Using Source Control Software with PowerShell by Trevor Sullivan

This chapter covers:

* When you should use source control with PowerShell
* Basic introduction to using source control
* Managing source control repositories from PowerShell

Windows PowerShell is a phenomenal tool for automation, but with great power comes great responsibility as well. Over time, you will find that your script repository will grow larger and larger, and you frequently will be modifying scripts to provide additional functionality, fix bugs, or make your code more readable. As your repository grows and morphs, it is important to maintain a history of changes to your script files. The benefits of keeping track of history are many, most importantly including the ability to revert to an old version of a script if you make a breaking modification.

In this chapter, I would like to describe the benefits of using source control and how to perform basic source control functions. It is my belief that you will begin to appreciate the benefits that using source control software can provide to you, and will seek to further understand the capabilities therein.

* 1. Requirements

For this chapter of PowerShell Deep Dive, you will want to have several pieces of software available to you:

* Microsoft Windows 7/8 system running Windows Management Framework 3.0
* Microsoft .NET Framework 4.0 (required for PowerShell v3)
* TortoiseHg – available from <http://tortoisehg.bitbucket.org>
* Mercurial (included with TortoiseHg)

As of this writing, the Release To Manufacturing (RTM) version of Windows PowerShell 3.0 is available for Windows 7. It is extremely stable, and offers an array (no pun intended) of new functionality over version 2.0. PowerShell version 2.0 is the version built into Windows 7 out of the box. To determine which version of PowerShell you are running, simply type the following command at the console:

$host.Version;

You can download the Windows Management Framework Core 3.0 release from this URL: <http://goo.gl/ixOkQ>.

* 1. When to use Source Control

As discussed in the introduction, source control should be used in any circumstance where having the ability to roll back, or review and audit changes, is crucial. In production environments, the smallest change can have a massive impact, and having the ability to rapidly revert to known-working code is vital.

Source control systems are great to keep track of small projects as well. Rather than having a single, large code repository full of random script code, it is advisable to subdivide your code into logical groupings. PowerShell modules are good candidates for self-contained Mercurial repositories, as they generally are designed to represent a cohesive block of functionality.

Another great use of source control is to integrate with a continuous integration system, which can build different versions of your software project, no matter where you’re at in the development cycle. Having this integration means that you can rapidly deploy any software version into a production environment, and avoid interrupting your development cycle to manually roll back code to a known working version.

* 1. Introduction to Mercurial

Mercurial is a very popular version control software (VCS) package in today’s time. Due to its decentralized nature, it is very portable software, meaning that a code repository can be synchronized with any other team member’s repository. With this in mind, it is not necessary to have a centralized server, although in a team environment, having a central server would still be desirable.

Like most other VCS packages, at the root level, Mercurial has a concept called a “repository.” A repository keeps track of changes to files that have been added to it. Thanks to the TortoiseHg graphical user interface (GUI) tools, you can easily create a new repository. In Mercurial parlance, this is known as repository “initialization.”

Once you have created a repository, you will want to “add” files to that repository, which instructs Mercurial to keep track of change to those files. When files are added, changed, or removed, those changes are detected during the next “commit” operation, and can be reviewed for errors prior to finalizing the commit. When you commit changes to a source code repository, you have the opportunity to add a “commit message” which helps yourself, and other developers, to identify what changes have occurred since the last commit operation.

As you develop your code going forward, you will continue the process of “adding” or “removing” files from change tracking, and when you make significant changes, you will regularly “commit” them back to the repository. Once you have reached a certain point, you may be ready to share your code with others. In this case, you can push or pull your code to or from a remote repository. Thanks to Mercurial’s decentralized nature, you are not forced to synchronize with any central repository, unless all of the developers on a project agree to it up front. Therefore, you can push/pull code changes with any replica of your repository. If you are working on an open source project, you might synchronize your local repository with a hosted service, such as Microsoft’s CodePlex open source website, Bitbucket.org, or any other source code hosting provider who supports Mercurial repositories.

* + 1. Command Line versus GUI

The TortoiseHg package includes both GUI tools and the Mercurial command line executable, called *hg.exe*. For nearly all operations either tool can be used, and it’s up to you to determine which one you are most comfortable with. It has been my experience that using the GUI tools can be much more convenient to view changes, insert commit messages, add & remove files, and so on, but there are many people who prefer to stay true to the command line tools.

One benefit of using the TortoiseHg GUI instead of the command line utility is that it will save your settings and credentials for a remote repository. This helps to avoid the necessity of typing out the remote repository URL, and your credentials, each time you want to synchronize (push / pull) your local repository with the remote one.

Using the command line over the GUI has its benefits as well, however. For example, you can use the command line to automate common tasks in Mercurial, and save yourself some precious typing time. We’ll explore more about how to use Mercurial from PowerShell later in this chapter. Most of the tasks you perform using the command line interface consist of a structure similar to:

hg.exe <Action> <ParametersForAction>

If you’re just getting started with Mercurial for the first time, you can always type either of these commands to help you learn the command line utility:

hg.exe help

or:

hg.exe help <action>

The “actions” you can perform are very user friendly, and easy to remember. Additionally, many of the actions have aliases, making them even easier to type when you’re deep in the middle of a coding session.

I believe that you will find both the command line interface and graphical user interface (GUI) to be beneficial tools as you get started with Mercurial.

* 1. Common Source Control Operations

During your development workflow, you will interact with a number of core functions in any version control software. In this section, we will go over some of the most common source control operations that you should become familiar with in Mercurial.

* + 1. Initializing a Repository

As discussed in the previous section, the first step to using Mercurial is to “initialize” a code repository. Let us assume that we will be creating a PowerShell module under the path: c:\code\Modules\ConfigMgr2012. You can perform the repository initialization using the command line tool as follows:

Listing 1.1 Initializing a Mercurial repository

# Set the working directory (repository root)

Set-Location -Path c:\code\Modules\ConfigMgr2012;

# Initialize the repository

hg.exe init;

Below is a figure demonstrating the TortoiseHg UI control to initialize a repository.

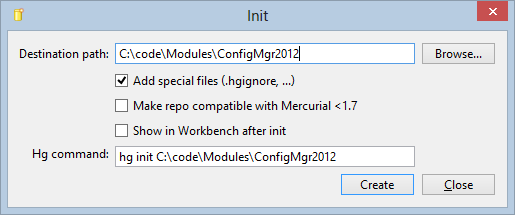


Figure 1-1 The TortoiseHg repository initialization screen

Once the code repository has been created, you’ll notice that a .hg folder gets created in the root, along with a .hgignore file. These files are used by Mercurial to keep track of the repository, and should not be modified by an end user under normal circumstances.

* + 1. Adding Files

Once you’ve initialized a code repository, you’ll want to add some files to the repository, so that Mercurial keeps track of changes to them. Again, you can use the command line or GUI for this purpose. Using hg.exe, you would run the command below.

Listing 1.2 Adding Files to a Mercurial Repository

# Set the working directory

Set-Location -Path c:\code\Modules\ConfigMgr2012;

# Create a new, empty script file

New-Item –ItemType File –Name NewScriptFile.ps1;

# Add the file to the repository

hg.exe add NewScriptFile.ps1;

Below is a screenshot of what the TortoiseHg GUI tool looks like to add files to a repository. You can simply uncheck the files that you do not want to add to the source control repository, and leave checked the ones that you want added.

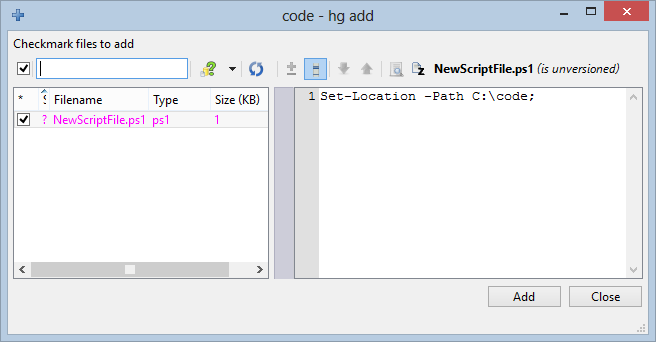


Figure 1-2 The TortoiseHg add files screen

**Note**

Keep in mind that adding files does not create any changes to the repository. In order to actually make the newly added files become part of the repository, you must commit them, which will be demonstrated in the next section.

* + 1. Committing a new Changeset

Once you have added files to your repository, you will want to commit the changes to the repository. Performing this action creates what is known as a Mercurial “changeset.” A changeset represents a series of changes that have been performed since the last changeset was committed.

In order to commit a new changeset using the command line, simply run the command similar to the following:

Listing 1.3 Committing a new Mercurial changeset

# Set the working directory to the repository

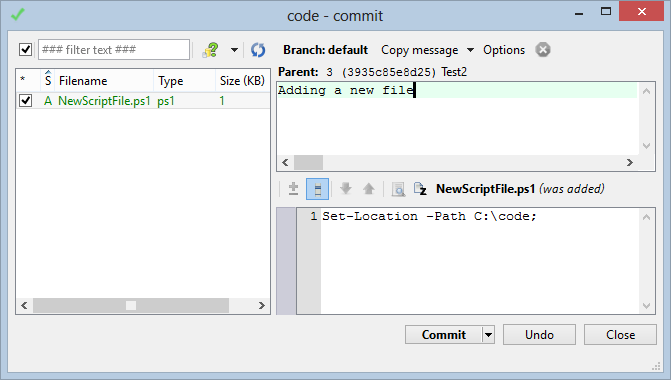
Set-Location –Path c:\code\Modules\ConfigMgr2012;

# Commit the changeset

hg.exe commit –m "Commit message" –u "Trevor Sullivan ([pcgeek86@gmail.com](mailto:pcgeek86@gmail.com))"

The Commit dialog in TortoiseHg looks similar to the screenshot below.

Figure 1-3 The TortoiseHg commit dialog



* + 1. Removing Files

Similar to the add action is the remove action. The remove action simply removes a file from change tracking in the Mercurial repository that you are currently operating on (the current working directory). To remove files from change tracking, simply use this command:

# Set the working directory (repository folder)

Set-Location -Path c:\code\Modules\ConfigMgr2012;

# Remove a file from the repository

hg.exe remove NewScriptFile.ps1;

Below is a screenshot that shows what the TortoiseHg remove dialog looks like.

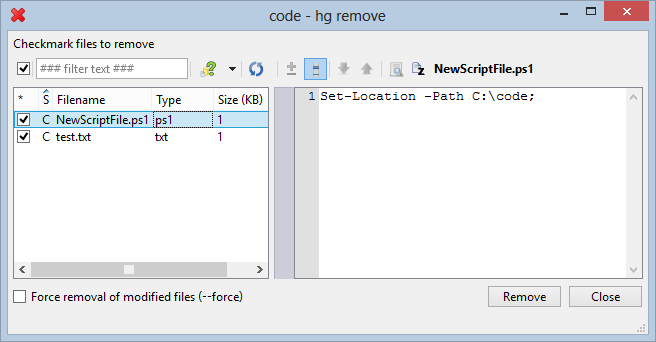


Figure 1-4 The TortoiseHg remove file dialog

* 1. Using Mercurial from PowerShell

As an end user of Mercurial, you can automate many of its functions from PowerShell. This is easy to do, since the product was built around the command line. There is unfortunately not a .NET library for Mercurial available. You can, however, easily call the command line utility, which we have discussed at some length already. Automating Mercurial tasks can help to streamline your workflow as a software developer, and can also be incorporated into build scripts that run on a continuous integration service such as Cruise Control .NET, JetBrains TeamCity, or similar.

* + 1. Script to Initialize a Repository

Here is an example of how to wrap a Mercurial repository initialization command in PowerShell code:

# Define the New-MercurialRepository function

function New-MercurialRepository {

[CmdletBinding()]

param (

[Parameter(Mandatory = $true)] [string] $Path

)

# Ensure that the folder path exists first

if (Test-Path -Path $Path -PathType Container) {

# Initialize the code repository

Start-Process -FilePath hg.exe -ArgumentList ('init "{0}"' -f $Path);

}

}

# Test the function

New-MercurialRepository -Path c:\code\Modules\ConfigMgr2012;

* + 1. Script to Commit a Changeset

Here is another example of a PowerShell function, which demonstrates how to perform a commit operation using a PowerShell script cmdlet. There are parameters to pass in a commit message, the path to the Mercurial repository, and an optional username. By default it uses the username that you are logged into a Windows session with.

function Commit-MercurialChangeset {

[CmdletBinding()]

param (

[Parameter(Mandatory = $true, HelpMessage = 'Please enter a commit message.')]

[string] $Message

, [Parameter(Mandatory = $true, HelpMessage = 'Please enter the path to the repository.')]

[string] $Path

, [Parameter(Mandatory = $false)] [string] $Username = $env:USERNAME

)

# Get a reference to the .hg folder, to ensure that it is a valid Mercurial repository folder

$Path = '{0}\.hg' -f $Path;

# Ensure that the path to the .hg folder exists

if (Test-Path -Path $Path -PathType Container) {

# Commit the changeset

$Arguments = 'commit -u "{1}" -m "{0}"' -f $Message, $Username;

Write-Host -Object $Arguments;

$Process = Start-Process -FilePath hg.exe -ArgumentList $Arguments `

-NoNewWindow -PassThru -Wait -WorkingDirectory $Path;

# Write the process exit code to the PowerShell pipeline

$Process.ExitCode;

}

else {

# Throw an exception if the path does not exist

throw ('Path ({0}) does not exist!' -f $Path);

}

}

Commit-MercurialChangeset –Message 'Test commit message.' -Path c:\code\Modules\ConfigMgr2012 -Username 'Trevor Sullivan';

* 1. Summary

In this chapter, we have discussed when to use a version control system (VCS) with PowerShell projects, and some basics of how to get started with Mercurial as a VCS. Additionally, we have examined how to call Mercurial commands from PowerShell using wrapper functions, and looked at the two main methods of interfacing with Mercurial repositories from the GUI and command line. I trust that this chapter has given you some new ideas, and can help take your coding workflow to the next level.

* 1. Bio

Trevor is a pursuer of all types of knowledge, and is particularly passionate about Information Technology and technical communities. He has been working primarily with Microsoft solutions since 2004, and is entirely self-taught. Trevor has received public recognition from Microsoft for his contributions to the Windows PowerShell community, and he has also written several guest blogs for the Microsoft “Hey, Scripting Guy!” blog and PowerShell Magazine. One of his more notable achievements is the release of the PowerEvents module for PowerShell, which is available on Microsoft’s CodePlex open source website. Trevor also enjoys giving presentations, and has presented on the well-known topic of PowerShell & WMI, in addition to two guest appearances on the weekly PowerScripting podcast.