powershell_presentation_notes.md

# Why EnergyCAP Developers should love MS PowerShell)

Author: Kurt Kroeker

Last Updated: 3/11/2019

## Introduction

Kurt Kroeker, Sr. Software Developer at EnergyCAP, Inc. for 10 years. I've built A/P, interval data, and bill reformatters, EDI scripts, SQL scripts, workflow scripts for EnergyCAP. So you can tell that I mostly work on the back-end side EnergyCAP software, which is now the place where I am.

Much more comfortable being in front of people when I have a violin, guitar, mandolin, whatever in my hands.

"When all you have is a hammer, every problem looks like a nail." PowerShell as my Swiss Army Knife, and you just add your own blades over time.

Informal poll: Raise hand if you are a...

- Rank beginner to Powershell
- Pretty decent with it
- You're a PowerShell ninja and want to come up and do this talk for me

That was my last attempt to get out of this presentation. :)

## Agenda

- Why should I care about PowerShell?
- What is PowerShell?
- Tips working with PowerShell
- Resources
- Questions?

## My PowerShell Background

Back in the day (2012), Miami-Dade County (MDC) needed to put together an automated workflow for data flowing in and out of EnergyCAP. This workflow included supporting 2 environments (Test and Production), importing EDI bills, executing 4 AP interfaces, setting bill flags, calling Installed Client's external task to run audits, and (eventually) calling ECO v3's APIs (V7 didn't exist yet in any form). Logging was required for all of these functions, and they required file manipulation, dealing with JSON data, and even sending emails with summaries of the log.

Chris Houdeshell recommended PowerShell as a solution to me and helped me get the project off and running, and it grew into the monstrosity now in Josh Condo's capable hands.

Over the years, I've found more and more uses for PowerShell when automating repetitive tasks in my development work. Some examples are adding a library of time-saving functions to help me get around my repositories and EnergyCAP resources, checking for obvious mistakes in report release ZIPs, creating "mini" report ZIPs and installing them, interacting with external APIs (Localise), and even looking up obscure passwords.

Don't remember how to do it; figure it out and then write a PowerShell script for it!

We mentioned a couple weeks ago that we wanted to reduce dependency on specialists for things we depend upon. Becoming more comfortable with PowerShell is a good way to move this initiative forward.

## Why should I care about PowerShell?

- It is POWERFUL
  - Scripting, especially when it has hooks into as many things as PowerShell does, helps you do a lot of time-consuming stuff very quickly and accurately
  - It's GREAT for chopping data quickly, especially XML, CSV, and JSON
  - Your favorite .NET APIs are always available to you
  - It's fast - you can type and tab faster than you can point and click
- It is EVERYWHERE
  - It's in all versions of Windows since XP
  - It's in Azure
    - Deploy ARM
    - Nice suite of built-in PowerShell commands for working with Azure
  - Since August 2016 with PowerShell Core, it's cross-platform and open source
  - It's in EnergyCAP code:
    - Automation scripts built for clients like MDC
    - ECAP codebase: `C:\ecap\energycap\contrib\Loco`, `build.ps1`, etc.
    - Reports repository
  - It is embedded in Visual Studio Code, Visual Studio
- It is FAMILIAR
  - Thanks to aliases, you already have a handle on PS syntax that you bring from other systems
  - C# developers and people familiar with COM objects will enjoy having familiar APIs at their fingertips

## What is PowerShell?

- Microsoft's reimagined command-line interface for working with their technologies
- Like cmd.exe on steroids; includes everything I can think of from cmd.exe; access to system environment variables, directory navigation, execution of scripts, file I/O, interactions with RESTful APIs, PC administration, and much more.

### Brief History Lesson

Every version of Windows has always had its CLI (command line interpreter). Back MS-DOS days, everything was done via the CLI! (my first code was "cd "). However, cmd.exe and batch files had limited functionality and couldn't automate GUIs very well.

Microsoft introduced Windows Script Host and VBScript in Windows 98. Once nice feature added here was the ability to script access to COM objects. EnergyCAP installed client took advantage of this via Virtual Account scripts, Batch name formulas, and EDI scripts. However, poor documentation and security concerns gave it a bad reputation.

By 2002, MS was developing a new CLI called Monad, focused on automating core administrative tasks. Monad was renamed to be PowerShell in April 2006, and then v1 was released at the same time.

That means PowerShell is available as far back as Windows XP. Early versions didn't include cmdlets for Azure and web technologies because they were maturing; I've seen this feature set grow over time.

13-year-old technology! Not staying stagnant...I learned quite a few things about PowerShell since starting to prepare this presentation! Two of the most interesting changes:

- Unit Testing framework that shipped with Windows 10 (2015)
- PowerShell Core for all OSs (2016)

## PowerShell concepts

Now that we have a little historical context out of the way, I want to introduce how this language works. Let's introduce the concepts:

- PS Console

Get to it from Start Menu, Run interface (Win + r), cmd.exe

`Get-Location` tells you where you are

- commandlets

Commandlets are basically PowerShell functions which encompass programmatic work of some kind. Think of them as functions.

Structure of the commands is "Verb-Noun"

E.g `Get-Verb` gets a list of verbs approved for use in PowerShell commands. As you will see, I'm not very consistent with my naming schemes, but hey, I'm getting better.

E.g. `Get-Command` will get you all the available commands in the PowerShell ecosystem. Includes Windows cmdlets, 3rd-party cmdlets, and your custom cmdlets.

If you're good about naming your functions, you can even easily filter the commandlets with the `-Verb` and `-Noun` params to `Get-Command`

`Get-Member` is a useful command to know. If you want to get a list of all the available properties and methods for a PS object, this is what you need.

A couple veeery useful ones: `Get-ChildItem` `Invoke-SqlCmd` `Out-File -encoding UTF8` (BE CAREFUL WITH ENCODING)

- aliases

PowerShell automatically creates aliases for the cmdlets. Many of them have been borrowed from other CLIs to make PowerShell easy to use for Unix and Linux users.

E.g. `dir`, `ls`, and `gci` are all aliases for the `Get-ChildItem` cmdlet.

You can create your own aliases with `Set-Alias`

- variables

PowerShell variables are extremely flexible. They are *not* type safe unless you explicitly set the object type during initialization, and even then, PowerShell is happy to convert types for you if a conversion exists.

Here's how you initialize typed variables: [string], [xml]

`<response error=""> <station name="UNVX" city="University Park" state="PA" country="US" latitude="40.800000" longitude="-77.850000" elevation="1158"/> </response>`

What are my available variables? Use `Get-Variable`

- script files

Don't type it out every time!

PowerShell scripts are stored in files with a "*.ps1" suffix. They may have code for a single function, or they may be libraries of functions.

You can pass parameters to PS1 files, you can even make them mandatory

`[Mandatory($true)]`

- functions

While you can contain PowerShell code at the file level, you can also declare multiple code functions within a single PS script file. These can be available within a single PS script execution or, when loaded, they can be used over and over again.

`function GetRandomNumber() { $rand = Get-Random -Maximum 100 -Minimum 0 return $rand }`

- pipelines

  "Pipelines act like a series of connected segments of pipe. Items moving along the pipeline pass through each segment."

  (source: https://docs.microsoft.com/en-us/powershell/scripting/learn/understanding-the-powershell-pipeline?view=powershell-5.1)

- operators

  -eq, -ne, -gt, -like, -and, -or

  `Where-Object, if()`

  Can use -like with wildcard * character

## Tips working with PowerShell

### Things to Love

- Love the tab key

PowerShell cmdlets do not display their arguments, and you're not using your mouse, so you can't hover over and see what's available in the PowerShell prompt. Use the tab key to view the available arguments and, in some cases, their possible arguments.

- Love pipelines and $_.

You can avoid traditional for-loops by "piping" the results of one command into another. PowerShell gives you a powerful contextual

`ForEach-Object` and `Where-Object`

- Love your PowerShell profile

If you get tired of loading PS1 files whenever you want a common function available, add the functions you use the most to your Microsoft.PowerShell_profile.ps1 file. Save it to your C:\Users\kurtk\Documents\WindowsPowerShell folder. You might even want to put it in source control.

- Love the up-arrow

This is common to most CLIs, but you can always get your commands back by hitting the up arrow to get the previous command, even from session to session.

- Love hash tables

If you want a quick and dirty object or set of objects to work with, you'll love hash tables. They're easy to initialize, and look a lot like dynamic objects in C# or regular objects in JavaScript:

`$myHashTable = @{ Name = "Kurt Kroeker"; Age = 31; Occupation = "Software Guy" }`

They even give you autocomplete for properties!

- Love working with JSON and CSV

PowerShell comes with niceties for working with data in CSV and JSON format:

`ConvertFrom-CSV` and `ConvertTo-Csv` `ConvertFrom-Json` and `ConvertTo-Json`

- Love Invoke-RestMethod

- Love the Windows PowerShell ISE

From Start Menu: "PowerShell ISE"

Nice IDE (or...ISE!) for composing and debugging PowerShell commands. Includes variable inspection, debugging with step-through and step-over, etc.

From PowerShell: `ise` or `powershell_ise`

- Love .NET!

```
Add-Type -Path .\EnergyCap.Data.dll

$context = New-Object EnergyCap.Data.Entity.EcDbContext "Data Source=devsqlwin01\sql2017;User
ID=esuser;Password=e2isnotis;Initial Catalog=vnext_abbvie", 2, 1024

$context.SystemUsers | Where-Object { $_.systemUserID -eq 2 }
```

## Gotchas

Working with strings

- Watch your delimiters!

PowerShell allows both single and double quotes to be used for delimiters. However, only double quotes honor string interpolation. Compare the following statements.

`$foo = "foo"` `"we have some $foo string interpolation here"` `'Sorry bub, no $foo string interpolation here'`

While you can usually choose single or double quotes when working with short strings, when you have to work with stringified JSON, you might have both single and double quotes in the data. Here-Strings are your friend:

`$json = @" {"placeCode":"KURTS_APOSTROPHIED_BUILDING","placeInfo":"Kurt's Apostrophie'd "Building"","parentPlaceId":1,"placeTypeId":2,"primaryUseId":null,"weatherStationCode":"UNV","buildDate":null,"address":{"addressTypeId":"1","country":"US","line1":"","line2":"","city":"State College","state":"PA","postalCode":"16803","latitude":"","longitude":"","weatherStationCode":"UNV"}} @"

`ConvertFrom-Json $json`

- Old code on the internet

Since PowerShell has evolved quite a bit over time, make sure you're always checking the timestamps on the articles and code samples you read. Something that used to be really hard (e.g. JSON manipulation before PowerShell 3.0 and `ConvertFrom-Json` ) might have become really easy.

- Is it an array?

Sometimes you may find that variables which you expected to be an array. For example, the much-used `Get-ChildItem` returns an array of files OR a single file (if there was only 1). Fore example:

`(gci *.ps1) -is [system.array]` `(gci *.dll) -is [system.array]`

If you find that you're in a scenario like this, you can always check for array-ness to make sure you code produces the expected results.

- Calling assemblies with arguments

Getting the syntax right was super frustrating when I wanted to execute an EXE with some arguments. However, I think this has gotten better with more recent versions of PowerShell. Here's an example of callings some assemblies with arguments from PowerShell:

```
dotnet .\ThermaCAPtureStats.dll -search "vance"
```

- . vs. .\

When you're navigating directory structures in PS, you can execute PS1 files directly by using the `.\myFileName.ps1` syntax. The command as expressed here will simply RUN the script.

However, if you use the period, you can *include* the PS1 module for use, if it contains functions you want to use: `. .\myFileName.ps1` will both *execute* the script AND register the functions for use within the PowerShell session.

## Subjects Not Covered

- Execution policies
  - I (we?) have RemoteSigned, meaning any code downloaded from the internet will not be executed by accident
- Azure resource interactions
- PC user and administrative management
- GUIs and PowerShell
- PSCustomObject creation
- Using external DLL dependencies (Add-Type, .NET [reflection.assembly])
- try/catch with PowerShell
- Microsoft Powershell Gallery (https://www.powershellgallery.com/)

## Resources

- Installing PS on macOS: https://docs.microsoft.com/en-us/powershell/scripting/install/installing-powershell-core-on-macos?view=powershell-6
- Playing sounds with PowerShell: https://devblogs.microsoft.com/scripting/powertip-use-powershell-to-play-wav-files/
- Is my PS variable an array or not: http://thephuck.com/scripts/easy-way-to-check-if-your-powershell-variable-is-an-array-or-not/
- Type Safety in PowerShell: http://www.winsoft.se/2009/01/type-safety-in-powershell/
- PowerShell tutorial: https://www.tutorialspoint.com/powershell/index.htm
- PowerShell in Wikipedia: https://en.wikipedia.org/wiki/PowerShell
- Testing PowerShell scripts: https://devblogs.microsoft.com/scripting/what-is-pester-and-why-should-i-care/
- PowerShell dev in VS Code: https://docs.microsoft.com/en-us/powershell/scripting/components/vscode/using-vscode?view=powershell-6
- Save Excel as CSV: https://michlstechblog.info/blog/powershell-export-excel-workbook-as-csv-file/
- Setting ExecutionPolicy: https://www.mssqltips.com/sqlservertip/2702/setting-the-powershell-execution-policy/
- Version of .NET used by PowerShell: https://stackoverflow.com/questions/3344855/which-net-version-is-my-powershell-script-using