**Why Developers should love PowerShell**

Author: Kurt Kroeker Last Updated: 10/11/2019

Intro

* Welcome to "Why Developers should love PowerShell"!

**About This Talk/Agenda**

* **Talking**
  + Kurt's PowerShell Background
  + Why should I care about PowerShell?
* **Demos**
  + Introduction to PowerShell
  + Tips working with PowerShell
  + PowerShell Gotchas

**DISCLAIMER:** I have a lot to cover, so if you’re following along on your computer and have an issue with something you saw me do, please write down your questions for the end of the session and I’ll hang around afterward for a little bit, too.

**Show of Hands: How many already using PowerShell on a daily basis?**

**What is PowerShell?**

* Microsoft's reimagined scripting terminal for working with their technologies (successor to cmd.exe and VBScript); it’s a glorified command prompt which has access to a huge array of functionality.

**Swiss Army Knife**

I’m super enthusiastic about PowerShell because I find it a very useful tool. <**show hammer**>

* "When all you have is a hammer, every problem looks like a nail." – not only one purpose!
* I'd like to encourage you to think of PowerShell as your Swiss Army Knife because you can use it to solve all kinds of problems right now, but one where you can just add your own blades over time to satisfy your and your organizations’ needs.

**My PowerShell Background**

Back in 2012, Miami-Dade County (MDC) government needed to put together an automated workflow to keep data flowing in and out of EnergyCAP. This workflow included:

* File system manipulation
* Executing multiple EXEs that transformed accounting data and accepted input parameters
* Querying SQL server databases
* Making REST API calls
* Logging was required for all of these functions
* Even sending emails with summaries of the log

What kind of technology could solve this problem? **POWERSHELL!**

Did anybody attend Chris Houdeshell’s session yesterday, "Recovery by Design: A Postmortem Adventure"?

Chris recommended PowerShell as a solution and helped me get the project off and running. It was a very successful project, and now EnergyCAP, Inc. relies heavily on PowerShell to automate similar workflow automation scripts for many of our clients. We’re invested heavily in it.

Not just valuable to EnergyCAP; valuable to me as a developer! 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 mistakes made in manual processes in in our team, interacting with external APIs, web scraping, managing credentials for my scripts.

It's become a tenet of mine: Don't just remember how to do it; figure it out and then write a PowerShell script to do it!

**Why should I care about PowerShell?**

1. 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, and available PowerShell IDEs are lightweight and well-featured. The rinse/repeat cycle is very fast.
2. It is EVERYWHERE
   * It's in all versions of Windows since XP
   * It's in Azure
     + Deploy ARM
     + PowerShell Azure
     + 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 probably in your organization somewhere now; or you will run across it in your career.
   * It is embedded in Visual Studio Code, Visual Studio
   * There are hooks into Powershell from some platforms we use such as OctopusDeploy, JAMS to provide saved credentials, variables, etc.
3. 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 data structures and APIs at their fingertips
   * If **YOU** spend the time building up your PowerShell "swiss army knife", you will end up with a very customized, very comfortable experience

**PowerShell Console Concepts**

So let's get to some demos and see how PowerShell looks, in action!

* PS Console
* Get to it from Start Menu, Run interface (Win + r), cmd.exe
* Chances are you already have a terminal open right in front of you in Visual Studio or VS Code.

Among other things, Get-Location tells you where you are (**your first cmdlet!**)

* commandlets
* Think of commandlets as PowerShell functions which encompass programmatic work of some kind
  + Analogous to command line programs in Linux like ls or grep
  + Cmdlets perform an action and/or return a value
* Cmdlets are built into PowerShell, installed along with other apps you install, or installed by you when you install a module
* Cmdlets are different from PowerShell "functions" in that they are actually written in compiled .NET code using the PowerShell SDK, while functions are pure PowerShell syntax
  + They both function the same in actual use

Structure of the commands should always be "Verb-Noun"

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

E.g. Get-Help gets help documentation for the provided commandlet

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.

If you're good about naming your functions, you can even easily filter the commandlets with the -Verb and -Noun params to Get-Command. You lose this benefit if you don't stick to the Verb-Noun format. This concept of a naming scheme makes PowerShell very discoverable. Also, the online documentation for PowerShell is EXCELLENT and often includes multiple examples built-into the docs (don’t have to go all over online to find recipes for use, although they’re there, too).

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.

* Show variable mutation in console
* Here's how you initialize typed variables: [string], [xml]
* What are my available variables? Use Get-Variable.
  + NOTE: I am using a PS Profile, so I have a lot more variables that you likely will.

**DEMO: XML contained in cd\_catalog.xml**

* pipelines

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

Important to note that in PowerShell, complex objects and arrays of objects are piped from command to command; contrast with Linux where pipes are only passing around strings.

**DEMO: demo\_pipelines.ps1**

Resource: <https://docs.microsoft.com/en-us/powershell/scripting/learn/understanding-the-powershell-pipeline?view=powershell-5.1>

**PowerShell Script File Concepts**

* 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, or complex scripts filled with business logic.

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

[Parameter(Mandatory = $true)]

**DEMO: demo\_scriptParameters.ps1**

* 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 into session memory, they can be used over and over again.

* operators

-eq, -ne, -gt, -lt, -like, -and, -or, -not

Where-Object, if()

Can use -like with wildcard \* character

-match -notmatch -replace -split

Can use regex operators

-whatif

You can see what WOULD happen without actually performing the action

**DEMO: demo\_operators.ps1**

* Error Handling

**DEMO: demo\_errors.ps1**

**Tips working with PowerShell**

**Things to Love**

* Get-Help, -Command, -Member

I mentioned the high discoverability of PowerShell functionality earlier. I want to quickly review them a bit more, and dig deeper into why why these work so well and how you can leverage the power of comment-based help.

Get-Help is good for your code, not just pre-packaged code; good practice to document using comment-based help. Keep your PS documentation up to date along with your source code.

**DEMO: demo\_commentBasedHelp.ps1**

* 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 variable $\_. to reference the looped item.

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 concise but powerful dictionary-like object to work with, you'll love hash tables. They're easy to initialize and act 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, including properties with spaces in the names!

**DEMO: Do a live HashTable**

* Love working with JSON and CSV

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

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

**DEMO: cd\_catalog.xml** **DEMO: places.json**

**DEMO: SalesJan2009.csv**

Format-Table may be useful to you as well when you're trying to view data.

* Love Invoke-RestMethod

**DEMO: demo\_apiCalls.ps1**

* Love VS Code

PowerShell extension for VS Code makes debugging PS scripts easy and enjoyable. Another option is available, but no longer supported (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!

I just need to keep pointing out that everything you're seeing in this session is in the context of the .NET or .NET core framework. The same structures, APIs, libraries are available to you throughout, so it will feel very familiar to .NET developers.

**Gotchas**

**. 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. This is called "dot sourcing" the script.

**DEMO: dotsource demo\_functions.ps1**

**ExecutionPolicy**

PowerShell has some very nice security features to protect users from malicious script execution, one of them being the ExecutionPolicy.

Get-ExecutionPolicy

There are a number levels of ExecutionPolicy. From most stringent to least stringent:

* Restricted - terminal scripting only; can't execute script files
* AllSigned - all script files can be executed only if they're signed
* RemoteSigned - remote script files can be executed only if they're signed
* Default - sets the default execution policy
* Bypass - nothing is blocked and there are no warnings or prompts

If you find script files or modules online that you want to try, you may find yourself having to negotiate the ExecutionPolicy, which may require elevated access to change. I recently hit this with a script I needed to resize a batch of images. I couldn't install the module; but I reviewed the script within the module and easily adapted it.

Resource: <https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.security/set-executionpolicy?view=powershell-6> Resource: <https://gallery.technet.microsoft.com/scriptcenter/Resize-Image-A-PowerShell-3d26ef68>

**Working with strings**

* Watch your quotes!

PowerShell allows both single and double quotes to be used for declaring strings. 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

$json

Additionally, if you have a complex object to use with string interpolation, you need to do a little more. Consider this more complex example:

Get-ChildItem | ForEach-Object { "The length of this file is $\_.length" } Get-ChildItem | ForEach-Object { "The length of this file is $($\_.length)" }

In this example, you need to surround the property reference with another $(). Then the interpolation will work correctly.

**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 are not. For example, the much-used Get-ChildItem returns an array of files OR a single file (if there was only 1). For example:~~

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

~~Scenarios like this expose themselves where you expect a loop to execute multiple times but they error! Get-Service also subject to this effect.~~

~~Resource:~~[~~https://devblogs.microsoft.com/powershell/same-command-different-return-types/~~](https://devblogs.microsoft.com/powershell/same-command-different-return-types/)

**~~DEMO: "length" property means two different things for results from Get-ChildItem~~**

~~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. It's easy to get confused by PowerShell's behavior when it comes to the syntax for passing arguments into EXEs. Here's an example of calling some assemblies with arguments from PowerShell:

**DEMO: demo\_consoleArgs.ps1**

**PowerShell and TLS**

PowerShell is configured to make all web requests using TLS 1.0, so if you're making web requests against a web server with TLS 1.1+ enabled, you'll quickly discover that Invoke-RestMethod and Invoke-WebRequest calls will fail, perhaps with an error message like this:

The request was aborted: Could not create SSL/TLS secure channel.

You can configure PowerShell to communicate with TLS 1.1 and 1.2 by modifying the ServicePointManager's SecurityProtocol setting. Try running this snippet in PowerShell and retry your request:

[System.Net.ServicePointManager]::SecurityProtocol = [System.Net.SecurityProtocolType]::Tls12;

Please note that this will ONLY affect your current PowerShell session. To make it stick, add this line to your PowerShell Profile.

**Subjects Not Demoed**

* Splatting arguments into functions/commandlets
  + Use @ instead of $ for splatting...
* Azure resource interactions
* PC user and administrative management
* GUIs and PowerShell
  + Building using Windows Forms...I prefer named parameters, Read-Host, Get-Credential
  + I saw a desktop automation module out there
  + Selenium has a PowerShell module; it's nice :)
* PSCustomObject creation
* Add-Type to use DLLs, even create your own!
* try/catch with PowerShell; error handling with $Error array of "ErrorRecord" types with the exception type and other details within, if you drill down.

**Q & A**

* Q: Why should I use PowerShell Core vs. regular?
  + PSCore feature set is smaller b/c .NET Core is newer
  + Run PS-based automation in other OSs
  + .NET Core generally faster than .NET Framework
  + PowerShell core is open source; see the guts
  + NOTE: can't use ISE in PSCore...use VSCode insteads
* Q: Why should I use Visual Studio Code vs the ISE?
  + VS Code
    - Is much nicer for managing projects/workspaces
    - More customizable than the ISE (e.g. default terminals for PS or PSCore)
    - Debugging experience was prettier
    - However, integrated PS terminal was not as nice; quirks of text input, autocomplete not as nice; no colors
    - Code snippet completion is a little annoying
  + ISE
    - Already installed everywhere
    - Decent debugging experience
    - ISE is not included as part of PowerShell Core; you'll have to use VS Code

**Resources**

* Installing PS on macOS: <https://docs.microsoft.com/en-us/powershell/scripting/install/installing-powershell-core-on-macos?view=powershell-6>
* 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>
* Niceties for working in git with PowerShell - posh-git: <https://github.com/dahlbyk/posh-git>
* Microsoft Powershell Gallery (<https://www.powershellgallery.com/>)