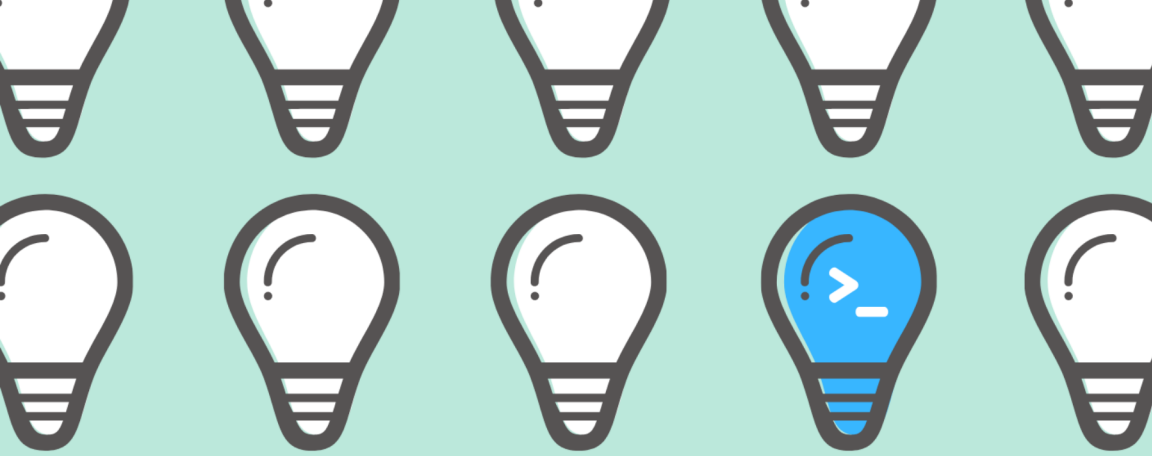
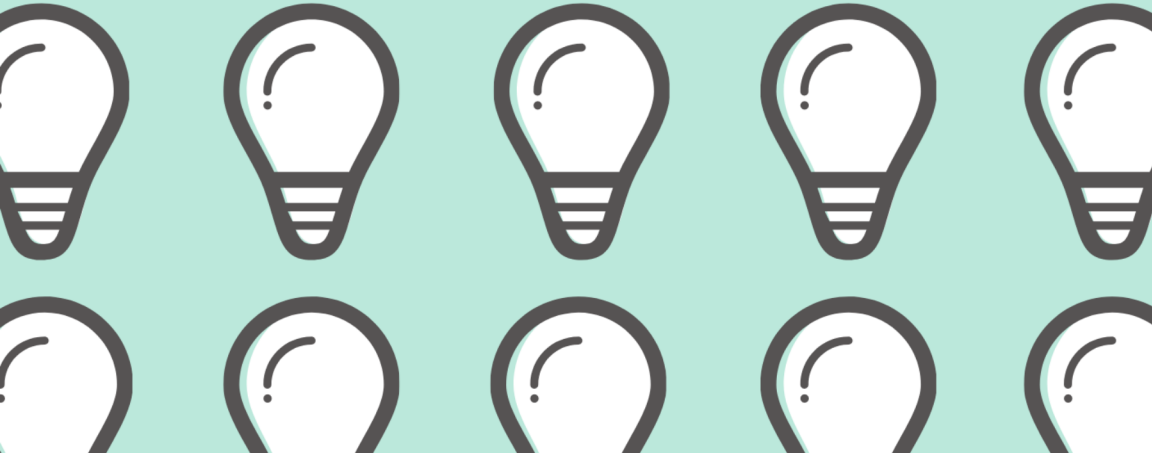


DON JONES

SHELL OF AN IDEA

the untold history of PowerShell



Shell of an Idea

The untold history of PowerShell

Don Jones

This book is for sale at http://leanpub.com/shell-of-an-idea

This version was published on 2020-06-11

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Also By Don Jones

The DSC Book

The PowerShell Scripting and Toolmaking Book

Become Hardcore Extreme Black Belt PowerShell Ninja Rockstar

Don Jones' PowerShell 4N00bs

Instructional Design for Mortals

How to Find a Wolf in Siberia

Tales of the Icelandic Troll

PowerShell by Mistake

The Culture of Learning

Alabaster

Let's Talk Business

Power Wave

The Never

Onyx

Be the Master: Special Edition

Be the Master

Sparks!

Superior Wave

To the PowerShell team, past and present. For all you've done, and for all you'll do, thank you.

Contents

Foreword

Nietzsche once said, "What we do is never understood, but only praised or blamed." There are people that praise PowerShell and there are people that blame PowerShell, but few people understand PowerShell. One of those few is Don Jones. It is hard to overstate Don's importance in the PowerShell story; he is the proverbial "First Follower." (Do yourself a favor and spend three minutes searching for a video called "First Follower: Leadership Lessons from Dancing Guy" and watch it now. No—I mean "now." I'll wait.)

I met Don in a bar in Las Vegas. He told me he wrote books—"really fast and really good." He said he understood IT pros, that they were going to love Monad, but that it would take 10 years before it became mainstream. I told him he was wrong and explained all the things we were doing to avoid the "crossing the chasm" problem. As I recall, his response was "Ok. Sure." If you've never talked with Don, it is hard to convey his ability to politely tell you that you are full of it, but it isn't worth his time or effort to explain it to you. If you hear "OK" *or* "Sure" you are really in the weeds, but "Ok. Sure." is a clue. You can guess where this story is going. In the end, Don was right—about that and so many other things along the way. It is a good thing I ignored him because if I had a clue about the difficulty that was in store for us on the path to take the great ideas in Monad and ship it as PowerShell, I would have ordered us another round and moved on to something else.

In *Shell of an Idea,* Don tells some of the many stories behind the creation of PowerShell and its design. It is a story of a group of amazing engineers struggling to forge a whole suite of new technologies into a coherent experience—all the while fighting a multi-year game of internal politics whack-a-mole. In case it is not clear—we were the moles. Microsoft's embrace and mastery

of GUIs brought it such overwhelming success that it was found to be a monopoly. So when I came to the company talking about the importance of command line interfaces and programmatic shells… well you can imagine how well that went over. The only reason we were funded to do a shell was to compete with Linux and even then, it was deemed so unimportant that I had to take a demotion to work on it full time.

The world has an unfortunate habit of giving all the credit to a single hero. The reality is that big things happen because of teams. So it is with PowerShell. I came up with several of the core foundational concepts and architectural principles, but PowerShell is unequivocally the product of a team of awesome engineers. Bill Gates used to say that Microsoft was great at finding and hiring the world's best software talent but failed at getting their IQs to add up. I am most proud I was able to create an environment that allowed a group of some of the world's best engineers to ship their ideas in a way that their IQs added up. This book is the story of those engineers, their ideas, and the messy path to adding it all up and shipping in an environment actively trying to kill the project at every step along the way.

Jeffrey Snover
May 2020

Introduction

My history with Windows administrative automation goes back a long way, at least to my 2003 book, *Managing Windows with VBScript and WMI* (Addison-Wesley). A bestseller of the time, it put me on the map as someone who spoke about, taught, and wrote about Windows automation. It helped drive my first Microsoft MVP Award recognition in 2004 and made it natural for me to jump into Windows PowerShell–then called Monad–when it hit the scene in 2005.

I was honored to co-present with Jeffrey Snover at TechEd Europe 2006 in Barcelona, where Microsoft formally launched PowerShell and introduced it to the world. I wrote the first published book on PowerShell, *Windows PowerShell: TFM* (SAPIEN Press) and have in total written or co-authored close to a dozen books on PowerShell. *Learn Windows PowerShell in a Month of Lunches* (Manning Books) remains a go-to bestseller for newcomers, and *PowerShell In Depth* (Manning) is still a top reference for PowerShell admins. I co-founded PowerShell.org, launched the PowerShell + DevOps Global Summit with my partner, and was named "PowerShell's First Follower" by Jeffrey Snover at the first Microsoft Ignite events in Chicago. I coined the terms "Toolmaker" and "Toolmaking" within the PowerShell world, and have been an advocate for strong practices and patterns. I even substituted for Jeffrey Snover as a speaker at the TechEd North America 2007 conference. Suffice to say that PowerShell has been an *enormous* part of my life and career.

Over the years, I've made a ton of good friends in the PowerShell community, which is easily one of the friendliest and most down-to-earth group of technologists I've ever met. My office has a small collection of the thoughtful, tongue-in-cheek mementos they've

given me: a CIA challenge coin, a Lego minifig of myself, a beer stein with PowerShell and Disney's Figment character etched into it, and more. My career has taken me away from the day-to-day engagements with both the technology and that audience, but they've both been such a huge part of my life and career that I can never step fully away.

In fact, that was the genesis for this book: I just can't let PowerShell go. It's not only been important to me, but it's also been hugely important and impactful to so many people in the industry. And yet PowerShell almost never happened. In fact, it almost never happened more than once. Were it not for a team of passionate visionaries willing to make the occasional possibly-career-limiting moves, PowerShell–and all the positive impact it's created–wouldn't have existed. PowerShell might have just been a port of Unix' KornShell, or it might have just been a WMI querying tool. Or it might not have been a thing at all.

There's a lot of untold story under the shell, and it's a story I wanted to tell. Much of PowerShell's core team have moved on to other teams or even to other companies. Nobody's getting any younger. I felt it was time to capture their stories and the shell's story while I could still track everyone down. Some bits of the story have been told at conferences or in other venues, but it's never been pulled together into one place–and it's never been told in its entirety.

If you've worked with PowerShell, then *Shell of an Idea* should provide some fascinating backstory to it. If you haven't worked with PowerShell but you're at least conversant with computers and systems administration, then you're in for a real treat. As much as possible I've tried to wrap context around the stories so that you can see where they fit into the world, and what PowerShell struggled against and sought to solve.

I've also included a number of quotes, solicited through my blog at DonJones.com[1]. These may seem out of context as I present them,

[1] http://donjones.com

but they're intended to provide some background for the people that PowerShell has impacted the most. I've edited these as lightly as possible for length and clarity because I feel that the effect of the shell's story is just as important as the story itself. Here's one example:

> PowerShell changed my life… I realize that such a statement may seem exaggerated, but every Powershell enthusiast can relate in some way to the overwhelming benefits and career opportunities that learning PowerShell has given them.
>
> I had worked in an operations management position for five years and dreaded going into work every day. The stress was awful, but the monotony was worse. I was 30 years old, had a wife and two boys under two years old, and living on my single income. I was afraid it was too late to change careers and find work I actually enjoyed. However, my older brother who worked as a system administrator for a large tech company told me about how he used PowerShell in his job and loved it. He was a PowerShell enthusiast and thought I could learn it and open up an opportunity to change into the IT field. He actually gave me the book he used that helped him learn, *Learn PowerShell In a Month of Lunches*.
>
> That was about six months ago, and I was fortunate that an opportunity opened up at my current company in our IT department soon after I started learning. The little I knew at the time allowed me to get my foot in the door and gave me the opportunity to make an immediate impact and learn in a practical way.
>
> PowerShell is easy to learn, incredibly practical, and useful in most every environment. Since I started, I've scripted automated tasks that run daily, created GUI tools for our Operations department and many more

> things like interacting with web APIs, and more.
>
> I now love what I do, and I'm excited about the career and financial opportunities this new path will help provide for me and my family. I will forever be a PowerShell evangelist and look forward to continuing to gain a more in-depth knowledge, and hopefully have the opportunity to teach and train others on how Powershell can potentially change their lives as well.
>
> –Aaron

How can you not want to read more of the story of a technology that can generate that kind of feeling? "Easy to learn," "incredibly practical," and "I now love what I do;" those aren't statements we often see all attached to a single technology, right? The journey to create a product that engenders those remarks must be amazing.

It's easy, as we sit in front of our monitors and tap away on our phones, to forget that the story of technology is a story of people. It's about visionaries who see problems and try to solve them, who take on some small piece of the world and try to make it at least a little better. It can be difficult for us everyday folks to look at the end result and be impressed by it. What I hope you take away from this book, though, is that those amazing end results come in tiny, often-difficult steps. If you're willing to take those little steps and push through the hurdles, you can make just as big of a difference. The people who brought PowerShell to life are just ordinary people who shared a vision and worked hard to make it a reality.

This is their story, and I'm proud to share it. I hope you enjoy it.

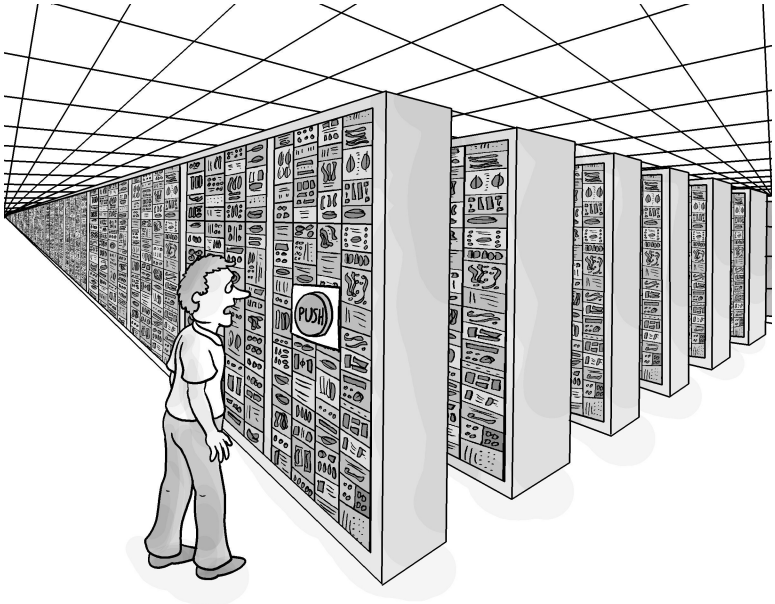
Don Jones

Cast of Characters

HISTORY

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

A Shell of a Problem



"All I have to do is push a button on each one…"

To understand the history of PowerShell and its subsequent impact, you need to understand a bit of Microsoft history. At least, a simplified version of a tiny piece of it.

Until 1993, Microsoft Windows was a desktop operating system, meaning it ran on individual computers used by individual people. Most of those computers were fairly large, beige boxes that sat on or under a desk. Laptop computers at the time were pretty primitive

and bulky compared to what you might see today. The Windows of the day was v3.1, and it couldn't even participate fully in the more primitive computer networks of the day. Home users tended to rely on dial-up services like America Online rather than the always-on Internet we take for granted these days. That Windows was shortly succeeded by Windows for Workgroups, the first fully network-capable Windows operating environment and the first Windows arguably created with business scenarios specifically in mind. But even then, Windows for Workgroups could really only *join* a network; there was no version of Windows capable of *hosting* a network. Networks of the time were hosted by a server, with the most common servers running either a product called NetWare 3.1, produced by a company called Novell, or a variant of the UNIX operating system.

Unix (as it is more commonly styled nowadays) had been around for a while by then, primarily at military research facilities and large universities but also running on the enormously expensive mainframe[2] and midrange computers used by the largest enterprise companies. It featured robust networking, and in fact provided the underpinnings of what would become today's Internet. Unix at the time was both incredibly complex to use (compared to Windows) and incredibly expensive; a single Unix computer could easily represent an investment in the tens of thousands of dollars, and required specialized training to operate.

NetWare was common in small- and medium-sized businesses. It was fairly complex to install and manage, and it included its own protocols for network communications. NetWare could run on smaller, cheaper computers, and it was somewhat simpler for network administrators to learn to use.

Perhaps most importantly, many decent-sized companies didn't have a network at all; they had a midrange computer like an IBM

[2]To be fair, these were technically "minicomputers," but the word "mini" in our modern context completely understates the size and cost of the things. I'll incorrectly refer to them as "mainframes" just to help set the right tone.

AS/400[3] that handled all of the business's computing. Users connected to these midrange computers through "terminal emulation" cards that plugged into their bulky desktop computers, and by using applications that ran on Windows. Essentially, these terminal emulators turned each desktop into a "dumb monitor" (literally not much more than a television hardwired to the midrange machine) capable of sending keystrokes and displaying whatever the midrange sent back. Truly huge companies often had a mainframe that was basically just a giant equivalent to a midrange, such as the Digital Equipment VAX line of computers. Again, people often connected to these via "dumb terminals" that were wired directly to the computer.

The point is that in the early 1990s computer networking wasn't a big thing for most businesses, and you needed specialized personnel to build and run a network if you did have one. It wasn't like today, where your smartphone can join a wireless network with a couple of taps, and you can set up your own home WiFi network just by plugging in a box and running a setup application on your phone.

So, the landscape of the time: big companies had a single enormous computer, and perhaps a bunch of pricey Unix machines [4] to supplement it. Smaller companies maybe had a few Windows or MS-DOS desktop computers connected to a NetWare server. Without networking, it was a pretty big pain to own more than a handful of computers, and so nobody really bothered. Even a giant tech university like MIT could probably have counted up all the computers they owned without much effort–a marked contrast to today's world, where most of us have a pocket computer (called a smartphone), maybe a laptop or tablet, maybe a wrist computer (smart watch), a gaming machine, and more. In the early 1990s, a single human being didn't run around owning a half dozen computers. Today, we probably can't accurately count how many

[3] These are referred to as IBM's System i computers now, after having briefly been rebranded to iSeries.

[4] Unix workstations were sized more like a modern PC, but still cost thousands and thousands of dollars.

of them are on the planet.

In 1993 Microsoft launched Windows NT, its first truly business-grade edition of Windows. Most critically, it launched Windows NT Server, which was their first operating system capable of hosting a robust network. Windows NT wasn't yet anywhere near the class of a midrange or mainframe operating system, but it could certainly compete with Novell NetWare as the centerpiece of a small- or medium-sized business network. Even large companies started buying Windows NT, but not to replace their AS/400 or VAX machine, mind you! In most cases, Windows NT snuck into the environment, purchased by a single department that was tired of not getting the computing resources they wanted from their company's midrange or mainframe. Compared to NetWare, Windows NT was easy to set up and straightforward to operate. It adopted the same graphical user interface that had made Windows so popular on the desktop. It was an easy sell: "add a file or print network as easily as opening up your word processor!"

This was a truly critical point in computing history: suddenly, *everyone* could have a server capable of hosting shared files, providing shared printing services, and other basic tasks. Servers were cheaper, and Windows NT made it easy for almost anyone to set up a network. Network computing had been democratized and commoditized, and almost every business wanted in on it. The number of servers installed in the world's companies began to proliferate *markedly*. Microsoft followed Windows NT (initially versioned 3.1) with Windows NT 3.51 and then Windows NT 4. Dropping the "NT," they then followed with Windows 2000 Server and then Windows Server 2003.

By 2003, Windows–and the many applications that ran on it–had grown up a *lot*. It was fully capable of taking on numerous enterprise-class workloads such as messaging, collaboration, databases, and more. Microsoft–themselves an AS/400 company for much of their history–committed to running their own company on Windows Server, and had made the jump by the 2003-2005

timeframe.

Here's the advantage of moving your computing from giant, million-dollar AS/400 and VAX systems to smaller, commodity servers running Windows Server: it's cheaper. Sure, a single Windows Server might not be able to do messaging *and* filing *and* printing *and* databases *and* whatever else, but you could buy three dozen Windows Server machines for *far* less than the price of a single AS/400. In the late 1990s, as the public Internet and World Wide Web came online and proliferated, people quickly realized that having *more, cheaper* servers was often better than having one expensive one. Want to stand up a website that can survive the traffic when your company gets mentioned on Oprah? Have that website served up by an entire building full of cheap web servers, each taking a small part of the overall workload. Today's cloud, in the form of Amazon Web Services, Microsoft Azure, Google Cloud, and others, exists entirely around the concept of "lots of cheap computers."

But here's the downside of all those servers: someone has to manage them. They need to be configured to work properly, and they need to *stay* configured. They need periodic security updates and bug patches. Patching one server back in 1994 was no big deal, but patching a building containing thousands of servers in 2003 became an entirely different thing.

And that's where Windows Server ran into trouble.

Sure, Windows Server was "as easy to run as the desktop you already know and love," but the ability to click through a wizard to install a patch didn't scale well. Having to run through the same wizard on ten computers–clicking Next, Next, Next, Next, Finish on each one–might be acceptable, but doing it for *a thousand* computers? Not so much. Even normal business processes like bringing on a new employee became a massive chore. In the old mainframe days, a new employee would get an account set up on the mainframe and a phone extension assigned to their desk,

and that was pretty much it from a tech perspective. Now? New employees needed a user account, an email mailbox, a folder to store their documents in, access to the company applications, and more. Completing all those provisioning tasks manually could take hours, if not *days,* and organizations big and small started to feel the pain.

Simply put, Windows started to bog down in terms of the labor it required, and so computers running Linux started giving Microsoft trouble in enterprise environments.

Linux, an open-source operating system that's based on, and largely compatible with, Unix was created mainly in response to the high cost of Unix operating systems. Linux is free to use in most cases, and it runs on the same cheaper commodity hardware that Windows Server can run on. Linux is a lot harder to administer, though. It favors cryptic commands typed into the computer, versus Windows' pretty icons and Next-Next-Finish wizards. The upside of Linux is that once you *do* learn to manage it, it's almost as easy to manage a hundred computers as it is just one. Instead of typing the commands into each computer yourself, you simply type them into a text file not unlike a word-processing document, and tell all of your computers to run that text file. The text file becomes a "script," like you might hand out to actors in Hollywood, with each computer reading their lines so that you don't have to.

Microsoft started struggling to close deals in large companies due in part to the perception that managing large batches of Windows Server machines took more labor than doing the same thing with cheap Linux machines. Windows Server cost money too, and organizations didn't often see the point in spending a lot of money on something that was painstaking to administer and maintain.

Nowhere was Microsoft's problem more evident than in a leaked white paper from August 2000. This was shortly after Microsoft had acquired Hotmail, a free email service hosting more than 100 million accounts and running entirely on Unix servers running the

FreeBSD variant of Unix. Microsoft employees were tasked with performing an analysis of what it would take to move from Unix to Windows as the base of Hotmail, and the results weren't rosy.

"It's easy to look at a UNIX system," the paper's author says, "and know what is running and why. Although its configuration files may have arcane (and sometimes too-simple) syntax, they are easy to find and change." But with Windows, "Some parameters that control the system's operation are hidden and difficult to fully assess. The metabase is an obvious example. The problem here is that it makes the administrator nervous; in a single-function system he wants to be able to understand all of the configuration-related choices that the system is making on his behalf." And then a real strike against Windows, from a manageability perspective: "GUI operations are essentially impossible to script. With large numbers of servers, it is impractical to use the GUI to carry out installation tasks or regular maintenance tasks." For Unix? "Most configuration setups, log files, and so on, are plain text files with reasonably short line lengths. Although this may be marginally detrimental to performance (usually in circumstances where it doesn't matter) it is a powerful approach because a small, familiar set of tools, adapted to working with short text lines, can be used by the administrators for most of their daily tasks. In particular, favorite tools can be used to analyze all the system's log files and error reports."

The paper goes on to really lay out why Unix' approach was better: "Over the years, UNIX versions have evolved a good set of single-function commands and shell scripting languages that work well for ad-hoc and automated administration. The shell scripting languages fall just short of being a programming language (they have less power than VBScript or JScript). This may seem to be a disadvantage, but we must remember that operators are not programmers; having to learn a block-structured programming language is a resistance point." Furthermore, "PERL… is more of a programming than scripting language. It is popular for repeated, automated tasks that can be developed and optimized by senior

administrative staff who do have the higher level of programming expertise required."

In other words: Windows was horrible at administrative automation, and VBScript wasn't helping. (As an aside, the Hotmail migration to Windows actually went off really well, with nobody realizing for months that the back-end migration had even happened.)

As the paper hints, Microsoft had first countered Unix' robust shell scripting history with Visual Basic Script, or VBScript. This scripting language was intended to let you manage Windows Server by typing commands into text files, just as Linux could do. But it was a programming language, not a shell scripting language. The barrier to entry for VBScript was high. You couldn't just run a command and then paste it into a script for long-term use; you had to *write code.* And the fundamental architecture of Windows Server wasn't the same as that of Linux, and it impacted Windows' ability to have an *effective* scripting language.

Linux, as with Unix before it, is a "text-based operating system." That's what the Hotmail white paper was alluding to: everything that tells the server how to behave–its *configuration*–is basically lines in text files. The operating system's means of communicating with other devices is similarly simplistic. Changing a text file is easy. Most Linux administrators quickly figured out the small number of tools that enabled them to change text files on hundreds of computers at once, effectively reconfiguring those computers with a single keystroke if needed. Sure, those tools were cryptic, with incomprehensible names like "grep," "sed," "awk," "cat," and more, but you only needed to learn them once. Once you did, the world of Linux administration was open to you. Learn a little, and you could do a lot.

Windows, on the other hand, is an API-based operating system. Each component inside Windows defines a set of interfaces that you use to tell it what to do. When you click an icon in Windows,

one bit of software uses those interfaces to tell another bit of software to do something, like open a file, send a message, or whatever. Automating Windows administration, then, is less about changing text files and more about some pretty serious computer programming. These interfaces are (for the most part) documented, but that documentation presumes you're an experienced software engineer. Sadly, the people hired to manage computer networks tend not to be experienced software engineers. In the Windows world, they were used to clicking icons, not coding programs of a hundred lines or more. VBScript helped a *bit,* but VBScript couldn't access all of the APIs needed to make Windows do everything it did. Eventually, someone using VBScript would run into a situation they simply couldn't handle, leaving them to go back to clicking icons to make stuff happen.

Worse, Windows' various APIs had all been created by developers who never expected anyone but themselves to use those APIs. Some APIs required you to use low-level programming languages like C or C++, while others could use more accessible, higher-level languages like VBScript. Still others were best used from Microsoft's .NET Framework, a set of APIs released in the late 1990s to make software development faster and more consistent. But .NET Framework didn't cover everything a server administrator might need. So this wasn't just a matter of Windows being based on APIs; it was also about Microsoft having changed their minds over the years on how those APIs were created and used. You could be a master in .NET Framework, for example, and still be unable to deal with some of the deeper, C++-based interfaces in Windows' core.

Lest you think Microsoft had been remiss in their architecture, rest assured that's not the case. Using APIs to wall off different components from each other is not only a standard practice, it's a *recommended* practice. APIs let multiple teams of people work on different subsystems without interference or dependencies on other teams. One team can do whatever they like with their piece

of software, knowing that all they need to do is publish an interface through which other teams could access whatever was needed. It's a bit like the radio in your car: you might not know how a radio works, but you can use the interface provided to change stations and adjust the volume. The back side of the radio sports another interface that lets the car supply power, antenna signals, and so on to the radio. If you buy a Ford truck, you're welcome to swap out the Ford radio for a Pioneer one, provided the Pioneer radio can support the same interface that your truck expects of a radio (which is why adapter cables exist).

A problem with interfaces, though, is that they can only give you the things their developer anticipated you would need. If your truck radio has no interface for taking a satellite radio signal, then there's nothing you can do about that, no matter how many adapter cables you have. And that's where Windows administrators often found themselves: if the developers of some Windows subsystem hadn't *anticipated* an administrator needing to do something, then the subsystem's interfaces wouldn't make it possible, and the administrator was out of luck.

And here's another problem Windows had: many of the teams who built Windows' various components assumed nobody would ever do anything other than click the pretty icons they'd created. For those components, it was essentially impossible to automate their administration because they simply had no interfaces through which to do so. It was, frankly, a bit of a mess, and it caused no end of frustration to Windows administrators who were managing a rapidly growing number of servers in their environments. This wasn't necessarily a bad decision on the part of those teams, because the *whole point* of Windows was its graphical user interface. For many of them, suggesting that people might need to administer using something other than icons and wizards approached heresy. Teams were *required* to deliver a comprehensive and easy-to-use graphical user interface. Anything else was often optional in terms of Microsoft's architecture standards, and optional things tend to

fall by the wayside when resources get tight and timelines get short.

Linux, to be clear, also *technically* relies on APIs. It's just that nearly every piece of Linux adopted "put stuff in text files" as their interface. If you want to reconfigure a piece of Windows–say, you need to add a user account to the company directory–you have to hope the directory subsystem's APIs offer a way to do that, and then you have to learn what data structure to pass them to make them do it. With Linux, you often just add a line to a text file. Notably, many recent Microsoft products have shifted to this text-based approach. With Microsoft Azure, for example, a specially formatted text file can be used to make Azure do almost anything.

But in the early- to mid-2000s, complex APIs still ruled Windows Server. It seemed like all the bits were there to automate *most* Windows administration, but they were scattered over a half dozen largely difficult and sometimes-incompatible languages and technologies. It's like going into an auto shop and realizing you need a set of metric sockets for the frame of the vehicle, Imperial sockets for the body, a torch welder for the roof, and a magic wand for the engine. If you can master *all* of the different tools then maybe you're fine, but it's a lot to wrap your head around.

This problem would have been solvable: you just need your Windows server administrators to be *really* broad in terms of the technologies they can support, and *really* fast at learning new things. Basically, if your admins are capable of being ersatz developers, you're fine. Except that wasn't the sales pitch Microsoft had been making for a decade. "Administer your network as easily as you use your own desktop!" had been the message, not "Learn four programming languages and spend all your time writing code!" The bulk of Microsoft's administrator audience wasn't up to speed on software programming, and in a lot of cases they weren't *interested* in learning languages like C#, C++, VBScript, or whatever else. Again, it's as if Microsoft had attracted a large audience of competent, intelligent, hardworking automotive mechanics, and then carried a nuclear reactor into the shop and said, "You can do this

too, right?" The audience was used to a certain level of consistency and abstraction that a graphical user interface affords, and they simply hadn't been prepared to have Windows' underlying inconsistencies and ugliness dumped in their laps.

Understand, too, that in 2003, Windows administrators tended to be paid markedly less than software developers with equivalent seniority, and in many cases less than similarly situated Linux or Unix administrators. The assumption that "managing Windows is easy!" was baked into their salaries, and the idea of suddenly being asked to take on a very different kind of role, without necessarily being paid more, didn't sit well.

This is the world that PowerShell (originally *Windows* PowerShell) was born into: Windows Server struggling to compete with Linux in large-scale companies, due in main part to the relative difficulty in automating Windows administration at scale. Under the hood, Windows was a hodgepodge of different interconnected systems, each one optimized for whatever its task was, and each one difficult to automate without knowing a half dozen or more different technologies and approaches.

The thing is, Microsoft had known this was a problem for quite a while, and their initial solution wasn't even aimed at Windows administrators.

Illustrating the Problem

Take the seemingly simple problem of adding a new user to a Microsoft Active Directory domain, a task that most large companies must perform several times each day.

Microsoft's first-class citizen approach was to use the Active Directory Users and Computers graphical user interface, or GUI. ADUC, as it's often called, was created as a "snap-in," or extension, to a generic GUI administration tool called the Microsoft Management

Console, or MMC. The idea with the MMC was to provide administrators with a single window–a "single pane of glass," in industry parlance–where they could do anything their jobs needed. Need to administer your company's Domain Name System? Add the DNS snap-in to the MMC. Need to do something with the Microsoft SQL Server? Add the right snap-in to the MMC. The MMC was part of Microsoft's "Common Engineering Criteria," or CEC, of the day, and it was an attempt to make all of the company's various GUI administration consoles more accessible and more consistent.

But you still had to click icons. A large company that brought on a dozen new employees or more every day could easily wind up with one or more human beings who literally did nothing but click buttons and checkboxes in the ADUC GUI. Many organizations rightfully saw that as a waste of human labor and looked for automation solutions.

But as we've learned, Microsoft was leagues away from having a cohesive automation story. In this instance, an administrator looking to automate Active Directory user creation might have to explore no less than *eleven* potential tools to see if any of them could get the job done:

- The Active Directory Services Interface (ADSI) Windows NT (WinNT), provider
- The ADSI Lightweight Directory Access Protocol (LDAP) provider–similar to its WinNT sibling, but with distinct capabilities
- Using a Csvde.exe command-line tool to import a comma-separated values file containing the new user data
- Running the Dsadd.exe command-line tool
- Using the LDAP Data Interchange Format (Ldifde.exe) tool
- Using a .NET Framework class–there were several potential ones to choose from–in a program
- Using Windows Management Instrumentation (WMI)

None of these tools accomplished exactly the same thing, although they all had overlap with each other. If you were creating a simple user account–a name and a password, perhaps–any of them might have done the trick. But companies also tend to log data like an employee's department, manager name, address, phone extension, and so on, and only *some* of those tools could do all of those. Still other attributes in Active Directory were accessible *only* from the GUI, so many administrators would spend days or weeks experimenting with various tools only to glumly return to the GUI after failing to find an automation tool that could do everything they needed.

The underlying reasons for all this were mainly political, and they were legion.

Microsoft product teams are largely autonomous, and often smaller than outsiders imagine. Although shipped as part of the Microsoft Windows Server operating system, Active Directory is its own product team, distinct from the base operating system. Teams–at least back then–tended to operate as self-contained fiefdoms, cooperating with other teams only at need, and typically only when sufficient political capital existed to compel cooperation.

Within the Windows operating system universe, including its many sub-components like Active Directory, the Common Engineering Criteria was one of the few documents that provided cross-team requirements. If the CEC said you had to provide administrative capabilities by means of an MMC snap-in then you *had* to do it, even if taking the time to do so meant sidelining some other features you'd hoped to work into your next release. Notably, the CEC in 2003 didn't touch on administrative automation at all, so it's no wonder so few Microsoft products of the time got automation right. Even when the teams *knew* they had an automation problem, they often didn't have the time or budget to address it.

A team was welcome to provide capabilities above and beyond the CEC, if they had the resources to do so. The automation tools

produced by the Active Directory team tended to focus on bulk import of users, because those bulk imports were a key scenario in migrating large enterprise customers from a competing solution. Enabling migration meant winning deals, which meant incoming revenue, so it's hardly surprising that those scenarios were the ones prioritized over automating day-to-day administration. Few technology executives of the time were sophisticated enough to consider "how will we manage this thing day-to-day" in their purchasing decisions, and–again, at the time–the ones who were tended to avoid Windows when they could.

It's worth noting that writing automation tooling isn't easy, which is another reason why few Microsoft teams committed to it. Developing an MMC snap-in *was* relatively easy: the MMC itself provided a lot of the code that was boring, such as presenting different views for data, intercepting user clicks and interpreting them as actions, and so on. The MMC was kind of a framework of functionality that was common to GUI-based administration, and so knocking out an MMC snap-in, while not trivial, wasn't a huge investment.

Nothing like the MMC existed for automation-enabled tools, though. Teams looking to create command-line tools, which could be more easily integrated into scripts, were *entirely* on their own. They had to develop their own command structure, write code to accept and interpret commands, develop output displays, and more. It's actually a *lot* of work, and given the competing priorities of the day–and the fact that the MMC was a CEC requirement–many teams simply couldn't afford the investment.

To get really specific, imagine that you're on a product team that handles Windows' Dynamic Host Configuration Protocol, or DHCP. DHCP is designed to automatically issue, track, and manage the addresses that computers need in order to participate on a network (your home WiFi router, for example, usually includes DHCP functionality so that your smart phone, laptop, and smart TV can all get on the network). In enterprise environments, critical

computers like servers often have a manually created reservation for their address so they get the same one every time they connect to the network. As a product team, let's say you've been interviewing customers and have figured out that the ability to bulk-manage reservations is really important to them. So you sit down to design a tool called "dhcpmanage." You come up with a few use cases:

- Customers might create a comma-separated values, or CSV, file in Excel that lists the reservations they want to create, and then import it by running `dhcpmanage -file reservations.csv`.
- Customers might want the tool to create a CSV file of existing reservations, and they might run `dhcpmanage -export current.csv`.
- The tool might also need to add or remove existing reservations one at a time, perhaps by running `dhcpmanage -add 192.168.13.12 -for 00:D3:32:EE:12:34:56:78`, or `dhcpmanage -remove 00:D3:32:EE:12:34:56:78`.

Let's say that's as far as you decide to go in your tool's initial release. Your team has a *lot* of work ahead of it! In addition to coding the basic functionality to add or remove reservations, you have to:

- Code the ability to read files
- Code the ability to write files
- Ensure your code can deal with an improperly formatted file or other error condition
- Write a parser that looks at what the user types and figures out which task they're trying to do
- Ensure any messages or errors your tool displays can be displayed in whatever language the computer is set to (Microsoft is a global company, and almost all tools have some level of localization)

- Run all of your code through testing to ensure it works

It's a lot of work. Even a tool of that simplicity may require a few hundred person-hours by the time it's designed, coded, tested, and made ready to ship. And the annoying part is that the work is more or less one-off, meaning that if another team needed to write a tool for *their* product, there's very little work they can leverage from other teams. Every tool is a new, start-from-scratch experience. That gets expensive.

The go-it-on-your-own approach to Microsoft product teams didn't help, either. With no framework for command-line tools in place at the operating system level, and with each team basically choosing their own destiny when it came to what they produced, the tools that *did* get produced were all but incompatible with each other. One team might produce a great command-line tool that could grab user information from a Human Resources database, but there might not be any way to link that data to a tool that could create new Active Directory user accounts. The walls between the tools reflected the political boundaries between teams.

Those walls and inconsistencies often proved to be a disincentive for Microsoft's customers. When every tool was unique, might not do *everything* the customer needed, might not play well with the other tools the customer was already using… well, a lot of customers just ignored what tools there were, because using them was more effort than it was worth. When the tools didn't get used, the product teams had a disincentive to make more tools, and so the problem just cycled and got worse.

As we've seen, it was a mess. But Unix wasn't a mess, right? So let's just copy that!

Let's Just Copy Unix

A common refrain amongst Microsoft's bigger critics was, "Why don't you just do what Unix does?" Unix, after all, was thriving in enterprises, and offered a rich set of command-line tools that enabled pretty much any kind of automation you could think of. And of course the same applied to Unix' open-source offspring, Linux.

But as we've discussed, Unix and Windows are very different kinds

of operating systems. What works for Unix won't automatically work for Windows; the two operating systems take radically different approaches to how they work, let alone how you administer them.

And let's be honest for a moment: Unix' command-line administration isn't exactly a piece of fine art. Yes, once you comprehend it, you can get the job done, and done well, but coming to comprehend it is a *huge* task.

Unix gets a lot of things right when it comes to the command-line: most of its tools are fairly atomic, which means they tend to do one thing and do that one thing well. You get one tool to change the owner of a file and another to change the permissions on the file. Atomicity is a good thing because it makes tools simpler to write and use, and makes them usable in a broader set of scenarios when an administrator is orchestrating several tools to act together.

But figuring out which Unix tool to use for a given task–heck, even figuring out what tools *exist* from the thousands that are out there–is really, really hard. Linux probably never would have taken off the way it did if it hadn't been for Google's ability to help new administrators figure out whether `grep`, `sed`, `awk`, or something else was the right tool for the job at hand. Prior to Google, you could walk by most Unix administrators' desks and find thick books containing command reference material.

And administrators needed that reference material, because learning to use one tool gave you almost no advantage when it came to learning other tools. One tool might require you to type `-m` to specify the name of a remote machine while another might want you to use `-Comp` or `-n` or `\c` or `--computer` for the same purpose. There was precious little consistency between the more than 3,300 command-line tools that existed in Red Hat Enterprise Linux, or RHEL, circa 2003, making each tool a unique and challenging experience for a new administrator.

Unix' tools–like the operating system itself–were text-based. That

means that when you ran a tool, whatever it produced was displayed on-screen as text. It's entirely possible to pass that text to another tool, which forms the foundation of automation, but it took a lot of work. For example, suppose you had a tool that could retrieve usernames from a database and another tool that could set up email accounts for those users. The first tool might list the username in the second on-screen column of data, occupying character positions 10 through 20, say. You'd have to pass, or "pipe," that data to a middleman tool that could extract just columns 10-20 before piping the result on to the email account tool. This kind of text parsing was part and parcel of every Unix admin's day-to-day life. And it was brittle: if the first tool's author revised it and moved the name information to columns 12-22, then everything you'd written would break and you'd have to go fix it all. As a result, tools were rarely revised in that way. Instead, a tool's author might just add a switch to change the output to a different format, giving you something *more* to learn and requiring that the author continue to support decades-old ways of doing things, just because changing it would probably break something else.

So Unix and Linux had some good things going for them, but they also had a lot of inconsistent, difficult cruft that had built up over the years, much of which it inherited from their decades-old Unix predecessors.

But sometimes it's actually easier just to reproduce the cruft that already exists than to think of something entirely new and better. Sometimes, "Just let us do what we've always been able to do," is all the mission calls for. Microsoft's next-generation shell *could* have simply been a faithful recreation of what had worked for so long on Unix.

In fact, it almost was *exactly* that.

Kermit

In the early 2000s, Intel–the world's largest producer of microprocessors and the biggest player in the PC architecture that most copies of Microsoft Windows ran on–had a bit of an embarrassing secret. Although Intel was famous for their Complex Instruction Set Computing, or CISC, processors, those chips were designed on a competing technology. Intel owns thousands of Sun Microsystems SPARC workstations that are based on Reduced Instruction Set

Computing, or RISC, chips and running Sun's Unix-variant operating system, Solaris. Intel wasn't thrilled with the fact that their own chips weren't powerful enough to design their next generation, and so CEO Craig Barrett started talking to Microsoft's Bill Gates about it.

The plan was to migrate Intel entirely to an Intel-based chip design platform, but more powerful chips couldn't provide the complete solution. Computers need powerful hardware, but they also need software, in the form of an operating system, that can fully leverage that hardware. Gates agreed to make the changes necessary to Windows to enable Intel's migration, and started dedicating resources inside Microsoft toward making it happen.

Nobody at Microsoft knew, of course, that Intel had already started a parallel effort to migrate their workstations to Intel-based machines running a Linux variant. Ultimately, Microsoft's efforts wouldn't achieve what they'd originally envisioned, but this is where the seed of PowerShell came into existence.

Windows NT had been designed to run multiple "subsystems," each essentially a mini operating system. The theory–never fully brought to reality, but a good theory nonetheless–was to enable Windows as a kind of super-operating system to run applications written for other operating systems. The Win32 subsystem would run Windows-native applications, a POSIX subsystem would provide basic Unix compatibility, an OS/2 subsystem would run IBM OS/2 applications, and so on. Windows integrated a "Services for Unix" application suite, which provided key interoperability mechanisms that let Windows play nicely on a Unix network.

Intel's biggest concerns for Windows boiled down to one main thing: a robust command-line shell, just as they had on their SPARC workstations. Ideally, they wanted a Unix shell that could run all the scripts and tools they'd already built for themselves. And so one of the teams Gates had funded within Microsoft was tasked with making it happen. Daryl Wray, a Program Manager in Microsoft at

the time, proposed to create an implementation of a popular Unix shell, KornShell, or ksh, to run on Windows, and he nicknamed his project "Kermit." Not after the famous frog Muppet, but after *Kermit the Hermit,* a children's book by Bill Peet about a crab. Crabs live in shells, you see, and Wray and team were setting out to create a new shell.

Wray's team wasn't trying to be overly ambitious, and they weren't trying to change the world. Implementing KornShell on Windows would simply let Intel's team run the command-line utilities that they were used to running. It was an eminently practical solution to the problem: "What is it you really need to do?" "Run these tools." "Okay, we'll make that happen."

Wray came from the Unix world and understood how Unix administrators worked. He understood their hard-won expertise with command-line stalwarts like `grep`, `sed`, and `awk`, and he understood how brittle Unix scripts could be. Moving a Unix script from one variant of Unix to another would almost always cause problems and require rewriting because the tools on different variants weren't always 100% consistent. The decision to port KornShell was made primarily because it lined up with what Intel's engineers were already doing and would ensure the least amount of breakage during a migration.

The Kermit effort lived within the Windows Client team in Microsoft. At the time, Windows Client and Server were considered distinct operating systems with some shared components. Because Kermit was intended to address a client-side issue, namely the existence of a command-line shell on chip design workstations, it was owned by the Client team. That would create some friction in the future, but for right now Kermit had two things going for it.

First, Kermit was a funded project with a team of around a dozen people. That's important, because putting together a team within Microsoft was always challenging, requiring business justifications, funding, and more. Those things were now out of the way.

Second, a Microsoft architect named Jeffrey Snover had caught wind of Kermit, and had big ideas for it.

A Manifesto

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

Culture

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

The .NET Framework Connection

Windows

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

Exchange

Windows, Again

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

v2

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

DESIGN

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

Design Decisions and Coding Stories

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

Decision-Making Principles

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

Usability Testing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

Being Verbose

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

Providers

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

Ctrl+C

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

Extensible Type System

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

How Parameters Became Cmdlets

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

Parameters: - vs. /

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

WHERE: The Elevation of the ScriptBlock

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

-WhatIf, -Confirm, and -Verbose

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

Punctuation Decisions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

Snap-ins vs. Modules

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

Namespaces

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

Verbs

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

COM

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

What's in a Name?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

Remoting and Buffering

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

Updatable Help

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

The Directed Graph

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

White on Blue

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

Creating a Language

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

The Security Story

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

Greatest Misses

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

MiniShells and AdminShells

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

Transactions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

Workflow

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

Data Streams

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

Tainted Data

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

COMMUNITY

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

The MVPs

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

Meet the MVPs

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

Adopting PowerShell

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

The Role of the MVP Community

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

Where are We Now?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

What Does the Community Look Like?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

Changing MVPs' Careers

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

Shout-Outs

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

The MVP-Microsoft Relationship

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

My PowerShell Story

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

Impact

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

CONCLUSION

Acknowledgements

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

APPENDICES

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

“PowerShell Makes Us All Better at Our Games”

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

“We Can Leapfrog Linux!”

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

Hula Monkey

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.

Personalities

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/shell-of-an-idea.