PSCONF.EU

2018

A Historical Architectural Tour of PowerShell (Part 2)

Bruce Payette
Microsoft Corp.
@BrucePayette

Agenda

- This is round 2 of (at least) a 2 round set of talks.
- In this session, we're going to trace PowerShell's evolution from V2 up to modern day.
- We'll look at some of the technical, cultural and business forces that drove this evolution
- The goal of these talks is both to present and to gather interesting (and hopefully useful) historical aspects of PowerShell's evolution.
  - The plan is to put all of this information into the GitHub repo so it will be available to everyone.
  - So if you have suggestions, anecdotes, questions or complaints, please let me know.

About Me

- Developer at Microsoft for 16+ years
- Before Microsoft worked on POSIX.2 Shell & Utilities at MKS and the Softway Systems
- On the PowerShell team for 16+ years
- Was lead developer for the language
- Also did a lot of the design for the pipeline (streaming) and command processors
- Author of “Windows PowerShell In Action”
- Email: brucepay@Microsoft.com
- Twitter @BrucePayette

PowerShell Version 2

- OK – we shipped V1. If we're good, maybe they'll let us do it again!
- In-place update to V1
  - Assemblies are in the GAC;
  - Work around rigid versioning; from the CLR perspective all types in version 2 where the same types as version 1
- A big release with many new features
  - The groundwork from V1 enabled us to add a ton of functionality in a relatively short time
- No *intentional* breaking changes between V1 and V2

InitialSessionState & Constrained Runspaces

- V1 supported multiple runspaces with `RunspaceConfiguration`
  - "Live object" shared across all runspaces created with it
  - Didn't cover all `SessionState` types, only cmdlets
- `InitialSessionState` introduced in V2 to fix this
  - `InitialSessionState` is used to *initialize* the runspace (i.e. the runspace gets a copy of the `InitialSessionState` object)
  - has support for all command types (functions, aliases, etc.) & variables
- `SessionState` entries had a `Visibility` property that controlled if a command was visible outside the runspace
  - When a command is compiled it is marked with as `CommandOrigin` *`Runspace`* or *`Internal`*
- Basis of Hosted Exchange RBAC model
  - Based on your account, they would generate an ISS for your role
  - Underpinnings of what became JEA

The [PowerShell] API

- V1 had a somewhat awkward set of host APIs.
- V2 attempted to remedy that with the `PowerShell` API
- This was a *fluent API* designed to make it easy to specify PowerShell commands in C# (or VB, Python, etc.)
  - Builder pattern
- Example

```
dir | sort Length | select –First 10

PowerShell.Create().AddCommand(“dir”).AddCommand(“sort”).AddArgument(“Length”)
   .AddCommand(“select”)
      .AddParameter(“First”, 10)
         .Invoke()
```

Modules 1

- We had CPAN envy big-time but for a module archive, first we needed modules
- Not originally planned for V2
  - Development of modules started very late in the development cycle
  - Most of the pieces were in place from `InitialSessionState` work
  - But the devil is in the details
- Module discovery, loading and unloading are dynamic
  - Module loading is a *runtime* operations
  - Done imperatively using cmdlets
  - Most languages have syntax to do this at compile time (so do we as of V5)
- Modules are first-class; visible to end-user as `ModuleInfo` objects.
  - Can invoke scriptblocks in a *module context*

```
& $myModuleInfo { $script:zork = 123 }
```

Modules 2

- The heart of a module is the `SessionState` object.
  - Same type of object that holds the global state
  - `SessionState` objects can be linked together (parent/child)
  - Pretty heavy weight.
- Different types of modules Script/Binary/Manifest
  - *Manifest modules* may only contain imported definitions
  - If a manifest imports a **root** module, the imported module takes on the type of the root module
- Modules are loaded only once (unless `-Force` is used)
  - Nested modules are not reloaded (by design)
  - All modules stored in a runspace-global module table, indexed by absolute path
- Removing a module is problematic
  - As long as there is a single reference to the module, it won't be unloaded

Modules 3 – Advanced Module-related Features

- Dynamic closures: scriptblocks that "wrap" a module

```
function counter {  # Counter factory function
    $count = 0
    { ( $script:count++ ) }.GetNewClosure()
}
$function:mc = counter # create counter function
```

- Modules as Custom Objects

```
New-Module –AsCustomObject { # custom object with 2 properties & 1 method
    $x = 1
    $y = 2
    function Print { "x=$x y=$y" }
    Export-ModuleMember –Function Print –Variable x,y
}
```

Remoting

- For a datacenter automation solution, lack of remote execution was a big obvious hole.
  - Almost shipped a PowerShell V1.5 to get PowerShell out early
- A basic design for remoting actually had been done in V1 and bits were implemented in the form of `Export-CliXML`.
  - Lossey serialization – only a small number of types serialized with fidelity; everything else is a property bag – for version insensitivity.
  - V1 plan had been to use Windows Communication Foundation as the transport layer.
- V2 plan was to use the new Web Services for Management (WSMan) standard and WINRM service
  - Uses the non-standard shell extension protocol
- Lot's of performance work before release but it's still an XML-based protocol on top of an XML protocol
  - Documented as MS-PSRP on top of MS-WSMV.

Integrated Scripting Environment

- Split-pane shell experience
  - Input in one pane, output in a separate pane
  - Seemed like a good idea at the time ☹
- Tab-completion based on (old) tokenizer API; see src:

  `…\src\System.Management.Automation\engine\parser`
  - Tab-completion worked in the editor as well as the command line
- Implementing the ISE drove a change into how the host threading module worked
  - `powershell.exe -mta/-sta` (PowerShell V1 was MTA more or less by accident)
  - Default for V2 was STA which made it possible to used XAML

    `[system.threading.thread]::CurrentThread.ApartmentState`
  - In V1, every invoke resulted in the creation of a new thread; V2 Introduced the ability to reuse the host (caller's) thread.

Other Stuff

- New operators in the language `-split`, `-join`
  - "Completed" the set of string/array operators
- Data Language
  - Constrained language regions in a script
  - Supported localization for modules along with `Import-LocalizedData`
  - Files containing only data with .psd1 extension (also used for module manifests)
- Block comments
  - <# A comment #>
- Ability to assign the output of a statement directly

```
$numbers = if ($x -eq $true) { "It was true" } else { "It was false" }
$randomNumbers = foreach ($i in 1..1000) {
                    Get-Random
                }
```

PowerShell Version 3

- Side-by-side with version 2
  - We moved to a new CLR version (CLR 4)
- Deliberately took some (small) breaking changes
  - Took advantage of the fact that we were going to a new CLR to justify fixing some things
- It took *YEARS* to get significant uptake of Version 3
  - Having to install a new CLR was a big impediment
  - People hate to change their software (except #TiredJoey)
- Themes: Multi-Machine Management, Coverage

Workflow (Codename M3P) - ZOMG!

- Windows Workflow Foundation (WF) has strict semantics, lexical scoping, etc.
- PowerShell had to include an entirely separate execution engine
  - Fortunately loaded on demand through a parser fiddle
- We decided that workflows should look just like PowerShell
  - Scripts convert to XAML; XAML references *Activities;*
  - Every command has a corresponding Activity.
  - New module type “xaml” for pure workflow
  - Workflows look like functions except for “workflow” keyword instead of “function” keyword
  - This was probably a mistake.
    - A lot of the value of workflow comes from presenting a restricted execution environment (see DSC).
    - We worked very, very hard to produce an execution environment with as few restrictions as possible (impedance mismatch)

Workflow Example: Highly Parallel

```
workflow mywf
{
    parallel # 3 statements in parallel
    {
        "one $(hostname)"
        "two $(get-date)"
        foreach -parallel ($i in 1..5) # 5 iterations in parallel
        {
            "number $i"
        }
    }
}

mywf -PSComputerName host1, host2, host3 # Start on three hosts
```

- But there is significant overhead in concurrent execution (runspace pooling)
  - So prefer *long-running remote* operations for workflow

CIM Cmdlets and CDXML

- Able to define cmdlets with CIMv2 providers and cdxml files
  - Declarative mechanism for defining cmdlets in XML
  - Provided support for both GUI and PowerShell experiences from one code base
- New module type "cdxml"
- Resulted in an *enormous* increase in coverage for PowerShell
- New *-CIM* CIMv2 cmdlets peered *-WMI* cmdlets
  - `Get-CimInstance` vs `Get-WmiObject`
- New object type for CIM `Microsoft.Management.Infrastructure.CimInstance` vs. old System.Management.ManagementObject type
  - CimInstance doesn't have methods; only properties

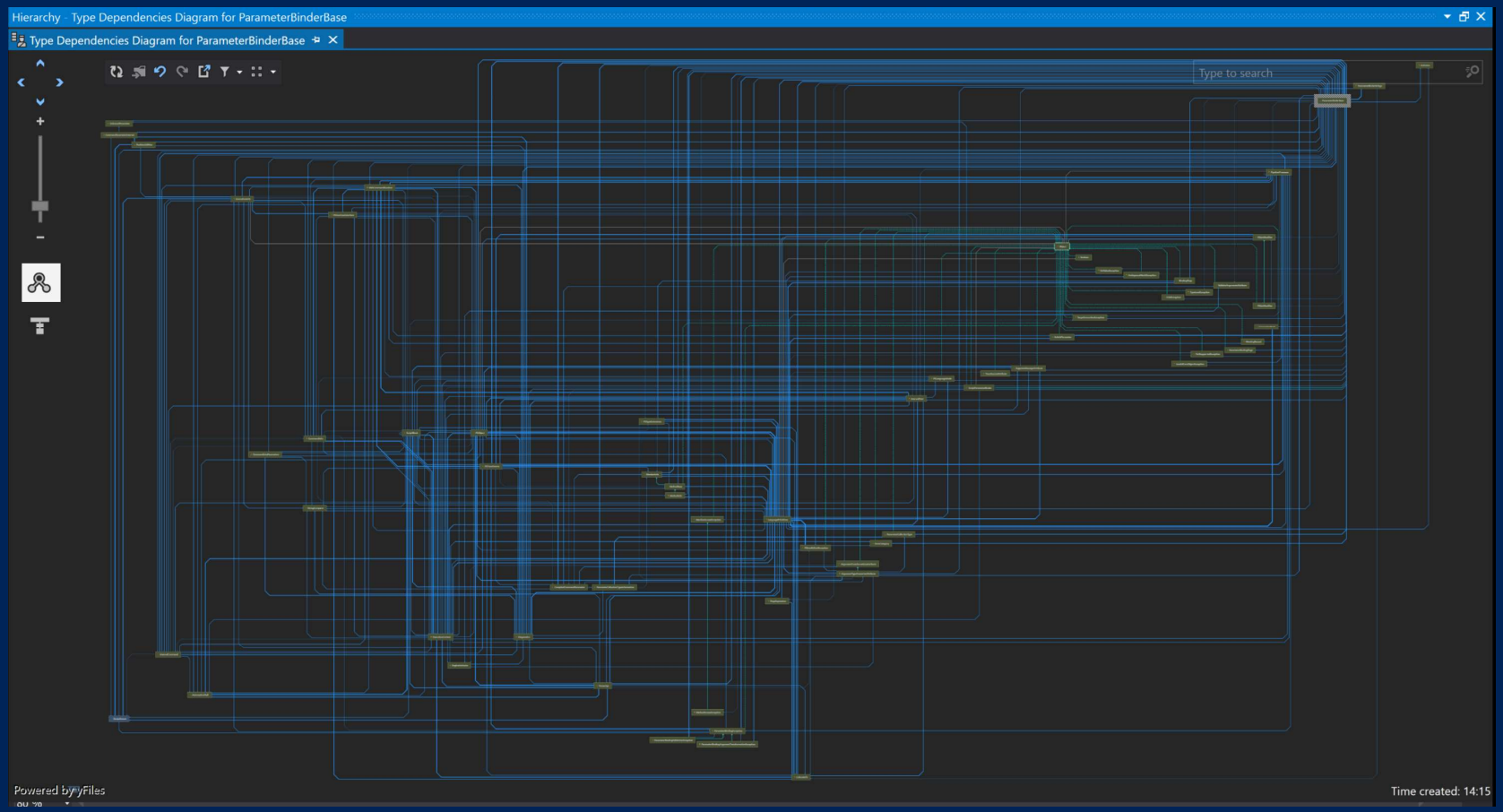
New Parser + DLR

- First version of the language parser and runtime was written to be very general
  - Recursive-descent, not optimized; had some performance issues; tokenizer was funny
- Version 2 introduced a new parser+tokenizer which is much faster
  - Still a recursive-descent parser optimized for PowerShell.

    …\src\System.Management.Automation\engine\lang
- Parser produces a *public AST* which allows **tooling** like Script Analyzer
  - Contains proper *extents* for tokens; allows for better error messages
- DLR expression trees for the runtime
  - Call-site caching for dynamic dispatch == much faster method dispatch
  - Blended interpreter/compiler for performance: Interpreted at first, then compiled after a certain number of iterations

    …\src\System.Management.Automation\engine\runtime
- Net result: language performance improves ~ 8-10x
  - But pipeline performance still sucks ☹

The Parameter Binder is COMPLEX (and slow)



Remoting Changes

- Connect/Disconnect for PowerShell sessions
  - Connect to a remote runspace ; Start a job ; Disconnect ; Go home ; Have a beer ; Reconnect
- Reliable connections
  - Transient connectivity issues won't cause the connection to be torn down
- Endpoint configuration files
  - `New-PSSessionConfigurationFile`, `Register-PSSessionConfiguration`
  - Declarative mechanism for creating constrained Runspaces; somewhat similar to module manifests

Integrated Scripting Environment

- Single-pane shell experience
  - User input and shell output mixed in the same window
- Outlining, line numbers
- AST-based *intellisense* instead of token-based tab-completion
  - Including dropdown lists
- Snippets
- Help window
- `Show-Command` (built-in to ISE and standalone cmdlet)
- XML syntax highlighting
- Scripting model improvements
  - Added tool windows
  - ISE options

Other Stuff 1

- Automatic member enumeration

```
(Get-Process).StartTime.Hour
```

- New Redirection Operations `Invoke-Something 4>> verbose.txt`
- `$using:` variables

```
icm –cn srv123 { echo $using:localVar }
```

- Added `$PSItem` as an alias for `$_`
  - Done to make PowerShell easier to learn; just made it more complicated
- Spaces allowed after the '.' in member accesses

```
(Get-Process).
 .StartTime
    .Hour
```

- Attributes (constraints) now allowed on any variable

```
[ValidateRange(1,10)] [int] $x = 1
```

Other Stuff 2

- Ordered hashtables, casting objects

```
[ordered] @{a=1; b=2} # using System.Collections.Specialized.OrderedDictionary
[System.Drawing.Point] @{X=1; Y=2}
[pscustomobject] @{x=1; y=2}
```

- `ForEach` statement no longer iterates over `$null`
  - Breaking change but more what people expected
- Comment based help improvements
- Added `--%` to make it easier to run executables
- New operators `-shl -shr -in -notin`
  - Note: PowerShell has more than 50 operators
- Module autoloading
  - Simplified interactive experience

Other Stuff 3

- Updateable help
  - No it's not a protocol
- Command discovery hooks:

```
$ExecutionContext.InvokeCommand.CommandNotFoundAction
$ExecutionContext.InvokeCommand.PostCommandLookupAction
$ExecutionContext.InvokeCommand.PreCommandLookupAction
```

- Changes to how the `PSObjects` are bound
  - V1 `PSObject.AsPSObject()` allowed multiple wrappers per object
  - V2 used lookaside so `PSObject.AsPSObject()` always returned the same `PSObject`
  - Yeah – this broke someone…

PowerShell Version 4

- In-place update to PowerShell version 3
- *Very* short (for Windows) release cycle ~ 1 year

Desired State Configuration Management

- Extension to PowerShell to allow you to do Declarative Configuration Management
  - At a certain level, similar in concept to Puppet/Chef, Ansible
- Client component: extensions to the PowerShell language
  - DSL for declaratively expressing the desired state of a target
  - Implemented as a general extension mechanism for DSL in the parser (runtime def'n only)
  - + cmdlets for managing and executing DSC configurations
- Second component: Local Configuration Manager (LCM) Agent
  - Runs on the managed node and enacts configuration
  - Ran as a CIM provider under WinRM
  - Depended on the task scheduler for cycles (15 minute granularity max.)
- Third component: resource providers for configurable elements
  - implemented in PowerShell or as CIMv2 providers in C++ (only the File provider)

DSC Pull Server

- Pull server allows DSC agent to pull configurations
  - Originally intended to be a sample (open source); ended up shipping as part of the product (close source)
  - Very limited capabilities esp. no support for managing content
  - REST for pull scenarios vs CIM for push scenarios
- Protocols are publicly documented through the Open Specifications program
  - Protocol designation is [MS-DSCPM]
  - Protocol is based on ODATA (see https://www.odata.org)
- Also allows pulling provider Modules
- Status reporting and aggregation

Some DSC Tribulations

- Had some heated “discussions” with Azure
  - Concern about possible confusion between DSC vs Azure templates

I’m not licking your cookie, I’m giving you a cookie!

Jeffrey Snover

Where and ForEach methods (or operators)

- New ubiquitous *methods* on all types for selecting and manipulating collections in *simple expressions*.

```
Write-Output (1,2,3,4,5).foreach{ $_ * 10 }
```

- Designed to minimize unnecessary punctuation
  - General syntax change – any method can be called with { } e.g. `$x.TakesSB{ “Hello” }`
- Feature driven by requirements for DSC configuration data processing
  - General purpose ***but minimally documented***

```
(dir).foreach{begin {$t=0} process {$t += $_.length} end {$t}}[0] -gt 20000

node $AllNodes.Where{ $_.Role -eq "WebServer“ }.NodeName
{ …
```

Other Stuff

- New default execution policy on Windows Server
  - the default execution policy is now `RemoteSigned`.
- Save-Help
  - Help can now be saved for modules that are installed on remote computers.
- Enhanced debugging
  - The debugger now supports debugging workflows, remote script execution and preserving debugging sessions across PowerShell session reconnections.
- `-PipelineVariable` switch
  - A new ubiquitous parameter to expose the current pipeline object as a variable for programming purposes
  - Added to support Azure Automation graphical workflow semantics

PowerShell Version 5, 5.1

- In-place update to PowerShell 4 (surprise ☺)

Classes 1

- First prototype used modules as classes – limited
  - Based on `New-Module -AsCustomObject { }`
- Goal for classes was "*Python equivalence*"
  - Explicitly not reinventing C#
  - Held weekly design meetings for a year
- PowerShell classes are *real* .NET types
  - Can extend existing types in the framework
- All types defined by the class keyword are public
- Classes defined in modules are *namespaced* by the module name
- All class members are public but can be marked hidden
  - Actually having private members would require the debugger to use private reflection

Classes 2

- Methods in classes have strict(er) semantics:
  - No uninitialized variables
  - Explicit return is required & return types are enforced
  - Lexical scope
- Classes are defined at compile time (static)
  - Mixing and matching with PowerShell dynamic semantics was problematic
- `using module` … loads the module at compile time
  - Referenced types are resolved at compile time
- Still to do
  - support for property setters and getters
  - using assembly … with metadata reader
  - Unloading/removing types

DSC Changes

- DSC agent went from being a scheduled task to a proper service
- Introduced the idea “partial configurations”
  - Core idea behind partial configurations was distributed responsibility
  - Different area owners (e.g. networking) would provide their fragment
  - Agent would aggregate the fragments into a single consistent configuration
  - This allowed area owners to independently service their fragment
- Able to implement DSC resource providers using classes
  - Instead of as a submodule
- DSC resource debugging in the LCM
- Pull server still sucks
  - Use Azure Automation DSC

PowerShell Gallery + OneGet (PackageManagement)

- Finally wanted to get our CPAN equivalent
- We stalked the halls of Microsoft talking to almost everyone who had a "gallery" of some sort.
  - There were a lot of them
  - Most of them inappropriate due to narrow focus or wrong target audience
- Nuget is where we landed
  - We had some issues with people not wanting to clutter up the purity of a dev gallery with icky management stuff (But we worked it out).
- OneGet universal module client
  - Project from previous release of server revived so we could take advantage of it
  - Driver-based mechanism to fetch stuff
- Galleries aren't easy
  - Security, quality, attribution, etc.; your processes need to be as automated as possible

PowerShell Version 6

- PowerShell goes open source. Head explodes!!!
- PowerShell goes cross-platform. Everything else explodes!!!
- Yeah – we broke some stuff. Deliberately.
  - New CLR which is, for the first time, is *not* a superset of the old CLR
  - New platforms to support
- We worked on this (especially the port to *nix) for a > year before announcing it
  - Just the technology
- Worked for another year before going GA
- We are side-by-side-by-side-by-side…
  - No GAC, self-contained installation
  - *Safe to install, safe to service*

Going Open Source was a LOT of New Work

- Created an entirely new set of processes and procedures
- Version control with GIT and Github
  - Migrated code from Windows *Source Depot* to Github with new(ish) layout
- Continuous Integration on multiple platforms
  - We use AppVeyor and Travis
- Community Governance/RFC Process; Breaking changes guidance
  - Stick to the processes you document (harder than we thought…)
  - *Maintainers* enforce code quality
  - *Committee* enforces guidance and functional evolution
- The work…
  - Building new tooling (build.psm1)
  - Moving build from Windows *razzle* to dotnet/cmake/build.psm1
  - Moving tests from *WTT* to Pester

V6 New features:

- Removed the convenience aliases (ls, cp, curl(!)) on *nix.
  - This was the subject of much debate
- The `more` functionality respects the Linux `$ENV:PAGER` and defaults to `less`.
- `-ExecutionPolicy` is ignored on non-Windows
- Default encoding is BOM-less UTF8 for interoperability on *nix.
- Background execution through '&'
- Automatic wildcard expansion for native command on *nix.
- Unicode escape parsing, new escape character for ESC (`e)
- Multiline,Singleline options to `-Split` operator
- No registry on *nix so we have Powershell.config.json file in $PSHOME
- Host application was rewritten: pwsh
  - Now defaults to –file behavior instead of –command; changed –Version to return version

V6 Breaking Changes

- No Workflow (no WF support in .NET Core)
- No ISE (no XAML support in .NET Core)
  - Alternate use VSCode
  - We are making lots of investments in VSCode
- No snapins (yay!)
- No WMI cmdlets (Get-WmiObject, etc.)
  - Use CIM cmdlets instead e.g. `Get-CimInstance`
- No `*-Counter` cmdlets
- Remove `AllScope` from most aliases
- No more direct cmdlet remoting with `-ComputerName`
  - On Windows, tt was COM-based; no COM on *nix so no direct remoting

The Future

- Driven by community and customers now more than ever
- You can *fix* your favorite bug
  - Or open an issue (bug or small feature) or RFC (feature), contribute to docs, processes, guidance, etc.
  - *But you can't change how existing features work or remove things*
- Breaking change contract

  https://github.com/PowerShell/PowerShell/blob/master/docs/dev-process/breaking-change-contract.md
  - Currently strong commitment to backwards compatibility
  - "*Open for Addition, Closed for Change*
- BUT the urge to fix (== change == break) things is strong
  - By Design" means it was *intentional*. It doesn't mean that it was *right*.
  - We should talk…

Summary

- PowerShell has had a long history over 16+ years & 6+ releases
- There is a strong rationale for most of the way PowerShell works
  - Even so, there may be a better way to do something
  - But please think about PowerShell as a whole; not as features in isolation
- Thanks.
  - brucepay@microsoft.com
  - @BrucePayette

Questions?

Next Steps

- Now: 15 min break

- Grab a coffee
- Stay here to enjoy next presentation
- Change track and switch to another room

- Ask me questions or meet me in a breakout session room afterwards

PowerShell Conference EU 2018

Simple Console Demos

```
# Embed one-liners right into your slides:

PS> Get-ExecutionPolicy
Bypass
```

Demo

Try and use demos instead of dead slides.

Please do not dive into zillions of pages of code.
Instead, if you can, turn your logical code blocks into functions, and store them in a module.

Share this module with your audience.

```powershell
# use this template for code samples
# and follow these instructions to show perfectly
# color-coded PowerShell code:

# paste code to PowerShell ISE, select it, and copy it
# to the clipboard
Paste-Code -ToISE | Select-Code | Copy-ToClipboard

# to insert it with full color-coding into a slide,
# paste the code to your PPT slide. It will be black,
# and there is a toolbutton labelled (Ctrl) in PPT.
Click-ToolButton -Choose SymbolWithClipboardAndBrush
# Click the toolbutton, and choose the button that shows
# a clipboard with a brush. This will add color-coding
# back to the pasted code
```

PLEASE NOTE: You do not need to use PowerShell ISE to code, or to demo. Use whatever editor you like best.

These steps use PowerShell ISE to color-code your code correctly and consistently, and insert the color-coded code into the slide. Please help make all code look consistent. Many thanks!

about_Author

- Feel free to add information about yourself
- You may advertise for you or your company

- Keep slide at the end of your presentation

- Do not use presentation time for your bio
  - Instead, your bio is published in conf materials