Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

PowerShell for Developers with Admin Tasks

Repetitive actions, such as starting up remove services, reloading apps, and kicking off test suites, can be automated easily using PowerShell — Microsoft's highly productive shell language.

March 05, 2013
URL:http://www.drdobbs.com/windows/powershell-for-developers-with-admin-tas/240150040

PowerShell is the best unkept secret in the Windows developer world. Although it is highly lauded by its users, it has not gained a lot of traction with the Windows developer crowd. It is all the rage with the Windows admin crowd, however. Powershell is tightly integrated with the operating system, .NET framework, and most Microsoft server products (SQL Server, IIS, TFS, etc).

In this article, I discuss why PowerShell should be used more by developers and I show you some useful stuff you can do with PowerShell. Finally, I'll go over my PowerShell profile and show how I use PowerShell every day to make my life easier.

PowerShell: The Conspiracy Theory

Historically, Windows had the worst shell ever: command.com, cmd.exe coupled with the worst terminal Window ever. That had two important outcomes:

- A strong aversion to working from the command-line on Microsoft operating systems
- Excellent GUI tools to perform tasks

PowerShell 1.0 became available in 2006 for Windows XP. However, it became ubiquitous (that is, installed by default) only in Windows 7 and Windows Server 2008 R2. During the five year delay, the trend toward mobile apps, Web applications, and the cloud emerged and the attention of the developer crowd shifted away from PowerShell.

But this neglect is unwarranted. Let me quickly show you why before diving into the meat of the article. Here are some cool one-liners.

Generate the integer between 5 and 20 (inclusive):

```
5..20
```

Pick a random number between 1 and 100:

```
1..100 | Get-Random
```

Play a video:

```
(New-object –COM WMPlayer.OCX).openPlayer("Your video")
```

Calculate total size of current directory in MB:

```
"{0:N2}" -f ((Get-ChildItem . -Recurse | Measure-Object -Property Length -Sum).Sum / 1MB)
```

Progress bar while computing total size of directory (OK, not a one liner, but very cool)

```
$files = Get-ChildItem . -Recurse
$total = 0
For ($i = 1; $i -le $files.Count-1; $i++)
{
    Write-Progress -Activity "Calculating total size..." -status $files[$i].Name -percentComplete ($i / $files.Count * 100)
    $total += $files[$i].Length
    Start-Sleep -Milliseconds 50
}

Write-Host "Total size: $($total / 1MB)"
#Write-Host "Total size: {0:N2}MB" -f ($total / 1MB)
```

Introductory PowerShell

PowerShell is object-based. UNIX (Linux, Mac OS X) shells are text-based. The idea is that using the shell pipeline on the UNIX variants, you can glue together any number of small tools that eat and spit text lines to create arbitrarily complex work-flows. This works and gives you infinite flexibility. But, at a price. The price is that you often have to perform some elaborate text processing between commands in the pipeline to adapt the output of one command to the input expected by the next one. This can lead to cumbersome and brittle pipelines that are both wonderfully concise, and yet completely unintelligible for the uninitiated.

PowerShell is different: Commands eat and spit out .NET objects. This approach is better suited to the Windows environment where many of the system

facilities are exposed through APIs and object models (Registry, COM, .NET, ADSI, WMI, Certificate store). This aspect makes it easy to user PowerShell for GUI front ends. For example, Figure 1 shows a form with a check box, date picker and OK button that was easy to assemble using free tools such as Sapien's PrimalForms GUI generator. *Dr. Dobb's* has previously discussed the ins and outs of writing GUIs in Powershell by hand.

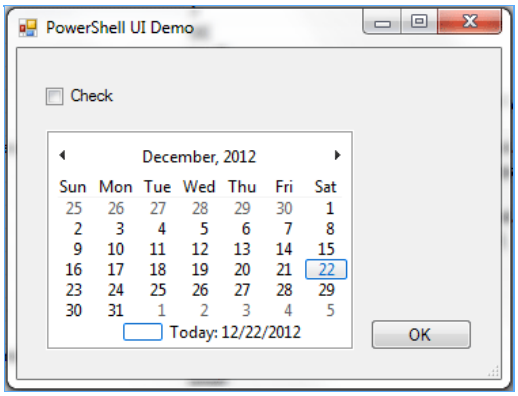

Figure 1: A GUI applet written in Powershell.

The "How" of PowerShell

PowerShell was designed for Windows system administrators and power users. It was optimized for interactive use on the command-line first, and for scripting second. If you have some experience with UNIX shells, PowerShell will sometimes feel very familiar and sometimes quite alien. The reason is that the UNIX-based operating systems and Windows evolved along very different trajectories. UNIX shells were optimized to manage a text-based ecosystem in which everything is or is made to appear as a text file. PowerShell was designed to manage the API-based Windows ecosystem.

The PowerShell team did a good job evaluating the needs of their target audience. They also looked at how sysadmins operate in the UNIX environment, then compared this environment to the Windows environment, distilled all this input, and came up with an architecture and design for a consistent, flexible, and extensible command execution engine and shell.

Before we get started, let's do a quick refresher on Powershell: `$variable` describes a variable name. Most commands rely on Cmdlets, which are named objects such as `Set-ExecutionPolicy`. Many of these commands have multiple built-in aliases. For example, the `Get-ChildItem` command has three official aliases: `dir`, `ls`, and `gci`. The `dir` alias is there to ease the transition for old Dos/Windows hands. The `ls` alias is to placate the UNIX crowd, and `gci` is there because if you have no previous experience with other shells, then it's the "PowerShell-y" way to define aliases (first letter of every word in the command). This is a very thoughtful, considerate, and humane approach that, when added to the fact that PowerShell is case insensitive, makes reading other people's code (or even your own code from a few months ago) challenging.

Finally, a word about operators: The comparison operators in PowerShell use mnemonics like `-lt` (less than), `-gt` (greater than), and `-eq` (equal to), instead of the familiar symbols = (or ==), !=, <, and >. The reason is a tip of the hat to ancient tradition. In particular, the long-time shell output redirection symbol is `'>'`. PowerShell designers wanted to keep this tradition, so they created the set of comparison operators `-lt, -eq, -neq, -le, -ge`, which takes a little getting used to.

Starting Up

Presuming you have PowerShell installed, you need to set the security policy to allow the running of scripts. You do this by typing `Set-ExecutionPolicy RemoteSigned`. This will allow you to launch PowerShell scripts locally. PowerShell, being an administrator's tool first and foremost, is configured by default to not allow running scripts as a security measure. This is a sensible default on remote machines, but you will definitely want to run scripts on your machine.

Next, set your profile. The PowerShell profile is where you can collect all the stuff you use commonly, so it's available in every session. There could be several profile files with the more specific files overriding more general ones. To create a PowerShell profile that applies to all users type:

```
new-item -path $env:windir\System32\WindowsPowerShell\v1.0\PowerShell_profile.ps1
-itemtype file -force
```

The path to the current profile is always in the `$profile` global variable. You can verify it exists by typing: `Test-Path $profile`.

Once you have a profile, you can edit the file and add your stuff. Remember to execute the profile after each change, so your changes will be available in the current `. $profile` session.

Don't try to learn PowerShell in a methodological manner. Play with it. Even playing may feel strange at first. The best way to go is to look for cool stuff other people have done and just stick it in your profile and use it.

Once you feel comfortable in the PowerShell environment, start expanding your horizons. Check your profile and look at all the stuff you stole from other people. Read a little about the commands they use and their parameters. Try to understand the tricks and idioms they use. PowerShell has an excellent help system. Explore it. `Get-Help` (or just the the alias `help`) is your friend. If it's too much work, you can also just type a Cmdlet name followed by `-?` to get help on it. For example, `Get-Service -?`.

If you want more extensive help with explanations of each parameter and lots of examples, add the `-detailed` switch:

```
help Get-Service -detailed
```

Beyond Cmdlet help, PowerShell also provides meta help about different topics. To find out which topics are available, type:

```
help about_
```

and PowerShell will display all the topics. Then you can just pick one, such as `'History'` and type: `help about_history`.

A Developer's PowerShell Profile

I'm not a system administrator and I have a strong aversion to long pipelines of commands with convoluted parameters. If you ask me how to find a service that was installed at-most three days ago and its name is longer than 10 characters, I'll have to look it up and do some trial and error until I get it right. I use PowerShell just like I use Bash on Mac OS X and Linux. I define a bunch of functions and aliases in my profile and I rely on them for my command-line needs. I rarely type commands with parameters. This saves me some typing, but more importantly, it saves me from the mental burden of keeping most of the complexity in my head.

Navigation

I almost always work in deeply nested folder hierarchies. I hate typing long path names on the command-line, and even tab completion can take too long if you have many sibling sub-directories — all with the same prefix. I also regularly visit a relatively small set of locations. So my standard practice, if I want to run a command that accepts a path, is to navigate to this location and run `"<cmd> ."`. To be able to navigate quickly to various locations, I define little functions that consist of a series of `'cd'` statements:

```
function cdr { cd 'c:\r' }
function cdp { cdr; cd ProjectCool }
function cdb { cdr; cd Branches }
function cdt { cdr; cd Testing }
```

Locations and programs

There are several common locations I use multiple times in the profile or full paths to various programs:

```
$program_files = 'C:\Program Files (x86)\'
$editor = $program_files + 'Notepad++\notepad++.exe'
$rabbitmq_sbin = $program_files + "RabbitMQ Server\rabbitmq_server-2.8.2\sbin\"
$rabbitmq_ctl = $rabbitmq_sbin + 'rabbitmqctl.bat'
$rabbitmq_server = $rabbitmq_sbin + 'rabbitmq-server.bat'
$rabbitmq_service = $rabbitmq_sbin + 'rabbitmq-service.bat'
$dot_net_framework = 'C:\Windows\Microsoft.NET\Framework\v4.0.30319\'
$install_util = $dot_net_framework + "installutil.exe"
$root_dir = 'c:\root'
```

Path

There are several locations I add to the path. For these locations, I define short names because they'll be used through the `PATH` and not directly in the command-line, so I don't need significant names.

```
$pd = 'C:\Python27\' # Python dir
$py3d = 'C:\Python32\' # Python dir
$ps = $pd + 'Scripts' # Python scripts
$ipd = $program_files + 'IronPython 2.7.1\' # IronPython dir
$ptd = 'C:\PuTTY\'
$td = 'C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE'
$components = $ipd, $pd, $ps, $ptd, $td, $env:PATH
$env:PATH = [string]::join(';', $components)
```

Aliases/Functions

I accumulated several useful functions I use frequently (which are sometimes used by other functions in the profile). I use descriptive names for the functions, and if I plan to use them interactively, I also define short aliases. For example, the `ElevateProcess()` function enables me to run a program with Administrator privileges. I added the alias `sudo` for quick interactive use:

```
function ElevateProcess
{
    $file, [string]$arguments = $args;
    $psi = new-object System.Diagnostics.ProcessStartInfo $file;
    $psi.Arguments = $arguments;
    $psi.Verb = "runas";
    $psi.WorkingDirectory = get-location;
    [System.Diagnostics.Process]::Start($psi);
}

set-alias sudo ElevateProcess
```

Another example is a little function to find all the aliases to a command. It is very simple and just uses an existing Cmdlet with a switch, but it's easier for me to remember via an alias to it:

```
function GetAliasNames([string]$command)
{
  Get-Alias -Definition $command
}

set-alias gan GetAliasNames
```

I also use lots of small functions that really work as aliases with arguments. For example, I often want to kill all the instances of a certain process by name. This can be done pretty easily in PowerShell with: `Stop-Process -ProcessName <process name>`. You can use the `kill` alias instead of `Stop-Process`, but it's still too verbose for me. So I defined a couple of functions to kill Erlang (when working with local RabbitMQ server) and Visual Studio (very useful when installing plugins or when it hangs):

```
function kvs { Stop-Process -ProcessName devenv }
function kerl { Stop-Process -ProcessName erl }
```

Quick Command Launchers

Another time-saver is quick program launchers. I use short functions to launch various programs (sometimes with arguments). Here are some examples:

```
function ipy
{
    $iron_python = $ipd + 'ipy64.exe'
    & $iron_python $args[0] $args[1] $args[2]
}

function py3
{
    $python3 = $py3d + 'python.exe'
    & $python3 $args[0] $args[1] $args[2]
}

function mongo
{
    & "C:\Users\Gigi\Downloads\mongodb-win32-2.0.4\bin\mongo.exe"
    $args[0] $args[1] $args[2] $args[3] $args[4]
}
```

Prompt

The prompt is important in interactive use. You can put pretty much anything you want there. I like to know where I am, so the current directory is a no-brainer. I don't care much about the host because this information can be displayed in the console window title (it doesn't change from one command to the next normally). I also like to know the time a command completed because I often run several lengthy commands in multiple console windows over and over (for example, for deployment after a code change) and I want to be sure of the last time I ran a particular command. There is nothing remarkable about these requirements. The only twist I added to my prompt is that I don't want to display the full path. As a developer, I often work with multiple workspaces with deep directory structures. If I display the full path, it takes a significant part of the line and doesn't leave enough space to type the command itself.

Displaying just the last part of the path doesn't work for me either because I often work on multiple copies of the same project from different branches in parallel. The solution I came up with is to compress most of the long prefix of common trees into a unique mnemonic abbreviation. For example, when I work in the Testing tree, I collapse the path `C:\root\ProjectCool\Testing` to `[X]`. A path like `C:\root\ProjectCool\Testing\Tools` shows up as `[X]\Tools` and is very easy to distinguish from the same project in the `ProjectCool_featureX` branch, whose path is `C:\root\ProjectCool_FeatureX\Testing\Tools` but shows up as `[x2]\Testing\Tools\`.

The trick is to always have good abbreviations for the trees and branches I currently work on. To set the prompt, you need to have a `Prompt()` function in your profile that returns a string. In the code below, I keep a hash of path prefixes to abbreviations and I check whether the current working directory matches any of the prefixes — and add the current time, too.

```
function Prompt
{
    $hash = @{
      "c:\users\gigi" = "~"; }
     “c:\root\ProjectCool” = “[X]”;
     “c:\root\ProjectCool_FeatureX” = “[X2]”;


    $result = $PWD.Path
    $fullPath = $PWD.Path.ToLower()
    foreach ($name in $hash.Keys)
    {
        if ($fullPath.StartsWith($name))
        {
            $path = $PWD.Path.Substring($name.Length);
            if ($path[0] -eq "\")
            {
                $path = $path.Substring(1)
            }
            $result = $hash[$name] + " " + $path;
        }
    }
    (get-date -f HH:MM:ss) + " " + $result + "> "
}
}
```

Text Searching (grepping)

PowerShell is all about objects, but it doesn't neglect text processing. The equivalent of the UNIX/Linux workhorse `grep` is `Select-String`. For some reason, the PowerShell designers didn't alias it to `grep`, so it is not as discoverable. I defined my own alias to it: `set-alias grep Select-String`.

When working with objects, the PowerShell equivalent to `Select-String/grep` is `Select-Object`. This is normally more powerful and robust. I defined a couple of functions that help me `grep` over a bunch of objects by converting each object to its text representation, then using `Select-String` on the result. I

aliased this concoction as `sgrep`:

```
# Convert a collection object to a collection of their textual representation
function Objects2Strings
{
    $input | ForEach-Object $_ { $_.ToString() }
}

# Convert input to strings (good for searching in filenames without digging into file contents)
function GrepFromPipe
{
    $input | Objects2Strings | select-string -Pattern $args[0]
}

set-alias sgrep GrepFromPipe
```

Conclusion

PowerShell was designed with the administrator in mind, but it can be amazingly useful in the hands of developers, too. If you find yourself frequently performing tedious and error-prone tasks, consider using PowerShell to make your life easier. PowerShell is powerful, well-designed, and has a good ecosystem and community. It will also be around for a long time given its adoption, ongoing development, and integration with Microsoft products.

Gigi Sayfan specializes in cross-platform object-oriented programming in C/C++/ C#/Python/Java with emphasis on large-scale distributed systems, and is a long-time contributor to Dr. Dobb's.

Update: This article was updated (that is, shortened) on 28 May 2013 due to possible legal action from Roblox for mention of one of their tools in passing.