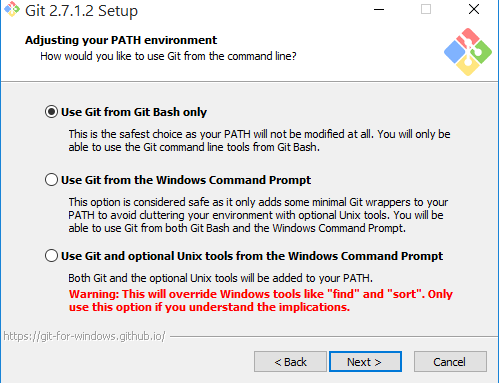
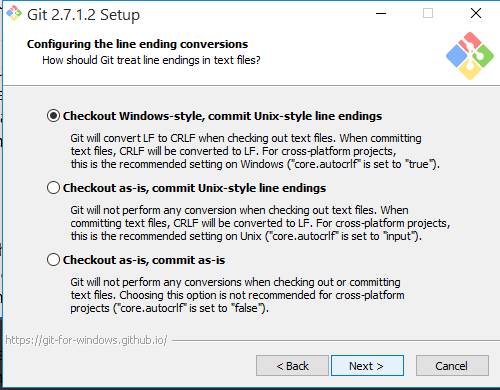
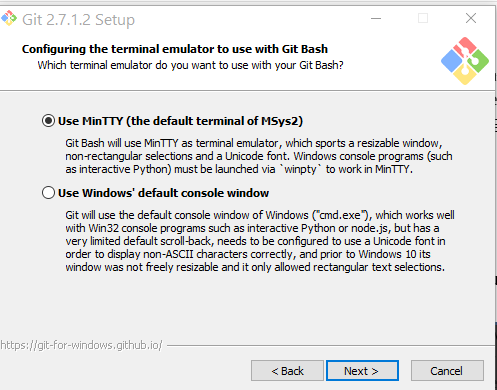
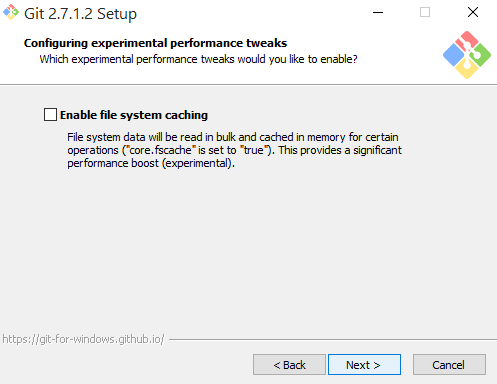
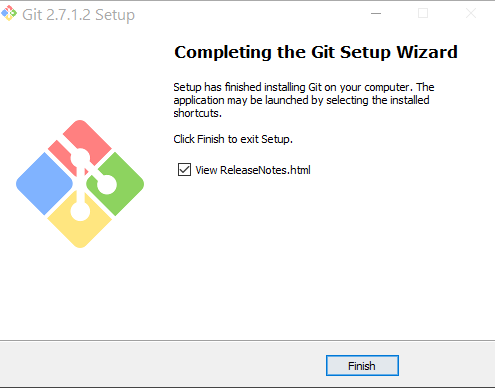


# Interacting with the Git Bash command line

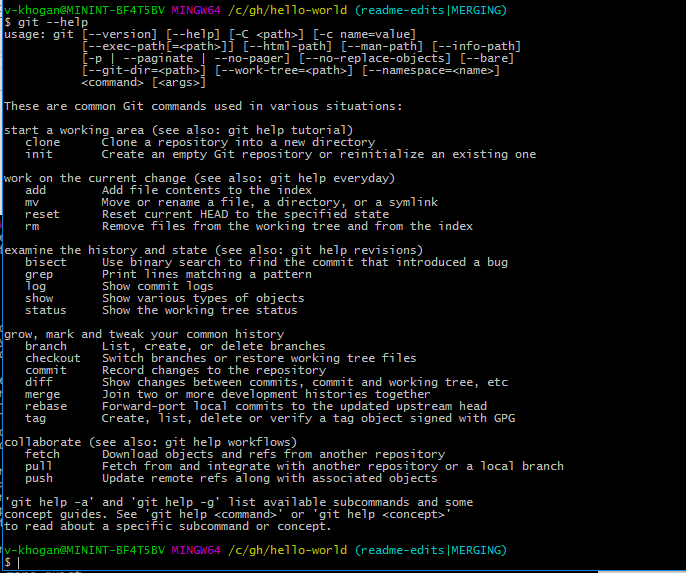


# Appendix

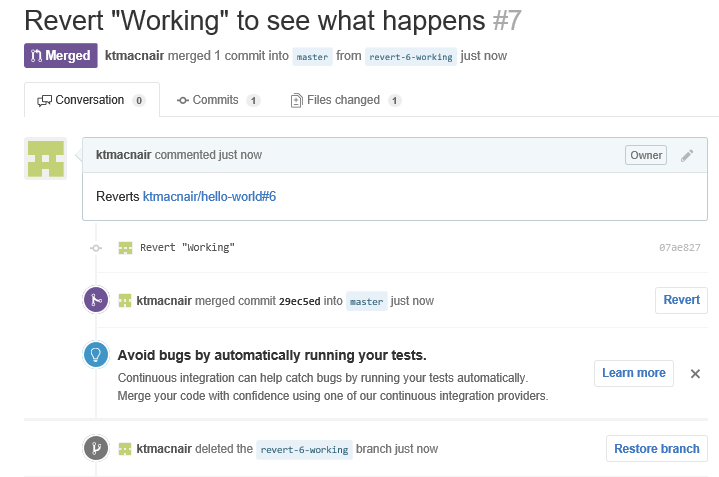
## Git install details

* 1. Components
     1. 
     2. 
     3. 
     4. 
     5. 
     6. 
     7. 

## Some Git commands for reference



## More magic: reverting changes already merged via pull request



I made several separate commits in my local clone, using a working branch. The last time I made an edit to local files in Visual Studio was around 7:05 p.m.

I submitted one pull request (which included all 4 of my commits), asking the system to pull my changes in to the master repo.

Back on GitHub, I merged the request and deleted the working branch.

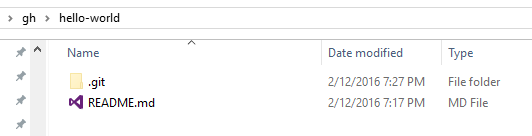
Changed my mind, decided to revert the merge, and clicked the **Revert** button.

The system created a new pull request to revert the changes.

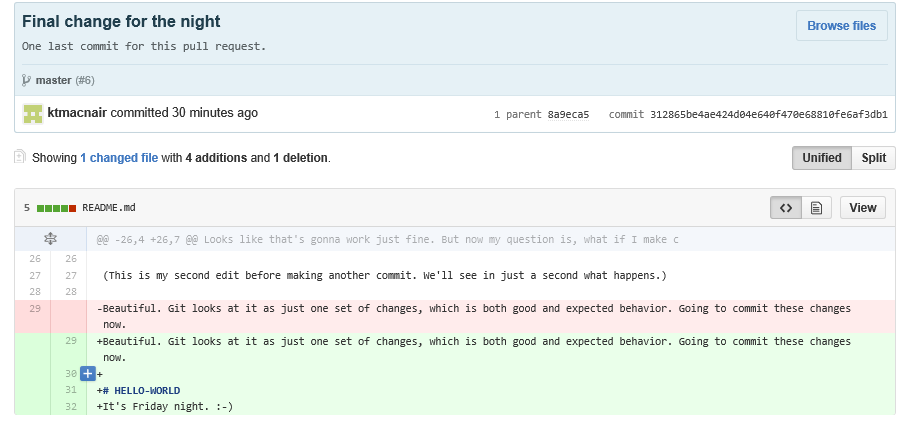
I merged *that* request, which got rid of my earlier changes and put things back to where they were before I merged them before.

Now comes the spooky part.

I went back to GitHub desktop app, it showed that I had no local changes, and I opened the repo in Explorer. And the file directory showed me that the last time my markdown file was changed was at 7:17 p.m.—*nearly 20 minutes after I last edited that file*.



All those changes I made are really, truly gone-gone-gone from the file. Crazy! But what, you say? I shouldn’t have reverted those changes? Never fear. Through more magic, there’s still a record of those changes on GitHub, I’d be able to re-create them. I’d have to go back through each individual commit from that pull request to make sure I got everything, but at least my work wouldn’t be lost.



Now I can delete the working branch from my local clone, and call it good.