The Administrator Crash Course

Windows PowerShell v2

Realtime publishers

Don Jones

Introduction to Realtime Publishers

by Don Jones, Series Editor

For several years now, Realtime has produced dozens and dozens of high-quality books that just happen to be delivered in electronic format—at no cost to you, the reader. We've made this unique publishing model work through the generous support and cooperation of our sponsors, who agree to bear each book's production expenses for the benefit of our readers.

Although we've always offered our publications to you for free, don't think for a moment that quality is anything less than our top priority. My job is to make sure that our books are as good as—and in most cases better than—any printed book that would cost you $40 or more. Our electronic publishing model offers several advantages over printed books: You receive chapters literally as fast as our authors produce them (hence the "realtime" aspect of our model), and we can update chapters to reflect the latest changes in technology.

I want to point out that our books are by no means paid advertisements or white papers. We're an independent publishing company, and an important aspect of my job is to make sure that our authors are free to voice their expertise and opinions without reservation or restriction. We maintain complete editorial control of our publications, and I'm proud that we've produced so many quality books over the past years.

I want to extend an invitation to visit us at http://nexus.realtimepublishers.com, especially if you've received this publication from a friend or colleague. We have a wide variety of additional books on a range of topics, and you're sure to find something that's of interest to you—and it won't cost you a thing. We hope you'll continue to come to Realtime for your educational needs far into the future.

Until then, enjoy.

Don Jones

Copyright Statement

PowerShell Crash Course Week 1

By now you've probably gotten the message loud and clear that Windows PowerShell is pretty important; Microsoft is adding it to more and more products, and going forward, the company's plan is to incorporate PowerShell throughout all of its business products as a baseline administrative layer. No, the GUI isn't going away—it's still called "Windows," after all—but in most cases, the GUI will simply be running PowerShell commands under the hood. In some cases, the GUI will be "de-emphasized," meaning the GUI might not surface *all* of the product's administrative functionality.

If you're ready to get started in PowerShell, and have no experience, this is the crash course for you. If you have a bit of Unix or VBScript experience, try to remove that from your brain: PowerShell will look familiar, but it's really something very new and different.

I encourage you to continue exploring beyond this crash course, too. For example, visit http://windowsitpro.com/go/DonJonesPowerShell to find tips and tricks and FAQs and to ask questions, or drop by the PowerShell team's own blog at http://blogs.msdn.com/powershell for "insider" information. You'll also find a lot of in-person PowerShell instruction at events like TechMentor (http://www.techmentorevents.com) and Windows Connections (http://www.winconnections.com).

How to Use this Crash Course

I suggest that you tackle a single crash course item each day. Spend some time practicing whatever examples are provided and trying to complete tasks that make sense in your environment. Don't be afraid to fail: Errors are how we learn. Just do it in a virtual environment (I recommend a virtualized domain controller running Windows Server 2008 R2) so that you don't upset the boss! Each "Day" in this crash course is designed to be reviewed in under an hour, so it's a perfect way to spend lunch for a few weeks. This book will be published in five-day increments, so each chapter corresponds to a single week of learning.

By the way, this crash course isn't intended to be comprehensive—I already co-authored *Windows PowerShell v2: TFM* and don't intend to re-write the same book here! Instead, this is designed to get you up and running quickly with the *most crucial elements* of the shell. I'm skipping over a lot of stuff to get right to the really good bits, and in some cases, I may gloss over technical details simply because they don't contribute to speedy understanding of the most important things. You should obviously keep exploring; in my blog (linked earlier), for example, I go into a lot of these little details in one short article at a time. You'll also find PowerShell video tips on http://nexus.realtimepublishers.com, and those can help you embrace some of the more-detailed things that I might skip here.

Pre-Requisites

PowerShell v2 comes preinstalled on Windows 7 and Windows Server 2008 R2; it's available as a free download (from http://download.microsoft.com) for Windows XP, Windows Server 2003, Windows Vista, and Windows Server 2008. Be sure to download the right version for your operating system (OS) and architecture (32- or 64-bit). You'll need, at a minimum, .NET Framework v2 installed. Ideally, get the latest version of the Framework, or at least v3.5 Service Pack 1, because that enables maximum Windows PowerShell features.

Windows Server 2008 R2 ships with a number of PowerShell modules that connect PowerShell to Active Directory (AD), IIS, BitLocker, and other technologies. Generally speaking, these modules will only run on Windows Server 2008 R2 or, if you install the Remote Server Administration Toolkit, on Windows 7. There's a trick called *implicit remoting* (I'll get to it) that lets you access these cmdlets from older OSs that have PowerShell v2 installed, but you'll need at least one Win7 or Win2008R2 machine on your network to *host* the cmdlets for remote use.

Week 1, Day 1: Commands, Cmdlets, and Aliases

PowerShell is *not,* first and foremost, a scripting language. It's a shell, not unlike Cmd.exe. It's written in .NET instead of C++ or something, but in the end it's a text-based command-line window. You type commands, hit Enter, and they run—and you see results.

Go on, try it. Run **Ping**, **Ipconfig**, **Net Share**, or whatever other commands you may know. They'll work. Those three specifically are *external commands,* meaning they exist as standalone .exe files. PowerShell also has native commands, which are called *cmdlets* (pronounced "command-lets"). The difference with these is that they *only* run within PowerShell; you can't get to them from Cmd.exe or from Explorer or anywhere else. Examples include **Dir**, **Cd**, **Del**, **Move**, **Copy**, and **Type**. Yep, those look just like command names you're probably familiar with. You can try **Ls**, **Cat**, and **Cp** while you're at it, because those will all work, too.

But they won't work in quite the same way that you're used to. Those are actually *aliases,* or nicknames, to PowerShell cmdlets. The real cmdlet names are things like **Get-ChildItem, Set-Location, Remove-Item, Move-Item, Copy-Item,** and **Get-Content.** The aliases exist to give you something a bit easier to type that corresponds to the MS-DOS-style command names that you're probably familiar with. These new cmdlets—and their aliases—work similarly to those old-school commands, but they get there in a different way. For example, try running **Dir /s** in PowerShell; you'll probably get an error. Run **Help Dir** and you'll see why you got an error: There's no /s parameter.

In PowerShell, parameters all begin with a dash (-) not a slash (/), and parameter names tend to be full words, like *recurse.* So running **Dir -recurse** will work just fine. Actually, you don't have to type the full parameter name; you only need enough so that the shell can uniquely identify it. **Dir -r** will probably work fine. And no, you can't build an alias that would make "Dir /s" work; aliases are just a *nickname* for the cmdlet name. Aliases don't have any effect on the parameters of the cmdlet.

That **Help** command is going to be your new best friend—or it had better be, if you plan to master PowerShell. Run **Help *** for a list of all help topics; notice the numerous "about" topics that don't relate to a specific cmdlet but instead cover background concepts. That's your manual (in fact, you can use the Unix-style alias **Man** instead of **Help** if you prefer). Run **Help *service*** to see everything that has to do with services, or **Help *user*** to see if there's anything in there to deal with user accounts.

While we're at it, notice the cmdlet names: They have a specific naming pattern, consisting of a verb, a dash, and a *singular* noun (it's **Get-Service**, not "Get-Services"). The verbs come from a strictly-controlled set: It will always be **New** and not **Create,** and you'll never see **Delete** in place of **Remove**. These consistent verb names, combined with common nouns such as Service, Process, EventLog, and so forth, make it easier to guess at a cmdlet name. Can you guess the cmdlet provided by Exchange Server to retrieve mailboxes? **Get-Mailbox**. What cmdlet in the AD module might retrieve a user account? **Get-ADUser**. Yeah, sometimes you'll see a product-specific prefix, such as "AD," attached to the noun. That helps distinguish it from other kinds of user accounts that might exist in your environment.

Try running some single commands. Stuck for ideas? Run **Help *** to get a list of help topics, which will include all the cmdlets, then run **Help *cmdlet-name* -example**. Adding the **-example** parameter retrieves a list of examples for that particular cmdlet—very handy!

Week 1, Day 2: Output

Running **Dir** and so forth certainly produces text on the screen. But let's look at why text is bad.

In the Unix world, everything is text based. Run a command that generates a list of running processes, and you'll see a text list laid out as a columnar table. A Unix admin looking for a particular process name might pipe that text to a command like Grep, telling it to filter out entire rows based on the contents of columns 8 through 16, which contained process names. That kind of text parsing has been a common task in most shell-based administration. It's tough because it requires exactitude—and often a lot of experimentation—that goes beyond the actual administrative task. PowerShell doesn't work that way.

Instead, PowerShell cmdlets produce *objects,* which are essentially a specialized data structure in memory. Instead of putting information into tables and lists, information goes into this specialized structure. The benefit of this structure is that you can ask the shell for a single piece of information, and it can instantly retrieve it without you having to know exactly how the structure is built. In other words, you don't ask the shell to look at the contents of columns 8 through 16; you just ask it to look at the process' names. The shell knows which bit of the data structure contains the name so that you don't *have* to know.

This turns out to be pretty powerful, as you'll see shortly. But you might ask yourself why PowerShell cmdlets *still seem to produce text.* Well, that's because the shell knows that you, poor human being that you are, can't comprehend the wondrous data structures stored in your computer's memory. So when the shell finishes running commands, it converts all those objects into text-based tables and lists for you. Essentially, it retrieves *some* of the objects' attributes from the in-memory data structure and dynamically constructs a text-based table or list.

You can actually exercise a tremendous amount of control over this process, or just sit back and let the defaults take over. If you aren't liking the defaults, there are really three steps you need to take.

Step 1: Find Attributes

Every object that the shell works with has numerous attributes or *properties.* For example, a process has properties for its name, ID, memory consumption, and so forth. To see a list of them all, *pipe* the object to **Get-Member** (which has an alias, **Gm**). For example, **Get-Process | Gm** will show you the properties of a process; **Get-Service | Gm** will show you the properties of a service.

> **Hint**
>
> Any cmdlet that uses **Get** as its verb will usually produce some kind of object that can be piped to **Get-Member**.

Make a note of the properties that you think you'd like to see. Just write down their names, for now.

Step 2: Pick a Layout

PowerShell offers three default layouts: tables, lists, and a wide list. Decide which one you think will work for your needs. Keep in mind that tables can only hold so many columns without truncating information, so if you've selected a LOT of properties, a list might be appropriate. A wide list only displays a *single* property—something else to keep in mind. Once you've chosen a layout, you'll pipe your objects to it: **Get-Process | Format-List**, for example, or **Get-Service | Format-Wide**, or **Get-EventLog Security -newest 20 | Format-Table**. The aliases for those Format cmdlets are **FL, FW,** and **FT**, respectively.

> **Hint**
>
> The shell isn't case-sensitive about cmdlet or alias names. FT is the same as Ft, ft, and fT.

Step 3: Add Your Properties

By default, the shell decides what properties are shown in a table or list, and it defaults to the Name property for wide lists. Customize that by just providing a comma-separated list of properties: **Get-Process | FL Name,ID,VM,CPU**. If you just want every property shown, use * for the property list: **Get-Service | FL ***.

Additional parameters of the Format cmdlets enable further customization. See if you can answer these questions:

- If you use a table with only a couple of properties, such as **Get-Service | FT Name,Status**, you'll notice a lot of unused space on the screen. What parameter of **Format-Table** might eliminate that extra space and instead automatically size each column for its contents?
- If you include too many columns, **FT** may truncate their contents. What parameter would instead force it to word-wrap that information?
- **FW** defaults to two columns. How can you have a list of four columns?

Format, Then You're Done

A trick about the Format cmdlets is that they *consume* whatever you pipe into them. They output a special kind of formatting instruction that really only makes sense to the shell itself. Try running **Get-Process | FT | GM** and you'll see what I mean. The practical upshot of this is that a Format cmdlet will almost always be the *last thing* on the command line. "Almost always?" Yes—the one type of cmdlet that can understand Format output is an Out cmdlet: **Out-Host, Out-Printer, Out-File,** and so forth. In fact, **Out-Host** is the one used by default in the console window, but you can pipe formatted output to the other Out cmdlets to redirect output to a printer, a file, or elsewhere:

- If you direct output to **Out-File**, the file width defaults to 80 characters. What if you wanted to pipe out a much wider table of information? Is there a parameter that would let you modify the logical width of the file?
- What parameters allow you to specify a destination printer when piping output to a printer?
- Can you pipe output directly to an Out cmdlet without using a Format cmdlet, such as **Get-Service | Out-File test.txt** ?

Week 1, Day 3: The Pipeline

You've already started piping stuff from one cmdlet to another, so it should come as no surprise that PowerShell cmdlets run in a *pipeline.* Essentially, a pipeline is just a sequence of cmdlets:

Get-Service | Sort Status -descending | Format-Table -groupBy Status

The pipe | character separates each cmdlet. Each cmdlet places things into the pipeline, and they are carried to the next cmdlet, which does something with them. Then that cmdlet places something into the pipeline, which carries it to the next cmdlet…and so on. This can create some pretty powerful one-line commands, and thanks to the shell's cmdlet naming syntax, they can be pretty easy to figure out. For example, consider this pseudo-command:

Get-Mailbox | Sort Size -descending | Select -first 10 | Move-Mailbox Server2

That's not the exact correct syntax, but hopefully it conveys the power of the shell's cmdlet interaction. The trick is that every cmdlet gets to decide what kind of input it will accept. In other words, you can't just pipe anything to anything. This would make no sense:

Get-ADUser -filter * | Stop-Service

There's no reason why piping a user account to **Stop-Service** should make sense, and in fact it won't work. So how can you tell what a cmdlet is willing to accept as input from the pipeline? There are (as will become a theme in this crash course) three steps.

Step 1: Determine Your Output

First, cmdlets that produce output are, as we already learned, producing *objects*. The thing is, not all objects are built the same. A process looks very different from a service, for example, which is entirely different than an event log entry or a user account. So, each object has a *type name*, which simply describes the kind of object you're looking at. Piping objects to **Get-Member** reveals their type name. Run **Get-Service | Gm** and see if you can find the type name. Go on, I'll wait.

…waits…

It's a ServiceController, right? You can often just take the last segment of the type name. Now there's just one trick: All objects are technically of whatever type they are *and* they are the more generic "object" type. That's like saying you're a *Homo sapien*, which is a very specific type name, and that you're also an *organism*, which is much more generic. "Object" is just a very generic classification for object types.

Ok, so now you know what you have as your cmdlet output: A generic "object" as well as some more specific type name. Now, what can you do with it?

Step 2: Find Matching Input Types

Run **Help Stop-Service -full** (you'll find that the -full help is often the most useful). Start looking at the breakdown for each parameter. Notice how each parameter has the option to "Accept pipeline input?" It's False for many of them. In fact, for **Stop-Service**, the first one that's True is the -inputObject parameter. More specifically, it accepts pipeline input ByValue, it says in the help. Looking at the parameter definition, you'll see that the *type* of object it accepts is ServiceController. Wait, where have we seen that before?

So here's how it works:

1. You pipe object(s) from one cmdlet to another.
2. The receiving cmdlet looks at the *type name* of the incoming objects.
3. The receiving cmdlet looks to see whether any of its parameters will accept pipeline input ByValue for *that* type name.
4. If it finds one (and there will be zero or one, but not more), the input objects are "given" to that parameter.

So that's why this works (and it'll crash your machine, so don't run it):

Get-Service | Stop-Service

It works because **Get-Service** produced ServiceController objects. Those were piped to **Stop-Service,** which quickly realized that the -inputObject parameter was willing to accept objects *of that type* (which is what ByValue means). So those services were handed off to the -inputObject parameter, specifying the services that should be stopped.

Reading that help file a bit more, you'll notice that -Name also accepts input ByValue. Its value type is String, meaning if you pipe in a string of characters, they'll be attached to the -Name parameter, specifying the service(s) to stop:

"BITS" | Stop-Service

"BITS","TrustedInstaller" | Stop-Service

> **Tip**
> When you make a comma-separated list of values, PowerShell treats them as a single group, so those two values are piped in as a unit. Because they're both of the String type, they'll both attach to the -Name parameter, and both services will be stopped.

And how did we know that "BITS","TrustedInstaller" were strings? By using **Get-Member**, of course!

"BITS","TrustedInstaller" | GM

They're clearly identified as a System.String (just "String" for short) by the output of **Get-Member**.

Step 3: When Types Aren't Enough

Now, the shell isn't super-smart. It will *try* to do stuff that doesn't make sense, if *you* tell it to. For example, consider this:

Get-Process | Stop-Service

Makes no sense, right? Well, let's look at it from the shell's point of view. **Get-Process** produces objects that, according to **Get-Member**, are of the System.Diagnostics.Process type. Great. Looking through the help for **Stop-Service**, I don't see any parameters that will bind a Process object ByValue; I also don't see any that would bind the more-generic "object" ByValue. So accepting pipeline input ByValue will fail.

But the shell has a backup plan: Accepting pipeline input ByPropertyName. For **Stop-Service**, you'll see this only on the -Name parameter. What does this mean?

1. You pipe object(s) from one cmdlet to another.
2. The receiving cmdlet looks at the *type name* of the incoming objects.
3. The receiving cmdlet looks to see whether any of its parameters will accept pipeline input ByValue for *that* type name.
4. If it doesn't find one, which is the case in this example, it will look to see what parameters accept pipeline input ByPropertyName.
5. It will then attach those parameters to *the properties of the incoming object(s) that have a matching name.* In other words, if the incoming objects have a Name property, the value of that property will go into the -Name parameter of **Stop-Service** simply because the names match.

Remember, this is "Plan B," so it only goes into effect when nothing could be bound to a parameter ByValue. So we're sitting here looking at a -Name parameter that wants to take pipeline input ByPropertyName. We've given it Process objects. Do those Process objects have a Name property? According to **Get-Member,** they *do indeed!* So the Name property of the input objects will be passed to the -Name property of the cmdlet. The result is that the **Stop-Service** cmdlet will try and stop services *based on their process name.* In many cases, it will be able to do so because a service's name is often the same as the name of its process when the service is running.

So you have to be a bit careful with this business of piping objects from one cmdlet to another—sometimes it'll work better than you think, which might be worse than you want.

Week 1, Day 4: Core Cmdlets

Now that you know how to pipe stuff from one cmdlet to another, you might want to learn some of the cmdlets that can let you manipulate objects in the pipeline. I'm going to briefly introduce these and give you a quick example, but I'm going to expect you to *read the help* to learn more about them—and I'll ask some finishing questions to help encourage that independent research:

- **Sort-Object**, or its alias **Sort**, rearranges objects in memory. Just specify the property you want to sort them by, such as **Get-Process | Sort ID**.
- **Select-Object** does a lot of stuff, and you'll often see its alias, **Select**. For now, focus on its ability to grab just the first or last objects in the pipeline: **Get-Process | Sort VM | Select -first 10**.
- **Measure-Object** counts objects. If you specify a property, you can also have it average the values for that property, assuming it contains numeric values. Its alias is just **Measure**: **Get-Process | Measure vm -average**
- **Import-CSV** and **Export-CSV** read and write Comma-Separated Values (CSV) files, like this: **Get-Service | Export-CSV servicelist.csv**
- **ConvertTo-HTML** creates an HTML table. You'll probably want to write the HTML to a file: **Get-EventLog Security -newest 10 | ConvertTo-HTML | Out-File security-events.htm**

Now's the time for that independent research. How would you:

- Change the sort order of **Sort** to be descending instead of ascending (which is the default)?
- Get the *last* 20 objects from the pipeline using **Select**?
- Change the delimiter of a CSV file to a pipe | character instead of a comma?
- Display not only the average value for processing physical memory but also the minimum and maximum values and the total physical memory used?

The more comfortable you become reading the help, the more capabilities you'll find!

Week 1, Day 5: Configuration Baselines

This is the last tip for your first week of PowerShell, and it's a doozy. First up is a pair of cmdlets that read and write XML-formatted files: **Import-CliXML** and **Export-CliXML**. For example, let's export all the running processes to a file:

Get-Process | Export-CliXML baseline.xml

Now, launch a couple more processes:

Notepad
Calc
Mspaint

Now for a fun new cmdlet called **Compare-Object**, or **Diff** as us slow typists like to call him. This cmdlet isn't that good at comparing text files (remember, PowerShell kinda hates text), but it's *awesome* at comparing sets of objects. Consider this command:

Diff (Ps) (Import-CliXML baseline.xml)

Couple of fun things happening there. First, **PS** is just an alias for **Get-Process**. The really fun thing is the placement of the parameters. You see, I should really have written the command like this:

Diff -referenceObject (Ps) -differenceObject (Import-CliXML baseline.xml)

This time, I'm including the actual parameter names. But I looked in the help (as I'm sure you did), and saw that -referenceObject is *positional,* and occupies the first position. The -differenceObject parameter is also positional, and occupies position 2. With these *positional* parameters, I don't need to type the parameter name so long as I put the parameter *values* into the correct positions. Thus:

Diff (Ps) (Import-CliXML baseline.xml)

The parentheses are doing something special. Just like in algebra, they tell the shell to execute whatever is inside the parentheses first. The result of whatever's *inside* the parentheses is passed to the parameter. So the result of **Get-Process** (which is a bunch of process objects) is passed to -referenceObject, and the result of **Import-CliXML baseline.xml** is passed to the -differenceObject parameter. Basically, I'm comparing two sets of processes: the current set and the set that I had exported to a CliXML file earlier.

The results are horrible. Oops. That's actually because *everything about processes is constantly changing*, including their memory use, CPU use, and so on. I'd do better to just compare a single, unchanging property—like name:

Diff (Ps) (Import-CliXML baseline.xml) -property Name

Ah, there are some results. I can now see which objects were present in the left side (the current processes) but not in the right (the baseline). So this is a difference report of my current configuration versus my baseline configuration. Imagine what other types of objects you could export and compare in this fashion!