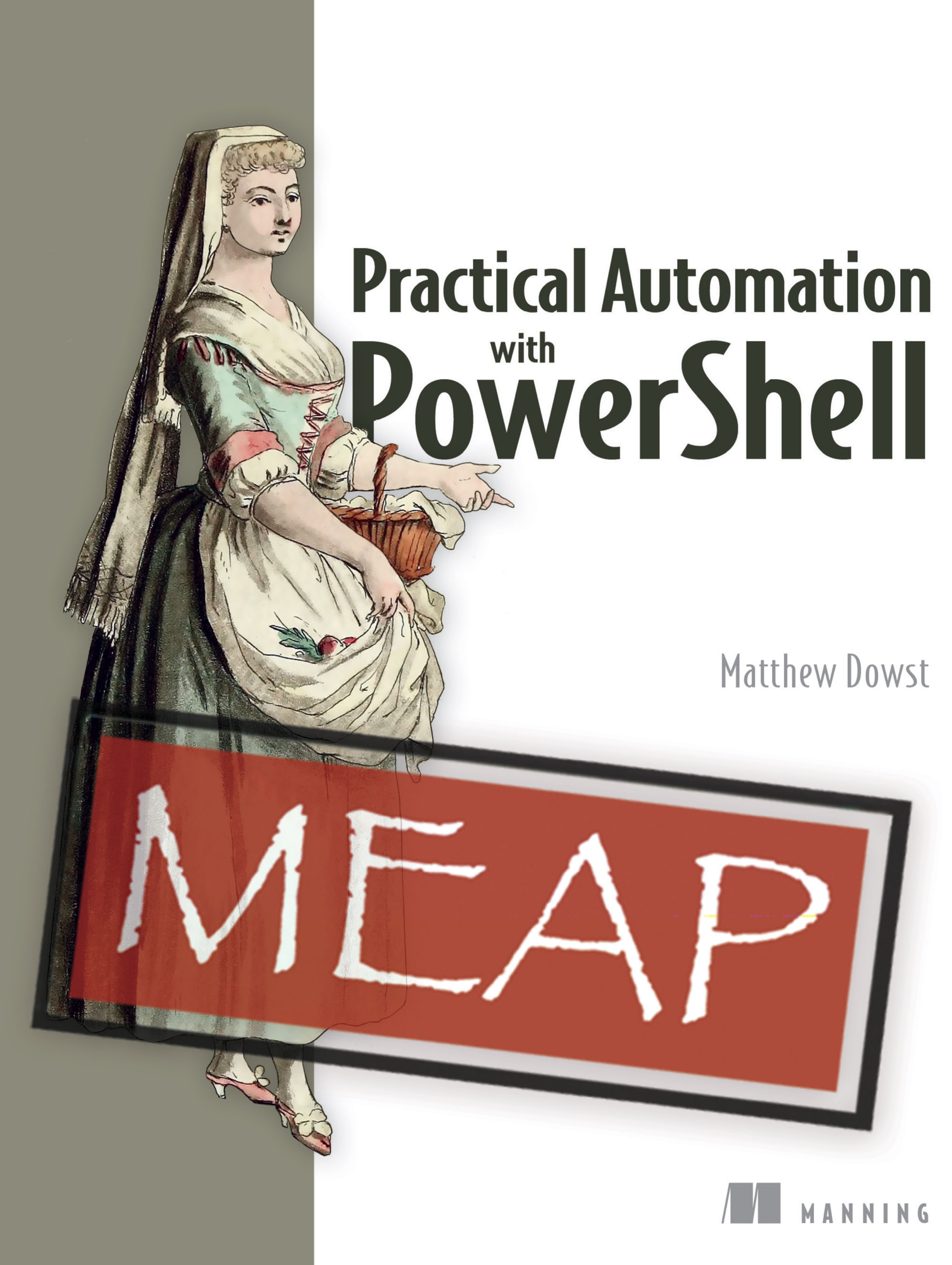

Practical Automation
with
PowerShell
Matthew Dowst
MEAP
MANNING

MEAP Edition
Manning Early Access Program

Practical Automation with PowerShell

Version 5

Copyright 2022 Manning Publications

For more information on this and other Manning titles go to
manning.com

©Manning Publications Co. To comment go to liveBook
Licensed to Tausif Khan <tsfahmed2@gmail.com>

welcome

Thank you for purchasing the MEAP for *Practical Automation with PowerShell*. This book is for IT professionals who are familiar with PowerShell. Maybe you have used PowerShell as a command-line tool, or even written a few custom scripts or modules, and now you are ready to take your skills to the next level by creating enterprise-ready automations. This book goes beyond simple coding examples, and teaches you how to think like an automation engineer, using real-world scenarios along the way.

Like many IT professionals, I started my career working at a help desk. During my time there, I scripted everything I could. Over the years, I have honed and developed techniques that have let me go from making simple scripts, to creating enterprise automation solutions with PowerShell. I also have the unique experience of working for a company that does custom application development, and IT infrastructure support. So, a few years ago, I posed the question: Is it easier to teach a developer infrastructure, or is it easier to teach an infrastructure background how to code? The answer is, by far, teaching someone with an infrastructure background how to code. You don't need to have a degree in computer science to create enterprise-ready automations. However, knowing all the idiosyncrasies of IT infrastructure is only something that you can learn through experience. So, by learning a few fundamental principles, you can start automating your various workloads in no time.

This book is broken into three sections, each covering a different part of the automation spectrum. Part one will cover how to design your automations for long-term management and reusability. It will also provide guidance on creating unattended automations and scheduling them. Part two contains the technical how-to guides for automating different workloads with PowerShell. This includes securing sensitive data, connecting to remote resources, and making smart/adaptive automations. Finally, part three will cover how you can share those automations with a wider audience, including sections on sharing scripts amongst your team, executing scripts on end-user devices, and providing front-end interfaces to your users.

Be sure to post any questions, comments, or suggestions you have in the liveBook discussion forum. Your feedback is invaluable in ensuring that you get the most out of this book.

—Matthew Dowst

©Manning Publications Co. To comment go to liveBook
Licensed to Tausif Khan <tsfahmed2@gmail.com>

brief contents

PART 1: GETTING STARTED WITH AUTOMATION

1 Why automate with PowerShell

2 Get started automating

PART 2: WRITING SCRIPTS

3 Scheduling automation scripts

4 Handling sensitive data

5 PowerShell remote execution

6 Making adaptable automations

7 Working with SQL

8 Cloud-based automation

9 Working outside of PowerShell

10 Automation coding best practices

PART 3: MANAGING AUTOMATION SCRIPTS

11 Sharing scripts among a team

12 Sharing scripts with end-users (front-end)

13 Where to store your code

14 Future-proofing your code

APPENDIXES

A Development environment set up

B Cloud environment setup

C Helper scripts

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

1

Why automate with PowerShell

This chapter covers

- **How to conceptualize your automation needs.**
- **Why you should automate with PowerShell.**
- **How to know when PowerShell is the right tool for the job.**
- **How you can get started automating your workloads today.**

Every day across all industries, IT professionals are tasked to do more with less, and the best way to achieve that is through automation. However, many companies do not see IT automation as an asset. More often than not, automations are cobbled together by some sysadmin in his or her spare time. This often leads to situations where the automation becomes less efficient than doing the task manually.

I am sure at some point you have tried using one of the codeless automation platforms like IFTTT, Flow, Zapier, or any other of the ones out there. If you are like me, I am sure you found the features a little too basic. They are great for personal "one-off" type automations, but to really get what you need out of them, they require customization beyond what their simple GUI can provide.

This is where PowerShell can shine. PowerShell is a task automation framework with a simple and intuitive scripting language. PowerShell includes command-lets (cmdlets) that allow you to do similar tasks available in admin consoles and provides a framework where those tasks can be chained together to execute a series of logical steps. The cmdlets allow you to manage and automate the entire Microsoft ecosystem (Azure, Office 365, Microsoft Dynamics, etc.), as well as other platforms like Linux and Amazon Web Services (AWS). By harnessing the potential of PowerShell and learning a few fundamental principles of automation, any IT professional can become an automation guru.

On top of being asked to do more with less, the IT industry is moving to a more "infrastructure as code" model. I have the unique experience of working for a company that

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

specializes in infrastructure consulting and custom application development. So, I posed the question, is it easier to teach someone with an application development background infrastructure or is it easier to teach someone with an infrastructure background to write code? The answer is, without a doubt, it is easier to teach someone with a deep knowledge of infrastructure to write code. Most of you are already familiar with working in command-line interfaces, using batch/shell files, and running PowerShell scripts. Therefore, the leap to writing code specifically for automations is not as significant as trying to teach someone all the idiosyncrasies of infrastructure.

1.1 What you'll learn in the book

This book does not aim to show you just how to write PowerShell scripts. There are already hundreds of resources out there on just writing code. Instead, the goal is to show you how you can use PowerShell as an automation tool by understanding:

- How you can leverage PowerShell to automate repeatable tasks
- How to avoid common automation pitfalls with PowerShell
- How to share and maintain scripts for your team
- How to front-end your scripts for end-users

We will achieve this goal by using real-world examples that IT professionals run into every day. You will work through both the technical side of writing the code, the conceptual side of why the code is structured the way it is, and how you can apply that to your automation needs.

1.1.1 Building Blocks

It is easy to let yourself become overwhelmed when trying to automate tasks. To avoid this, the approach we will take throughout this book is the concept of creating reusable building blocks. This concept will get you automating tasks on day one and provide you with a framework to continually expand upon your earlier work.

Take, for instance, the typical scenario of adding a user to a group. Your current process could look something like this:

1. A user submits a ticket request to be added to the group.
2. A member of the help desk is assigned the ticket
3. The assigned person checks in Active Directory for the group and doesn't find it.
4. They check Exchange and don't find it.
5. They check Azure AD, find the group, and add the user to it.
6. They update and resolve the ticket.

This is an easily repeatable process that is perfect for automation. Your ultimate goal would be to have the process fully automated, so no one would ever need to touch the ticket, but this can take time. However, you can start saving the help desk time right away if you create building blocks. Looking at the current process, you can easily see that the most significant amount of time used is finding where a group exists (and with the growing cloud and hybrid environments trends, this will only worsen).

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

So, your first building block could be a simple script the help desk technicians can run that checks all environments to discover and returns where the group exists. This would start saving them time right away while you work on the other pieces of the automation. From there, you can build the additional automations that can take the output from the discovery script and actually add the user to the group. Then, you can create a workflow to kick off the automation when the ticket is submitted, find the group owner, and add an approval step to the ticket for them. Then once approved, have it add the user to the group and close the ticket.

Not only will each step along the way make measurable improvements, but it can also help you with designing the later steps of the automation. It can do this by helping you identify scenarios you may not have thought of. Take, for example, the first block for discovering where the group exists. What happens if a group with the same name exists in two environments? Did you design it to only return the first one or to return all of them? If you designed it to return multiple entries, can your automation handle multiple values being passed to it? What happens if the username is different in different environments? The list can go on and on. What the building blocks do, is help you identify these scenarios and plan your later automation actions around them. They can also prevent you from wasting time planning for scenarios that might not actually be relevant.

In turn, these lessons learned in your earlier automations will help when you begin to expand your automation offerings. For instance, after you complete automating adding users to groups, you decide to automate the user provisioning process. Since you already have the functionality to add users to groups, you already some of the building blocks you will need for your user provisioning automation.

The topics covered in this book are designed to help you create these different building blocks. You will learn how to use the functionality available in PowerShell to create these building blocks that you can share between automations and between team members so that everyone can use them for their automations.

1.2 The Anatomy of Automation

When designing any automation, there are four key things you need to consider. These are the automation’s:

- Goal
- Triggers
- Actions
- Future

The automation goal is what that automation needs to accomplish. The trigger is what is initiating or starting the automation actions. The actions are the steps taken during the automation. And the future is what it will take to maintain this automation as a whole and the individual building blocks.

We can use a very common real-world examples to help illustrate each part of the anatomy of the automation. For example, if you have a web server that keeps going offline because the logs fill up the drives. These logs cannot be deleted because they are required

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

for security audits. So, about once a week, you have to find files over 30 days old, compress these old logs, then move them to long-term storage.

1.2.1 Automation Goal

The automation goal is what you are trying to achieve with this automation. While the goal of the automation may seem easy to define, you need to ensure that you take into consideration all aspects of the automation.

In our log file cleanup example, it would be easy to say that our goal is to prevent the drives on the web-server from filling up, but that is just scratching the surface. If that was our goal, we could just delete the old logs. However, we also know these logs are required for security audits. So, our real goal is to create an automation process that will prevent the drives from filling up while ensuring no data is lost, and it will be accessible on the rare occasions it is needed. This gives an all-up overview of the automation and can be used as a checklist when designing your actions.

For example, if we change the last part and say the data needs to be accessed regularly, it could change our actions. In this case, compressing the files and moving them to long-term storage would not be the best option. You could instead move the files to a larger storage array. This would make them easier to access while still preventing your drives from filling up.

Now that you know what you want your automation to achieve, you can start planning the steps it needs to take to get there.

1.2.2 Triggers

Triggers are what start your automation. Broadly speaking, there are two types of triggers, polling triggers and pushing triggers. Polling triggers are ones that check-in with endpoints and pushing triggers are ones that are initiated by an outside event. Understanding the difference between these and how they work will have a significant impact on your automation journey.

Polling triggers are ones that routinely check-in to a system for specific conditions. Two typical implementations of these, and ones we will use throughout this book, are *Monitors* and *Schedules*.

A monitor is something that checks-in and waits for a specific condition to occur. This can be anything from watching an FTP site for file uploads, to monitoring an inbox for emails, confirming a service is running, or any other number of things you need to watch for. Monitors can run continuously or on a recurring interval.

The choice to use a continuous or interval-based monitor will depend on balancing the automations needs vs. costs. For example, let’s say you are monitoring a file share for a file to be written. If you know that the file only gets written once an hour, it would be a waste of resources to have your automation check every 60 seconds for it.

While a monitor might run on a regularly recurring schedule, a scheduled automation is different in that the trigger itself doesn’t check for a condition before running subsequent steps. Instead, it will just run every time it is scheduled. Some common examples include cleaning up files, data synchronization, and routine maintenance tasks. Like with a monitor,

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

you need to carefully consider the needs of your automation when stepping up your schedule.

A push trigger is when an outside event initiates the automation. For example, a common push trigger is an HTTP request like a webhook. They can also include calls from other automations. Also, most service desk tools have a workflow engine that can trigger automations when a certain request is received. These are just a few examples of automated push triggers, but any external interaction can be considered a push trigger.

A simple button or the execution of a command-shell can be push triggers as well. The important thing to remember is that push triggers are initiated by any outside event, where polling triggers are the ones reaching out to the endpoint.

Back to the example of cleaning up the web server logs, you need to figure out what would be the best trigger to use, Polling or Pushing. In this case, a polling trigger makes sense since the web server has no way to reach out. Now you need to determine if it should be a Monitor or Schedule. Usually, a monitor is for things you need to take immediate or near-future actions on. For instance, a service has stopped, or a network connection has dropped. Since this is more of a maintenance task, a schedule would make the most sense. Next, you need to determine your recurrence interval.

You already know that you have to clean up these logs at least once a week. So logically, a trigger with a recurring interval of less than one week would be best. You also know that a new log file gets created after a certain number of lines. You look and see there are about 3-4 logs generated daily. Therefore, a once-daily job would be a good option because anything less would be overkill, and anything over would run the risk of the logs growing too large.

Once you determine your trigger, it is time to move on to the core part of your automation, the actions.

1.2.3 Actions

Actions are what most people think of when they think of automation. The actions are the operations your automation performs to achieve the automation goal. Automations can consist of multiple different actions, sometimes referred to as steps. You can classify actions into three main categories *Logic*, *Tasks*, and *Logging*.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

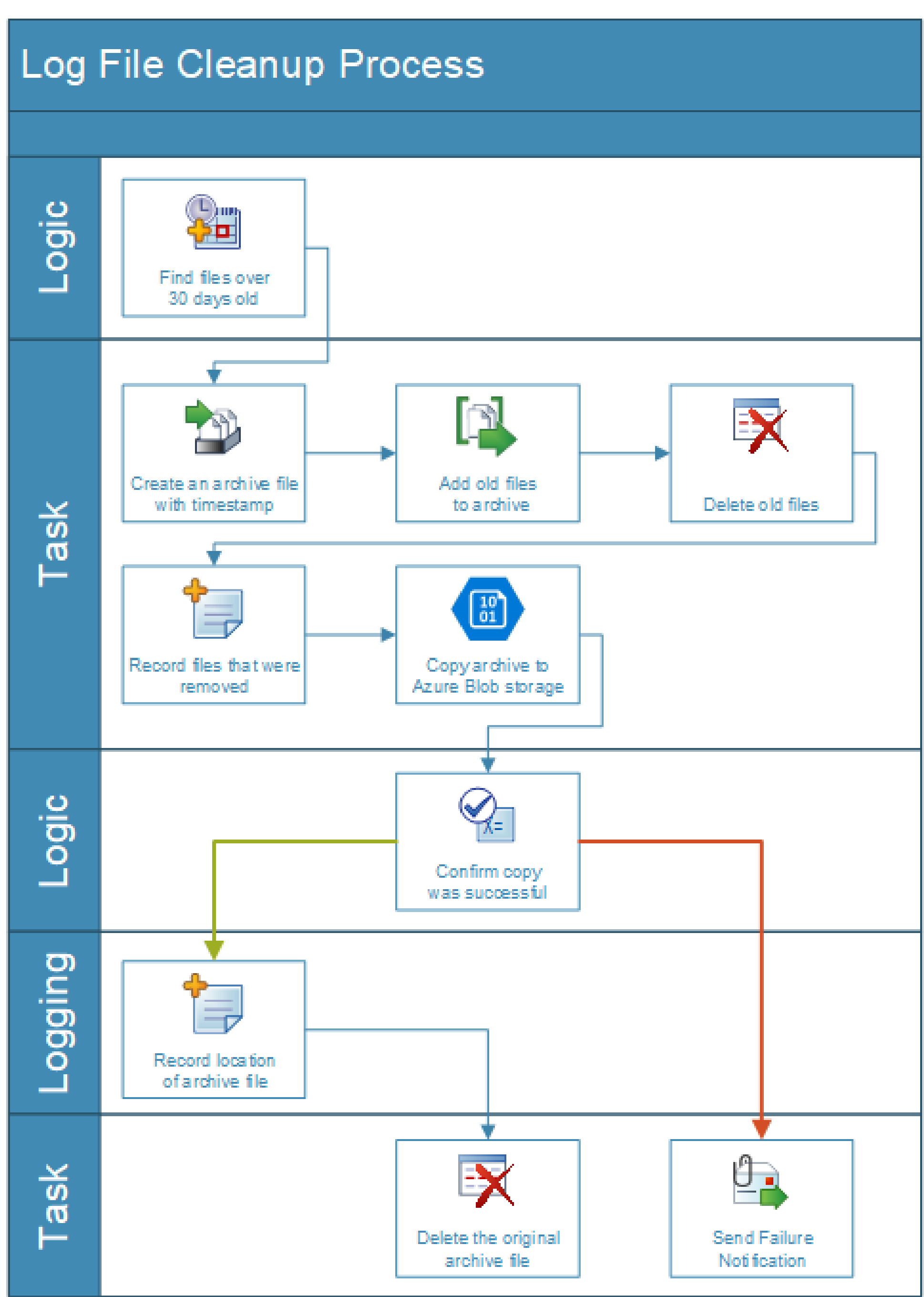


Figure 1: The steps for a file cleanup automation process by type

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

Logic actions are the actions that control the flow of your automation. This includes conditional constructs (your typical if/else conditions), loops, waits, error catching/handling, and handling of variables or other runtime data.

Tasks are the actions performed against the endpoints. Or, in other words, if it is not a Logic or Logging action, it's a Task. The best way to think about it is Logic actions are the brain, and Tasks are the hands.

Logging, like the name implies, is the recording of your actions. Your logging can consist of output from both Logic and Task actions. While Logging actions could be considered Tasks, I prefer to think of them separately because they are not directly involved in completing the automation goal. However, they will be directly involved in you creating successful and maintainable automations.

Looking at our example for cleaning up log files, we can identify the actions we need to take and what type of actions they are.

1. Find logs over 30 days old (Logic)
2. Create an archive file with timestamp name (Task)
3. Add old files to archive (Task)
4. Remove old files from the drive (Task)
5. Record which files were removed and the name of the archive file (Logging)
6. Copy archive file Azure Blob for long-term storage (Task)
7. Confirm copy was successful (Logic)
   If not, stop the process, and send a notification
8. Record location of the new file (Logging)
9. Remove the original archive file (Task)

1.2.4 The Future

A few years back, I went to a customer to help them automate their user provisioning processes. During the discovery phase, I was told users had to be created in a specific directory, left for one hour, then moved to their proper directory. Of course, I asked why and was told it would allow the users to sync with an in-house application. It turns out the person whose job it was to add and remove users from this application decided that they would automate the process. At the time, all users existed in the same directory. So he built this automation, and it saved him 30-60 minutes of work a week. However, over time things changed.

The company had expanded and needed to provide different policies to different users, so they created different directories. Then they noticed that certain users were not being created in this in-house system. By this time, the person who wrote to the automation was long gone, and no one else understood how it worked. So, they would just add them into the one directory, wait until the hourly sync ran, then move them to the proper directory. What had initially saved one person 60 minutes of work a week, was now costing others a couple of extra minutes for each user they created. Which in the long-term means this automation is costing them more than it ever saved. This is a classic example of not planning for the future.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

No one can predict the future, but you can certainly plan for it. No matter what step of the automation process you are working on, you need to ask yourself how difficult will this be to maintain? When you do this, think back on your experience and consider how the requirements might change over time.

In our log cleanup scenario, we said our first action was to find log files over 30 days old. One of the first things that should have come to mind is, what happens if the drive starts filling up faster and you need to change it to 14 days? How difficult would it be to change? If you created the number of days as a variable, it would not be difficult at all. Versus if you hardcoded the number of days in your scripts, and you had to go back and make multiple changes.

Another scenario that might not be as straightforward is, what if they add a second log folder? First, you need to ask how likely this scenario is. If it is likely, then you can ask yourself, is it worth writing the automation to handle multiple folder paths, or could I do something as simple as just running it twice, once for each path?

Another aspect to consider is the question, what if I have to change it from daily to hourly? Again, ask yourself if this is a likely scenario. Then if it is, determine what it would take to change the automation to hourly. It might seem like a simple answer to say, change the filter from days to hours, but you also need to look at how this could affect other actions. For instance, when you create the archive file, are you adding a timestamp to the name? If so, does it include hours? If it doesn't, then you may create a situation where you end up accidentally overwriting data.

The answers to any of these questions are going to depend on your unique requirements. Of course, you will not be able to predict every possible situation, but if you keep these questions in the back of your mind the entire time, and know how you can address them using PowerShell, you will be more prepared when changes do occur.

You also need to be aware of getting caught in the weeds. If you noticed, my first response to any question was, how likely is this scenario? You can get so bogged down in accounting for different scenarios that you'll never accomplish anything, or you'll make your logic so complex no one other than you will be able to understand it. It is a delicate balancing act and one we will continually touch on throughout this book.

1.3 Why PowerShell for Automation

As I mentioned earlier, PowerShell was built as a task automation framework. It is purpose-built with IT automation in mind and includes many different tools to help you achieve your automation needs.

1.3.1 Easy

First and foremost, PowerShell is an easy language to pick up. One of the often overlooked advantages of PowerShell is the fact that it is both a scripting language and a command-shell. The commands you use in the shell prompt are the same commands you use when creating a script. So even if you've never created a PowerShell script before, the transition from command-shell to scripting is not a major one. It can be as simple as saving the

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

command you just wrote as a script file, and you have now just made your first automation script.

It was designed to be a very intuitive language adopting a verb-noun syntax to make it easy to find and remember commands, known as cmdlets. For example, to get the services from the OS, you use the `Get-Service` cmdlet. To stop a service, there is the `Stop-Service` cmdlet, and to start it again, the `Start-Service` cmdlet. Plus, if you can't remember a cmdlet or how to use it, there are the `Get-Command` the `Get-Help` cmdlets to help you out.

IT Admins are not developers, and this book doesn't expect you to be either. So there is no need for you to install an overly bloated integrated development environment (IDE) or know how to run a debugger. Although you can use one if you choose, and you can also use something as simple as Notepad. There are numerous editors with syntax highlighting that support PowerShell. Some free ones like Visual Studio Code, NotePad++, and Sublime. As well as some paid ones like PrimalScript and PowerShell Studio.

1.3.2 Power of .Net Framework

Since PowerShell is built on the .Net framework, it is a much more powerful tool than most people realize. It has the built-in ability to make .Net calls directly inside of scripts. This allows you to gain the extra benefit of using .Net without the need to learn a lower-level language like C# or compile any custom code.

1.3.3 Portable

Another great benefit of PowerShell is its portability. With the introduction of .Net Core, which lead to PowerShell Core, it has become a true cross-platform solution. Not only will PowerShell work on Windows, Linux, and macOS, but there are numerous platforms with PowerShell support. Everybody knows that PowerShell works great and is supported across the entire Microsoft stack, including System Center, Exchange, and SQL Server. Azure has Azure Functions and Azure Automation that you can use to run PowerShell scripts against both your cloud and on-premises environments. Amazon Web Services has a similar service that supports PowerShell with its Lambda offering.

You can use PowerShell across a multitude of different automation platforms. These include most, if not all, IT Automation platforms like ActiveBatch, Control-M, CA Automic, vRealize Orchestrator, and many others. Its reach even goes beyond the automation platforms to include configuration management tools (Puppet, Chef, Ansible) and build automation tools (Jenkins, Azure DevOps, GitLab).

It is also supported in most Service Desk platforms that have workflow automations as well. For example, ServiceNow can trigger a specific PowerShell script when a particular service request is submitted.

1.3.4 Secure

As with all things IT related, security is a huge factor in automation. When you think about some of the access levels automations require, it can become scary. Luckily, PowerShell has been purpose-built to handle multiple different security risks, many of which we will cover later in this book.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

You have the ability to securely sign your scripts to ensure that they can't be manipulated to do things they were not intended to do. It also has what is referred to as, Just Enough Administration (JEA). JEA uses the principle of Least Privilege to delegate only the required permissions to your automation.

PowerShell also has multiple methods for handling authentication. It was originally built with Windows in mind, so it has Windows Integrated Authentication built into it. You can also use secure credential objects and certificates to authenticate against most modern protocols, including OAuth and SSH.

1.3.5 Extensible

By using PowerShell, you are not limited to what is native to the language. It has interfaces with many different APIs, including the Win32 API. You can also talk directly to WMI and COM objects.

In addition, PowerShell can be extended through the use of modules. Modules can be written in PowerShell, using a lower-level language like C# and compiled into a dll, or a combination of the two. This extends the range of possibilities to include the entire extension library of .Net. That has over 200,000 unique packages.

1.3.6 Microsoft's commitment

Microsoft first introduced PowerShell in 2006 and has continued its development non-stop ever since. PowerShell has been included in every release of Windows, both client and server, since 2009 when it was included in Windows 7 and Windows Server 2008 R2.

Microsoft has really embraced the community and helps it grow. With the introduction of PowerShell Core, Microsoft has made PowerShell an open-source project where the community can help to further expand the capabilities. Microsoft is so dedicated to the PowerShell community, that they hold monthly calls that anyone can join to learn about what the PowerShell Team is working on, and ask questions they may have.

> **NOTE**: You can keep up with all changes, submit feedback, find links to the monthly call, and records of past calls in the PowerShell-RFC GitHub repo: https://github.com/PowerShell/PowerShell-RFC

With Microsoft's move to a cloud-first strategy, they have continued to show their commitment by making the pledge that anything you can do in the Azure portal, will have a corresponding PowerShell command. This shows that where Microsoft and the markets go, they will be sure that PowerShell follows.

1.4 What PowerShell isn't

While PowerShell is capable of many things, there are a few things it is not. It does not have a front-end like forms users can fill in. It is also not a job scheduler and does not have built-in triggers like webhooks. While it is technically possible to achieve some of this functionality through PowerShell, it is not practical. There are other tools out there that are built specifically for these tasks, and many of them support PowerShell.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

1.4.1 Job Scheduler

PowerShell itself does not have a built-in job scheduler. You may be aware of the Register-Scheduled Job cmdlet, but all that did was create PowerShell jobs in the Windows Task Scheduler. To achieve true cross-platform support with PowerShell Core, this functionality was removed from version 6.0 and up. You can still use Task Scheduler to schedule and run your PowerShell scripts in Windows, just like you can use Cron in Linux, but there are other tools out there purpose-built to handle things like automation jobs.

If you are already using tools like Jenkins, Ansible, or Control-M, you can use PowerShell inside of these platforms to fulfill your automation requirements. The best part is your automations will then be platform agnostic. For example, if you invested your efforts in a solution like IFTTT or System Center Orchestrator, you are now locked into those platforms. If that software is deprecated, changes its licensing, or takes away functionality, your only course of action is to recreate your entire automation. But if you build your automations with PowerShell in Jenkins, and your company decides to move to Ansible, you can easily transfer your automation scripts from one platform to another with minimal effort.

1.4.2 Front-End

The same can be said for things like front-end forms. A front-end is just a way to gather information for your automation. You can technically build forms in PowerShell, and there are instances where it makes sense, but there are a lot of caveats to it. Like with job schedulers, there are numerous tools available that make creating and presenting forms simple and easy.

You can build all the actions for your automations in PowerShell and then front-end it through any means you like. For instance, you can make a SharePoint list to collect the information you need for your automation in a few minutes. Then, all you need to do is build a simple trigger that passes the required information to your automation. Then, if you want to move to ServiceNow, no problem. All you need to do is remap your trigger from SharePoint to ServiceNow, and your automation will continue to function like before.

1.5 Choosing the right tool for the job

One of the biggest mistakes you can make when trying to automate something is trying to make a tool do something it is not designed to do. So, before you begin any PowerShell automation project, you need to determine if it is the best tool for the job.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

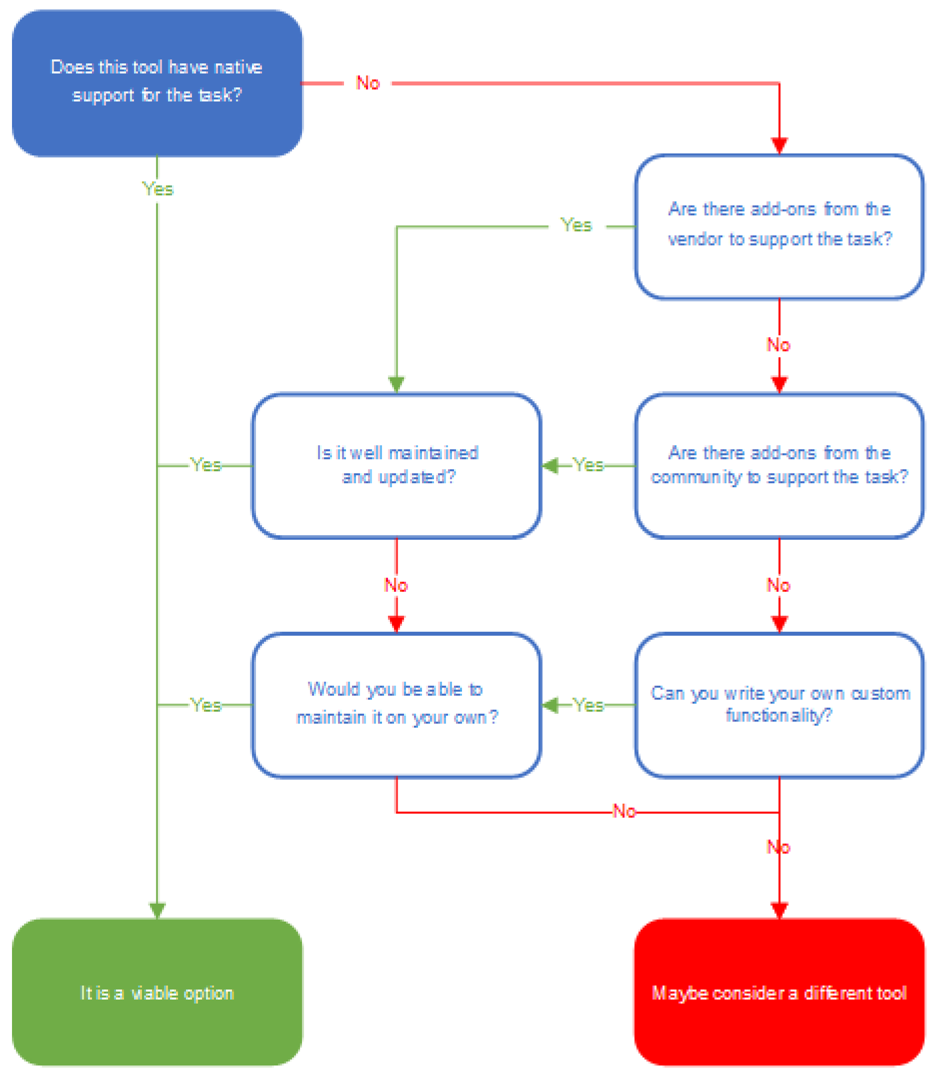


Figure 2: The PowerShell decision tree can be used to determine if PowerShell is the right tool for the job

For example, if you are setting up resources in Azure, I would not recommend using Python. Not because Python itself is a bad tool, far from it, but because Python does not have support for Azure resources. You can actually do it by invoking the Azure Command Line Interface (CLI) through Python, but doing this can lead to a whole other set of issues. Now your Python script is dependent on having Azure CLI installed. Since Azure CLI is a stand-alone application and not a package for Python, you will need to build specific checks into your script to ensure that the files you need are available. Also, your script is now

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

dependent on a platform that supports both Python and Azure CLI. This dramatically increases the complexity of your automation and makes it much less portable.

Now, if you choose PowerShell for this task, you can use the Azure PowerShell modules created and maintained by Microsoft to perform your tasks. All the functionality to check for and resolve dependency issues are built into PowerShell. With 2-3 lines of code, you can make your script completely portable to any other system running PowerShell.

I am not saying PowerShell is the end all be all, but for certain workloads, PowerShell just makes sense. Now with PowerShell Core, the number of tasks that you cannot automate with PowerShell is growing smaller and smaller, but is not zero. If you need to do technical analysis as part of your automation, such as calculating and plotting Bollinger bands, I would not recommend PowerShell. In this case, the Panadas library in Python is leaps and bounds above anything available in PowerShell.

1.5.1 Automation Decision Tree

So, how do you determine if PowerShell is the right tool for the job? One way is by using the decision tree in figure 2.

When using the decision tree, you need to look at all aspects of the automation process you are creating. Let us take our previous example of archive old log files to Azure storage blobs. The first action was to find files over 30 days old. Running that through the decision tree would look something like this:

1. Does this tool have native support for all the tasks I need to accomplish? **Yes, PowerShell has built-in functionality to work with file systems.**

There is no need to continue with the other questions because the first one is a definitive yes. The next few actions in the process will be similar. For instance, creating the archive file:

1. Does this tool have native support for all the tasks I need to accomplish? **Yes, the Compress-Archive cmdlet is native to PowerShell.**

However, not all actions will be so straightforward. Take, for example, the action to copy the files to Azure Blob Storage:

1. Does this tool have native support for all the tasks I need to accomplish? **No**
2. Are there modules/add-ons from the company that can accomplish the tasks? **Yes, Microsoft has an official Azure Storage module.**

Again, this is pretty cut and dry because we know Microsoft creates official PowerShell modules to support all Azure functionality. But there will be instances, even within the Microsoft ecosystem, where the answer might not be so clear. For example, let us say the action to log which files were removed, you need to write these to a SQL table.

1. Does this tool have native support for all the tasks I need to accomplish? **No**
2. Are there modules/add-ons from the company that can accomplish the tasks? **There is a SqlServer module from Microsoft, but it requires that the SQL Server SQLCMD utility is installed, and it is not available via the public PowerShell Gallery.**

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

3. If not, are there modules/add-ons from the community that can accomplish the tasks? **Yes. The module dbatools is available in the PowerShell Gallery.**
    a) Is it maintained and updated? **The GitHub repo has over 15,000 commits, 200 contributors, and the last update was a few days ago.**
4. How difficult would it be to write custom functionality? **It is possible to query SQL directly from PowerShell using the System.Data.SqlClient class that is native in .Net.**
    a) Will it be difficult to maintain? **There may be differences between .Net and .Net Core for the SqlClient class.**

As you can see, there is always a multitude of ways that you can accomplish the task. It will then be your job to make an informed decision on which tool or tools are best suited for the job. You may find that no one tool can meet all your needs, and that is fine too. When using PowerShell, you can easily switch between different solutions to accomplish your goals. And after reading this book, you'll be able to easily identify tasks that you can utilize PowerShell for.

1.5.2 No need to reinvent the wheel

One of the great things about PowerShell is the large community that loves to share its knowledge. At the time of this writing, over 6,400 different PowerShell modules are available in the official PowerShell Gallery. On top of that, there are numerous websites, forums, and blogs dedicated to PowerShell. So chances are if there is something you are trying to do with PowerShell, someone has already done it, or something similar.

There is no need to write every single line of code in your scripts from scratch. I encourage you to go out and see what other people have done. Learn from their mistakes and experiences. I cannot tell you how many times I've seen a block of code to do XYZ, and I think to myself, "why did they do it that way?" Then I write it another way, run into a problem, then realize, "oh, that's why the other script did that."

At the same time, do not just copy and paste code from someone's blog post into your script and expect everything to work. Instead, look and the code the other person wrote. Figure out what exactly it does and how it accomplishes it. Then you can implement it into your script with the confidence that it will work, and most importantly, that you will be able to maintain it.

1.6 What you need to get started today

While PowerShell Core is a cross-platform tool, most examples in this book will be running in a Windows environment. While I would recommend using Windows 10 or Windows Server 2019, you should be able to follow along using any version of Windows that supports Windows PowerShell 5.1 and PowerShell Core 7.

Unless otherwise specified, you can assume that everything in this book is written for PowerShell Core. If you have not already done so, I highly suggest making the switch to Visual Studio Code. Unlike the traditional Visual Studio, Visual Studio Code is a free, lightweight code editor that is open-sourced, cross-platform, and very community driven. It

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

supports most common programming and scripting languages, including Windows PowerShell and PowerShell Core, allowing you to work with both side by side.

Some of the examples in this book work with third-party platforms or cloud solutions. Since it is impossible to write a book that covers everyone's needs, all platforms used in this book are either free or have a free trial you can use. These include Jenkins, Azure, AWS, and GitHub. You can refer to appendix A for complete details on the environments and tools used.

1.7 Summary

- PowerShell is a powerful high-level language designed with IT automation in mind that is easy to pick up and start using.
- You can use PowerShell to create reusable building blocks that can be shared among automations and your team members.
- To create truly successful automations, you need to be able to conceptualize the process and plan for the future.
- PowerShell is an extensible and portable tool that makes it perfect to fit most automation needs.
- It can work hand in hand with other tools and platforms to meet every need you have quickly and easily.
- It has a large community and is backed by one of the largest tech companies in the world.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

2

Get Started Automating

This chapter covers

- **Identifying what tasks you can automate.**
- **The concept phases automations.**
- **Examples of how to create reusable functions.**
- **How to store your functions in module.**

As you saw in the last chapter, most IT departments have grown beyond the days where one person could write a script for something like creating accounts. To be a truly agile and cost-effective department, your automations need to be as well. Unfortunately, many organizations are hesitant to start automation projects because of their past experiences with their projects not meeting expectations or going way over budget. I have talked with numerous people and companies that have had bad experiences with automation projects. However, I have not once come across an automation project that failed because of the technology. In every case, it was poor processes, planning, execution, or any combination of those.

In this chapter, you will learn how to make your automation project a success by using the concepts of phases and building blocks and how that applies to PowerShell. You will see how you can take a simple script and turn it into a reusable building block you can use anywhere. You will do this by taking a simple script to clean up old log files and turn it into a building block by thinking like an automator.

You will also learn how you can store these building blocks for use across multiple automations. Whether you are writing a simple script to automate a repetitive task or something much more extensive, knowing how to use a phased approach to your automation can save you a lot of time, money, and stress.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

2.1 What to automate

If you are reading this book, then it is probably safe to assume you have asked yourself, what should I automate. While, the answer you most likely want to hear is, everything! The genric answer is any repetitive task that takes you less time to automate than to perform. However, like many things in the information technology field, the answer is not always so simple. You need to consider multiple factors to determine if something is worth automating, and as you will see, it may not always be a straight return on time invested.

It is easy to say if it takes you less to automate it than it takes to do it manually, then it is worth automating, but that is not the complete story. You need to take into consideration the following:

1. Time - How long it takes to perform the task?
2. Frequency - How often someone performs the task?
3. Relevancy - How long will the task/automation be needed?
4. Implementation - How long will it take to automate?
5. Upkeep - How much long-term maintenance and upkeep will it require?

The first two items, how long and how often, are usually the most straightforward numbers to figure out. As well as, the business side of things like; how long will this task be relevant? For example, if you automate a job that will go away after the next system upgrade, then you may not have enough time to recoup your time invested.

The implementation and upkeep costs can be a little more challenging to calculate. The implementation time and costs are things you will learn to get a feel for the more you automate. Just remember to factor in things like the cost of the tools, platforms, licenses, etc. For determining the upkeep, you need to look at the technology-based maintenance tasks like platform maintenance, API changes, and system upgrades.

Once you have answers to these questions, you can calculate the amount of time you can spend automating the task to see if it is worth your time. If you multiple the time, frequency, and relevancy, you can get your current cost. Then add your implementation plus the upkeep over the relevancy period. If your current cost is greater than your automation cost, then the task is worth automating.

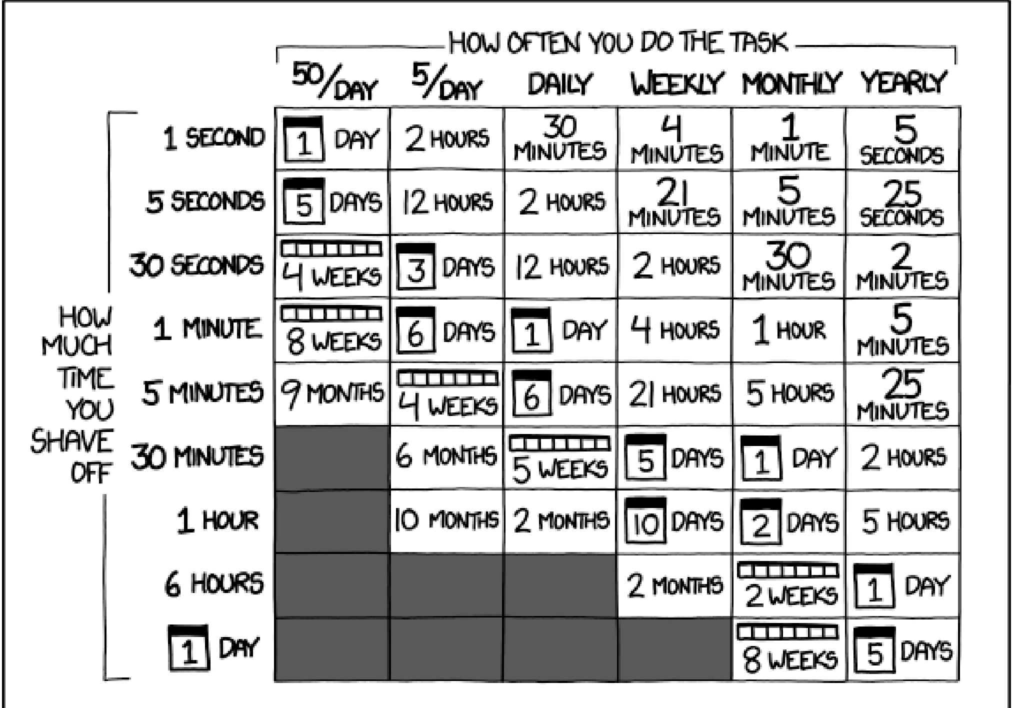
$$Time \times Frequency \times Relevancy > Implementation + (Upkeep \times Relevancy)$$
$$Current\ Cost > Automation\ Cost$$

Figure 1, from XKCD.com, is a great tool you can use to get a quick idea of whether or not a task is worth your time and effort to automate. It shows the amount of time you can spend automating a task, with a 5-year relevancy, before you spend more time automating than you would performing the task manually.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

HOW LONG CAN YOU WORK ON MAKING A ROUTINE TASK MORE EFFICIENT BEFORE YOU'RE SPENDING MORE TIME THAN YOU SAVE?
(ACROSS FIVE YEARS)



HOW MUCH TIME YOU SHAVE OFF	HOW OFTEN YOU DO THE TASK					
	50/DAY	5/DAY	DAILY	WEEKLY	MONTHLY	YEARLY
1 SECOND	1 DAY	2 HOURS	30 MINUTES	4 MINUTES	1 MINUTE	5 SECONDS
5 SECONDS	5 DAYS	12 HOURS	2 HOURS	21 MINUTES	5 MINUTES	25 SECONDS
30 SECONDS	4 WEEKS	3 DAYS	12 HOURS	2 HOURS	30 MINUTES	2 MINUTES
1 MINUTE	8 WEEKS	6 DAYS	1 DAY	4 HOURS	1 HOUR	5 MINUTES
5 MINUTES	9 MONTHS	4 WEEKS	6 DAYS	21 HOURS	5 HOURS	25 MINUTES
30 MINUTES		6 MONTHS	5 WEEKS	5 DAYS	1 DAY	2 HOURS
1 HOUR		10 MONTHS	2 MONTHS	10 DAYS	2 DAYS	5 HOURS
6 HOURS				2 MONTHS	2 WEEKS	1 DAY
1 DAY					8 WEEKS	5 DAYS

Figure 1: The amount of time you can spend automating a task with 5-year relevancy.

At the beginning of your automation journey, it can be challenging to estimate the implementation and upkeep costs. However, this is something you will learn the more you do it. Until you become more comfortable with these estimates, a good rule of thumb is, if you think you can automate it in half the time saved, you will be almost guaranteed a good return on your investment.

Besides just being a repetitive task, there are other things to look for when determining what to automate. Anything prone to a high degree of human error is a great candidate for automation. Working with a large dataset or data transcription are two great examples of tasks that are ripe for automation. People make mistakes when typing. It gets even worse when they are dealing with lots of data in front of them. If you have ever had to create a complex Excel formula to manipulate some data, then you have already made a simple automation.

Even if the task at hand is not something you need to do repeatedly, creating a one-off automation could save you time. Plus, if you keep that automation, you can use it as a foundation if you have to perform a similar task in the future. An excellent example of this is

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

string manipulation tasks. For example, you have a text file with a bunch of poorly formatted text that you need to parse into columns and get into a spreadsheet. However, it might not be that many rows, and you could transcribe it or copy/paste it in a few minutes. Or you take it as an opportunity to hone your skills with using regular expressions, splits, substrings, indexes, replace, or any other number of string manipulation methods. Learning to use these correctly will be an invaluable skill in your automation journey.

Another place you can look for things to automate is in tasks that you may not need to do often, but they can be complex and time-consuming. If it is a task that is complex enough that you made yourself a checklist, then you also just made yourself an automation project plan. Start by automating one of the steps, then another, and another, and so on until you have a fully automated process. Then the next time this task comes up, instead of referring to a checklist or trying to remember each step of the process, you can just click a button.

The best way to get started on your automation journey is to find a simple task that you repeatedly do and automate it. It doesn't have to be a big task or anything fancy. Just think about something that will save you time.

You can also use automation to help you overcome obstacles or handicaps that might prevent you from being as productive as you would like to be. For example, I will freely admit that I am not the most organized person when it comes to my inbox. I would like to be, but I cannot always keep on top of things. I don't like to use Outlook rules because I want to be sure that I don't miss an alert, especially if I'm away from my desk. So, what ends up happening is I quickly read through my emails, but I don't always file them right then. As a result, over time, I end up with thousands of emails in my inbox. To combat this, I wrote a script that will file my emails for me. It moves messages to specific folders based on email addresses and keywords. Not only does this automation save me time and help me be more productive, but it also makes me happy, and at the end of the day, that's worth it to me.

One last thing to keep in mind is you do not need to automate an entire process, end to end. You may very well calculate that the cost of automating something would be greater than performing it manually. However, you may be able to automate certain portions of it to save time and give you a positive return on investment. A perfect example of this is barcodes. Barcodes allow cashiers and warehouse workers to quickly scan items instead of hand entering product codes. RFID tags would be even quicker, but the cost of implementing them has so far been higher than the cost of scanning a barcode.

The more experience you get with automation, the better you will become at determining was is and what isn't worth automating. Also, as you will see in the next section, by using a phased approach with reusable building blocks in your automation processes, you can set yourself up for creating bigger and better automations down the line.

2.2 The automation process

When looking at an automation project, it is easy to get overwhelmed. People will tell you to use things like the KISS principle (keep it short and simple). While that is easy to say, in practice, it is not always easy to do. When you have multiple systems talking to each other, complex logic, and ever-changing requirements, it may seem near impossible. This is where

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

the concepts of building blocks and phases come in. By using building blocks and phases, you can break down your complex tasks into small, simple steps.

2.2.1 Building blocks

As you saw in the last chapter, PowerShell provides an excellent way for you to create reusable building blocks to help you achieve your automation goals. No matter how complex the automation is, it can always be broken down into smaller and more simplified steps. Each one of these steps can then be tackled and become another building block for you to use. By breaking tasks down into smaller blocks, you can prevent yourself from becoming overwhelmed and provide clear goals that you can meet regularly. The majority of this book will cover helping you create these different building blocks that you can use across your automations.

Building blocks also allow you to build your skills over time. As you automate more and more, your skills will continue to grow. You will learn new techniques, not just in your coding but in the overall process. You may find a better way to perform a task using PowerShell. If you used building blocks, you can go back and update all your previous automations quickly and easily.

2.2.2 Phases

The adage "you have to be able to walk before you can run" applies perfectly to the world of automation. Your first few automations you make will not be pretty. Just like the first picture you ever drew or the first paper you wrote in school. It takes time and experience to build your skills. But, that doesn't mean you cannot start reaping the benefit of automation right away.

By breaking your automations into phases, you can create incremental benefits. Figure 2 represents how to apply a phased approach to see benefits from the start. Imagine you need to get from point A to point B. Sure, a car may be the fastest way to get there, but you have no idea how to build a car, let alone the resources. So, start small and work your way up. Build yourself a skateboard. Then upgrade to a scooter, then a bike, then a motorcycle, and finally build that car. Each step of the way, you will see improvements and continue to improve your process. Plus, you will see benefits from the very start, unlike if you set out to just build a car from the get-go. In that situation, you would be walking the entire time until you finally built the car.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

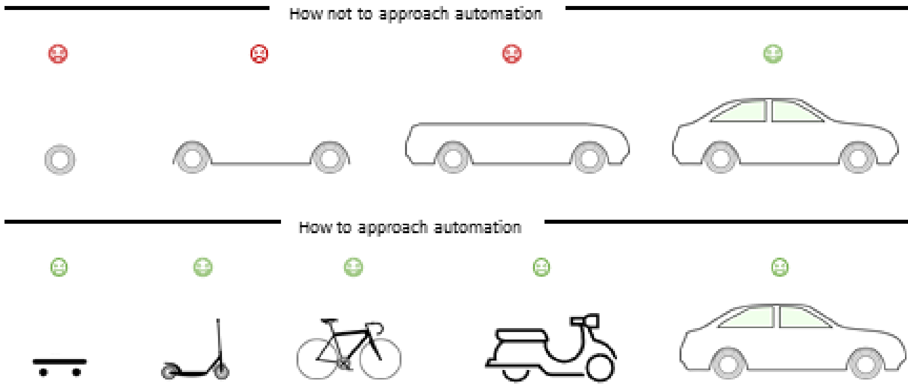


Figure 2: How a phased approach can allow you to start receiving benefits sooner.

During each phase, you will most likely be creating several building blocks. Furthermore, these building blocks you create will often be used across the different phases and improved upon from one phase to the next. For example, in figure 2, you learned to make a wheel in phase 1. Then, in phase 2, you improved upon that knowledge and made an even better wheel.

Phases also allow you to adapt and adjust the automation along the way. You can get feedback after each phase from the people using it. You may find out along the way there are things you did not take into account. Like in the figure 2 scenario after you created the skateboard, people told you it was great for part of the trip, but not the really muddy parts. You can take this feedback and adjust phase 2 to include larger wheels. Contrast this to the non-phased approach of jumping right in and building the car, then finding out it gets stuck in the mud. If you didn't build the suspension and wheel wells to fit bigger wheels, now you have a ton of rework to do.

2.2.3 Combining building blocks and phases

To demonstrate the concept of building blocks and phases in a more IT-centric way, you can look at the common automation scenario of provisioning a virtual machine. While there can be a lot to this process, you can break it down into a few phases.

1. Create a Virtual Machine
2. Install the Operating System
3. Configure the Operating System

While it would be great to tackle all this at once, it would be a massive undertaking, and you would not see any benefits until the very end. Instead, you can tackle one phase at a time. Providing yourself added benefits along the way. Start with phase 1, creating a virtual machine. The building blocks for this could consist of:

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

1. Select a host machine
2. Create a blank virtual machine
3. Allocate CPU and Memory
4. Attach Network Interface Card to the appropriate subnet
5. Create and attach virtual hard disks

Once you've finished phase 1 (creating a virtual machine), you can move on to phase 2 while you are already repping the benefits of phase 1.

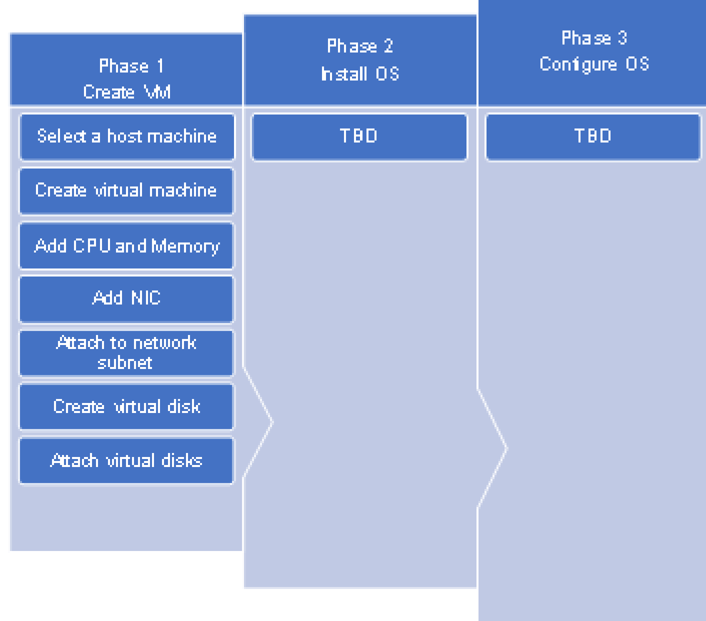


Figure 2: Virtual Provisioning Phased Approach (Phase 1)

In phase 2, you are going to install the operating system. Here you have a couple of options. You can create a template virtual hard disk with the operating system already installed. However, this would mean you have to maintain the template, including apply patches. Also, if you have multiple hosts in different regions, it could be a pain to make sure they all stay in sync. So, you decided to use your configuration management tools to install the operating system. This way, your image is consistent throughout your environment, and it is always up to date.

As you start building this part of the automation, you realize that your virtual machine needs to be on a specific subnet to receive the image. So, your building blocks may be similar to this:

1. Attach to operating system deployment subnet
2. Turn on the virtual machine

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

3. Wait for the operating system to install
4. Attach to production subnet

Since you already created a block to assign the virtual machine to a subnet in phase 1, you can reuse that code for blocks 1 and 4 in this phase. I made attaching to a subnet separate block because I've automated this exact scenario many times before and have run into the situation multiple times. If you combine assigning all the resources to one block, so you assign CPU, Memory, attached the network, and allocate the virtual hard disk, you cannot reuse it. If you want to connect to a different network, it wouldn't really hurt to reassign the CPU and Memory but allocating another virtual hard disk could cause significant issues. Suppose you do something like this that fine. Think of it as a learning experience. I still do it all the time myself. Plus, since you will have to create the building block to assign the subnet for this phase, there is no reason you can't go back and update blocks in the previous phase.

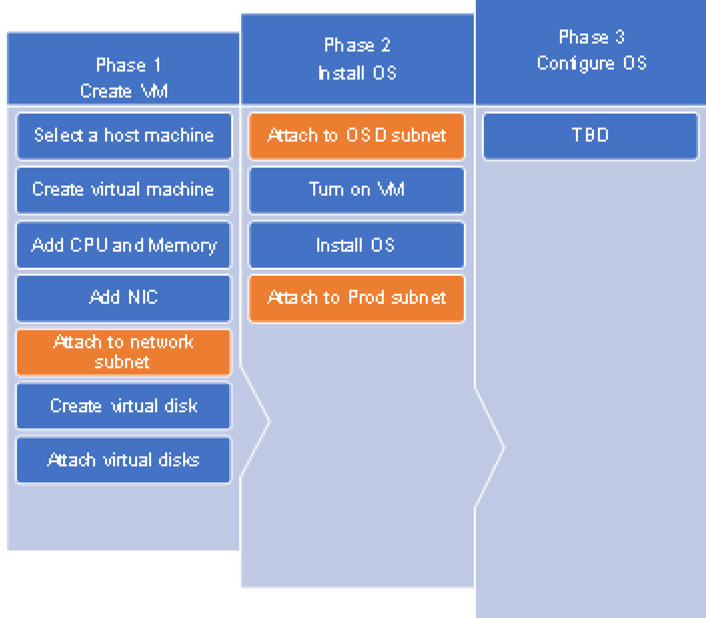


Figure 3: Virtual Provisioning Phased Approach (Phase 2) with shared components

Now you have 2 phases in production, and people are starting to see real benefits. On top of that, you are starting to learn what would benefit them in the next phase. You can talk to the people using the automation and discover what they would like to see in phase 3. It could be assigning a static IP address. Creating secondary data drives. Or any other number of things you may not have considered. Also, you don't have to stop after phase 3. You can add a phase 4 to install applications automatically.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

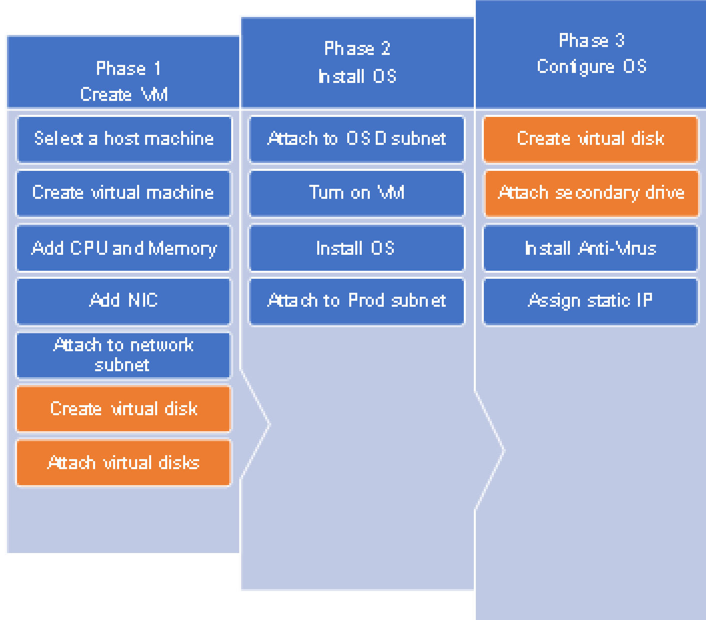


Figure 4: Virtual Provisioning Phased Approach (Phase 3) with shared components

The most significant benefit of combining the concept of build blocks and phases is its flexibility. Not just during the creation process but down the road as well. If your requirements or resources change, all you have to do is swap out the building blocks specific to that change. The process itself and the other building blocks will remain unchanged.

Imagine if your company decided to switch to a new hypervisor or move to the cloud. In these cases, you would need to redo phase 1. Then back to phase 2, all you need to do is swap the network assignment blocks with the new ones you built. All the rest of phase 2 stays the same. Or if they decided to switch to a different operating system. There would be little to no changes in phase 1, and maybe some minor changes to phase 2. So, no matter what gets thrown at you, you'll be able to adjust rapidly.

2.3 The anatomy of PowerShell automation

The previous section describes why using building blocks is essential to your success with automation. Which leaves you with the question, how does this translate to PowerShell? To answer that, we need to look at the anatomy of a PowerShell automation.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

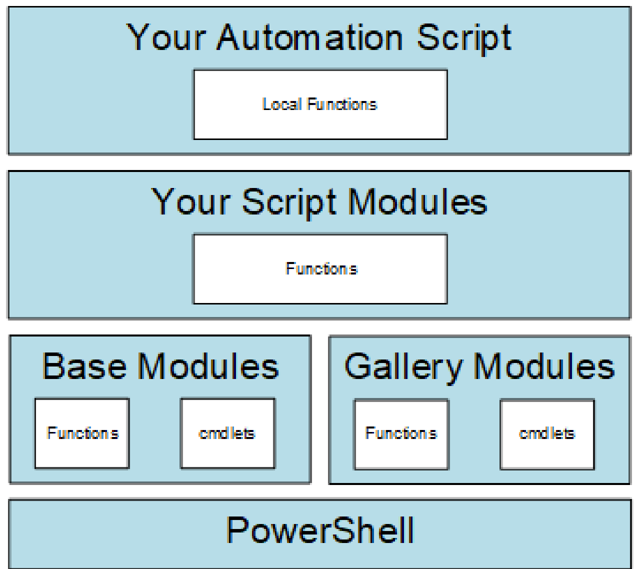


Figure 5: Anatomy of an Automation

As you can see in figure 3, we define an automation as a single script. A script in this instance is defined as a single PowerShell script (ps1) file, that when executed from start to finish, will perform all the required tasks of your automation. Now you can sit down and write a PowerShell script one line after another to perform each task you need. However, doing this will leave you with a tangled mess of code that is difficult to read, make changes to, and test. Instead, you can use functions to break up your code into small, easily testable, and manageable pieces of code. These functions are your building blocks.

A function in PowerShell is a collection of statements that you can define into a single command. For instance, if you want to get a list top 10 processes using the most CPU on your computer, you can use the `Get-Process` cmdlet. However, running just this cmdlet will return every running process in alphabetical order, but you want to sort it by top CPU utilization and limit the numbers of items returned. You also want to format the output to show the process ID, name, and CPU utilization with thousand separators. Unfortunately, now your command is starting to get pretty long and more complex.

This also may be something you want to run often. Instead of needing to remember and retype this entire command line, you can turn it into a function. Then you can then call that function with a single short command. You can also make your functions dynamic by defining parameters. For example, you can create a parameter to specify the number of items to return instead of hardcoding it to 10.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

Listing 1 Get Top N Processes

```
Function Get-TopProcess{    #A
    param(    #B
        [int]$TopN
    )
    Get-Process | Sort-Object CPU -Descending |    #C
        Select-Object -First $TopN -Property ID,
        ProcessName, @{l='CPU';e={'{0:N}' -f $_.CPU}}
}
```

#A Declare your function
#B Define the parameters
#C Run the command

Functions can contain calls to other functions and cmdlets. You can store your functions inside of your script or in a module. Once you have your function defined, you can call it like any other command in PowerShell.

```
PS P:\> Get-TopProcess -TopN 5


  Id ProcessName   CPU
  -- -----------   ---
1168 dwm           39,633.27
9152 mstsc         33,772.52
9112 Code          16,023.08
1216 svchost       13,093.50
2664 HealthService 10,345.77
```

A good rule of thumb when writing a function is it should perform one task that can be restarted if something goes wrong.

Take the example of an automation for archiving old logs. You want to find the old files, add them to an archive, then delete them from the folder. You can write a single function to do that, but what would happen if something goes wrong halfway through the removal process. If you restart the function, you could lose data when you recreated the archive file, and half the files are already deleted.

It is also good practice to write a function any time you find yourself writing the same lines of code over and over again. This way, if you need to make a change to your code, there is only one place to update instead of having to track down every single line you wrote it on.

However, there is also no need to go overboard with the functions. If what you need to do can be done with a single command or just a couple of lines of code, wrapping it in a function could be more trouble than it is worth. It is also best not to include control logic inside of your functions. If you need your automation to take specific actions based on certain results, then it is best to define that in the script.

2.4 Cleaning up old files (your first building blocks)

In this section, you are going to write a simple script (automation) to clean up old log files. In doing this, you will see how to think like an automator and apply the concept of building blocks to your script creation.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

As always, you start with your requirements gathering. You know that you need to remove old logs to keep the drive from filling up. You also understand the logs must be retained for at least seven years, but after 30 days, they can go into cold storage.

With that information, you can start designing phase 1. In this phase, you will find the files to archive, add the old files to an archive file, then remove the old files. Now that you have your basic design, you need to start thinking like an automator.

Figure 7: File Clean up phase 1 design

The first thing you need to consider is what variables your automation will need. This will help you determine the parameters for your script. In this case, you are going to need to know:

- The folder containing the log file
- Where to save the archive file
- What to name the archive file
- How old a file should be before being archived

The first two, getting the log folder and where to save the archive, are reasonably straightforward. In both cases, the input will be a folder path. But the next two, will require some additional considerations.

You know you need to filter the logs by date, so you only archive the ones you want. Since you want to archive files over 30 days old, you can do something as simple as subtracting 30 days from the current time. You can achieve this in PowerShell by using the AddDays method on a DateTime object and passing in a negative number. Since you want to make this reusable for other automations, you can make the date filter parameter a number value provided to the script. However, there are other things you will want to consider.

Since the value of the date filter needs to be a negative number, you can either expect someone using this automation to know that and enter a negative value, or you can have the script automatically flip a positive number to a negative one. However, in either case, you

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

can potentially end up setting the date to 30 days in the future, causing your script to archive files way too soon.

Luckily, with PowerShell, there are multiple ways you can handle this. For example, you can add logic to check if the value is positive and have your script automatically convert it to a negative number. Or you can confirm the date returned is in the past and either exit if not or attempt to fix it. Both of these options can be reusable functions, but in this case, they might be overkill. Instead, a more straightforward approach is to use the parameter validation functionality, which is native in PowerShell, to ensure the value passed is within the range you want it. And since positive numbers are easier for people to think about and enter, you can have your script require a positive number, then just automatically flip it to a negative one.

While any of the approaches mentioned above would be valid, we used the most simple, and therefore less error-prone process, following the KISS principle. If down the line you discover that even with the validation, people keep trying to send negative values, you can adjust your script to use one of the more complex solutions. The key here is the phased approach. You can continue to evolve your script as time goes on. While this problem was easily solved using parameter validation, the next one, setting the name of the archive file, might not be so straightforward.

When your automation runs, you will most likely want to create a new archive file, instead of adding to an existing one. Adding to an existing one can be dangerous because if something goes wrong, it can affect multiple days or even weeks worth of files. Also, a possible phase 2 could be to copy this archive to a cloud-based store. In this case, you would not want to recopy the same file repeatedly as it continues to grow larger. So instead, the safe bet is to create a new one every time the automation runs.

Now you know, the file name will need to be unique for every execution. Therefore it makes sense to add a timestamp to the file name. This means you need to consider how often the automation will run. If it runs once a day, you can make the timestamp, just day, month, and year. However, if it will run multiple times a day, you may need to add the hour, minutes, seconds, or even milliseconds to the name. Next, you need to consider what timestamp to use. You can use the current time, but that may make it difficult to find past logs without looking inside every archive. You can use the date of your filter, but this could get confusing if you ever change the number of days in the filter. Instead, you can use the timestamp from the newest file you are archiving. Now, if you need to search the archives, you can quickly determine which files would be in which archive simply by the name.

Based on all of this, you can no longer have the archive filename as a simple parameter. Instead, you can make a parameter for the archive filename prefix and create a building block (aka PowerShell Function) to append the timestamp value to it.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

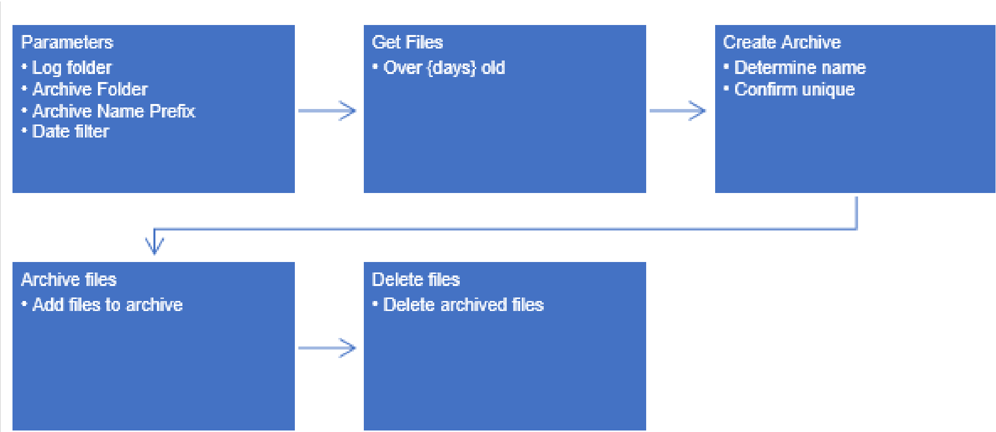


Figure 8: File Clean up phase 1 redesign

As you can see, something as simple as what to name your file can have more variables than you may have initially considered. But this serves as an excellent example of the mindset you need to have when creating your automations. You will see, throughout this exercise, examples of how to thinking like an automator.

2.4.1 Your first function

The code to create the timestamp, append it to the filename and folder, and confirm it is a unique file is a perfect example of when to create a function. Not only will making it a function allow you to maintain and test just that portion of the script. It is also functionality that can be useful in other automations.

Just like with the larger overall automation, you start with determining your parameters. In this case, you need the archive path, the file prefix, and the date value to use for creating the timestamp. Then you need to think about the tasks to perform.

When you think in terms of automation, you should be asking yourself questions like; what should happen if the folder in the ZipPath variable does not exist? Or, what if a file with the same name is already in the folder? To address these concerns, you can use some if-conditions along with the `Test-Path` cmdlet to test the path and the file.

Licensed to Tausif Khan <tsfahmed2@gmail.com>

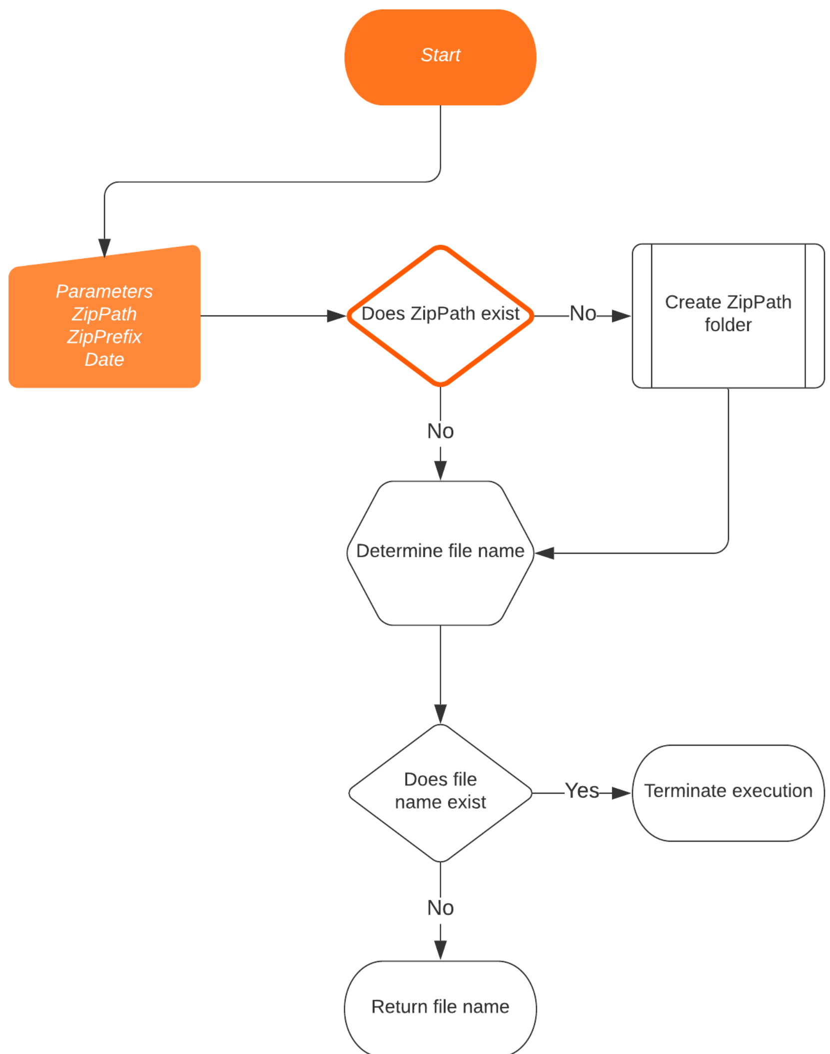


Figure 9: Archive file path function workflow

Now that you have the logic, you can move onto creating the function. However, before diving right in, let's cover a few best practices when creating a function.

You should always include the `[CmdletBinding()]` attribute at the beginning of any function. It provides your function with support for the default parameters to handle things

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

like verbose output and error handling. After the `[CmdletBinding()]` line, you should always include in the `[OutputType()]`. It tells PowerShell what type of value the function it will return. In this case, your function will return a string value for the archive file. So you'll set the value to `[OutputType([string])]`.

Neither the `CmdletBinding` nor the `OutputType` are required to create a function. But it is good practice to include it. As you get into more advanced functions, these will come into use, so it is good to start using them from the beginning.

Next, you will define your parameters in a `params` block. For each parameter, you set whether or not it is mandatory and the type of value. Again, neither of these are required when writing PowerShell functions. However, when you are writing functions that will be used in automations or shared with others, it is good to include them. This way, people can quickly identify what values are required and what data type they should. Also, it helps to ensure that the proper values are passed, and if not, PowerShell's built-in error handling will help prevent unforeseen circumstances that could arrive by the wrong value being sent.

Another thing you always want to include is the comment-based help section. This is left out in many of the examples in this book for brevity. But, I absolutely recommend you add it to all scripts and functions you create. Plus, if you are using VS Code, there is no excuse not to add it because VS Code can autogenerate it. All you have to do is type ## on the first line inside your function, and it will outline it for you. Then all you need to do is fill in the details.

Now to get into the actual execution of the function. The first thing you want to do is checks to see if the folder passed in exists. You can use the `Test-Path` cmdlet inside an if condition to do this.

The `Test-Path` cmdlet returns True if the folder exists and False if it doesn't. In this case of the archive folder, you want to know if it does not exist. So you need to reverse the logic by adding the `-not` keyword in the if statement. Thus, causing the command inside the if block to execute when False.

Instead of having your automation stop if the folder does not exist, you can create it using the `New-Item` cmdlet.

Controlling Function Output

If the New-Item cmdlet is not set to write to a variable, PowerShell will write it to the output stream. Anything written to the Output stream to your script will be returned from your function. In this case, the function would return the output from this command and the zip file path at the end, causing all sorts of unknown issues later in the script.

To prevent this from happening, you can add | Out-Null to the end of any PowerShell command to prevent it from writing to the output stream. Out-Null does not block error or verbose stream, so you can still use those with the command.

If you added the `[CmdletBinding()]` to the beginning of the function, you can use the `-Verbose` switch when calling the function. Verbose output is not written to the Output stream. Therefore it is not returned to the script or any variables. But when you include the `-Verbose` switch, the Verbose stream will be written to the screen. This allows you to confirm that the if-condition is working even though you needing to block the output from any command.

The next part of the script will just be some simple string concatenations to create the file name and set the path. When dealing with paths, in any automation, it is always best to use

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

the `Join-Path` cmdlet as it will automatically account for the slashes for you. This way, you do not have to worry if the value passed for the folder parameter contains a slash at the end or not.

Then finally, you use `Test-Path` again to confirm a file with the same name doesn't already exist. Expect this time you want to take action if it does, so you do not need to reverse the logic. If a file with the same name already exists, you can use the `throw` command to send a terminating error to the script, causing the entire process to halt. When designing anything for automations, you must be mindful of when you want your automation to stop if a particular condition is not satisfied. Like in the situation where the file already exists, you will want to terminate to prevent accidentally overwriting data.

And finally, the function ends by returning the value of the archive file path by outputting the variable to the script.

Listing 2 Set-ArchiveFilePath Function

```
Function Set-ArchiveFilePath{    #A
    [CmdletBinding()]    #B
    [OutputType([string])]
    param(    #C
    [Parameter(Mandatory = $true)]
    [string]$ZipPath,

    [Parameter(Mandatory = $true)]
    [string]$ZipPrefix,

    [Parameter(Mandatory = $true)]
    [datetime]$Date
    )

    if(-not (Test-Path -Path $ZipPath)){    #D
        New-Item -Path $ZipPath -ItemType Directory | Out-Null
        Write-Verbose "Created folder '$ZipPath'"    #E
    }

    $timeString = $Date.ToString('yyyyMMdd')    #F
    $ZipName = "$($ZipPrefix)$($timeString).zip"    #G
    $ZipFile = Join-Path $ZipPath $ZipName    #H

    if(Test-Path -Path $ZipFile){    #I
        throw "The file '$ZipFile' already exists"
    }

    $ZipFile    #J
}
```

#A Declare the function and set required parameters
#B Declare CmdletBinding and OutputType
#C Define the parameters
#D Check if the folder path exists and create it if it doesn't
#E Include verbose output for testing and troubleshoot
#F Create the timestamp based on the date
#G Create the file name
#H Set the full path of the zip file
#I confirm the file doesn't already exist. Throw a terminating error if it does

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

#J Return the file path to the script

2.4.2 Returning data from functions

When returning data from a function, it is best to save to a variable in the function. Then have just that variable on the last line of the function. You may see others that use `return` or `Write-Output` in these cases. While all three are valid ways to return values from a function to a script, they all also have their drawbacks.

The `return` command in PowerShell is different from the `return` command you will see in other languages. Just because you use the return command, it does not mean that is the only thing the function will return.

> **REMEMBER**, a function will return everything written to the Output stream. Therefore, the name return can be misleading, especially for people used to languages like C# or Java. Also, there are situations where you will want to return multiple streams from a function. If you are in the habit of adding return to every function, this will cause issues.

The `return` command does have its uses since it will stop the processing of a function. You can use it when you want to stop a function from executing under certain conditions and return what the value is at that time. There are multiple other ways to handle this, but if a return works well in your situation, go ahead and use it. I would also recommend adding some comments as to why above that line.

Some people prefer to use the `Write-Output` cmdlet for clarity reasons, as it expresses what is being written to the Output stream. However, others feel, just like with the `return` command, that it sets a false expectation that this will be the only thing returned. Also, using the `Write-Output` cmdlet can have performance impacts on your functions and has been known to cause issues with different data types.

For these reasons, it is best to use a single line with a clearly named variable to output the results to the Output stream on the very last line of the function. Returning values inside if/else statements or having `Write-Output` cmdlets mixed in with other commands can make it very difficult to read and understand where the output is coming from. Remember, with automations, you can be almost guaranteed you will need to revisit any script or function in the future. Why not make it easier for yourself or the next person who reads the script?

2.4.3 Testing your functions

As mentioned previously, one of the best reasons to make functions is to allow for easy testing. This is made even easier by the fact that PowerShell will save functions to memory. This will enable you to run multiple tests without worrying about dependencies or issues with other parts of your script.

To test the function you just created, we will use Visual Studio Code (VS Code). If you have not already done so, open a new file in VS Code, and enter the function from listing 2.

Press F5 to execute the code. Since the file only contains this one function, PowerShell does not execute the code. All that happens is the function is loaded into memory. Then you can execute the function in the terminal pane for your testing. We will cover things like Mock

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

testing in later chapters, so the commands you run will perform the actions they are designed to do.

Helper Scripts

I have included several scripts that you can use to help with testing throughout this book. These scripts are in the Help Scripts folder for each chapter or on the GitHub repository for this book. For this section, I have included the script file **New-TestLogFiles.ps1**. You can use this script to create a directory of dummy log files with different created and modified dates. This will allow you to test the functions and scripts in this section.

The first thing you need to do is determine the parameters to pass for testing. In this case, you need the `ZipPath` and `ZipPrefix`, which are both strings. Those can be easily passed as part of the command line. The final parameter Date requires a DateTime object. Here is where PowerShell's ease of use can really come in handy. Since you defined that the parameter is a DateTime object, PowerShell is smart enough to know how to parse a properly formatted string value into a DateTime object for you. This gives you the option to either create the DateTime object before calling the function as the script will do, or send a properly formatted string, that it will convert for you. Keep in mind, it must be a properly formatted string.

You can get a list of string format examples by running the command: (Get-Date).GetDateTimeFormats()

Once you have the values for your parameters, you are ready to begin testing. For the first test, set the `ZipPath` parameter to a folder you know does not exist. This way, you can test the folder creation statement in the function.

```
PS P:\> Set-ArchiveFilePath -ZipPath "L:\Archives\" -ZipPrefix "LogArchive-" -Date "2021-
       02-24" -Verbose

VERBOSE: Created folder 'L:\Archives\'
L:\Archives\LogArchive-20210124.zip
```

Note the `-Verbose` at the end of the command line. This tells the function to output any `Write-Verbose` statements that it executes. In this case, we received confirmation that the if-condition to check that the folder did not exist was true, and it created the folder. If you rerun the same command, you should see the same file name, but this time there should not be any verbose output. This tells you that the script correctly detects that the folder exists, so it does not try to recreate it. It also shows you that the New-Item command successfully created the folder the first time you ran the function.

```
PS P:\> Set-ArchiveFilePath -ZipPath "L:\Archives\" -ZipPrefix "LogArchive-" -Date "2021-
       02-24" -Verbose

L:\Archives\LogArchive-20210124.zip
```

For the next test, go ahead and create a zip file in the directory, using the name that the previous steps returned. Then run the command once more.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
PS P:\> Set-ArchiveFilePath -ZipPath "L:\Archives\" -ZipPrefix "LogArchive-" -Date "2021-
      02-24" -Verbose

Exception:
Line |
  24 |          throw "The file '$ZipFile' already exists"
     |          ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     | The file 'L:\Archives\LogArchive-20210224.zip' already exists
```

This time you will see that the function threw an exception, letting you know that the file already exists. Once your function is tested, it is ready to be added to your script.

A bonus of testing like this is you have just created some perfect examples to include in your comment-based help.

2.4.4 Problems to avoid when adding functions to scripts

When you add a function into a script, it must go before any lines that call it. This is because PowerShell scripts execute in sequence. So, you must always have the statement declaring the function before calling it.

Also, be very careful with functions stored in memory. If you run one script that loads a function into memory, then run another script in the same PowerShell session, the second script can use functions that only exist in the first script. However, if you create a new PowerShell session then run the second script first, it will error out because it does not contain the function. It worked the first time because the first script had already loaded the function into memory. For this reason, you should always create new PowerShell sessions in between tests. This prevents you from getting false positives in cases where items may be stored in memory.

Thankfully, VS Code provides a very easy way for you to do this. All you have to do is click on the little trashcan icon in the terminal. This will kill the session, then will ask if you want to start a new one. Click Yes, and you will have a brand-new clean session reloaded for you.

2.4.5 Brevity versus efficiency

One trap that people often fall into with PowerShell is the insistence on making scripts as few lines as possible. Unfortunately, this leads to scripts with unwieldy long commands that are impossible to read and test. As a result, they can often be less efficient than if they are broken up into multiple lines.

For example, in the automation you are creating, you need to get the files to archive. This is done using the `Get-ChildItem` cmdlet, then add them to an archive using the `Compress-Archive` cmdlet. This task can be done in a single line by piping the results of the `Get-ChildItem` cmdlet to the `Compress-Archive` cmdlet.

```
Get-ChildItem -Path $LogPath -File |  Where-Object{ $_.LastWriteTime -lt $Date} | Compress-
      Archive -DestinationPath $ZipFile
```

If you do combine these into one line, the output will be from the last command, `Compress-Archive`. Then when you go to delete the files, your script will not know which files were added to the archive. You would then need to rerun the Get-ChildItem cmdlet to

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

get the files to delete. Not only is this very inefficient, as you are querying the machine for the file again, but it can also lead to unintended consequences. For example, if a file has been added between the two times the `Get-ChildItem` cmdlet runs, you can end up deleting a file that wasn't archived.

That is not saying combining commands or using multiple pipelines is a bad thing. It really just depends on the context. A good rule to remember is to only query once. If you have a command collecting data and multiple steps use that data, then that command should not be repeated. Instead, the results should be saved in a variable and passed to the other commands that need it.

Besides efficiency, another good reason to break things up is for readability. For example, you can set the path, name, and timestamp in a single command, but it becomes a mess to read.

```
$ZipFile = Join-Path $ZipPath "$($ZipPrefix)$($Date.ToString('yyyyMMdd')).zip"
```

By breaking it up on to a couple of lines you can make it much more readable.

```
$timeString = $Date.ToString('yyyyMMdd')
$ZipName = "$($ZipPrefix)$($timeString).zip"
$ZipFile = Join-Path $ZipPath $ZipName
```

On the same token, breaking it up into too many lines can sometimes lead to large chunks of code that are not as clear.

```
$ZipFilePattern = '{0}_{1}.{2}'
$ZipFileDate = $($Date.ToString('yyyyMMdd'))
$ZipExtension = "zip"
$ZipFileName = $ZipFilePattern -f $ZipPrefix, $ZipFileDate, $ZipExtension
$ZipFile = Join-Path -Path $ZipPath -ChildPath $ZipFileName
```

There may be a perfectly good reason to do this. If you want the script to increment the file name instead of failing on a duplicate entry, it might make sense to have it broken down to separate variables on separate lines. Neither way is inherently right or wrong. It all depends on your context and needs.

So, remember, brevity and efficiency do not go hand in hand. Just because you can achieve something with a single command doesn't always mean it is a good idea.

2.4.6 Careful what you automate

The last step in the automation, deleting the old log files, might seem like a pretty straightforward one. However, if the thought of deleting files via an automated script does not give you pause, then perhaps you never heard the saying, "To err is human. To totally mess something up takes a computer." This rings especially true with automations. However, if you build them well, you can sleep soundly at night, knowing your automations will not be running wild throughout your environment.

With the clean up of the log files, you can quickly delete all the files found by the `Get-ChildItem` command using the `Remove-Item` cmdlet. Of course, you can assume all the files were added to the archive because the `Compress-Archive` cmdlet did not return any errors,

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

but we all know what assuming leads to. So, how can we ensure that each file was archived and is safe to delete? By creating a function that will do just that.

Like with everything in PowerShell and automations, there are multiple ways to achieve this. For example, you can use the `Expand-Archive` cmdlet to extract the archive to another folder, then check that each file matches. However, this would be very inefficient, prone to issues like not having enough disk space to extract the files, and will leave you with two sets of files to delete. Unfortunately, PowerShell does not have a cmdlet to look inside a zip file without extracting it. Fortunately, you are not restricted to just using PowerShell cmdlets. You can also call .Net objects directly in PowerShell, using a technique known as dot sourcing.

Using dot sourcing, you can create a function that uses the `System.IO.Compression` .Net namespace to look inside a zip file. Thus, allowing you to confirm each files' name and uncompressed size without needing to extract them.

How did I know to use the System.IO.Compression namespace?

After searching for ways to look inside an archive file in PowerShell came up empty, I performed the same search, but instead of PowerShell, I used C#. This brought me to a forum post about doing just that. Knowing that I could dot source, I was able to recreate the C# code using PowerShell.

Just like the last function, you will start this one with the `CmdletBinding` and `OutputType` attributes. However, since you are just performing a delete, there is no output, so the `OutputType` attribute can be left blank.

For the parameters, you will need to know the path of the zip file and the files that should be inside of it. The zip file path is a simple string, but the files to delete will need multiple values. Since PowerShell is an object-oriented language and the output from the `Get-ChildItem` cmdlet is saved in a variable, you can pass the object as a parameter as-s. This way, you can avoid needing to convert it to a string array or anything like that.

Since this function will perform an unreversable action, you will also want to include a `WhatIf` switch to help with testing. Switches work much like a Boolean value; expect you don't have to include True or False after it. Just listing the parameter in your command sets it to True. `WhatIf` is a popular parameter included in many PowerShell cmdlets. It allows you to see what the cmdlet would do without actually performing the action. Including it in your function will allow you to test the deletion process without actually removing anything.

Since you will be using a .Net class in your function, the first thing you will want to do is the `Add-Type` cmdlet with the full name of the .Net class. This cmdlet will load a .Net namespace into your PowerShell session. This ensures that you will be able to use the dot sourcing in the other command. In this case, it will be the namespace, `System.IO.Compression.FileSystem`.

Then you can call the classes in that namespace directly in PowerShell by writing the class name between square brackets. Then you can call the methods and constructors by adding two colons. For example, to get the files inside an archive, you use the `OpenRead` method in the `System.IO.Compression.ZipFile` class and save it to a PowerShell variable.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
$OpenZip = [IO.Compression.ZipFile]::OpenRead($ZipFile)
```

Next, you need to compare the files in the archive versus the files that should be in it. Using a `foreach` will allow you to go through each file, one at a time to confirm they are in the archive file by matching the name and file size. If found, they can be deleted. If they are not found, then an error message is sent to PowerShell. However, unlike the previous function, there is no need to stop processing if a couple of files are missing. So, in this case, you can use the `Write-Error` cmdlet instead of the `throw` command. The `Write-Error` cmdlet will send the error back to PowerShell but is not a terminating error like the `throw` command. Instead, this error is just recorded so it can be addressed later.

Since there is no output from this function, there is no need to add a variable result to the end.

Listing 3 Deleting Archived Files

```
Function Remove-ArchivedFiles {
    [CmdletBinding()]
    [OutputType()]
    param(
    [Parameter(Mandatory = $true)]
    [string]$ZipFile,

    [Parameter(Mandatory = $true)]
    [object]$FilesToDelete,

    [Parameter(Mandatory = $false)]
    [switch]$WhatIf = $false
    )
    $AssemblyName = 'System.IO.Compression.FileSystem'    #A
    Add-Type -AssemblyName $AssemblyName | Out-Null

    $OpenZip = [System.IO.Compression.ZipFile]::OpenRead($ZipFile)
    $ZipFileEntries = $OpenZip.Entries    #B

    foreach($file in $FilesToDelete){    #C
        $check = $ZipFileEntries | Where-Object{ $_.Name -eq $file.Name -and
            $_.Length -eq $file.Length }
        if($null -ne $check){    #D
            $file | Remove-Item -Force -WhatIf:$WhatIf    #E
        }
        else {
            Write-Error "'$($file.Name)' was not find in '$($ZipFile)'"
        }
    }
}
```

#A Load the System.IO.Compression.FileSystem assembly so you can use dot sourcing later
#B Get the information on the files inside the zip
#C Confirm each file to delete has a match in the zip file
#D If $check does not equal null, then you know the file was found and can be deleted
#E Add WhatIf to allow for testing without actually deleting the files

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

2.4.7 Putting it all together

Now that you have created your new function (aka Building Blocks), it is time to put everything together into a single script. Like with a function, your scripts should always start with comment-based help, the `CmdletBinding` and `OutputType` attributes, and parameter block. Then if you need it to import any modules, this should go directly after the parameters.

Before you enter any of the script code and logic, enter the script functions. While technically, functions can go anywhere as long as they are before any commands to call them, it will make things much easier to read and maintain if all functions are declared at the beginning. This will also make it easier to add or replace building blocks in a later phase.

For instance, in phase 2, you want to upload the archive to a cloud storage provider. You can build that function outside of this script. Then when you are ready to add it, you just need to copy and paste it in and add a line to call it. Then say down the line, if you change cloud providers, simply swap out that one function with one that uploads to the other cloud, and you don't have to change anything else in your script.

Once you add the functions, you can start adding code for the script. Which here will be to set the date filter, get the files to archive, get the archive file path, archive the files, and finally deleting them.

Listing 4 Putting it All Together

```
[CmdletBinding()]
[OutputType()]
param(
    [Parameter(Mandatory = $true)]
    [string]$LogPath,

    [Parameter(Mandatory = $true)]
    [string]$ZipPath,

    [Parameter(Mandatory = $true)]
    [string]$ZipPrefix,

    [Parameter(Mandatory = $false)]
    [double]$NumberOfDays = 30
)

Function Set-ArchiveFilePath{    #A
    [CmdletBinding()]
    [OutputType([string])]
    param(
    [Parameter(Mandatory = $true)]
    [string]$ZipPath,

    [Parameter(Mandatory = $true)]
    [string]$ZipPrefix,

    [Parameter(Mandatory = $false)]
    [datetime]$Date = (Get-Date)
    )

    if(-not (Test-Path -Path $ZipPath)){
```

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
        New-Item -Path $ZipPath -ItemType Directory | Out-Null
        Write-Verbose "Created folder '$ZipPath'"
    }

    $ZipName = "$($ZipPrefix)$($Date.ToString('yyyyMMdd')).zip"
    $ZipFile = Join-Path $ZipPath $ZipName

    if(Test-Path -Path $ZipFile){
        throw "The file '$ZipFile' already exists"
    }

    $ZipFile
}

Function Remove-ArchivedFiles {
    [CmdletBinding()]
    [OutputType()]
    param(
    [Parameter(Mandatory = $true)]
    [string]$ZipFile,

    [Parameter(Mandatory = $true)]
    [object]$FilesToDelete,

    [Parameter(Mandatory = $false)]
    [switch]$WhatIf = $false
    )

    $AssemblyName = 'System.IO.Compression.FileSystem'
    Add-Type -AssemblyName $AssemblyName | Out-Null

    $OpenZip = [System.IO.Compression.ZipFile]::OpenRead($ZipFile)
    $ZipFileEntries = $OpenZip.Entries

    foreach($file in $FilesToDelete){
        $check = $ZipFileEntries | Where-Object{ $_.Name -eq $file.Name -and
            $_.Length -eq $file.Length }
        if($null -ne $check){
            $file | Remove-Item -Force -WhatIf:$WhatIf
        }
        else {
            Write-Error "'$($file.Name)' was not find in '$($ZipFile)'"
        }
    }
}

$Date = (Get-Date).AddDays(-$NumberOfDays)    #B
$files = Get-ChildItem -Path $LogPath -File |    #C
    Where-Object{ $_.LastWriteTime -lt $Date}

$ZipParameters = @{
    ZipPath = $ZipPath
    ZipPrefix = $ZipPrefix
    Date = $Date
}
$ZipFile = Set-ArchiveFilePath @ZipParameters    #D

$files | Compress-Archive -DestinationPath $ZipFile    #E
```


Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
$RemoveFiles = @{
    ZipFile = $ZipFile
    FilesToDelete = $files
}
Remove-ArchivedFiles @RemoveFiles    #F
```

#A Declare your functions before the script code
#B Set the date filter based on the number of days in the past
#C Get the files to archive based on the date filter
#D Get the archive file path
#E Add the files to the archive file
#F confirm files are in the archive and delete

To test this script, you will need to have log files to clean up. If you have not already done so, run the New-TestLogFiles.ps1 included in this chapter's Helper Scripts to create the dummy log files for you to test.

Next, you can set the values to use for your testing in the terminal.

```
PS P:\> $LogPath = "L:\Logs\"
PS P:\> $ZipPath = "L:\Archives\"
PS P:\> $ZipPrefix = "LogArchive-"
PS P:\> $NumberOfDays = 30
```

In VS Code, you can run a single line or section of your code at a time by highlighting it and hitting F8. Unlike F5 that runs the entire script, F8 only runs the section of code that is highlighted. You can start testing by highlighting the functions and hitting F8. This will load the functions into memory

Next, running the lines to set the date and collect the files to archive.

```
PS P:\> $Date = (Get-Date).AddDays(-$NumberOfDays)
PS P:\> $files = Get-ChildItem -Path $LogPath -File |
    Where-Object{ $_.LastWriteTime -lt $Date}
```

You will notice there is no output. That is because the output is saved in the `$files` variable at this point in the script. But you can check the values of each variable by simply entering it in the terminal window. You can also use this to confirm that your date filter is working and that only the files you want to archive are included.

```
PS P:\> $Date
Sunday, January 10, 2021 7:59:29 AM

PS P:\> $files
    Directory: L:\Logs

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a---          11/12/2020  7:59 AM       32505856 u_ex20201112.log
-a---          11/13/2020  7:59 AM       10485760 u_ex20201113.log
-a---          11/14/2020  7:59 AM        4194304 u_ex20201114.log
-a---          11/15/2020  7:59 AM       40894464 u_ex20201115.log
-a---          11/16/2020  7:59 AM       32505856 u_ex20201116.log
…
```

Next, you can run the line to set the archive path and file name and confirm that it is set as expected.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
PS P:\> $ZipParameters = @{
     >>     ZipPath = $ZipPath
     >>     ZipPrefix = $ZipPrefix
     >>     Date = $Date
     >> }
     >> $ZipFile = Set-ArchiveFilePath @ZipParameters

PS P:\> $ZipFile
L:\Archives\LogArchive-20210110.zip
```

Then run the line to add the log files to the archive file.

```
PS P:\> $files | Compress-Archive -DestinationPath $ZipFile
```

Now you are ready to test the delete function. For your first test, add the `-WhatIf` switch to the end of the command.

```
PS P:\> Remove-ArchivedFiles -ZipFile $ZipFile -FilesToDelete $files -WhatIf
What if: Performing the operation "Remove File" on target "L:\Logs\u_ex20201112.log".
What if: Performing the operation "Remove File" on target "L:\Logs\u_ex20201113.log".
What if: Performing the operation "Remove File" on target "L:\Logs\u_ex20201114.log".
What if: Performing the operation "Remove File" on target "L:\Logs\u_ex20201115.log".
```

You should see the "What if" written to the terminal window for each file it would delete. If you check in file explorer, you should see that the files are still there. Then rerun the command, this time without the `-WhatIf` switch.

```
PS P:\> Remove-ArchivedFiles -ZipFile $ZipFile -FilesToDelete $files
```

This time there should be no output, but you can check file explorer, and the files will be gone.

After running through this test, you will want to test the entire script. When performing the final test, I recommend opening a brand new PowerShell window and calling the script directly. This way, you can ensure that there are values or functions in memory affecting the script. To do this, run the New-TestLogFiles.ps1 again to create new log files. Then open a new PowerShell terminal and run your script. If it works, then you are done.

2.5 Storing functions

In the last section, you created a PowerShell (ps1 file) with a couple of functions saved inside of it. Doing this works just fine for functions that you make for a very specific use case. However, functions declared inside a script are only available to that script. At that point, they become specialty use tools and not really building blocks. Of course, sometimes you need a specialty tool, and you can build those as needed, but at the same time, you do not want to have to recreate general-purpose tools for each and every script you create.

2.5.1 Scripts vs. Modules vs. Profiles

While storing your functions in a script file is fine for special-purpose tasks, it limits your ability to share or reuse them outside of that individual script. Two other options you have are to store functions are in Profiles and Modules.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

A PowerShell profile is a script that is loaded when you open a PowerShell session. You can save functions in a profile script and have them load each time you open PowerShell. The downside to this is a profile is specific to your profile on your machine. This means when you use PowerShell on another device, the functions that you are expecting to be there are not available. Yes, it is possible to set up a profile on a network share and have it sync, but that is a lot of work, and it is not guaranteed to function as you expect it. On top of that, different consoles have their own profiles. So, VS Code uses a separate profile file than the PowerShell console or PowerShell ISE. Also, profiles are designed to be specific to the user. Thus, allowing you to set your own customizations. This leads to them not being very friendly to share. All in all, profiles are great for your own personal customizations, but they are not good places to store functions.

The option that gives you the greatest flexibility to reuse and share functions is placing them inside a module. A PowerShell module is a collection of cmdlets, functions, classes, and variables. Modules are great for collaborating with a team because each member can add their own functions to it. In addition, it allows you to reuse the same function in multiple different automations. As you will see later in this book, modules also lend themselves very nicely to things like version control and unit testing.

2.5.2 When to add functions to a module

The question you should be asking yourself any time you are creating a function in PowerShell is, "would this be useful in another script?" If the answer to that question is yes or maybe, then it is worth considering putting it into a module.

Most modules you come across in PowerShell are very much system-based. For example, the Active Directory module or the Azure modules. They are all built with a specific tool in mind because they are usually created by the company or provider of that system. You can certainly stick to this pattern yourself or not. It is really up to you. Nothing is stopping you from creating a module to house a bunch of different useful, yet unrelated, functions.

For instance, you can create a single module used for managing user accounts. This module might have functions that reach out to Active Directory, Office 365, SAP, etc. While these are all separate systems, your module can act as a bridge between them, making your user management tasks much more manageable.

Again, it depends on your needs. Plus, once you see how easy it is to create and maintain a module, there should be no hesitation considering it.

2.5.3 Creating a script module

Creating a module in PowerShell can sometimes be as easy as renaming a file from a ps1 to a psm1, and it can be as complicated as writing cmdlets in C# that needs to be compiled to a DLL. A module that does not contain a compiled code is a **Script Module**. A script module provides a perfect place for you to store and share a collection of related functions. We will not go in-depth on module creation, as there can be a lot to it, and there are plenty of other books and resources that cover this topic. However, I would like to cover a few essential tips and tricks that you can use to get started creating script modules today.

At its most basic, a module can be a single PowerShell script file as saved a psm1. You can paste all the functions you want into a psm1 file, save it, and load it into PowerShell and

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

be on your way. But this does not lend itself well to versioning or testing. So, at a minimum, you should also include a module manifest (psd1) file. The module manifest can provide details about the module, such as which functions or variables to make public. But most importantly, it contains the modules version number, helping to ensure you always have the latest and greatest version.

Along with the psm1 and psd1 files, you can include additional script files (ps1) with your functions to load with the module. So, instead of creating one massive psm1 file with every function, you create a single script (ps1) file for each function and have them loaded by the psm1.

To show you how simple this process can be, we can take our log file cleanup script from the last section and move the functions from inside the script into a module.

As with anything, the first place to start is with the name. In this case, we can name the module FileCleanupTools. Then, all you need to do is create the folder structure, psd1, and psm1 files. The top-level folder, the psd1, and the psm1 should all have the same name. If they do not have the same name, PowerShell will not be able to load it using the standard `Import-Module ModuelName` command. Next, under the top-level folder, you should create a folder that matches the version number.

PowerShell allows you to have multiple versions of the same module installed at the same time. This is great for testing and upgrading proposes. As long as you nest your module files in a folder with the version number, you can pick and choose which ones you want to load.

> **LOADING DIFFERENT VERSIONS:** By default, PowerShell will load the highest version, so you need to include the -MaximumVersion and -MinimumVersion in the Import-Module command if you want to load a specific version.

The psd1 and psm1 will go inside the version folder. From here, the forward the file and folder structure is up to you. However, the standard practice, and one that I recommend, is to create a folder name Public. The Public folder will contain the ps1 files that will house the functions.

As a result, your typical module will start looking something like what you see in figure 6.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

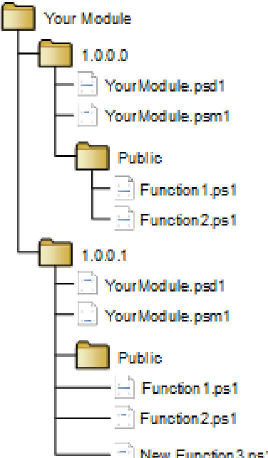


Figure 10: PowerShell Module Version Folders

While this may sound like a lot to remember, you can use PowerShell to streamline your module creation process. As you just saw in the log file clean up, PowerShell can create folders for you. PowerShell also includes the cmdlet `New-ModuleManifest` that you can use to create the psd1 file for you.

When you use the `New-ModuleManifest` cmdlet, you need to specify the module name, the path to the psd1 and psm1 files, and the module version number. You will also want to provide the author's name and the minimum PowerShell version the module can use.

You can do that all at once with the script in listing 9.

Listing 5 Create FileCleanupTools Module

```
$ModuleName    = 'FileCleanupTools'    #A
$ModuleVersion = "1.0.0.0"    #B
$Author        = "YourNameHere"    #C
$PSVersion     = '7.0'    #D

$ModulePath = Join-Path .\ "$($ModuleName)\$($ModuleVersion)"
New-Item -Path $ModulePath -ItemType Directory    #E
Set-Location $ModulePath
New-Item -Path .\Public -ItemType Directory    #F

$ManifestParameters = @{
    ModuleVersion = $ModuleVersion
    Author        = $Author
    Path          = ".\$($ModuleName).psd1"    #G
    RootModule    = ".\$($ModuleName).psm1"    #H
    PowerShellVersion = $PSVersion
}
New-ModuleManifest @ManifestParameters    #I

Out-File -Path ".\$($ModuleName).psm1" -Encoding utf8    #J
```

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

#A The name of your modules
#B The version of your module
#C Your name
#D The minimum PowerShell version this module supports
#E Creates a folder with the same name as the module
#F Creates the public folder to store your ps1 scripts
#G Sets the path to the psd1 file
#H Sets the path to the psm1 file
#I Creates the module manifest psd1 file with the settings supplied in the parameters
#J Creates a blank psm1 file

Now that you have your basic structure, you can add your functions to the Public folder. To do this, create a new PowerShell script file in the folder and give it the same name as the function. Starting with the `Set-ArchiveFilePath` function, create the file **Set-ArchiveFilePath.ps1**. Then do the same for the `Remove-ArchivedFiles` function.

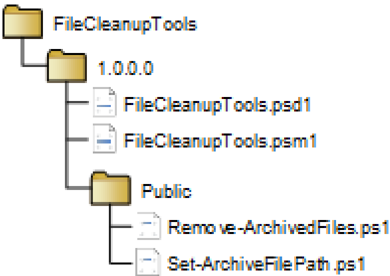


Figure 11: FileCleanupTools module files

From here, you will simply copy and paste the code for each function into their respective file. Be sure that you are only bringing in the function and no other parts of the script when you copy and paste. The file should start with the Function keyword declaring the function, and the last line should be the curly-bracket ending the function. The listing 10 shows what the **Set-ArchiveFilePath.ps1** file should contain.

Listing 6 Set-ArchiveFilePath.ps1

```
Function Set-ArchiveFilePath{
    [CmdletBinding()]
    [OutputType([string])]
    param(
    [Parameter(Mandatory = $true)]
    [string]$ZipPath,

    [Parameter(Mandatory = $true)]
    [string]$ZipPrefix,

    [Parameter(Mandatory = $false)]
    [datetime]$Date = (Get-Date)
    )

    if(-not (Test-Path -Path $ZipPath)){
```

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
        New-Item -Path $ZipPath -ItemType Directory | Out-Null
        Write-Verbose "Created folder '$ZipPath'"
    }

    $ZipName = "$($ZipPrefix)$($Date.ToString('yyyyMMdd')).zip"
    $ZipFile = Join-Path $ZipPath $ZipName

    if(Test-Path -Path $ZipFile){
        throw "The file '$ZipFile' already exists"
    }

    $ZipFile
}
```

Repeat this process for any other functions you want to add to the module.

One thing to note is that just the script files will not be automatically be loaded when you import the module. By default, PowerShell will run the psm1 file on import because it is listed as the RootModule in the psd1 manifest. Therefore, you need to let the module know which files it needs to run to import your functions. The easiest way to do this is by having the psm1 file search the Public folder for each ps1 file and then execute each one to load the function into your current PowerShell session.

The best part about loading the functions this way is that there is nothing you need to update or change when adding a new function. Simply add the ps1 for it to the Public folder, and it will be loaded the next time you import the module.

As long as your functions are in the Public folder, you can do this by adding the code in Listing 11 to your psm1 file.

Listing 7 Load Module Functions

```
$Path = Join-Path $PSScriptRoot 'Public'
$Functions = Get-ChildItem -Path $Path -Filter '*.ps1'    #A

Foreach ($import in $Functions) {    #B
    Try {
        Write-Verbose "dot-sourcing file '$($import.fullname)'"
        . $import.fullname    #C
    }
    Catch {
        Write-Error -Message "Failed to import function $($import.name)"
    }
}
```

#A Get all the ps1 files in the Public folder
#B Loop through each ps1 file
#C Execute each ps1 file to load the function into memory

If you use a different folder structure or multiple folders, you will need to update the first couple of lines, so it knows which folders to look in. For example, if you also add a folder for Private functions, you can pass both folders to the `Get-ChildItem` cmdlet to return all the ps1s between them.

```
$Public = Join-Path $PSScriptRoot 'Public'
$Private = Join-Path $PSScriptRoot 'Private'
$Functions = Get-ChildItem -Path $Public,$Private -Filter '*.ps1'
```

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

Once you have your module file created, you can run the Import-Module command and point it to the path of your module manifest (psd1) file to import the functions into your current PowerShell session. You will need to use the full path unless the module folder is inside a folder included in the `$env:PSModulePath` environment variable. But even if it is, it is good to use the full path for testing to ensure you are loading the correct version.

Also, when you are testing a module, be sure to include the `-Force` switch at the end to force the module to reload and pick up any changes you have made. You can also provide the `-PassThru` switch to ensure that your functions are loaded.

```
P:\FileCleanupTools> Import-Module .\FileCleanupTools.psd1 -Force -PassThru

ModuleType Version  Name             ExportedCommands
---------- -------  ----             ----------------
Script     1.0.0.0  FileCleanupTools {Remove-ArchivedFiles,
                                       Set-ArchiveFilePath}
```

Once your module is ready, you can remove the functions from the original script and add one line of code to import the module.

Listing 8 Moving Functions to Module

```
param(
    [Parameter(Mandatory = $true)]
    [string]$LogPath,

    [Parameter(Mandatory = $true)]
    [string]$ZipPath,

    [Parameter(Mandatory = $true)]
    [string]$ZipPrefix,

    [Parameter(Mandatory = $false)]
    [double]$NumberOfDays = 30
)

Import-Module FileCleanupTools    #A

$Date = (Get-Date).AddDays(-$NumberOfDays)
$files = Get-ChildItem -Path $LogPath -File |
    Where-Object{ $_.LastWriteTime -lt $Date}

$ZipParameters = @{
    ZipPath = $ZipPath
    ZipPrefix = $ZipPrefix
    Date = $Date
}
$ZipFile = Set-ArchiveFilePath @ZipParameters

$files | Compress-Archive -DestinationPath $ZipFile

Remove-ArchivedFiles -ZipFile $ZipFile -FilesToDelete $files
```

#A Replaced functions with the command to load the FileCleanupTools module

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

2.5.4 Module creation tips

There are a few things you can do to help yourself and others when you are creating a module.

USE COMMON NAMING AND STYLING

You may have noticed by now that the majority of PowerShell commands follow the same naming conventions and style. This includes using the Verb-Noun naming pattern for cmdlets and functions (Get-Module, Import-Module, etc.). When deciding what verb to use, it is best to stick with the list of approved verbs. You can find the list of approved verbs by running the command `Get-Verb`. Also, the noun should be singular—E.g. `Get-Command`, not `Get-Commands`.

Module names, parameter names, and variables use Pascal casing. Pascal casing is where each proper word in the name starts with an upper-case letter. For example, the module we just created is named, FileCleanupTools. Some people will use Camel casing for local variables and Pascal for global variables, but this is not universal. Camel casing is like Pascal casing, except the first word is lower-case (ex. fileCleanupTools).

Do not be afraid of long variable and parameter names. It is much better to be descriptive than short. For example, a parameter named `$NumberOfDays` is much clearer than one named `$Days`.

While none of these styles are required, you could create a function named `archive-File-Deleting`, and it would work fine. Of course, you would receive a warning when you import the module, but that's it. However, others used to the standard PowerShell naming conventions may find it challenging to find and use your functions.

SEPARATE PRIVATE AND PUBLIC FUNCTIONS

In our example, we create a folder named Public and put our functions ps1 files in it. This was done to let others know these are functions that they can use in their scripts. There may be times where you write what are referred to as helper functions. These are functions that other functions in the module can call, but the users should not need to call directly. You can have these for a whole host of reasons. For example, you may have a function that parses some data and is used by several functions in your module, but there would be no reason for the end-user to need it. In this case, it is best to make it a private function. To do this, you can create a second folder in your module named Private to hold these files. Then update the import in your psm1 to also import the Private folder.

The thing to keep in mind here is that just because they in the Private folder, it doesn't mean they will be hidden. The Public and Private folder names are just suggestions. You could name them anything. They are just a common way that others writing PowerShell modules use to help keep things organized. You can put them all together in one folder, or multiple subfolders, or really any way you like, as long as the psm1 file can find them. To make them private, you have to delist them from the manifest (psd1) file. If you look inside the FileCleanupTools.psd1 you created earlier, you will see the line `FunctionsToExport = '*'`. This line tells the module to export all functions that match this pattern. Since the pattern is just a single wildcard, everything will match. As there is no exclude line, the only way to exclude something is by not listing it. This is where the Public and Private folders

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

come in handy. You can simply update the manifest to only include the functions from the public folder. So it would look like this, for the FileCleanupTools.psd1.

```
FunctionsToExport = 'Remove-ArchivedFiles', 'Set-ArchiveFilePath'
```

When you do this, be sure to remember to update the manifest any time you want to add a new public function.

INSTALL CUSTOM MODULES IN PSMODULEPATH

Have you ever wondered how PowerShell knows where the files are when you type `Import-Module` and just give a module name and not a path as we did in the previous example? It will do this for any module installed in a folder listed in the `$env:PSModulePath` environmental variable. So it is best to always install your modules to a path listed in there. But be careful as there are system (AllUsers) and user-level (CurrentUser) scopes. If you are creating an automation that will run under the system account, it is best to use the AllUsers scope. The default for this on Windows is `$env:ProgramFiles\PowerShell\Modules`. You can confirm what folders are included by checking the `$env:PSModulePath` variable in your PowerShell console.

LISTING DEPENDENCIES IN THE MANIFEST

There is a parameter in the manifest file named, `RequiredModules`. This parameter will allow you to list modules that are required for this module. For example, if you wrote a module to work with Active Directory users, you would want to ensure that the ActiveDirectory module is loaded along with your module. The problem you may run into with the `RequiredModules` parameter is, it does not import the module if it has not already been imported, and it does not check versions. Therefore, I often find myself handling module dependencies directly in the psm1 file.

Listing 9 Import Required Modules

```
[System.Collections.Generic.List[PSObject]]$RequiredModules = @()
$RequiredModules.Add([pscustomobject]@{    #A
    Name = 'Pester'
    Version = '4.1.2'
})

foreach($module in $RequiredModules){    #B
    $Check = Get-Module $module.Name -ListAvailable    #C

    if(-not $check){    #D
        throw "Module $($module.Name) not found"
    }

    $VersionCheck = $Check |    #E
        Where-Object{ $_.Version -ge $module.Version }

    if(-not $VersionCheck){    #F
        Write-Error "Module $($module.Name) running older version"
    }

    Import-Module -Name $module.Name    #G
```

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
}
```

#A Create an object for each module to check
#B Loop through each module to check
#C Check if the module is installed on the local machine
#D If not found, throw a terminating error to stop this module from loading
#E If it is found, check the version
#F If an older version is found, write an error but do not stop
#G Import the module into the current session

2.6 Summary

- Knowing what to automate and what not to automate are keys to your success.
- Applying the concepts of phases and building blocks to your automation projects will help you see benefits sooner and provide a solid foundation to grow.
- Building blocks can be translated to PowerShell functions.
- Functions should only perform one action that can be restarted if something goes wrong.
- Functions should be stored in PowerShell modules to allow other automations and people to use them.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

3
Scheduling automation scripts

This chapter covers

- **How to schedule scripts.**
- **Considerations for scheduled script**
- **Creating continuously running scripts.**

When starting their PowerShell automation journey, one of the first things everyone wants to learn about is scheduling scripts to run unattended. In this chapter, you will learn more than just how to schedule a script. You will also learn some best practices, using some common real-world scenarios that will help ensure your scheduled scripts run smoothly. The concepts and practices used in these examples can be applied to any script you need to schedule.

It is tempting to say you can take any existing PowerShell script and schedule it with a job scheduler, but that is only part of the solution. Before jumping straight into scheduling a PowerShell script, you need to ensure that your script is adequately written to handle being run in an unattended manner. This includes many of the previously covered concepts, like ensuring dependencies are met and ensuring there are no user prompts.

There are several different types of scripts you will want to run on a scheduled basis. Two of the most common are Scheduled Script and Watcher Scripts. A scheduled script is set to run regularly, but not so often that it is continuously running. On the other hand, a watcher runs either continuously or at least every few minutes or so. This chapter will cover both types and the different things to consider when coding them, and the considerations to make when scheduling them.

3.1 Scheduled scripts

A scheduled script is run on a fairly regular basis but does not need to be real-time. Some good examples of these are scripts to collect inventory, check on user accounts, or check on system

©Manning Publications Co. To comment go to liveBook
Licensed to Tausif Khan <tsfahmed2@gmail.com>

resources, run data backups, etc. No matter what your script does, you need to take care of before setting it to run on a schedule.

3.1.1 Know your dependencies and take care of them beforehand

If your script is dependent on any modules, be sure that these are installed on the system running the script before scheduling it. You also need to be aware of how these dependencies may affect other scheduled scripts. For example, if you have two scripts that require different versions of the same module, you need to ensure that both versions are installed instead of just the highest version.

Do not try to have your scheduled scripts install modules because it can lead to all sorts of unintended consequences. For example, it could fail to properly install the module, causing the scheduled script to never execute successfully. Or you could create a situation where two scripts continually override each other, taking up valuable system resources and causing failures between each other.

3.1.2 Know where your script needs to execute

This one sounds simple, but there are situations where a script executing in the wrong environment or on the wrong server can cause you problems. A typical example is ensuring the script has network access to the required systems. For instance, if you need to connect to AWS or Azure from an on-premises script, you need to ensure no proxies or firewalls are blocking it.

There are also ones that may not seem as obvious. For example, if you want to force the Azure AD Connector to sync using PowerShell, that script has to run on the server with the connector installed. Another one I've run into multiple times is dealing with Active Directory replication. If you have a script that creates an Active Directory user, then connects to Exchange, you can run into problems due to replication. For example, if you create the account on a domain controller in a different site than the Exchange server, it may not be able to see the account when your script tries to create the mailbox, causing your script to fail.

3.1.3 Know what context the script needs to execute under

In conjunction with knowing your dependencies and where your script needs to execute is know what context it needs to run under. Most job schedulers can run scripts as a particular user or as the system. If you need to authenticate with Active Directory, SQL, or a network share, chances are you will need to run under a user context. If you are collecting data about the local machine, then it can run under the system account.

Knowing the context will also help in setting your dependencies. PowerShell modules can be installed at the user level or the system level. You may test your script under your account, by it fails to load the modules when run through the job scheduler. This can be due to the modules being installed under your account only. Therefore, I suggest you always install modules to the system level to avoid these types of problems.

©Manning Publications Co. To comment go to liveBook
Licensed to Tausif Khan <tsfahmed2@gmail.com>

3.2 Scheduling your scripts

As with most things in PowerShell, there are several different ways you can schedule scripts to run regularly. You can use anything from the built-in Windows Task Scheduler to enterprise-level job schedulers like Control-M or JAMS. Also, many other automation platforms have built-in schedulers like System Center Orchestrator/SMA, Ansible, ActiveBatch, and PowerShell Universal. There are also several cloud-based solutions that can run PowerShell scripts both in the cloud and in your on-premises environment. You will find these covered in the later cloud chapter. It will be up to you to choose the tool that best fits your environment.

A scheduled script is run on a fairly regular basis but does not need to be real-time. Some good examples of these are scripts to collect inventory, check on user accounts, or check on system resources, run data backups, etc.

Whichever tool you choose, the process remains the same.

1. Create your script
2. Copy it where the scheduler can access it
3. Ensure dependencies are meet
4. Set required permissions
5. Schedule it.

The Log File Clean Up script from chapter 2 is a perfect example of a script you would want to schedule to run. You can use it to practice creating a scheduled job using the Windows Task Scheduler, Cron, and Jenkins.

HELPER SCRIPTS: A copy of the script “Invoke-LogFileCleanup.ps1” and the module folder “FileCleanupTools” are available in the Helper Script folder for this chapter.

3.2.1 Task Scheduler

Task Scheduler is by far the most popular tool for scheduling scripts in a Windows environment. It does have one major drawback in that there is no central console for it, but it is easy to use and has been built into the Windows operating system since Windows NT 4.0.

When setting up any job to run through Task Scheduler, you need to take into consideration permissions. Permissions to access the script file and permissions to access the required resources. For this exercise, you can assume the logs are on the local machine, so you can run under the system account.

However, you could, for example, place the script file on a network share. Having the script in a network share is a great way to help maintain a single script file and not have individual copies on every server. The downside is you need to ensure that Task Scheduler can access it. The best way to do this is with a service account. You never want to use your personal account for a scheduled task. Besides the obvious security risk, it’s also a great way to ensure your account will get locked out the next time you change your password. Your other options include creating a completely unrestricted share or giving each computer that runs the script explicit access to the share. Both of which are huge security risks and can make maintaining a nightmare.

©Manning Publications Co. To comment go to liveBook
Licensed to Tausif Khan <tsfahmed2@gmail.com>

If you are reading this book, you are more than likely very familiar with the Windows Task Scheduler. However, I would like to cover a few things that you should consider when creating scheduled tasks for PowerShell scripts.

INSTALLING YOUR CUSTOM MODULE OPTIONS

Since this script uses functions in a custom module, you must copy the module folder somewhere the script can access it. The default path for the PowerShell modules, which will be automatically loaded at runtime, are:

- **PowerShell v5.1** : C:\Program Files\WindowsPowerShell\Modules
- **PowerShell v7.0** : C:\Program Files\PowerShell\7\Modules

To install the module for Task Scheduler to use, just copy the folder FileCleanupTools folder from chapter 2 to one of the folders above.

SECURITY OPTIONS

Typically you want your automations to run unattended. Therefore, you want to select *Run whether user is logged on or not*. From there, you have two options. First, you can select *Do not store password* to have the task run under the system context. This is fine as long as everything the script interacts with is on the local machine.

Second, if you need to interact with other systems, network shares, or anything that requires user authentication, leave *Do not store password* unselected. Then click on *Change User or Group* to select the service account. You will receive a prompt to provide the password when you save the task.

©Manning Publications Co. To comment go to liveBook
Licensed to Tausif Khan <tsfahmed2@gmail.com>

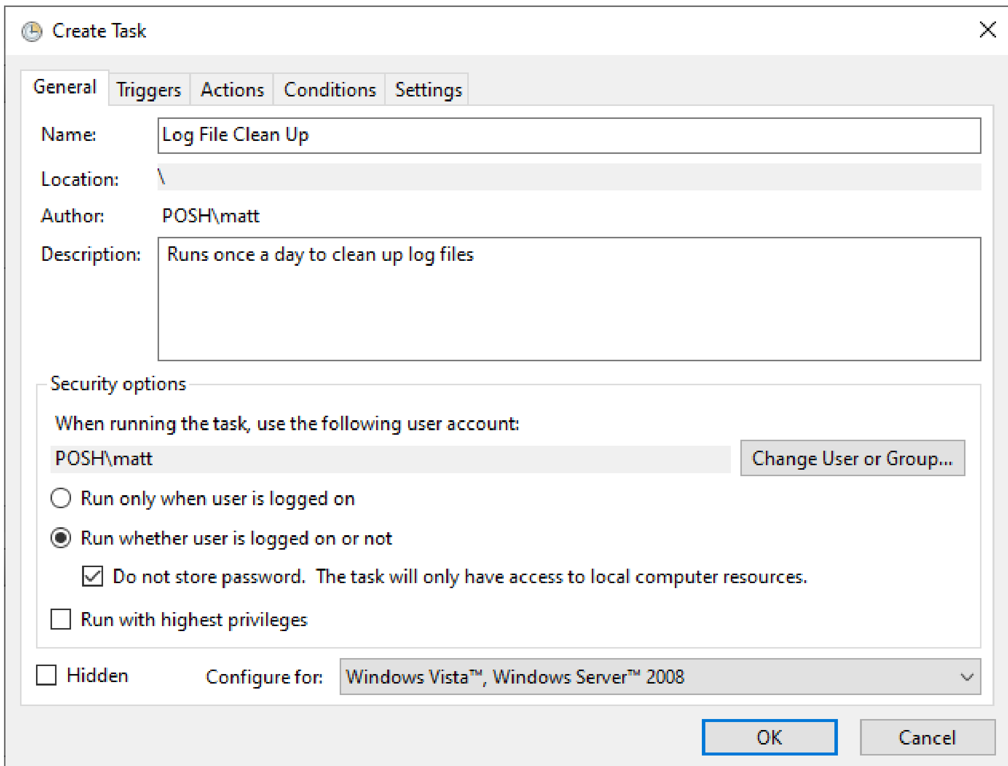


Figure 1: Create scheduled task window

CREATING POWERSHELL ACTIONS

When you create an Action to execute a PowerShell script, you cannot just set the *Program/Script* box to the PowerShell script file (ps1). Instead, you need to set the *Program/Script* box to the PowerShell executable. Then your script will go in the *Add arguments* box.

The default path for the PowerShell executables that will go in the *Program/Script* box are:

- **PowerShell v5.1** : C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
- **PowerShell v7.0** : C:\Program Files\PowerShell\7\pwsh.exe

Then in the *Add arguments* box, you provide the `-File` argument with the path to your script and the values to pass for any parameters.

```
-File "C:\Scripts\Invoke-LogFileCleanup.ps1" -LogPath "L:\Logs\" -ZipPath "L:\Archives\" -
      ZipPrefix "LogArchive-" -NumberOfDays 30
```

Also, if you are using PowerShell v5.1 or below, you will most likely also want to include the –`WindowStyle` argument and have it set to Hidden, so your script will run silently.

©Manning Publications Co. To comment go to liveBook
Licensed to Tausif Khan <tsfahmed2@gmail.com>

JOB LOGS

You can review the results of all past executions and see any errors in the History tab for each job. All job logs are written to the Event Viewer log Microsoft-Windows-TaskScheduler/Operational. While there is no central console to manage jobs, you can use event forwarding to collect all the job logs in one central location.

3.2.2 Create scheduled task via PowerShell

Since Task Scheduler does not have a central console for you to see and schedule across multiple computers at once, the next best thing is to create your scheduled tasks via a PowerShell script. This will help ensure that all computers end up with the same configuration.

When creating a scheduled task through PowerShell, you have two main options. One is to create a script where you explicitly define the required parameters, and the other is to export an existing scheduled task. Whichever way you choose, you will use the `Register-ScheduledTask` cmdlet that is part of the ScheduledTasks module. This module is included with PowerShell, so there is nothing special you need to install.

CREATE A NEW SCHEDULED TASK

Creating a scheduled task via PowerShell is very similar to the process of creating it in the console. You set the time interval for the trigger, define the actions, assign permissions, then create the task. Using the previous example of setting the Invoke-LogFileCleanup.ps1 script to run once a day at 8 a.m., let's look at how you can create this task.

First, you need to define your trigger. This can be done using the `New-ScheduledTaskTrigger` cmdlet. You can set triggers to run once, daily, weekly, or at logon. In this last case, you will use the `-Daily` switch and specify the `-At` parameter to 8 a.m.

Next, you need to define your action. This is done using the `New-ScheduledTaskAction` cmdlet. Here you need to supply the path to the executable and the arguments. Remember, the executable will be the path to the PowerShell executable and not the ps1 file. Then in the arguments, you supply the `-File` parameter, path to the ps1, and any parameters for that ps1.

When you enter the argument, it needs to be a single string, so watch out for things like escaping quotes and spaces. Safest to create it is by using single quotes on the outside and double quotes on the inside. When you create a string with single quotes in PowerShell, it takes the text between them as literal characters and does not try to interpret anything. You can see that in the snippet below, where the final string contains all the double quotes.

```
PS P:\> $Argument = '-File ' +
     >>    '"C:\Scripts\Invoke-LogFileCleanup.ps1"' +
     >>    ' -LogPath "L:\Logs\" -ZipPath "L:\Archives\"' +
     >>    ' -ZipPrefix "LogArchive-" -NumberOfDays 30'
PS P:\> $Argument

-File "C:\Scripts\Invoke-LogFileCleanup.ps1" -LogPath "L:\Logs\" -ZipPath "L:\Archives\" -
       ZipPrefix "LogArchive-" -NumberOfDays 30
```

Once you have your action and your trigger defined, you can create the scheduled task using the `Register-ScheduledTask` cmdlet.

©Manning Publications Co. To comment go to liveBook
Licensed to Tausif Khan <tsfahmed2@gmail.com>

When creating the scheduled task, you have several options to choose from when setting the permissions. By default, the schedule is set to *Run only when user is logged on*. However, since you will want it to run unattended, you will need to use the `-User` argument to set it to *Run whether user is logged on or not*.

Just like when creating it through the console, you can choose to have it run as the system account or using a service account. To use the system account, just set the value for the `-User` argument to *NT AUTHORITY\SYSTEM*. If you decide to use a service account, you will also need to supply the password using the `-Password` argument.

NOTE: Be aware that the password argument is a plain text string, so be sure not to save it in any scripts.

One more thing to consider before creating your task is the name. I strongly recommend that you create a sub-folder in the *Task Scheduler Library* to group similar automations. You can do this by adding the folder name followed by a backslash (\) before the task's name in the `-TaskName` argument.

Finally, you supply the `-Trigger` and `-Action` parameters with the trigger and action you created.

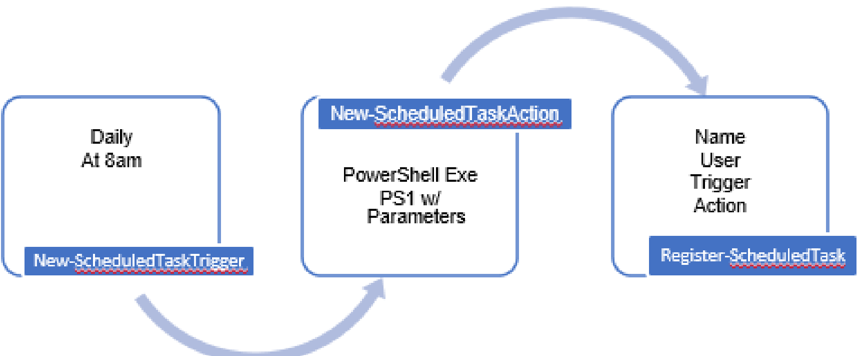


Figure 2:

Listing 1 Create Scheduled Task

```
$Trigger = New-ScheduledTaskTrigger -Daily -At 8am    #A

$Execute = "C:\Program Files\PowerShell\7\pwsh.exe"    #B
$Argument = '-File ' +    #C
    '"C:\Scripts\Invoke-LogFileCleanup.ps1"' +
    ' -LogPath "L:\Logs\" -ZipPath "L:\Archives\"' +
    ' -ZipPrefix "LogArchive-" -NumberOfDays 30'

$ScheduledTaskAction = @{    #D
    Execute = $Execute
```

©Manning Publications Co. To comment go to liveBook
Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
    Argument = $Argument
}
$Action = New-ScheduledTaskAction @ScheduledTaskAction

$ScheduledTask = @{    #E
    TaskName = "PoSHAutomation\LogFileCleanup"
    Trigger  = $Trigger
    Action   = $Action
    User     = 'NT AUTHORITY\SYSTEM'
}
Register-ScheduledTask @ScheduledTask
```

#A Create Scheduled Task trigger
#B Set Action execution path
#C Set Action arguments
#D Create the Scheduled Task Action
#E Combine the trigger and action to create the Scheduled Task

> **NOTE:** You may receive an access denied message when running the Register-ScheduledTask cmdlet. To avoid this you can run the command from an elevated PowerShell session.

After you run this, you should see your scheduled task under the folder PoSHAutomation.

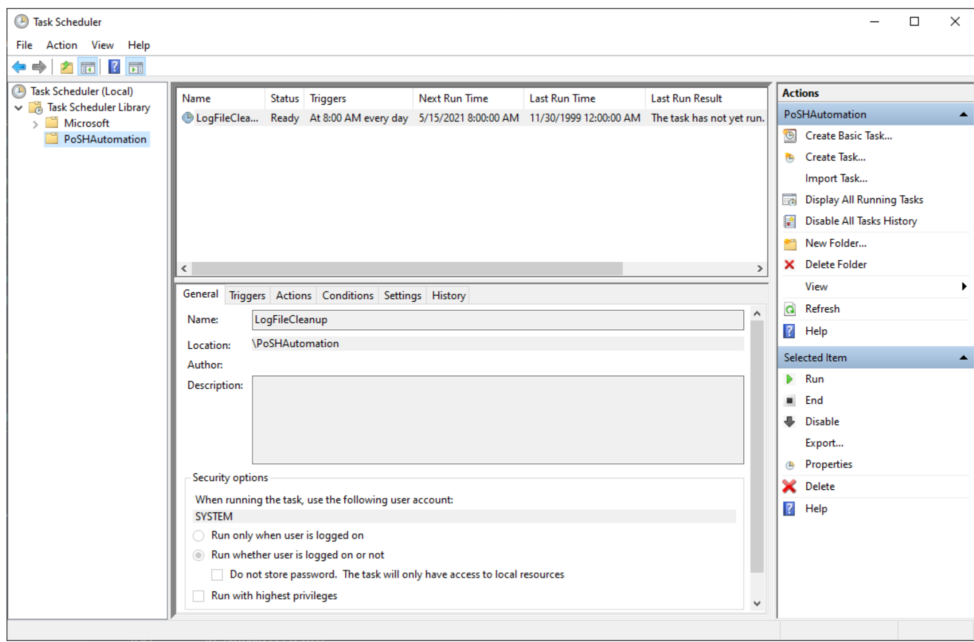


Figure 3: Task Scheduler custom folder

©Manning Publications Co. To comment go to liveBook
Licensed to Tausif Khan <tsfahmed2@gmail.com>

EXPORTING AND IMPORTING SCHEDULED TASKS

The best way to ensure that your jobs remain consistent across multiple computers is by using the import and export functionality available with Task Scheduler. You can use the `Export-ScheduledTask` cmdlet to export any task to XML. You can then use this XML with the `Register-ScheduledTask` cmdlet to recreate the task on any other Windows computer.

You first start by exporting the task and saving the XML output to a file. It is a good idea to save these to a file share so that you can easily import them to any other machine on your network. The `Export-ScheduledTask` cmdlet outputs a string with the XML data in it, so you can simply save it by piping the output to a file using the `Out-File` cmdlet.

```
PS P:\> $ScheduledTask = @{
     >>    TaskName = "LogFileCleanup"
     >>    TaskPath = "\PoSHAutomation\"
     >>}
PS P:\> $export = Export-ScheduledTask @ScheduledTask
PS P:\> $export | Out-File "\\srv01\PoSHAutomation\LogFileCleanup.xml"
```

From there, you can recreate the task on any other computer by importing the contents of the XML file, then running the `Register-ScheduledTask` passing in the XML string to the `-Xml` argument. However, you will want to note that even though the XML file contains the task's name, you still have to supply the `-TaskName` parameter. Well, luckily, you can convert XML directly to a PowerShell object. So, with a couple of extra lines of code, you can extract the name of the job from the XML to automatically populate the `-TaskName` parameter for you.

Listing 2 Importing a Scheduled Task

```
$FilePath = ".\CH03\Monitor\Export\LogFileCleanup.xml"
$xml = Get-Content $FilePath -Raw    #A
[xml]$xmlObject = $xml    #B
$TaskName = $xmlObject.Task.RegistrationInfo.URI    #C
Register-ScheduledTask -Xml $xml -TaskName $TaskName    #D
```

#A Import the contents of the XML file to a string
#B Convert the XML string to an XML object
#C Set the task name based on the value in the XML
#D Import the scheduled task

You can even take it one step further and import all the XML files from a single directory to create multiple jobs at once. You can use the `Get-ChildItem` cmdlet to get all the XML files in a folder and then use a `foreach` to import each one of them

Listing 3 Importing Multiple a Scheduled Tasks

```
$Share = "\\srv01\PoSHAutomation\"
$TaskFiles = Get-ChildItem -Path $Share -Filter "*.xml"    #A

foreach($task in $TaskFiles){    #B
    $xml = Get-Content $FilePath -Raw
    [xml]$xmlObject = $xml
    $TaskName = $xmlObject.Task.RegistrationInfo.URI
    Register-ScheduledTask -Xml $xml -TaskName $TaskName
}
```

©Manning Publications Co. To comment go to liveBook
Licensed to Tausif Khan <tsfahmed2@gmail.com>

#A Get all the XML files in the folder path
#B parse through each file and import the job

REGISTER-SCHEDULEDJOB

If you've been using PowerShell for a while, you may be aware of the cmdlet `Register-ScheduledJob`. This cmdlet is very similar to the `Register-ScheduledTask` cmdlet, with one major caveat. The `Register-ScheduledJob` cmdlet is not in PowerShell core. The way it works is entirely incompatible with .Net Core, and starting in PowerShell 7 has been blocked from even being imported using the PowerShell compatibility transport. Therefore, I highly recommend you switch any existing scripts from `Register-ScheduledJob` to `Register-ScheduledTask`.

3.2.3 Cron scheduler

If you are new to Linux or just not familiar with Cron, it is the Linux equivalent of Task Scheduler. Or really the other way around since Cron was originally built by Bell Labs in 1975. Either way, it is an excellent tool for scheduling recurring tasks on a Linux computer and is installed by default on pretty much every distro. It is a very robust platform with many options, but we are just going to focus on how you can use it to run PowerShell scripts.

Unlike Task Scheduler, Cron does not have a GUI. Instead, you control everything through command lines and a Cron Table file, know as CronTab, that contains all the jobs for that particular user on that computer. Like Task Scheduler, in Cron, all you need to do is set the schedule, set permissions, and set the action to call your script.

The script part is easy. It is essentially the same command you used for Task Scheduler, just with paths written for Linux. For example, your command to call the Invoke-LogFileCleanup.ps1 script would look something like this.

```
/snap/powershell/160/opt/powershell/pwsh -File "/home/posh/Invoke-LogFileCleanup.ps1" -LogPath
      "/etc/poshtest/Logs" -ZipPath "/etc/poshtest/Logs/Archives" -ZipPrefix "LogArchive-" -
      NumberOfDays 30
```

Prior to creating your Cron job, you can test the execution using Terminal. If your command runs successfully through Terminal, then you know it will run through Cron.

To create your Cron job, open Terminal and enter the command as follows.

```
crontab -e
```

This will open the Crontab file for the current user. If you want the job to run as a different user, enter the command with the `-u` argument followed by the account's username.

```
crontab -u username -e
```

If this is your first time opening Crontab, you may see a prompt to select an editor. Select your preferred one and continue.

Now it is time to create your job. The syntax to create the job is the Cron syntax for the schedule followed by the command. Again, we won't go into too much detail on Cron syntax as there are plenty of resources out there on it. Just know that it consists of 5 columns that represent minute, hour, day of the month, month, and day of the week.

©Manning Publications Co. To comment go to liveBook
Licensed to Tausif Khan <tsfahmed2@gmail.com>

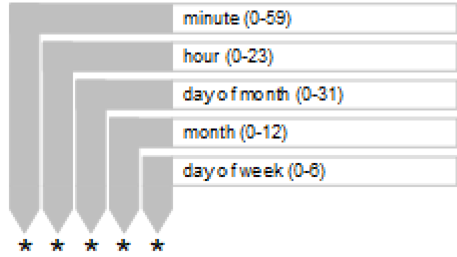


Figure 4: Cron Schedule

To run the script at 8 a.m., just like the Windows computer, your syntax will be * 8 * * *. Enter the syntax for the time interval followed by the command to execute, similar to the following.

```
* 8 * * * /snap/powershell/160/opt/powershell/pwsh -File "/home/posh/Invoke-LogFileCleanup.ps1" -
       LogPath "/etc/poshtest/Logs" -ZipPath "/etc/poshtest/Logs/Archives" -ZipPrefix "LogArchive-
       " -NumberOfDays 30
```

Then just save your changes and close the crontab file. As long as the Cron service is running on your computer, this job will execute at 8 a.m. every day.

3.2.4 Jenkins scheduler

As mentioned earlier, there are numerous tools out there that can support executing PowerShell. Jenkins is an open-source automation server, that while originally built as a continuous integration tool, has grown much larger. As with any tool, it has its pluses and minuses. A big plus is that it has a web UI that you can use to manage all of your jobs in one place. It also has the ability to use role-based access and store credentials. This will allow you to give others access to execute scripts on systems or environments without needing to provide them with explicit permissions to that system.

One downside to Jenkins is that executing PowerShell on remote servers can be tricky. Jenkins will execute the PowerShell script on the Jenkins server. So, if you need it to run the script on a different server, you will need to use PowerShell remoting. We will cover PowerShell remoting in depth in chapter 5, so for this example, we are fine with the script running on the Jenkins server.

If you followed the environment setup guide in the appendix, you should be all set to perform this exercise. We will once again schedule your disk space usage script to run on a schedule. This time using Jenkins.

Before you copy your script to Jenkins, there is one thing you will need to change. Jenkins does not have the ability to pass parameters to a script the same way you can from the command line. Instead, it uses environment variables. The easiest way to account for this is by replacing your parameter block and defining the parameters as values. Then set a value for each variable to an environmental variable with the same name. This will prevent you from having to rewrite every instance where the parameter is used inside the script. The parameters in the log file cleanup should look like this.

©Manning Publications Co. To comment go to liveBook
Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
$LogPath = $env:logpath
$ZipPath = $env:zippath
$ZipPrefix = $env:zipprefix
$NumberOfDays = $env:numberofdays
```

One thing to be aware of is Jenkins environmental variable should be all lowercase in your script. Once you have your parameters updated, it is time to create the job in Jenkins.

1. Open your web browser and log into your Jenkins instance.
2. Click *New Item*.
3. Enter a name for your project.
4. Select *Freestyle project*.
5. Click *OK*.
6. Check the box *This project is parameterized*.
7. Click the *Add Parameter* button and select *String*.
8. In the *Name* field, enter the name of your parameter *LogPath*.
9. In the Default Value field, enter the path to the log files.

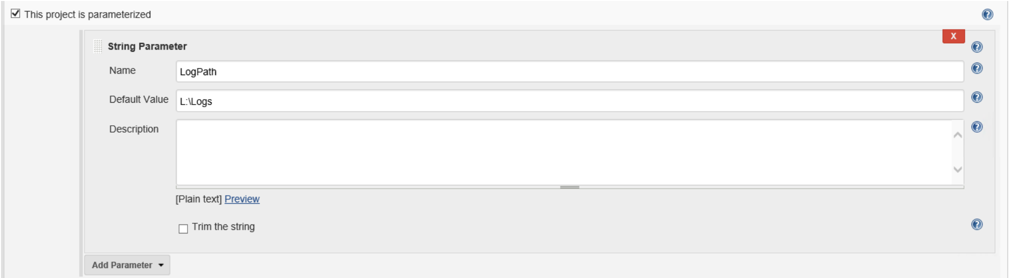


Figure 5: Add Jenkins parameters

10. Repeat steps 7-9 for the *ZipPath*, *ZipPrefix*, and *NumberOfDays* parameters.
11. Scroll down to the *Build Triggers* section.
12. Check the box *Build periodically*.
13. The syntax for the *Schedule* is the same as Cron, so to run at 8 a.m. every day enter * *8* * * *.

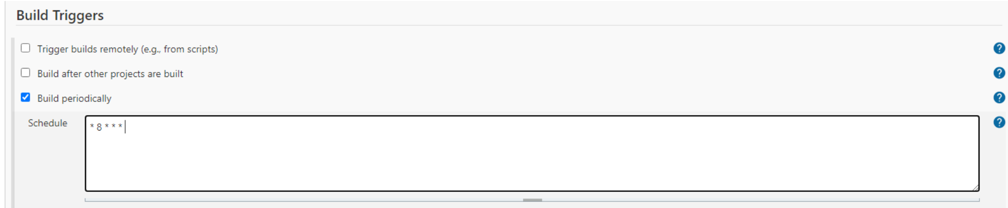


Figure 6: Jenkins Trigger

©Manning Publications Co. To comment go to liveBook
Licensed to Tausif Khan <tsfahmed2@gmail.com>

14. Scroll down to the *Build* section and click the *Add build step* button.
15. Select *PowerShell* from the dropdown.
16. Copy and paste your *Invoke-LogFileCleanup.ps1* script, with the replaced parameters, into the *Command* block.

Figure 7: Jenkins Script

17. Click *Save*.
18. You can test your job right away by clicking *Build with Parameters* button.
19. Click *Build*.
20. When your script is finished executing, you will see the job listed under the *Build History* section.

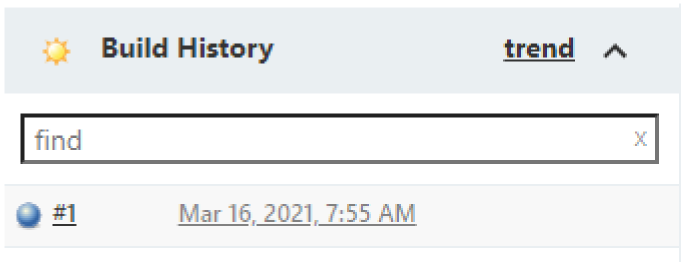


Figure 8: Jenkins Build History

21. If you click on any entry under the build history, you can then view the Console Output of your job.

Figure 9: Jenkins Console Output

3.3 Watcher scripts

A Watcher script is a scheduled script that runs either continuously or at least every few minutes. A good rule of thumb is any script that needs to run every 15 minutes or less should be considered a watcher. A typical example of this can be a file watcher, where you monitor a folder for new or updated files. Other examples are monitoring a SharePoint list for new entries or checking a shared mailbox for new emails. They can also be used for real-time monitoring, like alerting on stopped services, or an unresponsive web app.

When creating a watcher script, you need to consider all the same things you do with any unattended scripts (dealing with dependencies, making data dynamic, preventing user inputs, etc.). However, with watcher scripts, there is one thing that needs to be at the forefront of your mind during the creation process, and that is execution time. For example, if you have a script that executes once a minute and takes 2 minutes to run, you will run into lots of problems.

As you will see, one way to reduce your watcher script's runtime is by having it use an Action script. Think of it as a watcher script is monitoring for a specific condition. Then once that condition is met it the action script will invoke the action script. Since the action script runs in a separate process, your watcher script will not have to wait for the action script to finish executing. Also, you can invoke multiple action scripts from a single watcher, allowing them to run in parallel.

For example, consider a script that performs the following; once a minute, check an FTP site for new files, copy those files to different folders based on the names. If written as a traditional PowerShell script, it will process the files one at a time, which means that you need to account for the time it takes to determine where to copy it, and the amount of time it takes to copy. In contrast, if you invoke an action script, multiple files can be processed at once, resulting in a much faster execution time and not causing the watcher to continually wait.

Another advantage to using action scripts is that they execute as a separate process from the watcher script. So, any errors, delays, or problems they experience will not affect the action script.

©Manning Publications Co. To comment go to liveBook
Licensed to Tausif Khan <tsfahmed2@gmail.com>

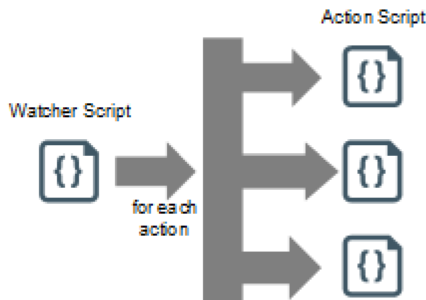


Figure 10: Watcher and action script interactions

The concept of watcher and action scripts is not something that is part of the PowerShell framework. It is an automation process that I have used for years when building automations. The concepts behind it can be translated to any language or platform. There were attempts to build it into the PowerShell framework years ago when PowerShell Workflow was introduced as part of the Service Management Automation (SMA) platform. If you never heard of PowerShell Workflow or SMA, don't feel bad; most people haven't. The advantages that it brought of parallel processing and resuming can now all be achieved natively in PowerShell.

The concepts you learn here can be made into building blocks that you can use through any automation you create.

3.3.1 Designing watcher scripts

Since watcher scripts tend to run every minute or so, the most important thing to consider is execution time. While the use of action scripts can help prevent your script from running too long, there is always the chance that an unforeseen situation will arise, causing your script to run longer than you intended. However, through some good coding practices, you can prevent this from causing problems with your automations.

Before creating your watcher script, you must first know how often it will need to run. You need to do this to ensure that your watcher script executions will not overlap with each other. If there is a potential for overlap, you need to design a way for your script to gracefully exit before that happens. You also need to develop the watcher script to pick up where it last left off. We will walk through these concepts while building a folder watcher.

Consider the following scenario; you need to monitor a folder for new files. Once a file is added, it needs to be moved to another folder. This monitor needs to be as real-time as possible so it will run once every minute. This is a common scenario that you could implement for multiple reasons, including monitoring an FTP folder for uploads from customers or monitoring a network share for exports from your ERP or payroll systems.

The first thing you need to do is break down the steps your automation will need to perform. Then determine which functionality needs to be in the watcher script and which should go in the action script. Keeping in mind that the goal is to have the watcher script both run as efficiently as possible and the ability to pick up from where the last one left off.

©Manning Publications Co. To comment go to liveBook
Licensed to Tausif Khan <tsfahmed2@gmail.com>

The first thing you need to do is get the files in the folder. Then for every file found, it needs to be moved to another folder. Since you are dealing with data, you want to be careful not to overwrite or skip any files. To help address this concern, you can have the script check if there is already a file with the same name in the destination folder. If there is, some of your choices are to:

- Overwrite the file
- Skip moving the file
- Error out
- Rename the file.

If you skip moving the file or error out, you will cause problems with subsequent executions because it will keep picking up that same file again and again. Renaming the file would be the safest option. However, while you are preventing data loss, duplicating data can be a problem of its own. If the same file keeps getting added to the source folder, it will just keep renaming and moving, causing massive duplicates in the destination folder. To prevent this from happening, you can implement a hybrid process to check if the files with the same names are indeed the same file.

You can check to see if the file size is the same, the last write times match, and the file hash is the same. If all of these values match, then it is safe to say it is the same file and can be overwritten. If any one of those checks fails, then it will be renamed and copied. This will help ensure that you don't overwrite any data, you are not duplicating existing data, and that all files are removed from the source folder.

So, the process for the automation will be:

©Manning Publications Co. To comment go to liveBook
Licensed to Tausif Khan <tsfahmed2@gmail.com>

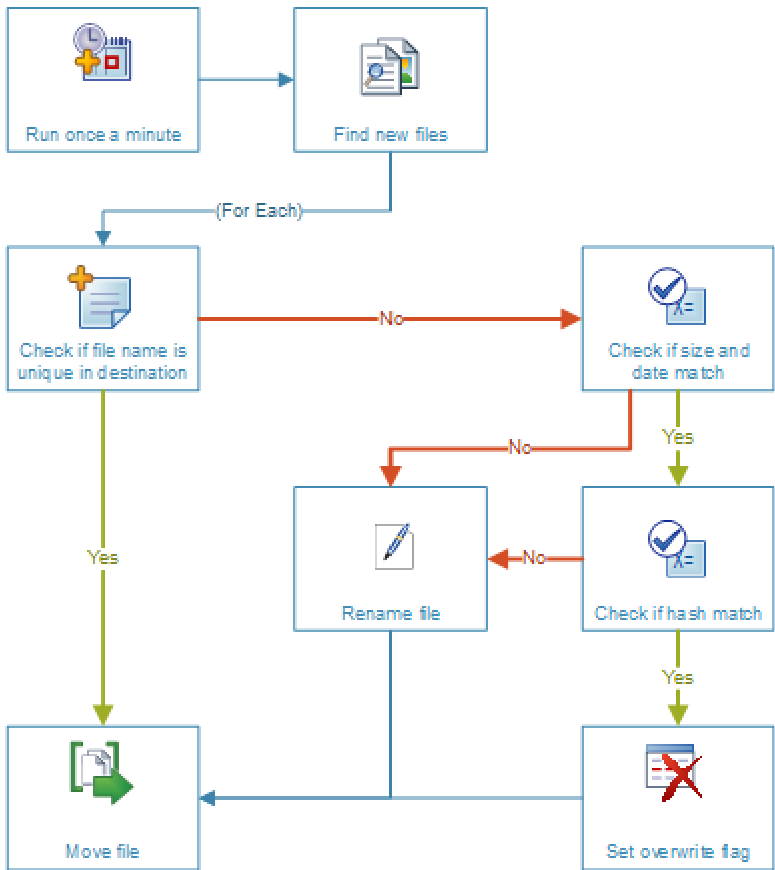


Figure 91: File watcher automation required workflow

The next step is to determine which steps will be in the watcher and which will be in the action script. Remembering you want as little processing done in the watcher, it would make sense only to have the first step of finding the new files in the watcher. Then let the action script handle everything else.

©Manning Publications Co. To comment go to liveBook
Licensed to Tausif Khan <tsfahmed2@gmail.com>

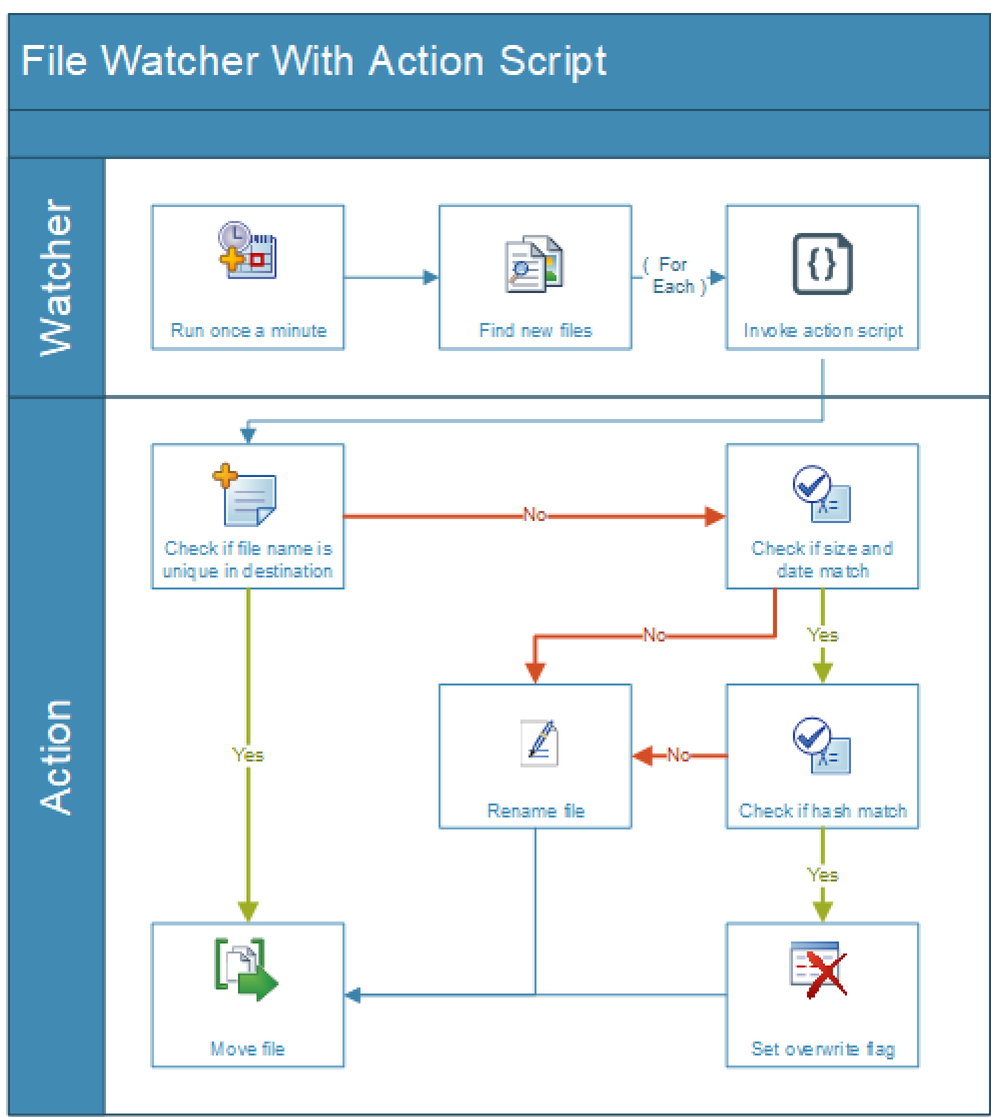


Figure 12: File watcher with action script execution

Now that you know what needs to go into the watcher script, you can start coding it, beginning with the command to find the new files in the folder. This can be done using the `Get-ChildItem` cmdlet. Since you want to ensure that your script runs as efficiently as possible, there are a few concepts you will want to follow regardless of the automation.

©Manning Publications Co. To comment go to liveBook
Licensed to Tausif Khan <tsfahmed2@gmail.com>

USE CMDLET BASED FILTERS

When you need to filter your results, try to use the parameters provided by the cmdlets rather than using the `Where-Object` filtering whenever possible. In this case, you will be using the `Get-ChildItem` cmdlet, which has a filter parameter. So, if you only need to return XML files, you can use `-Filter '*.xml'` to limit your results. When you use the Where-Object after a pipe, the `Get-ChildItem` cmdlet will return all items, then PowerShell will filter them, drastically increasing your runtime.

AVOID RECURSION

It is always best to scope your scripts to only pull the data they need. If you have multiple subfolders or nested Organizational Units, parsing through all of them can be very time-consuming. It can often be quicker to run multiple get commands scoped to individual subfolders than to run one at the top level with recursion.

For example, if the folder watcher needs to look into multiple subfolders, it would be quicker to list the specific folders than just listing the parent and having your script search through every single child folder. It may also consider creating a separate watcher for each folder.

Another example of this is a watcher I created to monitor user accounts in Active Directory. The Organizational Units (OU) structure was that each office had its own OU. Inside each OU were separate OUs for computers, users, admins, printers, service accounts, etc. I only needed to monitor the User OU under each site OU. So, if I scoped the script to the top level and told it to recurse, it would find everything I needed, but it would also waste time searching through all of the other OUs. Instead, I had one command return all the OUs directly under the top level. Then used a for each to look directly in the User OU under each one. Doing this caused the script to execute in 10 seconds versus the command that used recursion that took 90 seconds.

PROCESS IN ORDER

Since a watcher is time-sensitive, you always want to execute in the order received. In the folder watch example, you will want the files sorted by the date and time they were created so that they will be processed in the order received. This helps ensure that if the script stops for any reason, the subsequent execution will pick up right where the previous one left off.

Following these practices, you can now build out the basic structure of your watcher. Which will get the files in the folder, sort them by date, then invoke the action script for each file.

LOG ACTIONS

Another thing to consider is what to do if the action script fails. For example, if something goes wrong and the action script is unable to complete. Then every time the watcher runs, it may attempt to run the action script over and over. If the number of problem items grows larger than your concurrent job count, your entire automation could stop.

To protect against this, you can create a log to let your watcher know which actions have been invoked. Again, this will be different for every watcher, but the concept remains the same. For example, for the folder watcher, you can write the file's creation date before it invokes the action script. Then have the script filter on the last date from the log the next time it starts, preventing it from attempting to send the same files over and over.

©Manning Publications Co. To comment go to liveBook
Licensed to Tausif Khan <tsfahmed2@gmail.com>

AVOID UNNECESSARY COMMANDS

It may be tempting to add additional checks or conditions to a watcher to account for every situation, but you are sacrificing speed with every new command added. If you have multiple conditions, it may be best to break them into separate watchers or add them to the action script. This is especially true if you have more than one action that can be taken. A good rule to follow is one action per watcher. Not only will it speed up your execution, but it will make your code easier to maintain in the long run.

3.3.2 Invoking Action Scripts

There are many ways you can invoke one PowerShell script from another. You can use the `Invoke-Command`, `Invoke-Expression`, `New-Job`, and `Start-Process` cmdlets. For action scripts, the best option is to use the `Start-Process`. Unlike the other cmdlets, the `Start-Process` cmdlet executes the script in a separate process from the watcher script. This means if the watcher script stops executing or has an error, the action scripts running are not affected.

To invoke the action script using the `Start-Process` cmdlet, you need to pass the script path and parameter values, along with the path to the PowerShell executable. You can also pass the `–NoNewWindow` argument to keep from having a ton of windows pop up every time it runs. You will notice that the command argument is very similar to the arguments you used when creating a scheduled task earlier. That is because both are essentially the equivalent of running the command from a terminal or command prompt window.

When invoking your action script, you will want to ensure that the parameters are primitive types (strings, int, Boolean, etc.) and not objects. This is because different object types can have different behavior when passed in this manner, and it is difficult to predict how they will react. For example, in the folder watcher, you will want to pass the file's full path as a string versus the file object type from the `Get-ChildItem` cmdlet.

RESOURCE LIMITING

One thing to be cautious of when using the `Start-Process` cmdlet to invoke action scripts is overwhelming your system. This can happen easily since each action script is running in an independent process. So, in our example, if you all of a sudden have 100 files added to the folder you are monitoring, your watcher could end up trying to process all 100 at a time.

To prevent this, you can add a limit to the number of action scripts a watcher can have running at once. This will prevent you from accidentally firing off more jobs than your system can handle. By adding the `-PassThru` switch to the `Start-Process` cmdlet, you can save the processes to an array. Then once a certain number of concurrently running jobs reaches the limit, have it wait until one has completed before continuing to another.

3.3.3 Graceful terminations

As previously mentioned, you should aim for your script to complete in half the amount of the run interval. Therefore, you will want to monitor the execution time in your watcher scripts, so you can terminate it if it runs for too long.

Most job scheduling platforms have settings for what to do if the previous task is still running, and it is time for the next task to start. For example, in Task Scheduler, you can prevent the new

©Manning Publications Co. To comment go to liveBook
Licensed to Tausif Khan <tsfahmed2@gmail.com>

task from starting, start it in parallel, queue it, or stop the existing instance. My recommendation here would be to choose the stop the existing instance. The main reason behind this is because if something goes wrong in the script, this would kill the process and start it fresh, hopefully resolving the issue.

Take, for example, an issue with authenticating to a network share. You all know it happens randomly from time to time. It would be much better to have the script just start over and try to reauthenticate, then build in all sorts of crazy logic into your script to try and handle this.

Having your script automatically terminate itself after a certain about of time, you can ensure it executes at a point that will not affect the subsequent execution. Instead of just letting the task scheduler terminate it, and you have no control over when it will happen.

For your folder watcher script, we said it needs to run every 60 seconds, so you need to ensure that if it is still running after 30 seconds, you terminate it at a point of your choosing. An excellent tool to help you do this is the System.Diagnostics.Stopwatch .NET class.

The Stopwatch class provides you a quick and easy way to measure the execution times inside your script. You can create a new stopwatch instance by dot sourcing the class with the StartNew method. Once the stopwatch starts, the total time is in the Elapsed property. There are also different methods for stopping, restarting, and resetting the stopwatch. The following snippet shows how to start, get the elapsed time, and stop the stopwatch.

```
PS P:\> $Timer =  [system.diagnostics.stopwatch]::StartNew()
PS P:\> Start-Sleep -Seconds 3
PS P:\> $Timer.Elapsed
PS P:\> $Timer.Stop()

Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 2
Milliseconds      : 636
Ticks             : 26362390
TotalDays         : 3.0512025462963E-05
TotalHours        : 0.000732288611111111
TotalMinutes      : 0.0439373166666667
TotalSeconds      : 2.636239
TotalMilliseconds : 2636.239
```

To use this in the watcher script, just add the command `$Timer = [system.diagnostics.stopwatch]::StartNew()` to the beginning of your script. Then you can determine where the best place to stop the execution of your script will be.

Where you terminate will be different for every script, but typically you will want to terminate after the action script is called. If you terminate before, you run the risk of never making it to the action. You also want to avoid terminating before any of your logs are written. In the folder watcher script, the best place to terminate would be at the bottom of the foreach loop. This will ensure that the current file is sent to the action script and before it tries to process the next one.

3.3.4 Folder Watcher

Now that we have gone through all the parts of a watcher, let's put it all together in the folder watcher script. Following the design recommendations, it will start with declaring the stopwatch. Then it will pick up the log with the last processed files date in it, and since we don't want it to

©Manning Publications Co. To comment go to liveBook
Licensed to Tausif Khan <tsfahmed2@gmail.com>

error out the first time it runs, or if something happens to the log, you can use the `Test-Path` cmdlet to confirm it exists before attempting to read it.

Now you are ready to query the files and sort them based on the creation date. Next, it will write the creation time to the log for each file, then invoke the action script. After the action script is invoked, it will check if the number of running jobs exceeds the limit. If it does, it will wait until at least one of them finishes. Then confirm that the time limit is not exceeded. If not, it will continue to the next file. Once all files are sent to action scripts, the watcher script will exit.

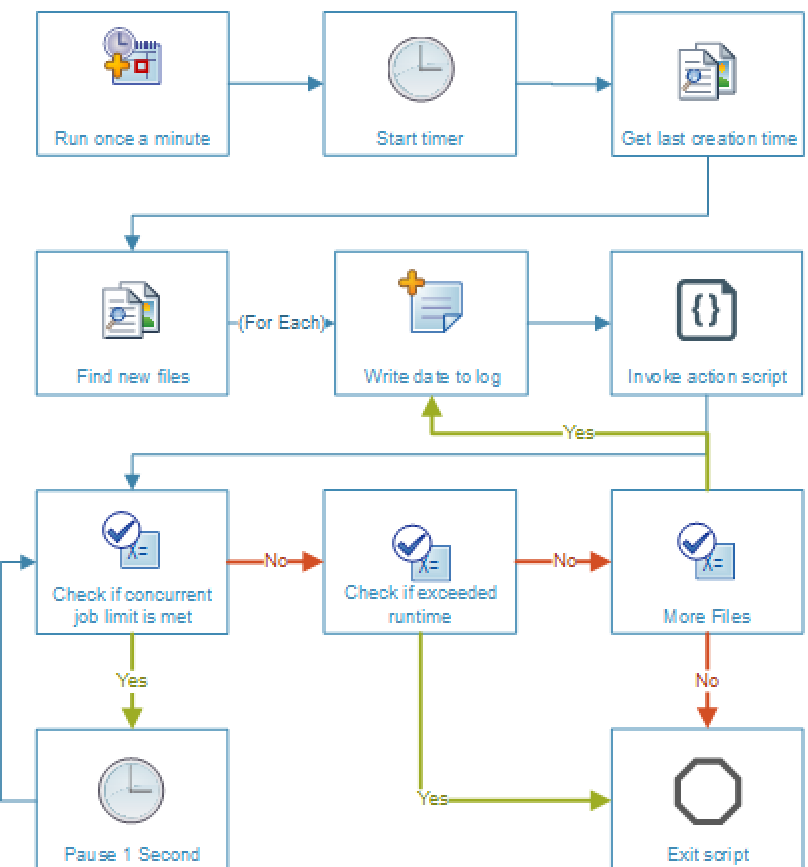


Figure 13: File watcher with action script invocation, concurrent job limiter, and execution timer

Listing 4 Watch-Folder.ps1

```
param(
    [Parameter(Mandatory = $true)]
    [string]$Source,

    [Parameter(Mandatory = $true)]
```

©Manning Publications Co. To comment go to liveBook
Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
    [string]$Destination,

    [Parameter(Mandatory = $true)]
    [string]$ActionScript,

    [Parameter(Mandatory = $true)]
    [int]$ConcurrentJobs,

    [Parameter(Mandatory = $true)]
    [string]$WatcherLog,

    [Parameter(Mandatory = $true)]
    [int]$TimeLimit
)

$Timer = [system.diagnostics.stopwatch]::StartNew()    #A

if (Test-Path $WatcherLog) {    #B
    $logDate = Get-Content $WatcherLog -Raw
    try {
        $LastCreationTime = Get-Date $logDate -ErrorAction Stop
    }
    catch {
        $LastCreationTime = Get-Date 1970-01-01
    }
}
else {
    $LastCreationTime = Get-Date 1970-01-01    #C
}

$files = Get-ChildItem -Path $Source |    #D
    Where-Object { $_.CreationTimeUtc -gt $LastCreationTime }
$sorted = $files | Sort-Object -Property CreationTime    #E

[int[]]$Pids = @()    #F
foreach ($file in $sorted) {
    Get-Date $file.CreationTimeUtc -Format o |    #G
        Out-File $WatcherLog

    $Arguments = "-file ""$ActionScript""",    #H
        "-FilePath ""$($file.FullName)""",
        "-Destination ""$($Destination)""",
        "-LogPath ""$($ActionLog)"""
    $jobParams = @{
        FilePath     = 'pwsh'
        ArgumentList = $Arguments
        NoNewWindow  = $true
    }
    $job = Start-Process @jobParams -PassThru    #I
    $Pids += $job.Id    #J

    while ($Pids.Count -ge $ConcurrentJobs) {    #K
        Write-Host "Pausing PID count : $($Pids.Count)"
        Start-Sleep -Seconds 1
        $Pids = @(Get-Process -Id $Pids -ErrorAction SilentlyContinue |
            Select-Object -ExpandProperty Id)    #L
    }

    if ($Timer.Elapsed.TotalSeconds -gt $TimeLimit) {    #M
```

©Manning Publications Co. To comment go to liveBook
Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
        Write-Host "Graceful terminating after $TimeLimit seconds"
        break    #N
    }
}
```

#A Start Stopwatch timer
#B check if the log file exists and set filter date if it does.
#C Default time if no log file is found
#D Get all the files in the folder
#E Sort the files based on creation time
#F Create an array to hold the process IDs of the action scripts
#G Record the files time to the log
#H Set the arguments from the action script
#I Invoke the action script with the PassThruswitch to pass the process id to a variable
#J And the id to the array
#K If the number of process ids is greater than or equal to the number of current jobs loop until it drops.
#L Get-Process will only return running processes, so execute it to find the total number running.
#M Check if the total execution time is greater than the time limit
#N The 'break' command is used to exit the foreach loop, stopping the script since there is nothing after the loop

3.3.5 Action scripts

When it comes to creating the action script, there are not as many limits and things to consider. They should follow all the standard PowerShell practices we've discussed regarding unattended scripts. Other than that, you may want to consider adding some logging to your script.

Since the action script runs independently, the watcher is not aware of errors in it. This means it won't report them back to the invoking system or include them in its logs. Therefore, it is always a good idea to have some form of logging in your action scripts so that you can be aware of any issues or failures.

We will cover logging more in-depth in later chapters. For the folder watch example, we can add some simple text file-based logging to the action script to record any errors. An excellent way to ensure that you adequately capture errors is by using a try/catch/finally block.

In the action script, you will create a function to perform all the duplicate checks and perform file moves. When you add the `[CmdletBinding()]` to the function, you can use the `-ErrorAction` argument and set it to Stop when you call the functions. During normal script execution, this would stop the script in the case of an error in the function. However, when used inside a try block, it will send the script to the catch block. If the function runs without any problems, then the catch block is skipped. Thus, regardless of there being an error or not, the finally block will always run.

For the action script in our example, you can set a variable with a success message under the function call in the try block. Then in the catch block, you can set that same variable to an error message. Then in the finally block, have it write that message to the log. If the function has an error, it will skip over the variable set in the try block and go right to the catch. Or if there is no error, it will set the success message, then skip the catch block. Then, in either case, the log file will be written to.

> **NOTE:** The variable $_ in a catch block will contain the error message. You can use this to record precisely what went wrong.

©Manning Publications Co. To comment go to liveBook
Licensed to Tausif Khan <tsfahmed2@gmail.com>

Taking what we now know and the tasks we identified at the beginning of this exercise, the tasks for this action script will be to test that the file exists. Then if it does, get the file object using the Get-Item cmdlet. This will allow you to get the item data you will need to use if a duplicate file is found. Remember, we are only passing in a string for the file path and not the file object.

Next, it will check if a file with the same name already exists in the destination folder. If one does, it will check if the files are the same. If they are, it will just overwrite it to clear it out of the source folder. If not, it will rename the file with a unique name. Then the file is moved to the destination folder.

Getting a unique name

There are several ways to ensure that your file has a unique name. These include using Guids or creating an interval variable that you can increment until a unique one is found. But when it comes to file names, nothing bets the FileTime string in the DateTime object.

All DateTime object contain the methods ToFileTime and ToFileTimeUtc. Invoking either of these methods will return a FileTime value, which is the number of 100-nanosecond intervals since January 1, 1601. By getting the current time and converting it to FileTime, you are almost guaranteed a unique file name if you add it to the file name. That is unless you have two files with the same name that are being moved within 100-nanoseconds of each other.

The FileTime value can also be parsed back into a standard DateTime format, giving you a record of when the file was renamed.

©Manning Publications Co. To comment go to liveBook
Licensed to Tausif Khan <tsfahmed2@gmail.com>

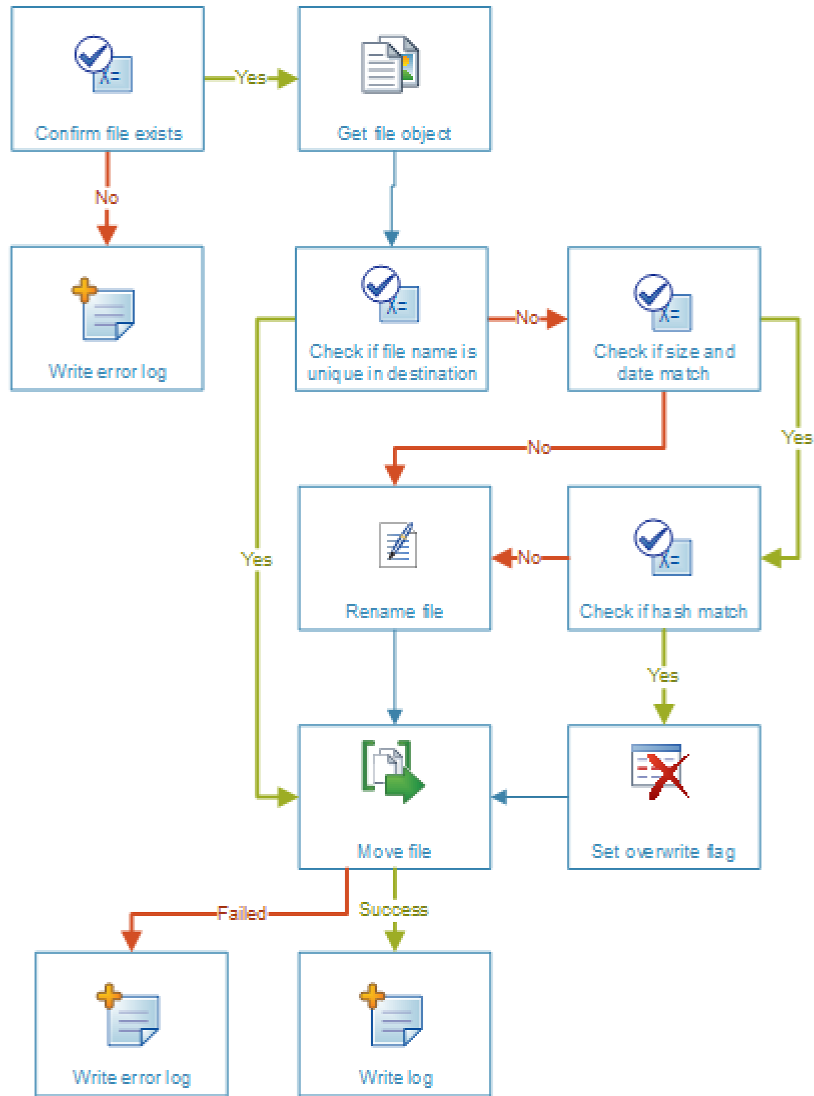


Figure 14: Action script with logging and error handling

Go ahead and create a second script named *Move-WatcherFile.ps1* and build out the action script.

©Manning Publications Co. To comment go to liveBook
Licensed to Tausif Khan <tsfahmed2@gmail.com>

Listing 5 Action Script with Logging and Error Handling

```
param(
    [Parameter(Mandatory = $true)]
    [string]$FilePath,
    [Parameter(Mandatory = $true)]
    [string]$Destination,
    [Parameter(Mandatory = $true)]
    [string]$LogPath
)

Function Move-ItemAdvanced {    #A
    [CmdletBinding()]
    [OutputType()]
    param(
        [Parameter(Mandatory = $true)]
        [object]$File,
        [Parameter(Mandatory = $true)]
        [string]$Destination
    )

    $DestinationFile = Join-Path -Path $Destination -ChildPath $File.Name

    if(Test-Path $DestinationFile){    #B
        $FileMatch = $true
        $check = Get-Item $DestinationFile    #C
        if($check.Length -ne $file.Length){
            $FileMatch = $false    #D
        }
        if($check.LastWriteTime -ne $file.LastWriteTime){
            $FileMatch = $false    #E
        }
        $SrcHash = Get-FileHash -Path $file.FullName    #F
        $DstHash = Get-FileHash -Path $check.FullName
        if($DstHash.Hash -ne $SrcHash.Hash){
            $FileMatch = $false
        }

        if($FileMatch -eq $false){    #G
            $ts = (Get-Date).ToFileTimeUtc()
            $name = $file.BaseName + "_" + $ts + $file.Extension
            $DestinationFile = Join-Path -Path $Destination -ChildPath $name
            Write-Verbose "File will be renamed '$($name)'"
        }
        else{
            Write-Verbose "File will be overwritten"
        }
    }
    else {
        $FileMatch = $false
    }

    $moveParams = @{
        Path = $file.FullName
        Destination = $DestinationFile
    }
    if($FileMatch -eq $true){    #H
        $moveParams.Add('Force',$true)
    }
```

©Manning Publications Co. To comment go to liveBook
Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
    Move-Item @moveParams -PassThru
}

if(-not (Test-Path $FilePath)){    #I
    "$(Get-Date) : File not found" | Out-File $LogPath -Append
    break
}

$file = Get-Item $FilePath    #J

$Arguments = @{
    File = $file
    Destination = $Destination
}

try{    #K
    $moved = Move-ItemAdvanced @Arguments -ErrorAction Stop
    $message = "Moved '$($FilePath)' to '$($moved.FullName)'"
}
catch{    #L
    $message = "Error moving '$($FilePath)' : $($_)"    #M
}
finally{    #N
    "$(Get-Date) : $message" | Out-File $LogPath -Append
}
```

#A Add new function to perform file checks when a duplicate is found
#B check if file exists
#C get the matching file
#D check if they have the same length
#E check if they have the same last write time
#F check if they have the same hash
#G If they don't all match, then create a unique filename with the timestamp
#H If the two files matched force an overwrite on the move
#I Test that the file is found. If not, write to log and stop processing
#J Get the file object
#K Wrap the move command in a try/catch with an error action set to stop
#L Catch will only run if an error is returned from within the try block
#M Create a custom message that includes the file path and the failure reason captured as $_
#N write to the log file using the finally block

Now your watcher and actions scripts are ready for action. Keep in mind, as with all automations, there is no way to predict every situation, so chances are you will need to make some tweaks to either script. However, just keep in mind execution time, concurrently running jobs, and logging, and you will be in good shape.

3.4 Running watchers

A watcher script should be able to be executed the same way you execute any script. You just need to ensure that the watcher can access the action script and that they can both access the resources they need independently. This allows you to use all the same tools you use for executing and scheduling any other PowerShell script.

©Manning Publications Co. To comment go to liveBook
Licensed to Tausif Khan <tsfahmed2@gmail.com>

3.4.1 Testing Watcher Execution

Before scheduling your watcher, you will want to test it thoroughly. On top of your standard functionality testing, you will want to measure the execution time. You can do this using the `Measure-Command` cmdlet.

The `Measure-Command` cmdlet allows you to measure the execution time of any command, expression, or script block. You can use it to test the execution time of your watcher script by using it in conjunction with the `Start-Process` cmdlet. Very similar to the way you call the action script from the watcher, you can use the `Start-Process` cmdlet to call the watcher script from another PowerShell session. Doing this will ensure that it runs into a separate session, very much like it will be when started by Task Scheduler or any other job scheduler. The only difference here is you are going to add the `-Wait` switch. This will ensure your script waits until the watcher has finished processing, ensuring you get an accurate measurement of the execution time. You can see in the snippet below how this looks when invoking the *Watch-Folder.ps1* watcher script.

```
PS P:\> $Argument = '-File ' +
     >>    '"C:\Scripts\Invoke-LogFileCleanup.ps1"' +
     >>    ' -LogPath "L:\Logs\" -ZipPath "L:\Archives\"' +
     >>    ' -ZipPrefix "LogArchive-" -NumberOfDays 30'
PS P:\> $jobParams = @{
     >>    FilePath = "C:\Program Files\PowerShell\7\pwsh.exe"
     >>    ArgumentList = $Argument
     >>    NoNewWindow = $true
     >> }
PS P:\> Measure-Command -Expression {
     >>    $job = Start-Process @jobParams -Wait}

Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 2
Milliseconds      : 17
Ticks             : 20173926
TotalDays         : 2.33494513888889E-05
TotalHours        : 0.000560386833333333
TotalMinutes      : 0.03362321
TotalSeconds      : 2.0173926
TotalMilliseconds : 2017.3926
```

As you can see, the TotalSeconds is well under the 30-second limit you wanted. But be sure to run multiple tests with different scenarios to ensure it stays under it. For example, if I put 100 files in the watcher folder, my execution time jumps to 68 seconds, well over the 30-second limit. When this happens, there are two things you need to consider.

First and foremost, you need to ask if it is possible that 100 files could be added in one minute, and if so, for how long. Because if your script takes 68 seconds to process 100 files, and 100 files are added every minute, you will have a massive backlog that will never clear. If this is the case, then you need to reconsider how your script is executing. So it would help if you asked yourself questions like:

- Could I increase the limit of currently running job action scripts?
- Is there a way I can increase the speed of execution of the watcher script?
- Can I split the load between two or more watchers, each grabbing a chunk of the files?

©Manning Publications Co. To comment go to liveBook
Licensed to Tausif Khan <tsfahmed2@gmail.com>

I cannot tell you the right answer because the answers to these questions will be different for every automation. Other automations may even have completely different questions. These are just examples to help you see the mindset you need to think about when creating your automations.

For instance, if you discover that the average number of files added in one minute is around 10, you will know your automation will handle the rare cases where 100 files are added. You may end up with a backlog for a few minutes, but it will quickly catch up. It just becomes a balancing act that you will need to monitor and adjust over time.

Also, if you find your watchers are exceeding their runtime on a regular basis, you can test the individual commands inside it using the `Measure-Command` cmdlet. This will help you determine where you may be able to rewrite certain portions or commands to help speed things up.

3.4.2 Scheduling Watchers

You can use all the same tools to schedule watcher scripts that you can to schedule monitor scripts. The only difference is you need to consider the execution time and what happens if it is exceeded. Even though you built-in a graceful termination, if the job scheduler has a setting for it, you should have it terminate the job if it is time for the next one to start, as a backup. You should never have two instances of the same watcher running at the same time.

Also, each instance of the watcher running should be started by the job scheduler. It may be tempting to create a job that runs once a day with a loop built-in to keep executing the same commands over and over for 24-hours. However, if something goes wrong during the execution, or the script has a terminating error, or the computer is rebooted, your automation will not run again until the next day.

3.5 Summary

- There are multiple different tools out there that will allow you to schedule PowerShell scripts. Including Task Scheduler and Cron.
- When creating a watcher script, the time of execution is extremely important.
- A watcher script should only monitor for events. Any actions that need to be taken should be performed by an action script.
- You should know how long your script should execute and build in a graceful termination if it runs too long.

©Manning Publications Co. To comment go to liveBook
Licensed to Tausif Khan <tsfahmed2@gmail.com>

4 Handling sensitive data

This chapter covers

- **Basic security principles for automations**
- **PowerShell secure objects**
- **Securing sensitive data needed by your scripts**
- **Identifying and migrating risk**

In December of 2020, one of the largest and most sophisticated cyberattacks ever was found to be taking place on systems across the globe. Security experts discovered that the SolarWinds Orion platform was the subject of a supply chain hack. Hackers were able to inject malware directly into the binaries of some Orion updates. This attack was such a big deal because SolarWinds' Orion is a monitoring and automation platform. Their motto, "One platform to rule your IT stack," makes it a very enticing target for bad actors.

More than 200 companies and federal agencies were impacted by this attack, including some big names like Intel, Nvidia, Cisco, and the US departments of Energy and Homeland Security. Expert suspect that this attack is responsible for other exploits found shortly after at Microsoft and VMware.

If someone is able to gain full access to an automation platform, not only are they able to perform any action that platform has permissions to perform, they also have access to all the information and data stored inside that platform. So, if you have a script with full domain admin rights or full global administrator, the bad actors will have those rights and privileges.

Even something that seems benign, like an internal FTP password or an API key, can be exploited in ways you might never consider. In the investigation of the SolarWinds exploit, investigators discovered that the password for the SolarWinds update server was saved in plain text inside a script that was in a public GitHub repository. At the time of writing, it has not been confirmed that this is how the hackers were able to inject their malware directly into the Orion updates. However, it is not a far stretch to see how it could have been the

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

cause and a straightforward one to avoid. This attack serves as a great reminder of several fundamental IT and automation security principles, which we will cover in this chapter.

Throughout this chapter, we will use a SQL health check automation as an example, but the principles apply to any automation. In this automation, you will connect to a SQL instance using PowerShell. Then check that the databases have their recovery model set to simple. If a database is not set to simple, it will send an email reporting the bad configuration. To accomplish this, you will need to retrieve secure data for both the SQL server connection and the SMTP connection.

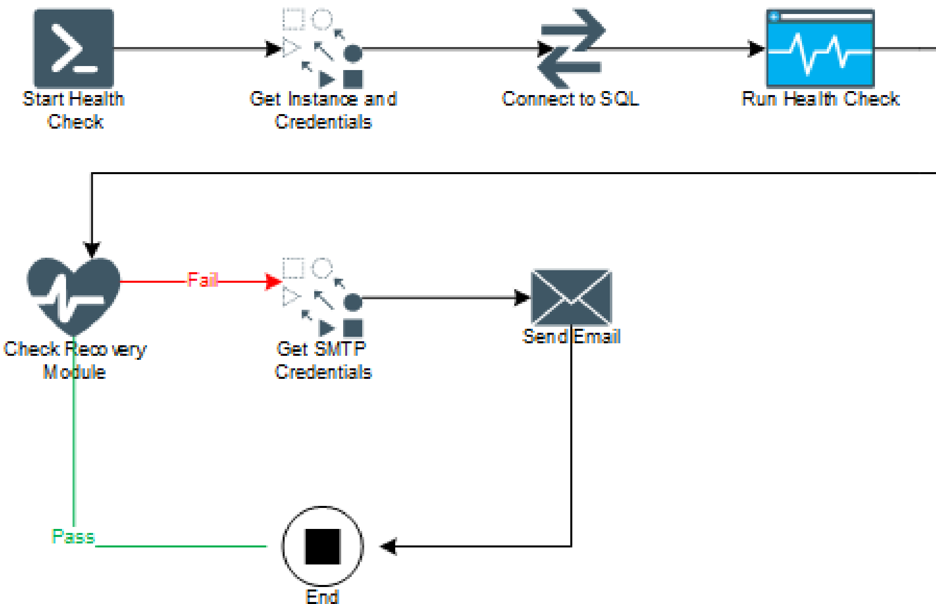


Figure 1 SQL health check automation to send a notification if recovery model is not set properly

SQL Health Check Prerequisites

To run the SQL health check, you need a SQL instance to test. I have provided a script in the chapter 4 Helper Scripts folder that will install and configure a SQL Server Express instance for you. I highly recommend that you use a virtual machine for this testing. If you set up the Jenkins server from appendix A, then I would recommend using that machine.

If you choose to use an existing SQL instance, you will need to manually install the `dbatools` and `MailoZaurr` modules from the PowerShell gallery.

You will also need a SendGrid account to send the notification emails. You can sign up for a free account using a personal or work email address at sendgrid.com. The free account can send 100 emails per day.

Finally, you will need to install KeePass to use as a secret vault. The information setting up the KeePass database is included later in this chapter. You can download and install KeePass from keepass.info or using Chocolatey with the command `choco install keepass`.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

4.1 Principles of automation security

Since every single company, platform, tool, and software is different, someone could write an entire 500-page book on handling specific security situations and not come close to scratching the surface. However, we will cover some key concepts that you can apply across the board to any automation you are creating.

To start with, any time you are creating automations, you need to ask yourself the questions:

- What are my risks if someone gains access to this automation?
- How can I limit the damage if some does access it?

4.1.1 Do not store sensitive information in scripts

First and foremost, you should never, under any circumstance, include any sensitive data inside of a script. While the SolarWinds example where someone gains access to your entire platform is an extreme one, the fact that a password saved in a script is that potentially what started it all is not. Someone unwittingly sharing a script with a plain text password in it online, losing a thumb drive with a copy of it, or sending it over an unsecured email to a vendor, is more common than most people would like to admit. To protect yourself from this and other possible breaches, you should never store sensitive data inside a script.

The question then becomes, what is sensitive data. Some of it is pretty obvious examples are:

- Passwords and secrets
- API keys
- SSH keys
- Private certificates
- Certificate thumbprints
- PGP keys
- RSA tokens

However, there is some information that may not seem as obvious. For instance, in the example of a SQL health check, a SQL connection string may seem innocuous, especially if it uses a trusted connection and not a username and password; but you are still giving someone a piece of the puzzle. Suppose a bad actor finds your script with the SQL connection string to the ERP database. In that case, they can quickly target their attacks, increasing the potential of gaining access to sensitive information before they are caught. The same can is valid for storing usernames in scripts. While the username itself is no good without the password, you have just given someone half of the information they need to gain access. Also, as you saw in the previous chapters, this makes your script less portable and harder to maintain.

Even when it isn’t innocuous, people will often time still put sensitive information in their scripts for a number of reasons. For one, they might not know how to secure it properly, or they may be on a time crunch and say they will do it later. And as we all know that later can often never come. However, as you will see later in this chapter setting up a secure store for

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

passwords and other information is quick and easy. And if you already have secure stores set up and available, you will have no excuse to put sensitive data in your scripts.

Just remember to ask yourself, would this information be of interest to an attacker? If the answer is yes, it should be considered sensitive data and stored securely outside of any scripts.

4.1.2 Principle of least privilege

Second, only to not saving passwords in scripts is the principle of least privilege. This principle states that an account should only have the permissions it needs and nothing more. Unfortunately, people often do not want to spend the time figuring out permissions to a granular level, or even worse, some vendors will just say that an account needs full administrator access. Sure, it is easier just to make an account an administrator and move on. But in doing so, you are creating an unacceptable level of risk.

In the case of our SQL health check script, you would not want to give the account database administrator (DBA) rights because it only needs to read information about the database. If the account has DBA rights and becomes comprised, the attacker can not only access all of your information; they can create backdoors and separate accounts that you might not find for months or longer. For the health check script, you don't even need read permissions to the databases. You just need to be granted the view server state permissions. This means that even if the service account is comprised, it can not be used to read the data in the database.

In another example, if you were an organization that used SolarWinds Orion, there would have been no way you could have prevented the attackers from gaining access. However, if you adhered to the principle of least privilege, you would not be as severely impacted as others who might do something like giving a service account domain administrator access.

The principle of least privilege does not apply to just things like service account permissions. It can also include things like IP or host-based restrictions. An excellent example of this can be SMTP relays. I have heard of situations where the username and password of a service account with permission to impersonate any Exchange user gets leaked. Hackers can then use that account to send emails that appeared to both people, and data protection filters, to come from a person inside the company. They can then submit payment requests to fake vendors.

In some cases, tens of thousands of dollars were stolen before anyone caught on. This could have all been avoided or at least made much more difficult if this account was only able to send these requests from a particular IP address or server. This is why I choose to use SendGrid in this automation example. Not only does it have uses an API key, so there is no user to comprise, but it also has IP-based restrictions and auditing to help prevent and detect unauthorized use.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

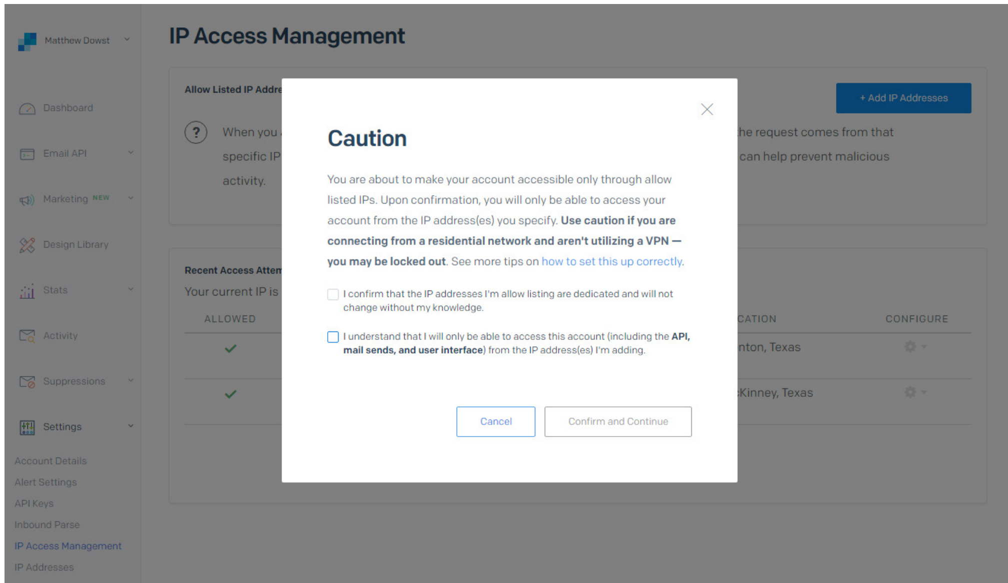


Figure 2 IP Access Management in SendGrid

4.1.3 Consider the context

An automation script intended to be run interactively versus one that is meant to be run unattended will have completely different security considerations. Suppose it is designed to be run interactively. In that case, you can do things like prompting for passwords, or run under that user context, use privileged access management solutions, or use delegated permissions.

However, suppose a script is going to run unattended. In that case, it will need a way to securely access sensitive data while preventing others from accessing the same data. Also, keep in mind the solution to this may not involve passwords at all. Consider our SQL health check scenario. If the SQL server uses Windows domain authorization, then there would be not needed to pass a username and password. You can use a job scheduler to run as that account and execute the script.

Another option could be to set the job to run locally on the SQL server. This way, it can run under the system context without needing a password. If you limit the script to executing under the system context on the database server, the only way, someone could exploit it would be to gain access to your database server. And if that happened, you would have much more significant concerns than someone being able to see your script.

4.1.4 Create role-based service accounts

I cannot tell you how many times I have sent someone a list of required service accounts, and the first thing they ask is, can these all use the same account? Unfortunately, this has become a far too common practice and is a very dangerous one. On the other side, some will

Licensed to Tausif Khan <tsfahmed2@gmail.com>

argue that managing hundreds of service accounts can, in itself, become a full-time job. This is why I like to use the concept of role-based access.

Let us take a look at the requirements for the SQL health check. First, we know we will need an account to authenticate with SQL and an account to send the email. In this case, I would recommend creating two separate accounts because they are interfacing with two completely different systems and performing different roles. Also, you will more than likely have more than one automation that will need to send emails. So, instead of setting up every service account with a mailbox or other SMTP replay permissions, you can create one account to send emails and share it between automations.

Taking it a step further, if you decide to create an automation to perform automated SQL maintenance or backups, you could reuse the service account for the health check because they fall under the same role of SQL maintenance. However, if you create an automation to perform SQL extracts, I would recommend creating a new account. Again, this is because it serves a different function that would not fall under the role of SQL maintenance.

If an automation reaches out to different systems, especially ones outside of your network, it is always best to use separate accounts. Even if they support things like single sign-on. This will help protect against situations where a vendor or a piece of software gets compromised. If that happens, you will have limited the impact by limiting that service account to that specific platform.

It becomes a balancing act that you will get a feel for and adjust over time. Just follow the concept of one service account per role and system, and you will help minimize the impact of any potential compromise.

4.1.5 Use logging and alerting

You will not be able to protect yourself against every threat out there. However, you can help to stop attacks as soon as possible through diligent logging and alerting. For example, if you have a service account that sends data from your on-premise data center to your cloud provider, and suddenly, you start seeing logins from Russia, you will know something is up, and you can put a stop to it. The recent advancements in identity and threat protection solutions have been a very welcome addition to the automation community. The ability to see when and where an account is being used makes spotting these situations much more effortless.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

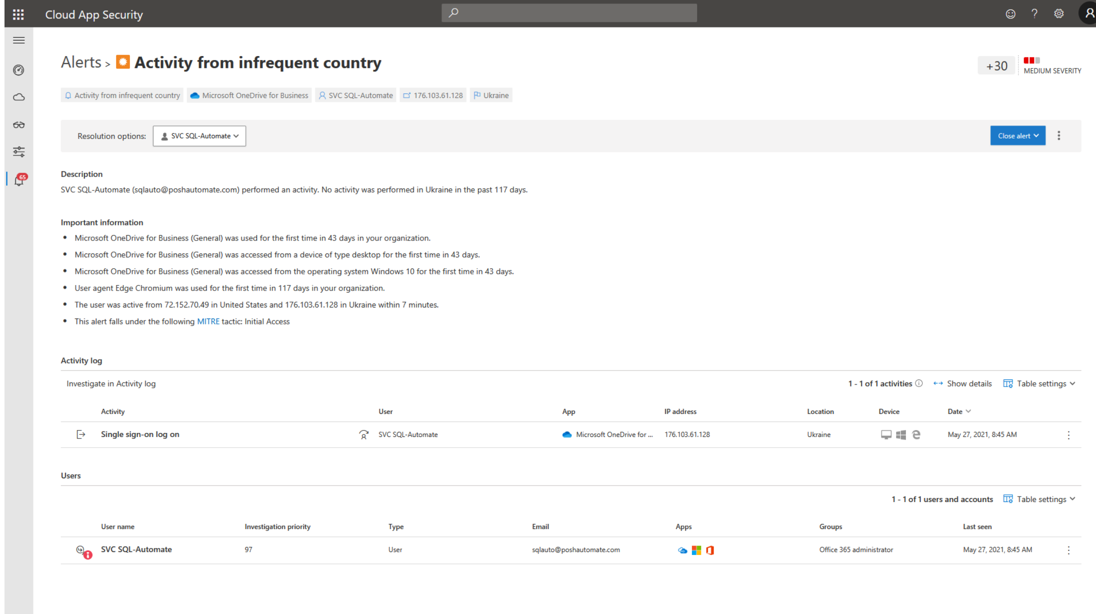


Figure 3 Alert on activity from an infrequent country from Microsoft Cloud App Security

Most enterprise-level password vaults have built-in logging, letting you see when and where passwords are accessed. Some even have AI behind them to be able to trigger alerts when anomalous activity occurs. Learn how to use and tune these alerts to your benefit.

Just like with the service accounts, it becomes a balancing act. Alert fatigue is a real thing. The quickest way to get your alerts ignored is by sending too many. Especially if many of the alerts are false positives. For instance, the breach of Target Superstores back in 2014 could have been stopped well before 40 million customers' credit and debit cards were leaked had someone not turned off a noisy alert.

4.1.6 Do not rely on security through obscurity

Security through obscurity (STO) is the reliance on using secrecy or obscurity to protect yourself. While it may not seem like a bad idea on the surface, it should never be relied on as the only means of security. A typical scenario you will see is changing the default ports for SSH and RDP. This works against scanners looking for those particular ports, but once the secret that you changed the SSH port is discovered, an attacker will adjust. It is no different than locking all the doors to your house, but you leave your house key under the doormat. Once someone knows this, they can get into your home with little to no problems.

That is not to say STO doesn't have its place. It should just never be relied on as the only means of security. One common and very insecure STO practice that I often see in PowerShell automations is the use of encoding to hide sensitive data in scripts. This is often done in situations where someone needs to run a script on an end-user's device as an

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

administrator. While the encoding does obfuscate the text, making it difficult for an end-user to see, a bad actor can reverse in seconds. Also, when it runs, the obfuscated values are decoded in memory as unsecured variables, making them easy to find.

Another example is compiling sensitive data into exe or dll. While this may seem more secure than encoding, it is not challenging to decompile an exe or dll. Also, you run into the same vulnerability of having the data written as unsecured variables in memory.

As you will learn throughout this chapter, there are many ways you can secure the sensitive data that automations need to access. Both at rest and during execution and are more secure than these examples.

4.1.7 Secure your scripts

Imagine you have an automation to collect health data from your servers. To keep from having to deal with service accounts and permissions, you create a scheduled task on each server to execute the script. Then to make your life even easier, you put the script on a network share. This way, if you need to update it, all machines will automatically get the new script the next they run.

Now imagine that script was in an insecure share, and a bad actor updated it to create themselves a local administrator account on every single server. Since it is running as the system account, it will have no problems doing that. This is why you must maintain strict access to all production scripts, regardless of where they reside. If you are using a platform like Jenkins or Azure Automation, you should set up at least two instances of them. One that people can access to develop and test their automations. And a second one to run production workloads that you have locked down on only a select few individuals.

Another way to prevent unauthorized script execution is by using coded signed scripts and setting your policies to block unsigned scripts from running. Code signing is a way to use a certificate to confirm the authenticity of a PowerShell script. The only way someone can swap out a script would be if they also had access to your code signing certificate. We will cover code signing more in-depth in chapter 6.

4.2 Credentials and secure strings in PowerShell

Back in the early days of PowerShell, when everything ran on-premises and was all connected to the same Active Directory domain, authentication was easy. You could use Task Scheduler or other job schedulers to run PowerShell scripts as a particular account. Nowadays, in your cross-platform hybrid environments, you can easily find yourself needing to authenticate across multiple environments for a single automation. In most cases, you will need to supply credentials, API keys, or other authentication data during the automation execution. To keep from having these stored in plain text in both files and memory, you can use SecureString and Credential objects in PowerShell.

We will dig deeper into using these inside your scripts and with the SQL health check in section 4.3, but before we do, it is good to know precisely what these object types are and why they are more secure.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

4.2.1 Secure strings

When you modify a standard string in PowerShell, it creates a copy of that string value in memory before changing it. Therefore, even if you delete the variable or set it to null, other copies of it might still exist in memory. Unlike a standard string object, a SecureString is pinned and encrypted in memory. The fact that it is encrypted in prevents memory dumps from being able to read them. Also, since it gets pinned and not copied, you know the value will be deleted from memory when you delete the variable or your process ends.

> **SECURESTRINGS ON LINUX AND MACOS:** Before getting started with SecureStrings, be aware that they should be limited to Windows systems only. This is because the SecureString object relies on a .Net Framework API, not a .Net Core API, so while you can use SecureString on Linux and macOS, the values in memory are not encrypted.

There are two ways that you can create a SecureString in PowerShell. You can use the `ConvertTo-SecureString` cmdlet or the `Read-Host` cmdlet. To use the `Read-Host` cmdlet, you add the *AsSecureString* parameter to it. This will prompt the user to enter a value that is then saved as a secure string inside the script.

```
PS P:\> $SecureString = Read-Host -AsSecureString
PS P:\> $SecureString
System.Security.SecureString
```

The other method uses the `ConvertTo-SecureString` cmdlet, which will convert an existing string to a secure string using the *AsPlainText* parameter. The thing to keep in mind here is that the plain text parameter will already be stored in memory, so it is not as secure as using the `Read-Host` method. If you included the plain text value anywhere in your script, it could be saved in the Event Logs and your PowerShell history. This is why it requires using the *Force* parameter to ensure you know the risks of using it.

```
PS P:\> $String = "password01"
PS P:\> $SecureString = ConvertTo-SecureString $String -AsPlainText -Force
PS P:\> $SecureString
System.Security.SecureString
```

You may be asking when you would use the `ConvertTo-SecureString` cmdlet. There are a few situations you will run into that you will need it. With the most common being making scripts transportable. By default, SecureStrings are encrypted based on the user and the device. If you export the SecureString and try to import it to a different computer, it will fail. However, you can provide a custom key that you can use to perform the encryption. You can then use this key on other computers to import the secure string. Of course, this leads to the need to protect that key because if it gets leaked, all your SecureStrings could be decrypted. As you will see later in this chapter, the PowerShell team at Microsoft has created a new solution that utilizes password vaults to use in place of having to create custom encryption keys.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

4.2.2 Credential objects

Credentials in PowerShell are stored as PSCredential objects. These are simply a PowerShell object that contains a standard string with the username and a SecureString object with the password. Like with SecureStrings, there are two ways you can create a PSCredential object.

The first way is to use the `Get-Credential` cmdlet and prompt the user for the credentials.

```
PS P:\> $Credential = Get-Credential
```

The other option is to manually create the PSCredential object by combing an existing unsecured string for the username and a SecureString for the password. Keep in mind that if you are creating a PSCredential object using unsecured strings, and those unsecured copies of the strings will still be in memory. However, they do have their uses, as you will see in section 4.4 when using Jenkins built-in vault.

```
PS P:\> $Username = 'Contoso\BGates'
PS P:\> $SecureString = ConvertTo-SecureString $Password -AsPlainText -Force
PS P:\> $Credential = New-Object System.Management.Automation.PSCredential $Username,
       $SecureString
```

One thing to keep in mind is PSCredentials can only be used by systems and commands that support PSCredentials. So, passing them to other PowerShell cmdlets is not a problem. However, if you need to call external systems that do not support PSCredential objects, you can convert them to a .Net NetworkCredential object. Some examples of when you would use this would be building custom database connection strings or authenticating with a web application. In addition, you can use them for basic, digest, NTLM, and Kerberos authentication.

```
PS P:\> $Username = 'Contoso\BGates'
PS P:\> $Password = ConvertTo-SecureString 'Password' -AsPlainText -Force
PS P:\> $Credential = New-Object System.Management.Automation.PSCredential $Username,
       $Password
PS P:\> $NetCred = $Credential.GetNetworkCredential()
PS P:\> $NetCred

UserName    Domain
--------    ------
BGates      Contoso
```

Be sure that you are aware that the NetworkCredential object will contain the password in plain text. However, this is wiped from memory once the PowerShell session is closed.

4.3 Storing credentials and secure strings in PowerShell

There used to be no quicker way to trigger an argument in any PowerShell forum than by asking how to store a password for use by a PowerShell script. You would receive a dozen different opinions, each with others arguing why you should not use that approach. This became even more heated after the release of PowerShell core and the discovery that the SecureString objects only exist in Windows.

Licensed to Tausif Khan <tsfahmed2@gmail.com>

Fortunately, the PowerShell team at Microsoft has been working diligently on this problem and has created the SecretManagement module. This module provides you with the ability to store and retrieve secrets and credentials from various password vaults. We will now set up the SecretManagement module for the SQL health check automation and any future automations you create.

4.3.1 The SecretManagement module

The SecretManagement module is an engine that you can use to access different storage vaults. It allows you to use various vaults, including Azure Key Vault, KeePass, LastPass, and many others. The SecretManagement module enables you to interact with any of these storage vaults using a predefined set of commands. It allows you to swap out the backend storage vault seamlessly without needing to update your scripts. It also allows you to access multiple different vaults from within a single automation.

In addition, the PowerShell team and Microsoft have released the SecretStore module, a storage vault built specifically for use with the SecretManagement module. The SecretStore module is not required to be able to use the SecretManagement module. You can use any other vault that has an extension built for it. To find a complete list of extensions, go to the PowerShell gallery and search for *SecretManagement*.

> **NOTE** At this time, the SecretManagement v1.0.0 is still only recommended for Windows environments as it still relies on the SecureString object. It will work on Linux and macOS but will be less secure because the SecureString will be unencrypted in memory.

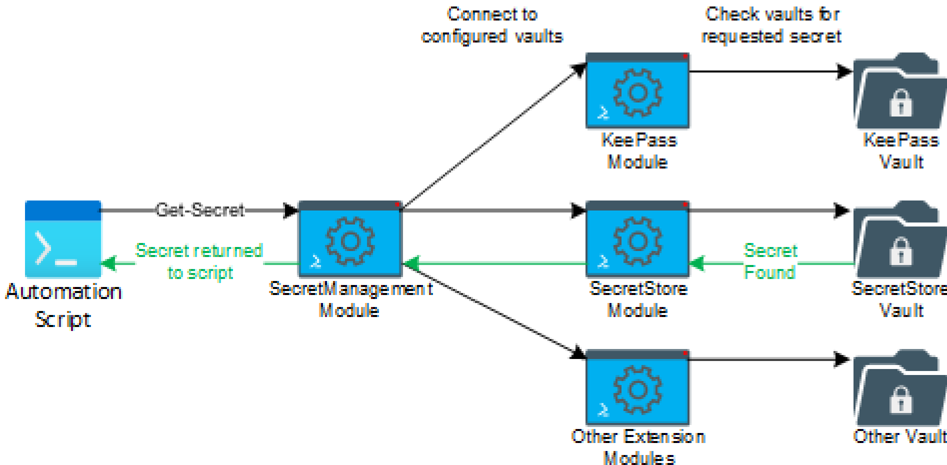


Figure 4 SecretManagement module and how it can access multiple different vaults

There is no setup required specific to the SecretManagement module. Aside from installing it. It just works as a translator for the different vaults. However, you will need to

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

configure the various vaults and register them with the SecretManagement module so it knows how to access them.

4.3.2 Set up the SecretStore vault

Using the SecretStore module is a great way to familiarize yourself with the SecretManagement module. However, it does have one major drawback in that any vaults you create are tied to that specific user on that specific machine. This means that you cannot share the vault with others or easily move it to another system. But the setup is simple, and you can get it up and running in a matter of minutes.

To show you how simple it is, we will set up the SecretManagement module, along with the SecretStore, to support the SQL health check automation.

INSTALL THE MODULES

The first step in the process is to install the two modules. They are both available from the PowerShell gallery, so you can install them by using PowerShellGet.

```
PS P:\> Install-Module Microsoft.PowerShell.SecretStore
PS P:\> Install-Module Microsoft.PowerShell.SecretManagement
```

CONFIGURE THE SECRETSTORE VAULT

Once you have the modules installed, you need to create a vault in the SecretStore. To do this, you will run the `Get-SecretStoreConfiguration` cmdlet. If this is your first time setting up the SecretStore, you will receive a prompt to provide a password. This is the password that you will use to access the vault.

```
PS P:\> Get-SecretStoreConfiguration
Creating a new Microsoft.PowerShell.SecretStore vault. A password is required by the
       current store configuration.
Enter password:
********
Enter password again for verification:
********

      Scope Authentication PasswordTimeout Interaction
      ----- -------------- --------------- -----------
CurrentUser       Password             900      Prompt
```

Before continuing, you need to consider the implications of needing to enter a password to access the vault. While it is more secure to have a password, it does not work well for unattended automations because it requires user interaction. This is where the fact that the vault is tied to a machine and user comes in handy.

Since we want the SQL health check to run unattended, you will turn off the password requirement. When doing that, it is also a good idea to set the interaction to *none* to prevent scripts from hanging if the password requirement is turned back on for some reason. If the interaction is set to *none* and a password is required, an exception is thrown, causing the script to stop.

Go ahead and disable the password and the interaction using the `Set-SecretStoreConfiguration` cmdlet.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
PS P:\> Set-SecretStoreConfiguration -Authentication None -Interaction None

Confirm
Are you sure you want to perform this action?
Performing the operation "Changes local store configuration" on target "SecretStore module
       local store".
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "Y"): y
A password is no longer required for the local store configuration.
To complete the change please provide the current password.
Enter password:
********
```

By turning off the password requirement, your area of vulnerability is the account and machine that contains the vault. In this case, you will want to make sure that you take all the appropriate actions to secure this account, including using a strong password and auditing the logins made by this account and this machine.

REGISTER THE VAULT

The previous steps were all setting up the vault in the SecretStore module. Next, you need to register the vault with the SecretManagement module, so it knows where to look for the secret values. You do this by passing the name of the extension module and the vault's name to the `Register-SecretVault` cmdlet. The vault’s name is what you will use to reference it in your scripts.

For the SQL health check, go ahead and create a new vault named *SQLHealthCheck*.

```
PS P:\> Register-SecretVault -ModuleName Microsoft.PowerShell.SecretStore -Name
       SQLHealthCheck
```

The SecretStore module allows you to create multiple vaults under the same profile. Doing this can help with the management and organization of your secrets.

Some vaults require you to provide information to *VaultParameters* parameters, but since the SecretStore is tied to the machine and user, there are no additional configurations required.

Setting a default vault

The SecertManagement module allows you to set a default vault, so you do not have to specify a name when retrieving secrets. This is fine for personal use but should not be relied upon for automations. You should always provide the vault name in automations to prevent failures if the default vault is changed.

4.3.3 Set up a KeePass vault

As mentioned, there are over a dozen different vaults that the SecretManagement module can use. So, we are going to take a look at setting up a second vault with KeePass. If you are not familiar with KeePass, it is a free, open-source password manager. And unlike other solutions vaults like LastPass or Azure Key Vault, it is an entirely offline solution.

One major difference between a KeePass vault and a SecretStore vault is that KeePass uses a database file that you can move between machines and users or even store on a network share. Also, like SecretStore, it allows you to bypass entering a password, which

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

works well for automations. However, KeePass can use custom key files to provide an extra layer of security.

You do not need to install KeePass on a machine to use the KeePass extension. The extension module contains all the requirements to access the database file. However, you do need to have it installed in order to create a database file.

CREATE KEEPASS DATABASE VAULT

When you create a KeePass database for use by the SecretManagement module, you need to consider the same things you do with the SecretStore. And that is how it will run. If you are using it for unattended automations, you will want to exclude using a Master password. However, it would be a good idea to use a key file.

KeePass uses a key file as an extra layer of security. Instead of providing a password, your automation can provide a key file. Since KeePass databases are not restricted to the machine that created them, the key file can help protect you in situations where a bad actor may get a copy of the database file. Without the key file, it will be useless to them.

However, if a database file can be accessed using only the key file, you need to ensure that both the database file and the key file are appropriately stored and protected from unauthorized access. And most importantly, that they are never stored in the same place.

Go ahead and install KeePass and create a database file named *SmtpKeePass.kdbx* with a key file named *SmtpKeePass.keyx*. In the new database wizard, be sure to uncheck the Master password box, then check Show expert options to create the key file.

INSTALL KEEPASS SECRETMANAGEMENT EXTENSION MODULE

Once you have KeePass installed and your database vault set up, you need to install the KeePass extension for the SecretManagement module so that you can use it through PowerShell.

```
PS P:\> Install-Module SecretManagement.KeePass
```

REGISTER KEEPASS WITH SECRETMANAGEMENT

Finally, you can register your KeePass database with the SecretManagement module using the `Register-SecretVault` cmdlet again. However, unlike with the SecretStore, you will need to provide parameters for the vault. These are different for each vault type. For KeePass, you need to provide the path to the database file, if a master password is required, and the path to the key file.

For the SQL health check, run the `Register-SecretVault` cmdlet to register the SmtpKeePass database you just created. Set the *Path* to the full path of the SmtpKeePass.kdbx database file, *KeyPath* to the full path of the SmtpKeePass.keyx key file, and set *UseMasterPassword* to false.

```
PS P:\> Register-SecretVault -Name 'SmtpKeePass' -ModuleName SecretManagement.KeePass -
       VaultParameters @{
    Path = " \\ITShare\Automation\SmtpKeePass.kdbx"
    UseMasterPassword = $false
    KeyPath= "C:\Users\svcacct\SmtpKeePass.keyx"
}
```

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

Once registered, you can add and retrieve secrets using the same commands you use for the SecertStore and any other vault attached to the SecretManagement module.

4.3.4 Choosing the right vault

The vault that you choose depends on your requirements. For example, SecretStore vaults are tied to a particular user and machine. While this does increase the security, it reduces the portability of your automation. In contrast, there is no such restriction on KeePass database files. However, since the KeePass file can be moved to a different machine, it is also more susceptible to be comprised if it is not properly stored and secured.

The nice thing about the SecretManagement module is that it can point to multiple vaults on the backend. So, you can use a combination of various vaults to meet your automation and security needs.

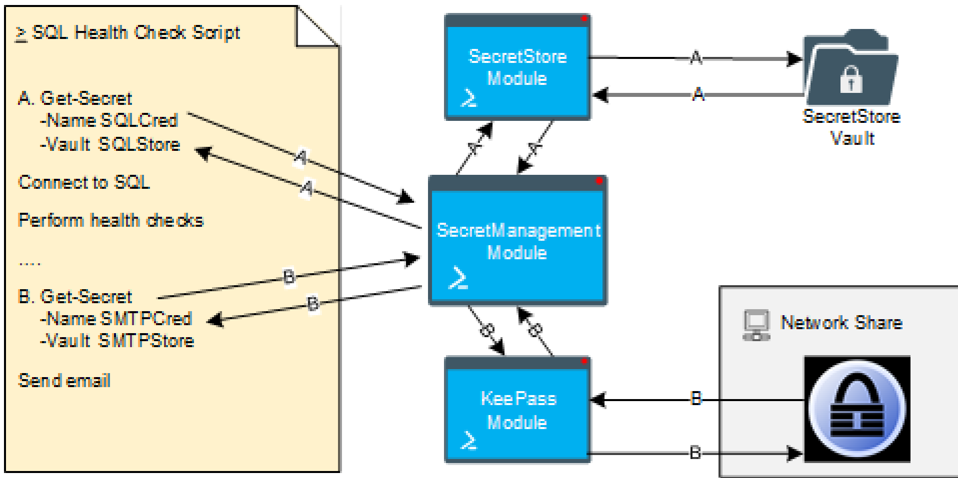


Figure 5 SecretManagement vault usage for the SQL health check automation

For example, in the SQL health check, you will use the KeePass vault to store the credentials needed to send the email via SendGrid, and the SecretStore to hold the secrets for connecting to the SQL instance. The reason for this is because you can have multiple automations that need to send emails.

If you create a new service account for each automation that needs to send an email, you will quickly run into a situation where you have so many accounts that you cannot manage them. On the other hand, if you have a couple of service accounts for sending emails used by multiple automations, you need to ensure that all automations can be easily updated if anything changes. This is where shared vaults come in handy. If you put the credentials for the email into a shared vault, you can update all automations at once if anything ever changes. While at the same time having a separate vault that contains the SQL connection information, providing you an extra layer of security. This way, if any one of your vaults gets comprised, the remaining automation will be unaffected.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

4.3.5 Adding secrets to a vault

Once you have your vaults created and registered to the SecretManagement module, you can add your secrets to them. You use the `Set-Secret` cmdlet to add a secret to a vault. You need to specify the parameters:

- Name - The name for the secret. This will be used to retrieve the value later.
- Vault – The name of the vault to store the secret. If not specified, the default vault is used.
- Secret - The object containing the secret. The supported object types are
    - byte[]
    - String
    - SecureString
    - PSCredential
    - Hashtable

The value for the *Secret* parameter should be the only one that contains sensitive information. Also, if a secret already exists in the vault with the same name, the `Set-Secret` cmdlet will overwrite it unless you use the *NoClobber* switch.

You will need to create secrets in both the SecretStore vault and the KeePass vault for the SQL health check automation. First, in the SecretStore vault, you will create a PSCredential object with the username and password for the SQL connection and SecureString with the name of the SQL server instance. Then in the KeePass vault, you will create the entries for the SendGrid API.

Table 4.1 Secrets required for the SQL health check script using the SecretsManagement module

Secret Name	Vault	Secret Object Type	Notes
TestSQL	SQLHealthCheck	String	<Your server name>\SQLEXPRESS *
TestSQLCredential	SQLHealthCheck	Credentials	Username* : sqlhealth Password* : P@55w9rd
SendGrid	SmtpKeePass	SecureString	Your email address
SendGridKey	SmtpKeePass	Credentials	When using SendGrid, the username will be *apikey*, and the password will be your API key.

* These are the default values if you used the setup scripts provided in the GitHub repo

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

Naming related Secrets

Whenever you have two or more secrets that are related to each other, as you do in the SQL health check, you should name them with the same prefix. Not only will it make it easier to keep track of, but it will also allow you to simplify your parameters by enabling you to pass one value that you can then append multiple suffixes to.

For example, to get the SQL secrets from table 4.1, you only need to pass the value 'TestSQL'. Then to get the credentials, you can simply append 'Credential' to the name.

```
$SqlName  = 'TestSQL'
$SqlCred  = "$($SqlName)Credential"
```

The same can be done for the SendGrid secrets by adding Key to the end of the name.

Create the entries for the SecretStore vault.

```
PS P:\> $SQLServer = "$($env:COMPUTERNAME)\SQLEXPRESS"
PS P:\> Set-Secret -Name TestSQL -Secret $SQLServer -Vault SQLHealthCheck
PS P:\> $Credential = Get-Credential
PS P:\> Set-Secret -Name TestSQLCredential -Secret $Credential -Vault SQLHealthCheck
```

Create the entries for the KeePass vault.

```
PS P:\> $SmtpFrom = Read-Host -AsSecureString
PS P:\> Set-Secret -Name SendGrid -Secret $SmtpFrom -Vault SmtpKeePass
PS P:\> $Credential = Get-Credential
PS P:\> Set-Secret -Name SendGridKey -Secret $Credential -Vault SmtpKeePass
```

Now that you have the secrets stored in the vault, you are ready to set up your automation script to retrieve and use them.

4.4 Using credentials and secure strings in your automations

As you will see, using the SecretManagement module to retrieve sensitive is a simple and easy process. However, there are still other viable options out there for storing and providing sensitive data to your scripts. Many popular automation and CD/CI platforms have ways of delivering credentials and other values to your scripts safely and securely. Using the SQL health check script, we will set up the automation first using the SecretManagement module and then using Jenkins with its built-in vault.

4.4.1 SecretManagement module

Once you have the vaults set up and registered to SecretManagement, you can start using them to retrieve secrets in your automations. Remember, the configurations are set to the user profile, so you need to be logged in as the user who will access the vault when configuring it. Then you need to ensure that your automation is running as that user. You can refer to chapter 3 to learn about scheduling automations to run as a particular user.

To retrieve a secret in your script, all you need to do is use the `Get-Secret` cmdlet. This cmdlet will return the first secret that matches the name you provided. This is why it is important to ensure that all of your secrets have unique names and why it is always good to include the *Vault* parameter. If you don't use the *Vault* parameter, the SecretManagement module searches through all vaults, starting with the default one.

Licensed to Tausif Khan <tsfahmed2@gmail.com>

By default, string secrets are returned as SecureString objects. However, if you need them returned as plain text, you can include the *AsPlainText* switch.

As you will see in the SQL health check script, you can use any combination of vaults and secrets you need inside your script. In this scenario, you need to retrieve the SQL server connection information from one vault and the SendGrid information from another. Also, when you retrieve the SQL server name and email address from the vault, these need to be converted into standard strings using the *AsPlainText* switch. Let us start by testing the SQL and SendGrid connections separately.

Starting with the SQL connection test, you can retrieve the SQL connection information from the SecretStore vault, then run a simple SQL query using the `Invoke-DbaDiagnosticQuery` cmdlet. If the command output contains data in the Results property, then you know it worked. If the Results property is empty, check the verbose output. If there is a permissions issue, it will be listed there. Try tweaking your SQL permissions and testing again until you receive the expected results.

Listing 1 Test SQL Connection information from SecretStore vault

```
$Secret = @{    #A
    Name  = 'TestSQLCredential'
    Vault = 'SQLHealthCheck'
}
$SqlCredential = Get-Secret @Secret
$Secret = @{    #B
    Name  = 'TestSQL'
    Vault = 'SQLHealthCheck'
}
$SQLServer = Get-Secret @Secret -AsPlainText

$DbaDiagnosticQuery = @{    #C
    SqlInstance   = $SQLServer
    SqlCredential = $SqlCredential
    QueryName     = 'Database Properties'
}
Invoke-DbaDiagnosticQuery @DbaDiagnosticQuery -Verbose
```

#A Retrieve credentials for SQL server connection.
#B Retrieve the SQL server name and convert it to plain text.
#C Execute a diagnostic query against SQL to test connection information from the SecretStore vault.

Next, you can test sending an email through SendGrid using the `Send-EmailMessage` cmdlet. In this case, you will retrieve this SendGrid API key and the email address from the KeePass vault. Then send a test email to the same email address. If you receive the email, you are ready to move forward with putting it all together in the SQL health check automation.

Licensed to Tausif Khan <tsfahmed2@gmail.com>

Listing 2 Test SendGrid Connection information from KeePass vault

```
$Secret = @{    #A
    Name  = 'SendGrid'
    Vault = 'SmtpKeePass'
}
$From = Get-Secret @Secret -AsPlainText
$Secret = @{    #B
    Name  = 'SendGridKey'
    Vault = 'SmtpKeePass'
}
$EmailCredentials = Get-Secret @Secret

$EmailMessage = @{    #C
    From       = $From
    To         = $From
    Credential = $EmailCredentials
    Body       = 'This is a test of the SendGrid API'
    Priority   = 'High'
    Subject    = "Test SendGrid"
    SendGrid   = $true
}
Send-EmailMessage @EmailMessage
```

#A Get the email address for the send from in plain text
#B Get the API key for SendGrid
#C Send test email with the SendGrid connection information from the KeePass vault.

Now that you have confirmed you can run a health check query and send an email, it is time to put everything together into an automation script, starting with the parameters. You will want to set the names of the vaults and the names of the secret objects as parameters, so you can easily use this script across different SQL instances. Since the related secrets are named with the same prefix, you only need to prompt for the prefix and then have the script append the suffix. So, instead of having a parameter for the SQL instance secret and a second one for the credential secret, you just need to create one to pass the prefix. Then in your script, append *Credential* to the name. You can do the same for the SendGrid API key by appending *Key* to the variable. Finally, you will need a parameter for the email to send to in the case of a failed check.

Next, the script will get the SQL connection information from the vault, then execute the *'Database Properties'* query from the `Invoke-DbaDiagnosticQuery` cmdlet to return the recovery model information for all the databases. Next, it will confirm that all of them are set to simple. If any are not set to simple, it will gather the information to send an email notification by getting the SendGrid secrets. Then it will create the email body by converting the PowerShell object containing the bad databases to an HTML table. And finally, it sends the email.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

Listing 3 SQL health check

```
param(
    [string]$SQLVault,
    [string]$SQLInstance,
    [string]$SmtpVault,
    [string]$FromSecret,
    [string]$SendTo
)
$Secret = @{    #A
    Name  = "$($SQLInstance)Credential"
    Vault = $SQLVault
}
$SqlCredential = Get-Secret @Secret
$Secret = @{    #B
    Name  = $SQLInstance
    Vault = $SQLVault
}
$SQLServer = Get-Secret @Secret -AsPlainText

$DbaDiagnosticQuery = @{    #C
    SqlInstance   = $SQLServer
    SqlCredential = $SqlCredential
    QueryName     = 'Database Properties'
}
$HealthCheck = Invoke-DbaDiagnosticQuery @DbaDiagnosticQuery
$failedCheck = $HealthCheck.Result |
    Where-Object { $_.'Recovery Model' -ne 'SIMPLE' }

if ($failedCheck) {
    $Secret = @{    #D
        Name  = $FromSecret
        Vault = $SmtpVault
    }
    $From = Get-Secret @Secret -AsPlainText
    $Secret = @{    #E
        Name  = "$($FromSecret)Key"
        Vault = $SmtpVault
    }
    $EmailCredentials = Get-Secret @Secret

    $Body = $failedCheck | ConvertTo-Html -As List |    #F
        Out-String

    $EmailMessage = @{    #G
        From       = $From
        To         = $SendTo
        Credential = $EmailCredentials
        Body       = $Body
        Priority   = 'High'
        Subject    = "SQL Health Check Failed for $($SQLServer)"
        SendGrid   = $true
    }
    Send-EmailMessage @EmailMessage
}
```

#A Retrieve credentials for SQL server connection
#B Retrieve the SQL server name and convert it to plain text.
#C Execute the Database Properties diagnostic query against SQL

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

#D Get the email address for the send from in plain text
#E Get the API key for SendGrid
#F Create email body by converting failed check results to HTML table
#G Send failure email notification

4.4.2 Using Jenkins credentials

Before the SecretManagement module, the best way to store credentials was by using a third-party platform that has its own store. One of these platforms is Jenkins. So while you can use the SecretManagement module for scripts that run through Jenkins, you can also use the built-in Jenkins store.

One of the advantages of using the Jenkins credentials is that you do not need to worry about having the different secret vaults and module extensions installed on every Jenkins server. Instead, they will all be able to get credentials from one global store. Also, it provides a GUI interface and logging of which automations have used which objects.

By default, you will have a store named Jenkins, which we will use to store the SQL health check automation values. You can set up different vaults and even apply role-based access to them, but for now, we are just going to use the global store.

In this example, we will update the SQL health check script to use Jenkins credentials and variables instead of the SecretManagement vaults.

To create credentials in Jenkins, you need to navigate to Manage Jenkins > Manage Credentials, then click on the store you want to use. In this case, it will be the Jenkins store. Once in the store, click on Global credentials, and create the credential objects listed in table 4.2.

Table 4.2 Secrets required for the SQL health check script using Jenkins

ID	Kind	Value
TestSQL	Secret text	<Your server name>\SQLEXPRESS *
TestSQLCredential	Username with password	Username* : sqlhealth Password* : P@55w9rd
SendGrid	Secret text	Your email address
SendGridKey	Username with password	When using SendGrid, the username will be *apikey*, and the password will be your API key.

* These are the default values if you used the setup scripts provided in the GitHub repo

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

Credentials that should be available irrespective of domain specification to requirements matching.

	ID	Name	Kind	Description
	TestSQL	TestSQL	Secret text	
	TestSQLCredential	sqlhealth/******	Username with password	
	SendGrid	SendGrid	Secret text	
	SendGridKey	apikey/******	Username with password	

Icon: S M L

Figure 6 Jenkins vault objects for the SQL health check automation

Now that you have the credentials and secrets created, you can use them in your scripts. But before you do, there are some things you need to know. Jenkins has its own methods for storing secret values and therefore does not have direct support for the SecureStrings and PSCredential objects in PowerShell. So, it loads the values into environment variables at the time of execution as standard unsecured strings. It has its own ways of preventing those variables from being written to the output logs or saved in memory. Therefore, when you need to use the values as SecureStrings or PSCredentials, you will need to convert them back from standard strings using the `ConvertTo-SecureString` cmdlet.

In the SQL health check script, you will need to recreate the credentials for the SQL connection and the SendGrid API key. The email address and SQL instance do not need to be converted because they are expected to be unsecured strings.

To get started, create a new Freestyle project in Jenkins. Then, under the Binding section, you will need to add two bindings for Username and password (separated) and configure them to use the values in table 4.3.

Table 4.3 Credential bindings required for the SQL health check script in Jenkins build

Username Variable	Password Variable	Credentials
sqlusername	sqlpassword	TestSQLCredential
sendgridusername	sendgridpassword	SendGridKey

Then add two binds for secret text and configure them to use the values in table 4.4.

Table 4.4 Secret text bindings required for the SQL health check script in Jenkins build

Variable	Credentials
sqlserver	TestSQL
sendgrid	SendGrid

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

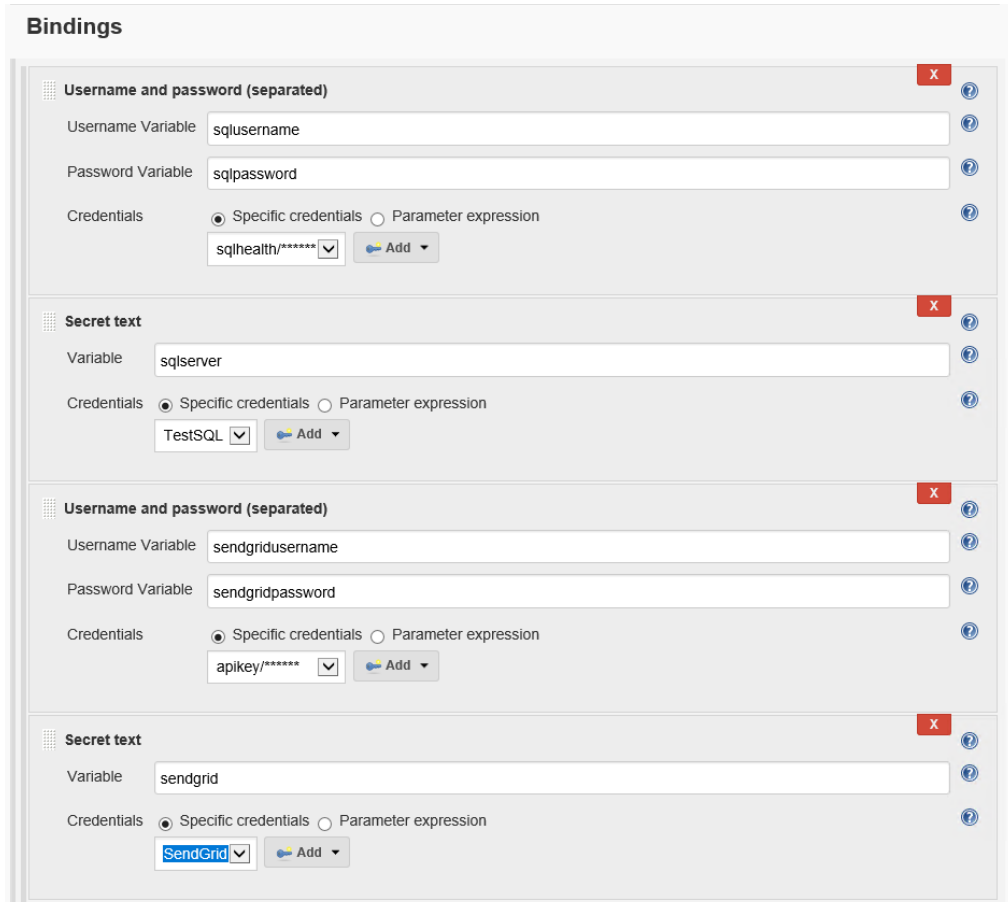


Figure 7 Jenkins bindings for the SQL health check automation project

Next, you need to update the SQL health check script to replace the calls to the SecretManagement vaults with the environment variables for Jenkins. Remembering that you will need to recreate the credentials.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

Listing 4 SQL health check through Jenkins

```
$secure = @{    #A
    String = $ENV:sqlpassword
    AsPlainText = $true
    Force = $true
}
$Password = ConvertTo-SecureString @secure
$SqlCredential = New-Object System.Management.Automation.PSCredential `
    ($ENV:sqlusername, $Password)

$SQLServer = $ENV:sqlserver    #B

$DbaDiagnosticQuery = @{
    SqlInstance   = $SQLServer
    SqlCredential = $SqlCredential
    QueryName     = 'DatabaseProperties'
}
$HealthCheck = Invoke-DbaDiagnosticQuery @DbaDiagnosticQuery
$failedCheck = $HealthCheck.Result |
    Where-Object { $_.'Recovery Model' -ne 'SIMPLE' }

if ($failedCheck) {
    $From = $ENV:sendgrid    #C
    $secure = @{    #D
        String = $ENV:sendgridusername
        AsPlainText = $true
        Force = $true
    }
    $Password = ConvertTo-SecureString @secure
    $Credential = New-Object System.Management.Automation.PSCredential `
        ($ENV:sendgridpassword, $Password)

    $Body = $failedCheck | ConvertTo-Html -As List |
        Out-String

    $EmailMessage = @{
        From       = $From
        To         = $SendTo
        Credential = $EmailCredentials
        Body       = $Body
        Priority   = 'High'
        Subject    = "SQL Health Check Failed for $($SQLServer)"
        SendGrid   = $true
    }
    Send-EmailMessage @EmailMessage
}
```

#A Replace Get-Secret call with Jenkins environment variables and recreate the credential object
#B Replace Get-Secret call with Jenkins environment variables
#C Replace Get-Secret call with Jenkins environment variables
#D Replace Get-Secret call with Jenkins environment variables and recreate the credential object

If you want to make this build dynamic so you can run it against different SQL instances, all you need to do is change the bindings from specific credentials to parameter expression. Then you can enter the name of the secrets to use at runtime.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

4.5 Know your risks

Throughout this chapter, you have seen, there is always a potential vulnerability introduced when creating unattended automations. However, by knowing what they are, you can help mitigate them. Unfortunately, many companies can be so scared of automation from a security perspective that they make it impossible to get anything done. That is why it will be your job to understand the risks so that you can explain them and the steps you have taken to mitigate them.

In our example, with the SQL health check, we were able to reduce the risks using multiple different techniques. First, we used an account specifically for running the health check, with the minimum required permissions. Along with a separate API key to use SendGrid for sending notifications. We also prevented any sensitive data from being stored in the script by using the SecretManagement module and Jenkins credential stores.

However, even though we significantly reduced the vulnerabilities, there are still risks that you need to be aware of and take the steps to properly account for them. For example, the KeePass vault can be copied by a bad actor and used on a machine outside of your network. However, by using a key file and storing that key file in a separate secure location, that database would be worthless to them. You will also want to ensure that only authorized personnel can access either file through appropriate file and share permissions.

On top of this, just using SendGrid, instead of an SMTP relay provides you additional levels of security. For example, you can create IP-based restrictions, view audit logs, and create alerts for anomalous activity. Therefore, even if your KeePass database is copied and someone gets a copy of the key file, they may not be able to do anything with it.

Another potential vulnerability to consider is that Jenkins passes the secret values to PowerShell as unsecured strings. However, Jenkins has built-in security to prevent these values from being written out to any output stream, and the PowerShell runtime is terminated after execution, clearing the memory used. So really, the only way someone could get the values would be to perform a memory dump during the script's execution. And if someone has access to your server and can perform a memory dump completely undetected, you have much bigger problems on your hands.

In all cases, we were able to reduce any risks added by introducing unattended automations. Of course, not even an air-gapped computer with an armed guard sitting next to it is 100% secure, but that doesn’t mean you shouldn’t do all you can to minimize risk. And no security officer should ever expect that. However, I can tell you from experience that being open and honest with your security team about the risks and the steps you have taken to reduce them can make getting your automations approved a much easier process.

4.6 Summary

- Following some basic principles like using role-based access, assigning the least privilege, and not relying on security through obscurity can help to ensure that your automations are safe and secure.
- SecureString and PSCredential objects can be used natively in PowerShell to keep your sensitive data secure during execution and at rest.
- The SecretManagement module can be used with multiple password vaults to provide

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

secure storage and access to sensitive data you need in your scripts.

- Many platforms have built-in vaults that you can use in place of the SecretManagement module.
- Knowing what your risks are, is the only way that you can reduce them to an acceptable level.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

5
PowerShell remote execution

This chapter covers

- **Designing scripts for remote execution**
- **PowerShell-based remoting**
- **Hypervisor-based remoting**
- **Agent-based remoting**

The ability to execute remote PowerShell commands is not only essential for recurring automations but is also a powerful tool to have in your arsenal for ad-hoc situations. Just think back on your career and remember times when you needed to gather large-scale information about your environment or apply a change across multiple servers at once. You will quickly realize this is a common situation for any IT department. And in some cases, especially security-related ones, time can be of the essence. Therefore, before these situations arise, you will want to have PowerShell remoting set up and know how to adapt your scripts for remote execution.

For instance, in May of 2021, security researchers identified vulnerabilities in several Visual Studio Code (VS Code) extensions. While discovering installed versions of VS Code may be simple, finding the installed extensions can present a significate challenge. This is because extensions are installed at the user level and not the system level. Therefore, a lot of scanning tools will not pick them up. Fortunately, all VS Code extensions contain a vsixmanifest file, which we can search for and read to identify installed extensions.

We will use this scenario through this chapter to demonstrate the different ways that you can execute PowerShell remotely and how you will need to adjust your scripts depending on which type of remote execution you use. You can then apply these same fundamentals principles to all remote execution automations. But before we get into that, let's quickly cover some of the basic concepts of PowerShell remoting.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

5.1 PowerShell remoting

When discussing PowerShell remoting, there are two things you need to understand. One is the remote execution protocols or how the machines talk to each other. The other is the remote execution context or how the remote sessions behave.

For clarity, the machine you will be making the remote connects from is the *client*. Any devices you are connecting to are the *servers*.

5.1.1 Remote Context

There are three main types of remote execution context;

1. Remote commands
2. Interactive sessions
3. Imported sessions

A remote command is when you execute a predefined command or script against a remote server. Most commonly, you use the `Invoke-Command` cmdlet for this in automation scenarios. It not only allows you to execute the commands on the remote server, but it also allows you to return the results to the client machine.

In the VS Code extension scenario, this type of remote execution is the best choice. For one, it allows you to execute predefined scripts and commands, which you will have. Two, you can return all the results to a single session, allowing you to view the results from all machines in a single place.

Interactive context is when you use the `Enter-PSSession` cmdlet to enter a remote session. This is the equivalent of opening and PowerShell prompt on the remote server. It is suitable for running one-off commands but does not lend itself very well to automation because the information from the commands is not returned to the local client.

Import context is when you use the `Import-PSSession` cmdlet to import the cmdlets and functions from a remote session into your local session. This allows you to use the commands without needing to install modules locally. It is most often used for Office 365 and Exchange-based automations. However, since the cmdlets are imported to the local client, it provides no way to interact with the remote server.

5.1.2 Remote Protocols

When you look into PowerShell remoting protocols, it is easy to get overwhelmed. There are so many acronyms, initialisms, and abbreviations that it can be hard to tell what anything is. For instance, PowerShell 7 supports WMI, WS-Management (WSMan), SSH, and RPC remoting. You will need to know which protocols to use based on the context required for the automation and the remote server's operating system.

WMI and RPC remoting have long been staples of PowerShell and Windows remoting in general. With the keyword here being Windows. If you have ever used a cmdlet that contains the `-ComputerName` parameter, then chances are you have used either WMI or RPC. These protocols work great but can be very limiting. Not only are they restricted to Windows, but there are a limited number of cmdlets that contain the `-ComputerName` parameter. Therefore,

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

to truly take full advantage of PowerShell remoting capabilities, you should use WSMan and SSH for executing remote commands.

The WSMan and SSH protocol create remote PowerShell sessions that let you run PowerShell under any remote context. Which one you use will depend on your particular environment. WSMan only works on Windows-based machines and can support local or Active Directory authentication. SSH can support both Windows and Linux but does not support Active Directory authentication.

> **WINRM** You will often hear the terms WinRM and WSMan used interchangeably. This is because WSMan is an open standard, and WinRM is Microsoft's implementation of that standard.

Just as there are mixed environments nowadays, there is no reason you can't use a mixture of protocols. In most cases, if a machine is domain joined, I will use WSMan; otherwise, I'll use SSH. As you will see, you can easily adapt your scripts to support both.

5.1.3 Persistent Sessions

When using the `Invoke-Command` and `Enter-PSSession` cmdlets, you have the option to either establish the session at the time of execution by using the `-ComputerName` argument or use an existing session. Also known as a persistent session. You can create these persistent sessions using the `New-PSSession` cmdlet.

Persistent sessions allow you to connect to the same session multiple times. You can also use them to create connections to multiple remote servers and execute your commands against all of them at once, providing you will parallel execution.

End-user devices and PowerShell remote execution

Remote PowerShell execution is generally limited to server devices and not end-user devices. The biggest reason for this is security. Allowing remote execution on end-user devices can leave you open to vulnerabilities if a device becomes comprised. And as you know, the risk of an end-user device become compromised is exponentially larger than a server. Also, servers' network configurations remain relatively static compared to end-user devices. With the growing trend of work-anywhere, there is no guarantee that an end-user device will be reachable over your WAN. Therefore, it is best to use configuration management software or MDM for managing end-user devices with PowerShell.

We will cover this in-depth in chapter 12.

5.2 Script considerations for remote execution

There are two types scripts of scripts we will talk about with regards to remote execution. The first is the scripts for execution on the remote server. In our scenario, it will be the script to find the VS Code extensions, but it can be any script you want to run remotely. The second is the control script. The control script runs on the local client and tells the remote servers to execute the script.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

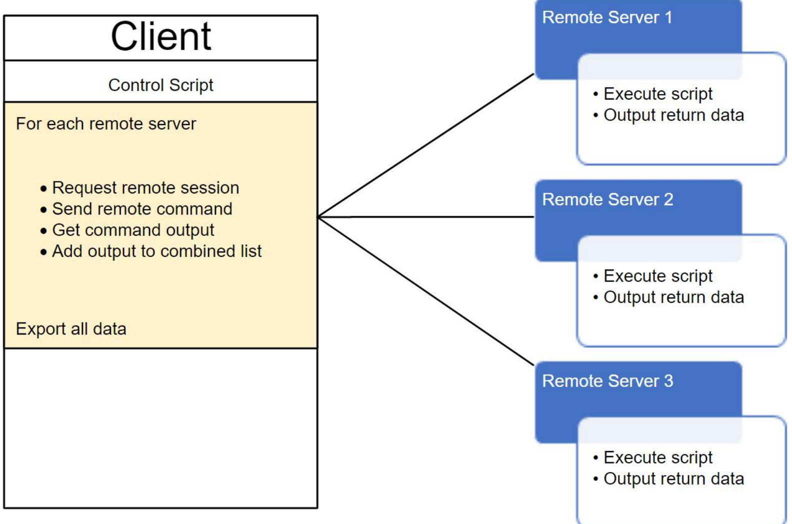


Figure 1 Control scripts are used to execute a PowerShell script or script block across multiple machines and return the data to a single place

For the majority of this chapter, we will be discussing and working with control scripts. You will design these control scripts to be reusable for any script that you need to execute remotely. However, before we dive in, there are a few things you need to be mindful of when creating a script you know will be used for remote execution.

5.2.1 Remote execution scripts

All of the automation script fundamentals we've discussed in the other chapters still apply when designing a remote execution script. These include ensuring the remote server has any required modules installed and that the script does not contain any commands that would stop and wait for user interaction. On top of these, you will want to ensure that any information returned from a remote execution is appropriately formatted and that your script can work on the required operating systems and PowerShell versions. Again, we will use the VS Code extension example to help illustrate these, but keep in mind these things apply to any remote PowerShell execution and not just this specific scenario.

Finding the installed VS Code extensions may seem like a reasonably simple task. You just need to search the VS Code extensions folder inside each user's home folders, gather the extensions found, and return the results. And if none are found, return a message stating that.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

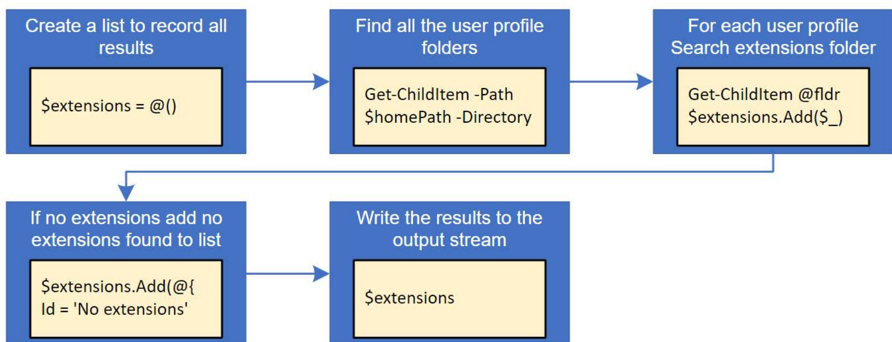


Figure 2 Search all user profiles for install VS Code extensions and return results

Since we know the cmdlets required to perform this task are built-in, we do not need to worry about module dependencies. We also know that they do not require user interactions. So, we can move on to the remote execution considerations.

The first thing you need to determine is what operating systems and PowerShell versions the remote machines are using. Ideally, you would want to write one script that you can run on all devices. This way, if you need to change something, you only have to change it once.

In the VS Code extensions scenario, you are searching in user home folders, and different operating systems have different home paths. In other scenarios, there could be different environmental variables, services, system paths, or any other number of things. Luckily, PowerShell has built-in variables to help you deal with this. The variables `$IsLinux`, `$IsWindows`, or `$IsMacOS` will return true or false depending on the operating system. Using these allows you to set your own variables for the specific operating system while leaving the rest of the script universal. For our scenario, you can create an if/else condition to set the home path based on the operating system and leave the rest of the script the same.

Ideally, all servers would be running PowerShell 7 or greater, but in reality, that is not always the case. There are plenty of situations where you need to ensure your scripts can run in Windows PowerShell and PowerShell Core. While the majority of PowerShell commands remained the same, there are some breaking changes between the two versions. It is also easy to fall into the habits of the newer versions. For example, the `$IsLinux`, `$IsWindows`, or `$IsMacOS` variables were introduced with PowerShell Core. So, to account for this, you could build some logic into your scripts to check PowerShell versions and end up with multiple different nested if/else conditions, or you could use your knowledge of PowerShell to your advantage. You know that prior to PowerShell Core, PowerShell only ran on Windows. Therefore, it doesn't have the `$IsLinux`, `$IsWindows`, or `$IsMacOS` variables. This means you can use a simple; if `$IsLinux`, use this path, else use the Windows path. Since the `$IsLinux` variable doesn't exist in Windows PowerShell, it will always use the else path. You can even throw an else if `$IsMacOS` in there if needed.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
if ($IsLinux) {
    # set Linux specific variables
}
elseif ($IsMacOS) {
    # set macOS specific variables
}
else {
    # set Windows specific variables
}
```

The next item to consider is how to structure any data you need to return from remote execution. Similar to executing a function, everything written to the output stream by the script will be returned to the control script. Therefore, you need to know what commands write to the output stream and only return the information you need.

Another item that seems obvious but is often overlooked is adding the machine's name to the information returned. If you run a script against ten machines and they all return data, it doesn’t do any good unless you know which machine returned it. While some remote protocols will automatically add the remote machine's name to the return data, others do not. So, it is best to have the script return the machine name in the output. This will guarantee that you will know which machine it was, regardless of the remote method used.

> **NOTE** **The environment variable to return the machine name** `$env:COMPUTERNAME` **doesn't work in Linux, but the .Net Core call** `[system.environment]::MachineName` **works in Linux and Windows.**

Also, it is good practice to add the date and time to the output, especially if you plan to execute the script multiple times.

One last thing to consider here is what to do if there is no data to return. For example, if no extensions are found with the VS Code extension script, there is nothing to return. However, that also means you don’t have a record that it ran. Therefore, you will want to include a check in your code when designing remote execution scripts to return something, even if the conditions are not met. In addition, you will want to ensure that this return is formatted the same way as if results were found, so your results can be stored together. You will see why this is important in the next section when we cover the control scripts.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

Listing 1 Get-VSCodeExtensions.ps1

```
[System.Collections.Generic.List[PSObject]] $extensions = @()
if ($IsLinux) {    #A
    $homePath = '/home/'
}
else {
    $homePath = "$($env:HOMEDRIVE)\Users"
}

$homeDirs = Get-ChildItem -Path $homePath -Directory    #B

foreach ($dir in $homeDirs) {    #C
    $vscPath = Join-Path $dir.FullName '.vscode\extensions'
    if (Test-Path -Path $vscPath) {    #D
        $ChildItem = @{
            Path    = $vscPath
            Recurse = $true
            Filter  = '.vsixmanifest'
            Force   = $true
        }
        $manifests = Get-ChildItem @ChildItem
        foreach ($m in $manifests) {
            [xml]$vsix = Get-Content -Path $m.FullName    #E
            $vsix.PackageManifest.Metadata.Identity |    #F
            Select-Object -Property Id, Version, Publisher,
            @{l = 'Folder'; e = { $m.FullName } },    #G
            @{l = 'ComputerName'; e = {[system.environment]::MachineName}},
            @{l = 'Date'; e = { Get-Date } } |
            ForEach-Object { $extensions.Add($_) }
        }
    }
}
if ($extensions.Count -eq 0) {    #H
    $extensions.Add([pscustomobject]@{
            Id           = 'No extension found'
            Version      = $null
            Publisher    = $null
            Folder       = $null
            ComputerName = [system.environment]::MachineName
            Date         = Get-Date
        })
}
$extensions    #I
```

#A Set home folder path based on the operating system
#B Get the subfolders under the home patch
#C Parse through each folder and check for VS Code extensions
#D If the VS Code extension folder is present, search it for vsixmanifest files
#E Get the contents of the vsixmanifest file and convert it to PowerShell XML object
#F Get the details from the manifest and add them to the extensions list
#G Add the folder path, computer name, and date to the output
#H If no extensions are found, return a PowerShell object with the same properties stating nothing found.
#I Just like an extension include the output at the end

Once you have created this script, please save it to your local machine with the name `Get-VSCodeExtensions.ps1`. This is the script you will be using to test your control scripts.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

5.2.2 Remote execution control scripts

When executing a script against multiple remote servers, you will want to have a control script. The control script will create the remote connection, run the remote command, and gather any information returned. Based on the type of remote execution you are performing, the cmdlets used will differ, but the overall process remains the same.

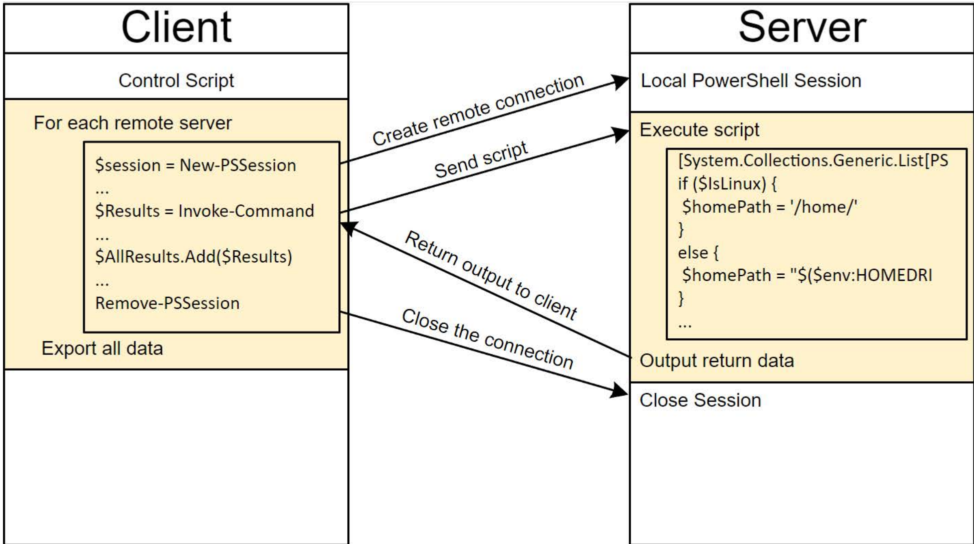


Figure 3 Control scripts are used to initiate PowerShell sessions across multiple machines and return the data to a single place

When designing your control scripts, one of the first things to consider is the execution time. For example, you can use a for each to loop through each remote device and run the script on it. However, these will not run in parallel. So, if your script takes 30 seconds to execute and you check 20 servers, it will take 10 minutes to complete. On the other hand, if you run the command on multiple servers at once, you can dramatically reduce execution time.

PowerShell offers multiple different ways to run remote commands sequentially. For example, some remote execution cmdlets, like the `Invoke-Command`, allow you to pass an array of computers. However, you need to be careful with this because if one computer fails, it could cause the others not to run. A better way to handle this is by using persistent connections.

Persistent connections allow you to establish the remote session before executing the command and will keep it active until you tell it to close. This will enable you to create an entire group of connections in a matter of seconds and execute the remote script against them all at once. It will also allow you to account for failed connections, so the rest of the remote devices are not affected. As you can see in figure 3 below, even if creating the

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

persistent connection takes around 1 second, you will end up saving time in the long run. And that is just with ten servers.

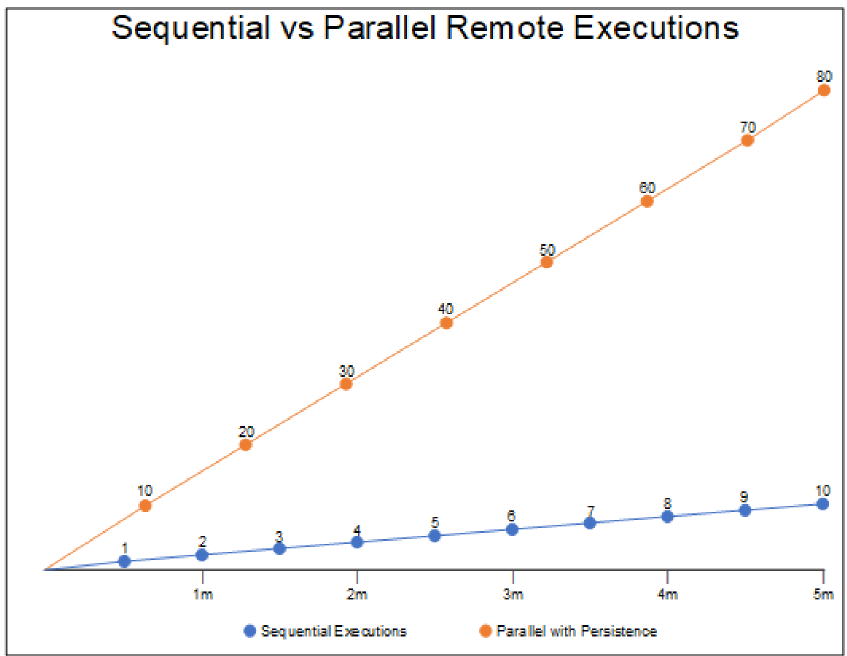


Figure 4 Comparison of the number of executions in a 5-minute window when using persistent connections for parallel execution versus individual executions

As with most things in automation, there is always the balancing act. You do not want to create so many connections at once that your computer and network slow to crawl. PowerShell remoting is limited to 32 persistent connections per session, but other remoting options may not have that limit. You will have to test and figure out what works best for you and your environment.

As with most things in PowerShell, there is always more than one way to accomplish something. Even in this chapter, you will see four different ways of executing remote scripts. In addition, there are other ways to run PowerShell processes in parallel, like for each parallel loops or using jobs, and those both present their unique challenges and advantages. However, from my experiences, persistent connections are the way to go when dealing with remote execution.

Aside from making the remote connections, the next most important thing is handling the data returned from each machine. If you run this script against five or ten computers, you can probably run them all at once and just save the output to a single variable. However, if you are executing against 50, 100, 1,000, or more, you must combine the executions' data.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

For example, if you have 50 machines to check, you can break them into five groups of ten. Then use an array to add the data together after each group finishes.

Consider a situation where you processed through half of the machines, and your network connection drops, or your PowerShell console closes. If you restart your script, it will resume from the beginning and check all of the devices again. This may not be a big deal if you are checking 50 machines, but what if you are checking 500 or 1,000 machines? Starting over will be a colossal waste of time. This is where saving your return data outside of PowerShell comes in handy. The easiest way to do this is by exporting it to a CSV file. PowerShell natively supports importing and exporting of objects to CSV using the cmdlets, `Import-Csv` and `Export-Csv`. And on top of that, CSV data is easily human-readable.

Formatting Data for Export-Csv

When using the Export-Csv cmdlet to append to an existing CSV file, you need to be aware that it will only include the fields that are currently in the header row. This is why it is crucial to return all data from your script using the same format.

If you export your results after each remote execution, you can reimport them if you need to restart the script. Then, all you have to do is check if the CSV file exists. If it does, load the data into a variable and then use it to filter your machine list to exclude those that have already been checked.

When filtering, you will want to ensure you use the value your script used to connect to create the remote session and not the name returned from the return data. For example, when using the `Invoke-Command` cmdlet, the `PSComputerName` property is automatically added to the output. By filtering on this, you will prevent duplicate submissions due to things like your control script using the FQDN and your script returning the NetBIOS or using DNS aliases.

One last thing to consider is creating a separate CSV export for any machines that fail to connect. This way, they are not stored in the same CSV as the actual results you want, and it provides you an excellent list to use for troubleshooting and fixing the failed connections.

Now, it is time to look into how you can execute your scripts on different remote systems and build your control scripts. First, starting with the native capabilities in PowerShell.

5.3 PowerShell remoting over WS-Management (WSMan)

When using PowerShell in an Active Directory environment WSMan is your best option. Not only does it support Active Directory authentication, but it is enabled by default on Windows server operating systems. You can also use Group Policy Objects (GPO) to enable WSMan remoting, making your setup more effortless.

5.3.1 Enable WSMan PowerShell remoting

Unfortunately, at this time, only Windows PowerShell can be controlled via GPO and not PowerShell Core. Therefore, if you want to execute remote command using PowerShell 6 or

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

above, you will need to run the `Enable-PSRemoting` cmdlet on each server. You can add the `-Force` switch to prevent prompts when enabling.

```
Enable-PSRemoting -Force
```

If you run the `Enable-PSRemoting` cmdlet and receive an error that one or more network connections are public, you can include the `-SkipNetworkProfileCheck` switch or make the connection private.

Also, the `Enable-PSRemoting` cmdlet will only enable remote PowerShell for the version you run the command in. So, for example, if you run the `Enable-PSRemoting` cmdlet in a PowerShell 7 session, it will not enable remoting in Windows PowerShell or the other way around.

5.3.2 Permissions for WSMan PowerShell remoting

By default, members of the *Administrators* and the *Remote Management Users* groups have permission to connect via PowerShell remoting to the server. Users in the *Remote Management Users* will over have the rights of a standard user unless they have additional permissions on the server.

For your VS Code extension scenario, you will want to give the user administrator privileges because they need to read the files inside each user profile.

5.3.3 Execute commands with WSMan PowerShell remoting

Now that your remote servers are set up, you are ready to start executing your remote commands. The next step will be determining how to perform the remote executions and recording the results. You will do this using a control script to invoke the name `Get-VSCodeExtensions.ps1` script you made earlier, but keep in mind this control script will be designed for use any other script you need to execute across multiple different systems

When using the `Invoke-Command` cmdlet, the script file only needs to be accessible to the local client and not the remote servers. You can also pass just a script block instead, which works well for one or two line commands. But when you are passing a complex script, it is best to save it as a separate script file. This will also allow you to make your control script reusable by not having the commands hardcoded.

Next, you need to provide the `Invoke-Command` cmdlet with the remote servers. As discussed earlier, you will want to create persistent sessions for each machine using the `New-PSSession` cmdlet. Then pass all the sessions to the `Invoke-Command` as an array to the `-Session` argument. Doing this will also allow you to isolate specific machines that failed to make the connection so that you can fix them separately.

When creating the array of remote sessions, you need to be careful to only add successful connections to the array. If you add a session that did not connect to your array, then pass that array to the `Invoke-Command`, it will error, and no commands will run on the remote server. To prevent this, you can wrap the `New-PSSession` command in a try/catch and set the `-ErrorAction` argument to `Stop`. Then if there is an error in the `New-PSSession`, your script will automatically jump to the catch block, skipping all other lines inside the try

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

block. You can see this in the snippet below. If the `New-PSSession` has an error, it will skip the line to add it to the array. Thus, ensuring your array only contains successful sessions.

```
try{
    $session = New-PSSession -ComputerName $s -ErrorAction Stop
    $Sessions.Add($session)
}
catch{
    Write-Host "$($s) failed to connect: $($_)"
}
```

To finish out the structure of your control script, you just need to add the list to collect all the returned data and the CSV import and export.

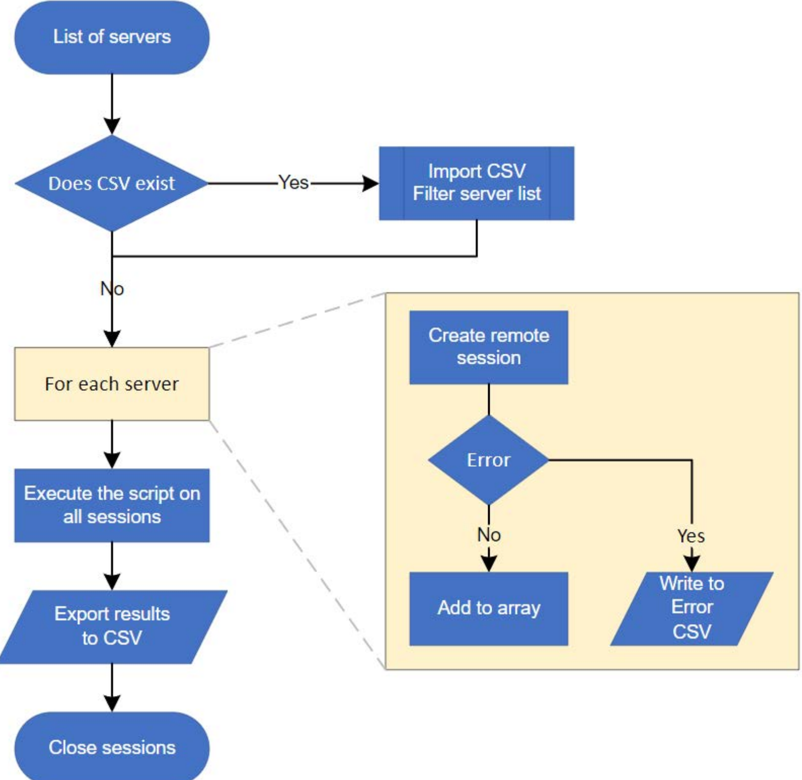


Figure 5 WSMan control script for remote script execution with persistent connections

The final step in the process is closing the remote sessions. When you create a session using the `New-PSSession` cmdlet, that session remains active on the local client and the remote server. To close it, you will use the `Remove-PSSession` cmdlet. This cmdlet will close the session, releasing the resources back to the remote server and closing the connection between the two machines.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

Listing 2 Execute local script against remote commputers using WSMan remoting

```
$servers = 'Svr01', 'Svr02', 'Svr03'    #A
$CsvFile = 'P:\Scripts\VSCodeExtensions.csv'    #B
$ScriptFile = 'P:\Scripts\Get-VSCodeExtensions.ps1'    #C
$ConnectionErrors = "P:\Scripts\VSCodeErrors.csv"    #D

if (Test-Path -Path $CsvFile) {    #E
    $csvData = Import-Csv -Path $CsvFile |
        Select-Object -ExpandProperty PSComputerName -Unique
    $servers = $servers | Where-Object { $_ -notin $csvData }
}

[System.Collections.Generic.List[PSObject]] $Sessions = @()
foreach ($s in $servers) {    #F
    $PSSession = @{
        ComputerName = $s
    }
    try {
        $session = New-PSSession @PSSession -ErrorAction Stop
        $Sessions.Add($session)
    }
    catch {
        [pscustomobject]@{    #G
            ComputerName = $s
            Date         = Get-Date
            ErrorMsg     = $_
        } | Export-Csv -Path $ConnectionErrors -Append
    }
}

$Command = @{    #H
    Session  = $Sessions
    FilePath = $ScriptFile
}
$Results = Invoke-Command @Command

$Results | Export-Csv -Path $CsvFile -Append    #I

Remove-PSSession -Session $Sessions    #J
```

#A Array of servers to connect to
#B Path to save results to
#C The script file from listing 1
#D Another CSV file to record connection errors
#E Test if the CSV file exists. If it does exclude the servers already scanned.
#F Connect to each server and add the session to the $Sessions array list
#G Add any errors to the connection error CSV file
#H Execute the script on all remote sessions at once
#I Export the results to CSV
#J Close and remove the remote sessions

5.3.4 Connect to the desired version of PowerShell

Before PowerShell 6, this is all you would need to execute a remote command. However, since PowerShell 6 and 7 are separate from Windows PowerShell, you may need to include the `-ConfigurationName` argument. If you don't specify this argument, then the cmdlet will

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

default to the value in the `$PSSessionConfigurationName` preference variable. Unless you expressly set this variable, it will default to using Windows PowerShell 5.1. Therefore, to use PowerShell 7 remotely, you need to specific `PowerShell.7` to the `-ConfigurationName` argument or set the value in `$PSSessionConfigurationName` variable.

The introduction of the `-ConfigurationName` argument means you have some additional items to consider in your automation. For example, if you use the PowerShell 7 configuration, your command will fail to connect to machines that don't have PowerShell 7 installed.

If you use the default of Windows PowerShell 5.1, you will need to ensure that your script can run in Windows PowerShell 5.1. Plus, as you see in the next section, SSH connections use the default on the remote machine. Since SSH is only supported in PowerShell 6 and later, you will need to ensure that your script will run in both PowerShell Core and Windows PowerShell.

As discussed earlier, most commands work the same in PowerShell Core and Windows PowerShell, but there are some breaking changes between them. Also, you are introducing complexity to your automation by trying to support both. In the long run, it is better to use the `-ConfigurationName` argument and fix any servers that are not configured correctly. Not only will it be cleaner this way, but you will be setting yourself up for future automations as well. But to keep things simple in this example, we will skip using it because the script can work in both versions.

5.4 PowerShell remoting over SSH

The Secure Shell Protocol (SSH) has been in use for Unix/Linux systems for over 25 years. However, in recent years Microsoft has started to introduce it in the Windows ecosystem. Starting PowerShell 6 SSH remoting can be done natively using OpenSSH. PowerShell can use SSH for remote connections between any combination of Windows, Linux, and macOS devices.

There are a few differences between SSH and WSMan remoting that you will need to be aware of. The first is SSH does not support Active Directory domain authentication. So, the accounts used for remote execution have to be a local account on the remote server. Also, SSH remoting does not support remote configuration. This means that you cannot specify the version of PowerShell to use on the remote server. Instead, PowerShell will automatically connect to the default version set on the remote server. There are also a few differences in the way you connect, which we will cover now.

5.4.1 Enable SSH PowerShell remoting

Unlike with WSMan, there is no command to enable SSH remoting for PowerShell. All of the configuration for SSH is done in the `sshd_config` file. Also, OpenSSH is not included in the PowerShell binaries, so you must install it separately. There are two components in OpenSSH, the client and the server. The client is for connecting to remote servers, and the server component accepts those connection requests. To enable SSH PowerShell remoting, you will need to perform the following steps.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

1. *Install OpenSSH*
2. *Enable the OpenSSH services*
3. *Set authentication methods*
4. *Add PowerShell to the SSH subsystem*

To get started, you need to install OpenSSH on the Windows device. If you are using Windows 10 build 1809 and later or Windows 2019 and later, OpenSSH is included as a feature. You can install it using the command below.

```
Get-WindowsCapability -Online | Where-Object{ $_.Name -like 'OpenSSH*' -and $_.State -ne
       'Installed' } | ForEach-Object{ Add-WindowsCapability -Online -Name $_.Name }
```

If you are running an older version of Windows, you can install a portable version of OpenSSH made for PowerShell. It is available on the PowerShell GitHub repository. https://github.com/PowerShell/OpenSSH-Portable

Next, you will want to ensure that the `sshd` and `ssh-agent` services are set to start automatically and are running.

```
Get-Service -Name sshd,ssh-agent |
    Set-Service -StartupType Automatic
Start-Service sshd,ssh-agent
```

On the client machine, you are only making connections from, this is all that needs to be done for now. For remote servers, you will need to configure OpenSSH to allow remote connections and to use PowerShell. To do this, open the `sshd_config` file on the remote server. For Windows, this is typically `%ProgramData%\ssh` and `/etc/ssh` for Linux. In this case, it will be the Linux server.

To get started, you can enable password-based authentication by setting the line with `PasswordAuthentication` to `yes`, or leaving it commented out because its default is yes. You will also want to uncomment the key-based authentication setting, `PubkeyAuthentication`, and set it to `yes`. You will eventually disable password-based authentication, but you need to leave it on until you configure key-based authentication in the next section.

```
PasswordAuthentication yes
PubkeyAuthentication yes
```

Next, you need to add a subsystem entry to let SSH know where the PowerShell binaries are.

```
# Windows
Subsystem powershell c:/progra~1/powershell/7/pwsh.exe -sshs -NoLogo

# Linux with Snap
Subsystem powershell /snap/powershell/160/opt/powershell/pwsh -sshs -NoLogo

# Other Linux
Subsystem powershell /usr/bin/pwsh -sshs -NoLogo
```

Note that the Windows path uses the 8.3 short name for the path. There is a bug in OpenSSH for Windows that does not allow paths with spaces in them.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

5.4.2 Authenticating with PowerShell and SSH

As you just saw, there are two methods to authenticating with SSH, passwords and keys. The big difference between these two, besides security, is passwords cannot be passed to the `New-PSSession` or `Invoke-Command` cmdlets. Instead, they must be typed at the time of execution. So as far as automations go, you will want to use key-based authentication.

For those unfamiliar with SSH, key-based authentication works by using key pairs. There is a private key and a public key. The private key is maintained on the local client initiating the connection. It is the equivalent of a password, so access to the private key must be strictly controlled. The public key is copied to the remote servers you want to access. You can generate a key pair using the `ssh-keygen` command. For our example of connecting from Windows to Linux, the Windows client will have the private key, and the public key is copied to the Linux server.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

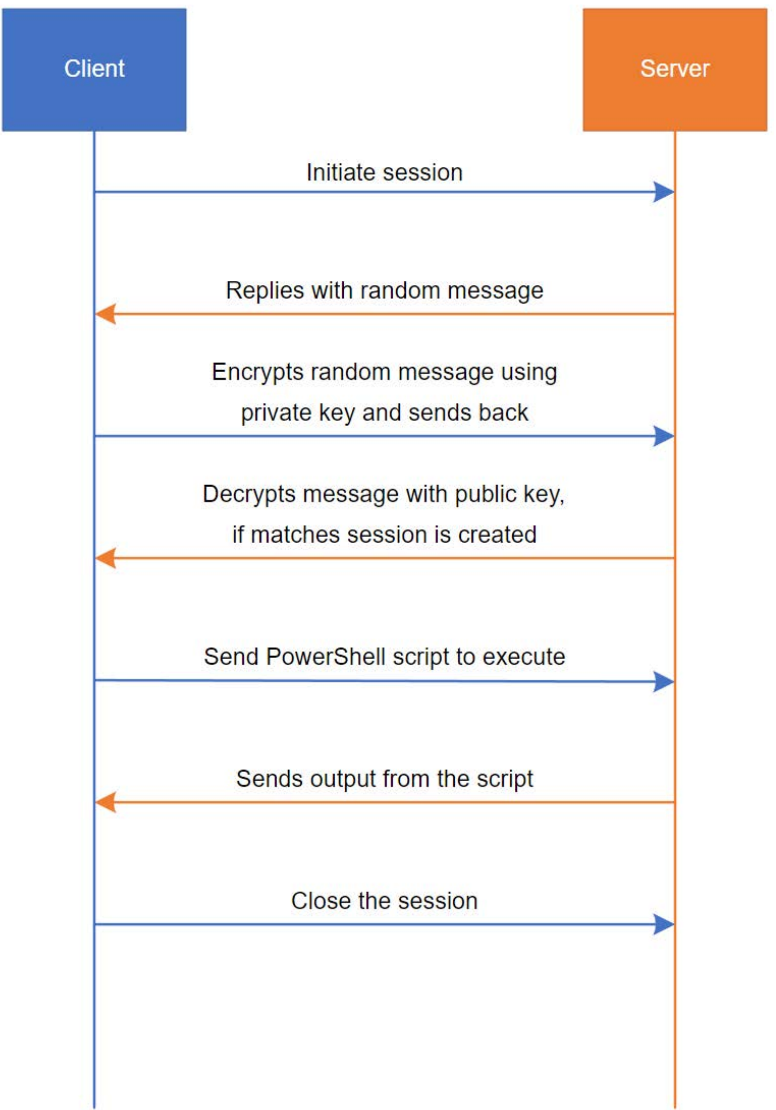


Figure 6 How SSH remote execution works with key pair authentication

After generating a key pair, you can store the private key using the ssh-agent for extra security. The agent will associate the private key with the user account on that system. You can then move the private key file to a secure storage location.

Another concept of SSH that you need to be familiar with is the known hosts. When you connect to a remote SSH server, it will provide a host key. This key is different from the authentication key pair. This key is unique to the host and used to identify it. It is helpful to prevent attacks using DNS redirects or other similar tactics. The first time you connect to a

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

remote server, it will prompt you to confirm the server's key. If you select yes, the server and its key are added to the `known_hosts` file on the local client. From then on, you will be able to connect without being prompted.

As you can tell there is a lot more that goes into setting up SSH remoting. But once you have everything configured, it will be smooth sailing. To illustrate this, we will walk through the steps below to set up a Windows client to use SSH to connect to a Linux server.

1. Generate key pair on Windows client
2. Add the private to the Windows ssh-agent
3. Enable password authentication on the Linux client
4. Copy the public key to the Linux client
5. Enable key-based authentication on the Linux client
6. Disable password authentication on the Linux client
7. Test connection from Windows to Linux

On the Windows client, open a PowerShell 7 prompt. First, you'll run the command `ssh-keygen` to generate your key pair. If you leave everything to the defaults, it will create the private key file (`id_rsa`) and the public key file (`id_rsa.pub`) in the `.shh` folder of your profile. Then you will want to import the private key to the ssh-agent, so you don't need to leave the private key file sitting around on the server.

```
ssh-keygen
ssh-add "$($env:USERPROFILE)\.ssh\id_rsa"
```

After running the ssh-add, you can move to private key file to a more secure storage location.

Now, you need to copy the public key to the remote servers. The best way to do this is by using ssh to copy it.

On the Linux server, ensure that password and key-base authentication are set to yes in the `sshd_config` file. Then run the command below from the Windows client to copy the key to the user's profile on the remote Linux server. Replace *username* with the account's name on the remote server and *hostname* with the name or IP address of the server. If this is your first time connecting, you will be prompted to add the machine to the trusted hosts and provide the password.

```
type "$($env:USERPROFILE)\.ssh\id_rsa.pub" | ssh username@hostname "mkdir -p ~/.ssh &&
       touch ~/.ssh/authorized_keys && chmod -R go= ~/.ssh && cat >>
       ~/.ssh/authorized_keys"
```

Now you can disable password authentication in the `sshd_config` file on the remote machine. To disable password-based authentication, you must uncomment and the `PasswordAuthentication` attribute to `no`. The default behavior is to accept password-based authentication, so having a hash (#) at the beginning of this line is the same as having it set to yes.

```
PasswordAuthentication no
PubkeyAuthentication yes
```


Licensed to Tausif Khan <tsfahmed2@gmail.com>

You should now be able to connect to the remote machine without being prompted. You can test this using the command below from your Windows client.

```
Invoke-Command -HostName 'remotemachine' -UserName 'user' -ScriptBlock{$psversiontable}
```

5.4.3 SSH environment considerations

Most individuals who use PowerShell regularly are used to working in Active Directory environments, where a lot of the authentication and account management is taken care of for you. However, since SSH only works on local accounts, you need to pay extra attention to your configurations.

For example, when using WSMan with a domain account, it is pretty much a given that you can authenticate to all the devices with the same username and pass combo. However, when using SSH connections, this is not always the case. When copying the public key to the remote devices, you can place it under any user profile you have access to on that device. If you use different account names during this process, it can cause you issues with your automation.

Therefore, you need to ensure that you either copy the public key to the same-named account on all servers or maintain a list of servers and the accounts associated with them. I prefer to use the same-named account because it makes automations easier and makes your environment cleaner and easier to manage.

5.4.4 Execute commands with SSH PowerShell remoting

You execute commands with SSH remoting the same way you execute them with WSMan remoting. The only difference is you need to use the `-HostName` and `-UserName` arguments instead of the `-ComputerName` and `-Credential` arguments.

Since you are using the `New-PSSession` to cmdlet to create the sessions, you do not need to change the `Invoke-Command` or any other commands in the script. You just need to update the `New-PSSession` to handle SSH connections. The only problem now is figuring out how to deal with the fact that there are different parameters for SSH and WSMan connections.

You could use the try/catch block in the script to attempt the SSH connection if the WSMan connection fails. The downside is that the `New-PSSession` cmdlet can sometimes take 20-30 seconds to return an error. If you are checking a large number of servers, this could drastically increase your execution time. To prevent this, you can add a simple port check to the script using the `Test-NetConnection` cmdlet. You can first test if a device is listening on port 5985, the default WSMan port. If that fails, you can test to see if it is listening on the SSH port of 22. Based on the results of the port test, your script will pick the appropriate connection to use.

Another issue you need to consider is that by default, in certain scenarios, SSH has prompts that rely on user interactions. The first is when the device you are connecting to is not already in the `known_hosts` file in your local profile. The second is when key-based authentication fails, and password-based authentication is enabled on the remote server. If you run into these situations during the script execution, it will hang waiting for input.

To resolve this, you can create a profile-specific `config` file on the client initiating the remote connections and configure it to fail in these situations. Then by using a try/catch, you

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

can record the reasons for the failures and address them afterward. To do this, simply create a file name `config` in the `.ssh` folder in your profile and add the following lines.

```
PasswordAuthentication no
StrictHostKeyChecking yes
```

You can also achieve this by using a one-line PowerShell command.

```
"PasswordAuthentication no\r\nStrictHostKeyChecking yes" | Out-File
       "$($env:USERPROFILE)/.ssh/config"
```

Now you do not have to worry about your automations hanging, and there is nothing else you need to change in your script to support it. So, all you need to do is add the logic for the `Test-NetConnection` and the `New-PSSession` parameters to support SSH connections.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

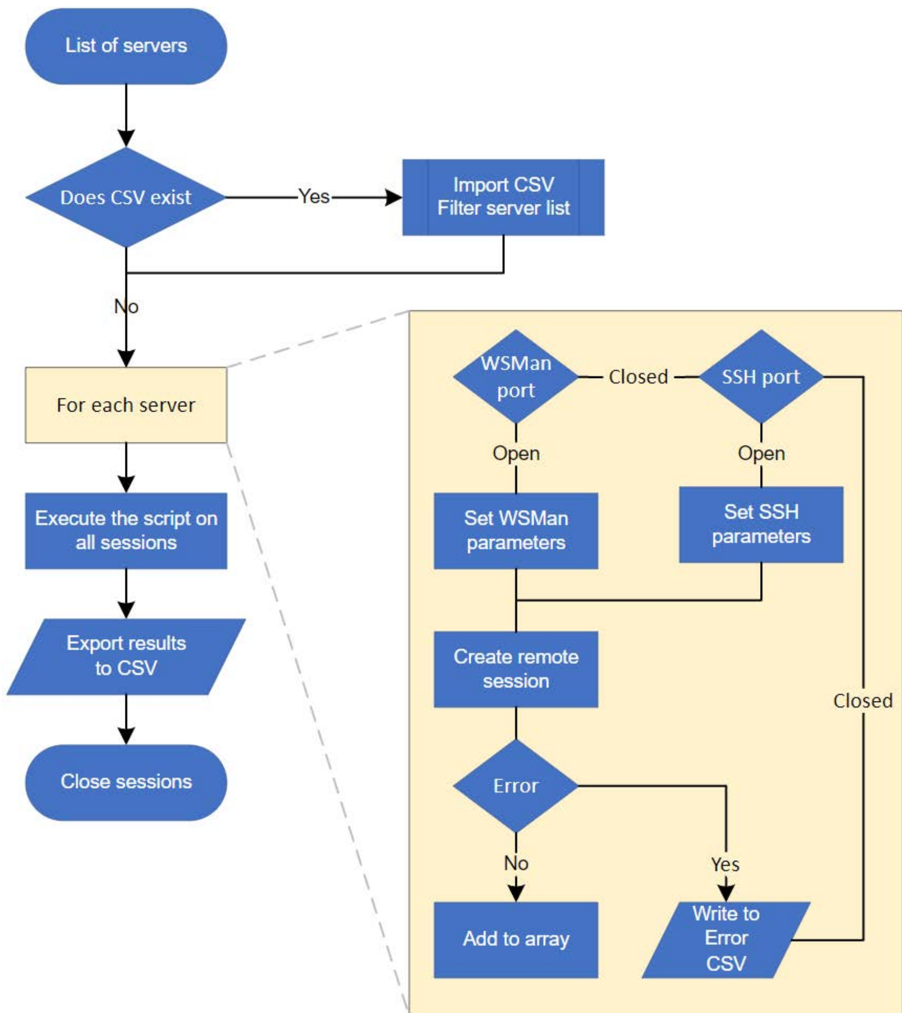


Figure 7 Control script for remote script execution with persistent connections using both WSMan and SSH protocols

Since the `Test-NetConnection` cmdlet returns true or false Boolean values, you can use it directly inside an if/else conditional statement and build the parameters based on the successful connection. Then if it is false on both, have it throw an error, so the catch block is triggered, and the error is recorded.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

Listing 3 Execute local script against remote computers using WSMan and SSH remoting

```
$SshUser = 'posh'    #A
$servers = 'Svr01', 'Svr02', 'Svr03'    #B
$CsvFile = 'P:\Scripts\VSCodeExtensions.csv'
$ScriptFile = 'P:\Scripts\Get-VSCodeExtensions.ps1'
$ConnectionErrors = "P:\Scripts\VSCodeErrors.csv"

if (Test-Path -Path $CsvFile) {
    $csvData = Import-Csv -Path $CsvFile |
    Select-Object -ExpandProperty PSComputerName -Unique
    $servers = $servers | Where-Object { $_ -notin $csvData }
}

[System.Collections.Generic.List[PSObject]] $Sessions = @()
foreach ($s in $servers) {
    $test = @{    #C
        ComputerName     = $s
        InformationLevel = 'Quiet'
        WarningAction    = 'SilentlyContinue'
    }
    try {
        $PSSession = @{    #D
            ErrorAction = 'Stop'
        }
        if (Test-NetConnection @test -Port 5985) {    #E
            $PSSession.Add('ComputerName', $s)
        }
        elseif (Test-NetConnection @test -Port 22) {    #F
            $PSSession.Add('HostName', $s)
            $PSSession.Add('UserName', $SshUser)
        }
        else {    #G
            throw "connection test failed"
        }
        $session = New-PSSession @PSSession    #H
        $Sessions.Add($session)
    }
    catch {
        [pscustomobject]@{
            ComputerName = $s
            Date         = Get-Date
            ErrorMsg     = $_
        } | Export-Csv -Path $ConnectionErrors -Append
    }
}

$Command = @{    #I
    Session  = $Sessions
    FilePath = $ScriptFile
}
$Results = Invoke-Command @Command

$Results | Export-Csv -Path $CsvFile -Append

Remove-PSSession -Session $Sessions
```

#A Added variable for the default ssh username to use
#B Remaining variables are unchanged

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

#C Set parameters for the Test-NetConnection calls
#D create hashtable for New-PSSession parameters
#E if listening on WSMan port
#F if listening on SSH port
#G if neither throw to the catch block
#H Create a remote session using the parameters set based on the results of the Test-NetConnection commands.
#I Remainder of the script is unchanged from listing 2

If you convert the variable at the beginning of this listing to parameters, it can be reused for any automation that requires you to connect to multiple remote machines and makes a great building block to keep around.

5.5 Hypervisor-based remoting

Unlike PowerShell native remoting, hypervisor-based remoting relies on an intermediary to execute PowerShell on a remote machine. However, like with native PowerShell remoting, you can use a control script to make these connections. This method uses a hypervisor to initiate the remote session. For example, Microsoft Hyper-V can use PowerShell Direct, and VMware uses the `Invoke-VMScript` cmdlet, which is part of their PowerCLI module. Even most cloud providers, include Azure and AWS, have this functionality available to their virtual machines.

The most significant advantage to hypervisor-based remoting over native PowerShell remoting is you do not have to have direct network communication with the virtual machine itself. Instead, you only need to be able to communicate with the host. From there, you can let the hypervisor integration tools take over. This can be indispensable for things like initial machine configurations or even enable native PowerShell remoting.

This can also come in handy dealing with machines in a DMZ and especially in the cloud. Another great example is the Azure Virtual Machine Run Command functionality. It allows you to run a command on an Azure virtual machine, and all you need is port 443 access to Azure. You do not need any network access to the virtual machine itself.

Figure 8 Remote script execution using Hypervisor-based remoting

As with all remote connections, hypervisor-based remoting has its own unique considerations and caveats. A big one to be mindful of is a virtual machine's name may not always be the same as the name in the guest operating system. So, you need to be aware of this when passing in the list of computers. In most cases, it will need to be the name of the virtual machine.

Another huge advantage that hypervisor-based connections have is the ability to have your control script turn on virtual machines that may be off. Then after it runs the script on

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

them, it can turn them back off. However, using this approach can present other problems. For example, a host may not be able to support turning on every virtual machine at once. Therefore, your best option would be to check each server individually, even though it will take longer to run in those situations.

In the previous example, you used a list of servers to create remote connections using either WSMan or SSH. Then it used those sessions to run the VS Code extension script on the remote servers. In this scenario, you will substitute the server list with a command to return all the virtual machines on a Hyper-V host. Then use PowerShell Direct to connect to each virtual machine.

As mentioned, many of these hypervisor-base remoting solutions have their own specific caveats, and Hyper-V PowerShell Direct is no exception. Your host and guest operating systems must all be Windows 10, Windows Server 2016, or later for this to work. Also, the control script must run from on the host machine with administrator privileges. As you can imagine, in clustered environments, this could pose a problem.

However, PowerShell Direct is supported using the same cmdlets as native PowerShell remoting. Since you will be processing each machine individually, there is no need to use the `New-PSSession` cmdlet. Therefore, your script can simply create the connection at the time of execution in the `Invoke-Command` cmdlet.

So, the script will get all the virtual machines on the host. Then for each one, it will turn on if required and wait for the operating system to respond. Then it will run the remote command, write the results to the CSV file, then turn the virtual machine off if it started it.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

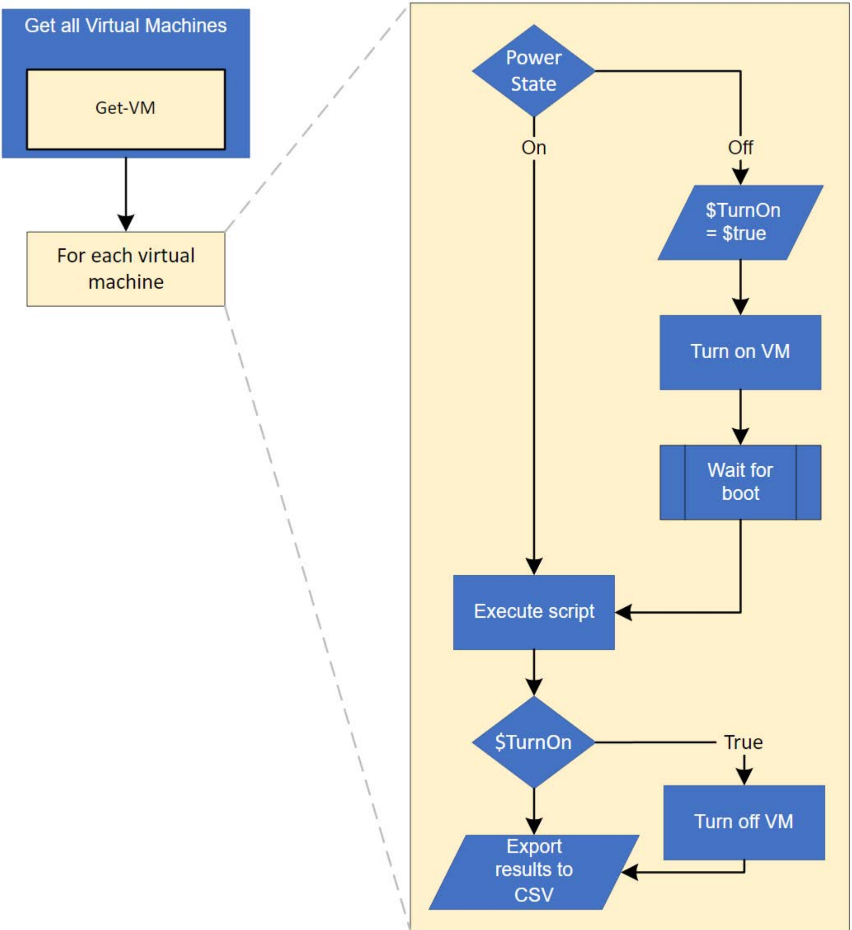


Figure 9 Control script for remote script execution using PowerShell Direct on Hyper-V

Before you put everything together, there are a few additional items to take into account. First, if you cannot authenticate to the virtual machine operating system using your current credentials, you will be prompted to provide a username and password. Just like with the SSH connections, this causes the script to hang waiting for the credentials. However, if you pass credentials to the machine and they fail, it will simply error out, which is the behavior you would want in an automated script. So, you can create the credential object using the `Get-Credential` cmdlet or using the `SecretsManagement` module. Even though `Get-Credential` requires user interaction, it is only once when the script starts, and for ease of the example, we will use it here.

The other item to consider is what happens if the virtual machine fails to turn on or the operating system does not properly boot. Addressing the issue of the virtual machine failing

Licensed to Tausif Khan <tsfahmed2@gmail.com>

to start can be handled the same way you dealt with the `New-PSSession` and use a try/catch, and have the catch use a `continue` to skip the rest of the loop.

The trickier one is dealing with the operating system not booting properly. You can determine if the operating system has started by the Heartbeat property of the virtual machine returning either `OkApplicationsHealthy` or `OkApplicationsUnknown`. So, how do you tell if a server is still booting or if the boot failed? Unfortunately, there is no perfect way. However, to prevent your automation from just sitting there waiting for a machine that may never boot, you can use a stopwatch to stop waiting after a predetermined amount of time. In this case, you can use an if statement to check if the allotted amount of time has elapsed and, if so, use a `break` command to quit the loop.

Listing 4 Connect to all Virtual Machines from a Hyper-V Host

```
$Credential = Get-Credential    #A
$CsvFile = 'P:\Scripts\VSCodeExtensions.csv'    #B
$ScriptFile = 'P:\Scripts\Get-VSCodeExtensions.ps1'    #C
$ConnectionErrors = "P:\Scripts\VSCodeErrors.csv"    #D

$servers = Get-VM    #E
foreach ($VM in $servers) {
    $TurnOff = $false
    if ($VM.State -ne 'Running') {    #F
        try {
            $VM | Start-VM -ErrorAction Stop    #G
        }
        catch {
            [pscustomobject]@{
                ComputerName = $s
                Date         = Get-Date
                ErrorMsg     = $_
            } | Export-Csv -Path $ConnectionErrors -Append
            continue    #H
        }
        $TurnOff = $true
        $timer = [system.diagnostics.stopwatch]::StartNew()
        while ($VM.Heartbeat -notmatch '^OK') {    #I
            if ($timer.Elapsed.TotalSeconds -gt 5) {
                break    #J
            }
        }
    }

    $Command = @{    #K
        VMId        = $Vm.Id
        FilePath    = $ScriptFile
        Credential  = $Credential
        ErrorAction = 'Stop'
    }
    try {
        $Results = Invoke-Command @Command    #L
        $Results | Export-Csv -Path $CsvFile -Append
    }
    catch {
        [pscustomobject]@{    #M
            ComputerName = $s
```

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
            Date         = Get-Date
            ErrorMsg     = $_
        } | Export-Csv -Path $ConnectionErrors -Append
    }

    if ($TurnOff -eq $true) {    #N
        $VM | Stop-VM
    }

}    #O
```

#A Prompt for credentials
#B Path to save results to
#C The script file from listing 1
#D Another CSV file to record connection errors
#E Get all the virtual machines on the host
#F Check if the virtual machine is running
#G Start the virtual machine
#H If the start command fails, continue to the next virtual machine
#I Wait for the heartbeat to equal a value that starts with OK, letting you know the OS has booted
#J If does not boot, break the loop and continue to the connection
#K Set the parameters using the virtual machine Id
#L Execute the script on the virtual machine
#M If execution fails, record the error
#N If the virtual machine was not running to start with, turn it back off
#O There is no disconnect needed because you did not create a persistent connection

If you use VMware, Citrix, Azure, AWS, or any other hypervisor or cloud provider, the cmdlets used will be different, but the concept remains the same.

5.6 Agent-based remoting

Like hypervisor-based remoting, agent-based remoting relies on an intermediate tool to execute this script. However, in this case, it is usually a 3rd party platform. There are numerous platforms that support this. These include Jenkins nodes, Azure Automation Hybrid Runbook Workers, HPE Operations Agents, and System Center Orchestrator Runbook Workers, to name a few.

These connections use a locally installed agent to execute the script directly on the remote device. They offer an advantage over PowerShell remoting because the agent will typically handle all the permissions and authentication.

We will not delve down into the nitty-gritty on setting these up, as each platform is unique. But we are going to discuss how you need to adapt your scripts when using these agents. These concepts can also apply to other scenarios, such as running a script via group policy or configuration management software.

The most significant difference with this form of remote execution is there is no control script. This means if your script is gathering information to return, you need to figure out how you will collect that data. Even if your script performs an action and not data collection, you will want to log its execution and results. Therefore, you will want to adjust your script to return data to one centralized location.

Depending on your environment, this location could be any number of things. Typically, in a domain environment, a file share would be a safe bet. However, when using mixed

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

environments, all servers may not have access to a single file share. In these cases, you can use an FTP site or cloud-based storage option to store the data. No matter which option you choose, the concepts you will learn here will remain the same. In all cases, you need to write the data to a centralized location while protecting against potential conflicts from multiple machines writing data simultaneously.

For example, if you decide to go the network share route, you can simply put an `Export-Csv` command at the end of the script pointing to a predetermined CSV file on a network share. Then to prevent accidentally overwriting the data from other devices, you can include the `-Append` switch. However, having multiple machines simultaneously writing to the same file can cause conflict and write errors. To prevent that from happening, your best option is to have each system write to its own file. Then on your end, you can write a script that will gather all the files at once and import them to a single object.

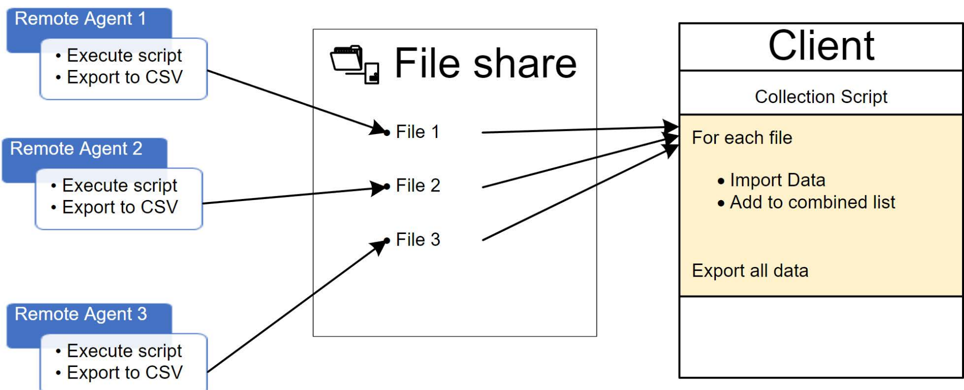


Figure 10 Example workflow for collecting data returned by remote agent executions

Now you need to consider how you will ensure that each server creates a unique file because you do not want to have two or more servers constantly overwritten the same file.

Depending on your environment, you may be able to get away with just using the system name to make your file name unique. However, in large or multiple domain environments, this may not always be the case. You can't even guarantee that using something like the device SIDs will produce unique values. Even the trick you used in chapter 3 of adding the timestamp to the file may not work because there is a chance that two computers with the same name will run the script simultaneously. It is a very small chance, but not one that would be out of the realm of possibility.

While there is no 100% foolproof way to ensure a unique value, but you can get pretty close by using a globally unique identifier, more commonly referred to as a GUID. A GUID is made up of 32 hexadecimal values split into five groups. Thus, there are 2^{128} different possible GUID combinations. This is more than the number of stars in the known universe. And the best part is you can create all the GUIDs you want by simply using the `New-Guid` cmdlet.

Licensed to Tausif Khan <tsfahmed2@gmail.com>

So, if you append the system name and a randomly generated GUID to the file name and you still end up with a duplicate name, you better run straight out and buy a lottery ticket.

Using these concepts, you can update the `Get-VSCodeExtensions.ps1` to write the results to a network share with a unique name with just a couple of extra lines added to the bottom.

Listing 5 Updated find installed Visual Studio Code extensions to output results to network share

```
$CsvPath = '\\Srv01\IT\Automations\VSCode'    #A
[System.Collections.Generic.List[PSObject]] $extensions = @()
if ($IsLinux) {
    $homePath = '/home/'
}
else {
    $homePath = "$($env:HOMEDRIVE)\Users"
}

$homeDirs = Get-ChildItem -Path $homePath -Directory

foreach ($dir in $homeDirs) {
    $vscPath = Join-Path $dir.FullName '.vscode\extensions'
    if (Test-Path -Path $vscPath) {
        $ChildItem = @{
            Path    = $vscPath
            Recurse = $true
            Filter  = '.vsixmanifest'
            Force   = $true
        }
        $manifests = Get-ChildItem @ChildItem
        foreach ($m in $manifests) {
            [xml]$vsix = Get-Content -Path $m.FullName
            $vsix.PackageManifest.Metadata.Identity |
            Select-Object -Property Id, Version, Publisher,
            @{l = 'Folder'; e = { $m.FullName } },
            @{l = 'ComputerName'; e = {[system.environment]::MachineName}},
            @{l = 'Date'; e = { Get-Date } } |
            ForEach-Object { $extensions.Add($_) }
        }
    }
}

if ($extensions.Count -eq 0) {
    $extensions.Add([pscustomobject]@{
            Id           = 'No extension found'
            Version      = $null
            Publisher    = $null
            Folder       = $null
            ComputerName = [system.environment]::MachineName
            Date         = Get-Date
        })
}
$fileName = [system.environment]::MachineName +    #B
    '-' + (New-Guid).ToString() + '.csv'
$File = Join-Path -Path $CsvPath -ChildPath $fileName    #C
$extensions | Export-Csv -Path $File -Append    #D
```

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

#A Add a variable with the path to the network share.
#B Create a unique file name by combining the machine name with a randomly generate GUID
#C Combine the file name with the path of the network share
#D Export the results to the CSV file

5.7 Setting yourself up for success with PowerShell remoting

I cannot emphasize strongly enough that you should know how to remotely connect to all systems in your environment and have them preconfigured. As you saw, there is no need to use a single remote connection type. You can certainly use a combination that makes sense for your environment. However, by having everything set up and your control script built, you can be ready for whatever situations may arise. And the concepts we covered with the VS Code extensions can apply to any script you need to run remotely.

To give a real-world example, I once had a customer call me in a panic because a bad update had been automatically pushed to their antivirus software. This bad update not only stopped a number of their business applications but had broken its own updating mechanism. The only resolution was to reinstall the application manually on 150+ servers.

They called looking for extra hands to help with all the manual reinstalls. But I informed them we had already written a control script to install an agent a few weeks before. After changing a few lines of code, we were able to reinstall the antivirus software on every server in under an hour.

The most remarkable thing about this is that we could handle it from one central location, even though they have a very disjointed network. They have a mixture of Windows, Linux, on-premises, and cloud servers. They also have to deal with remote offices that are not always in a trusted domain.

We used a combination of WSMan and SSH PowerShell remoting for all the servers in their data center. Then used the Azure virtual machine Run Command for some machines in Azure. And finally, since we had set up the servers in their remote offices as Azure Automation Hybrid Workers, we were able to update all those using the agent.

Through the use of PowerShell remoting, we saved this company many person-hours of manually connecting to and reinstalling an application. But, more importantly, we were able to get their business applications back online faster, saving them untold thousands in potential lost revenue.

5.8 Summary

- WSMan remoting works well in Windows Active Directory environments.
- For non-Active Directory environments or ones with Linux and macOS, you will need to use SSH remoting.
- Control scripts are used to execute a remote command against multiple servers and can be designed to use a mixture of remoting protocols.
- When using agent-based remoting, you will need to account for not having a control script.
- Hypervisor-based remoting works well for situations where other remoting is not an option but may not be a viable option for recurring automations.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

6 Making adaptable automations

This chapter covers

- **Using event handling to account for known errors**
- **Creating dynamic functions**
- **Using external data in your scripts**

One of the toughest challenges you will face with automation is figuring out how to make things as efficient and maintainable as possible. The best way to achieve that is by making your code as smart and adaptable as possible. As you will see, adaptable scripting can mean many different things. For example, something as simple as adding parameters to a function makes your code more adaptable. But in this chapter, we will take it to the next level by making functions that can account for potential known errors and resolve them and make a function that can create dynamic if/else statements on the fly. And at the end, you will see how you can tie all these functions together into a dynamic automation.

To demonstrate this, we will build an automation to perform some basic server setup tasks. This automation will be able to perform the following steps:

1. Install Windows Features and Roles
2. Stop and Disable unneeded services
3. Configure security baseline setting
4. Configure the Windows firewall

We will start with stopping and disabling unneeded services, which will provide a great example of using error handling in your script to do more than just report a problem or halt execution. Next, we will configure security baselines by providing the script a list of registry keys to check and update. Then we see how you can tie together all four steps listed above into a single automations using a configuration data file.

Since the scenarios in this chapter deal with changing system settings, I suggest creating a new Virtual Machine with Windows Server 2016 or newer for your testing. Also, since we

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

are dealing with a new server, all code will work in Windows PowerShell and PowerShell Core.

We will be putting all the code created into a module named `PoshAutomate-ServerConfig`. You can quickly generate the base structure you will need using the `New-ModuleTemplate` function from chapter 2.

Listing 1 Creating the PoshAutomate-ServerConfig module

```
Function New-ModuleTemplate {    #A
    [CmdletBinding()]
    [OutputType()]
    param(
        [Parameter(Mandatory = $true)]
        [string]$ModuleName,
        [Parameter(Mandatory = $true)]
        [string]$ModuleVersion,
        [Parameter(Mandatory = $true)]
        [string]$Author,
        [Parameter(Mandatory = $true)]
        [string]$PSVersion,
        [Parameter(Mandatory = $false)]
        [string[]]$Functions
    )
    $ModulePath = Join-Path .\ "$($ModuleName)\$($ModuleVersion)"
    New-Item -Path $ModulePath -ItemType Directory
    Set-Location $ModulePath
    New-Item -Path .\Public -ItemType Directory

    $ManifestParameters = @{
        ModuleVersion     = $ModuleVersion
        Author            = $Author
        Path              = ".\$($ModuleName).psd1"
        RootModule        = ".\$($ModuleName).psm1"
        PowerShellVersion = $PSVersion
    }
    New-ModuleManifest @ManifestParameters

    $File = @{
        Path     = ".\$($ModuleName).psm1"
        Encoding = 'utf8'
    }
    Out-File @File

    $Functions | ForEach-Object {
        Out-File -Path ".\Public\$($_).ps1" -Encoding utf8
    }
}

$module = @{    #B
    ModuleName    = 'PoshAutomate-ServerConfig'    #C
    ModuleVersion = "1.0.0.0"    #D
    Author        = "YourNameHere"    #E
    PSVersion     = '5.1'    #F
    Functions     = 'Disable-WindowsService',    #G
        'Install-RequiredFeatures', 'Set-FirewallDefaults',
        'Set-SecurityBaseline', 'Set-ServerConfig',
        'Test-SecurityBaseline'
```

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
}
New-ModuleTemplate @module    #H
```

#A This is the same function as Listing 5 in chapter 2
#B Set the parameters to pass to the function
#C The name of your module
#D The version of your module
#E Your name
#F The minimum PowerShell version this module supports
#G The functions to create blank files for in the Public folder
#H Execute the function to create the new module

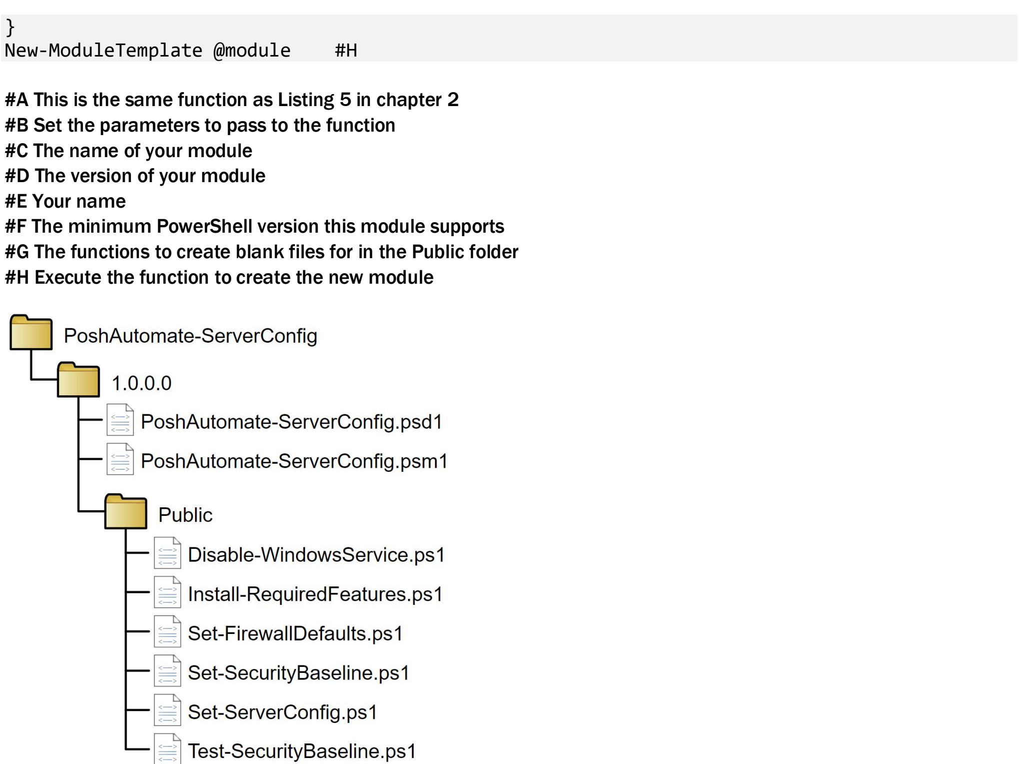


Figure 1 PoshAutomate-ServerConfig module file structure

Once you get the file structure created, you can add the following code to the `PoshAutomate-ServerConfig.psm1` to automatically import the functions from the Public folder.

Listing 2 PoshAutomate-ServerConfig.psm1

```
$Path = Join-Path $PSScriptRoot 'Public'
$Functions = Get-ChildItem -Path $Path -Filter '*.ps1'    #A

Foreach ($import in $Functions) {    #B
    Try {
        Write-Verbose "dot-sourcing file '$($import.fullname)'"
        . $import.fullname    #C
    }
    Catch {
        Write-Error -Message "Failed to import function $($import.name)"
    }
}
```

#A Get all the ps1 files in the Public folder
#B Loop through each ps1 file
#C Execute each ps1 file to load the function into memory

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

6.1 Event Handling

To see the true potential and cost savings of automation, you must be able to build event handling into your scripts. Any time you have to go in after an automation runs and fix something, make a note of it. If you see something that happens on a regular basis, it is a good chance you need to add some event handling to your script.

Take, for example, the first scenario in our automation of stopping and disabling services. Anyone who has worked with Windows knows that you can do this with a few lines of code.

```
Get-Service -Name Spooler |
    Set-Service -StartupType Disabled -PassThru |
    Stop-Service -PassThru
```

Now think about what happens if the service is not found. For example, if you pass the name of a service that does not exist to the `Get-Service` cmdlet, it will return an error. But, if the service does not exist, then there is nothing to stop or disable. So, is it really an error? I would say it is not an error but is something that you should record.

To prevent your script from throwing an error, you can choose to suppress errors on that command using the parameter `-ErrorAction SilentlyContinue`. However, when you do this, there is no way for you to know for sure that the service does not exist. You are just assuming that the reason for the error was that the service does not exist. But when someone suppresses the error message, there is no way to know for sure. For example, it could also throw an error if you do not have the appropriate permissions. The only way to know for sure is to capture and evaluate the error message using a `try/catch` block.

6.1.1 Using try/catch blocks for event handling

By default, a PowerShell script will stop executing when a command throws a terminating error except when that error happens inside of a `try` block. When there is a terminating error inside a try block, the script will skip the remainder of the code inside the `try` block and go to the `catch` block. If there are no errors, the script will skip the code in the `catch` block. You can also add a `finally` block that will execute last in all cases. So, let's see how we can use this with our services function.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

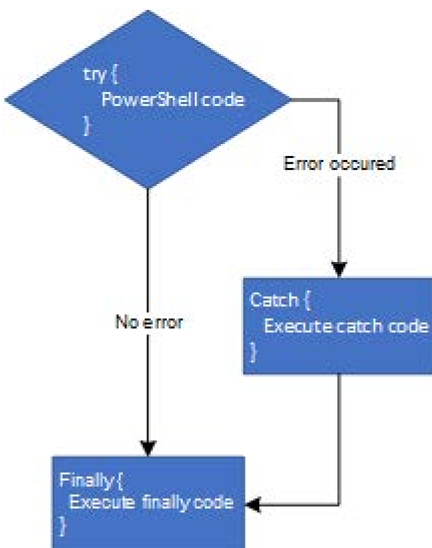


Figure 2 Try/Catch/Finally block flow

If you open a PowerShell command and enter `Get-Service -Name xyz`, you will see an error stating it cannot find the service. However, if you run that command again but wrapped in a `try/catch`, you will still see the same error. That is because this particular error is not a terminating error. Therefore the catch `block` is not triggered. So, to ensure the `catch` block is triggered, you can add `-ErrorAction Stop` to the end of the command to turn all error messages into terminating errors, ensuring that the `catch` block will be triggered.

```
try{
   Get-Service -Name xyz -ErrorAction Stop
}
catch{
   $_
}
```

When you run the command above into PowerShell, you will still see the error message but notice that the output is now white instead of red. This is because of the `$_` in the `catch` block. When a `catch` block is triggered, the error that caused it is automatically saved to the variable `$_`. This is what we can use to test that the error we received was the expected one.

Inside the `catch`, you can use an `if/else` conditional statement to check the error message. If it does not match the expected error, it will call the `Write-Error` cmdlet to let PowerShell error handling report the error but not terminate the execution. You should use non-terminating errors in situations where the subsequent steps can still process even though an error has occurred.

For example, you do not need to stop processing the other services in this scenario due to one of them failing. The other services can still be stopped and disabled. Then you can go

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

back and address the failures. However, if an error on a particular step would cause subsequent failures, then you would want to terminate the execution. For instance, if you were setting up a web server and IIS failed to install. There would be no need to continue with the steps to configure IIS because it is not there. In these situations, you can simply replace the `Write-Error` command with `throw`.

```
$Name = 'xyz'
try{
    $Service = Get-Service -Name $Name -ErrorAction Stop
}
catch{
    if($_.FullyQualifiedErrorId -ne
       'NoServiceFoundForGivenName,Microsoft.PowerShell.Commands.GetServiceCommand'){
        Write-Error $_
    }
}
```

In the next step, we want to set the service startup type to disabled. You do not need to stop a service before you disable it. And since we want to ensure they are disabled, it makes sense to put that command before the stop command. This way, if the function runs into an unexpected error in the stop process, we will have guaranteed that the service is at least disabled.

In this situation, we can put the `Set-Service` cmdlet directly under the `Get-Service` cmdlet because the `Get-Service` cmdlet has the error action set to stop. Thus, if there is an error in the `Get-Service` command, it will jump to the `catch` block, skipping the `Set-Service` command.

6.1.2 Creating custom event handles

Now that we have disabled the service, it is time to stop it. I am almost positive that everyone reading this book has run into a situation where you tell a service to stop running, and it just hangs. If you've ever experienced this through PowerShell, then you have most likely seen your console fill with warning after warning stating, "Waiting for service 'xyz' to stop..". And PowerShell will continue to repeat that message until the service stops or you manually kill the execution. Neither of which is an ideal situation in an automation scenario. So, let's take a look at how we can avoid this through some parallel processing.

Most cmdlets that act upon an outside resource and wait for a particular state will have an option to bypass that wait. In this scenario, the `Stop-Service` cmdlet has a `-NoWait` switch. This switch tells the cmdlet to send the stop command but do not wait for it to stop. Doing this will allow you to send multiple stop commands one after another without waiting for one to finish stopping. It will also allow you to create your own event handling to kill the process after a predetermined amount of time. So, we need to make the functionality to do the following:

1. Send the stop command to multiple services without waiting.
2. Check the status of the services to ensure they have all stopped.
3. If any have not stopped after 60 seconds, attempt to kill the process.
4. If any have not stopped after 90 seconds, notify that a reboot is required.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

Unlike with the `Get-Service` cmdlet, we do not care if the `Stop-Service` cmdlet throws an error. This is because regardless of what happens on the stop, the service has already been disabled. So, even if there is an error or it does not stop in the time we allotted, a reboot will be requested, which will ensure the service does not come back up. Therefore, there is no problem adding the `-ErrorAction SilentlyContinue` argument to the command in this situation.

Using jobs to bypass waits

Some cmdlets that you may want to run in parallel do not have a no-wait option. For example, downloading multiple files using the `Invoke-WebRequest` cmdlet. In these situations, you can use PowerShell jobs to run the command as background processes. You can see this in the snippet below, where two files are downloaded simultaneously as jobs. Then the `Get-Job` followed by the `Wait-Job` will pause the script until both jobs are complete.

```
Start-Job -ScriptBlock {
    Invoke-WebRequest -Uri $UrlA -OutFile $FileA
}
Start-Job -ScriptBlock {
    Invoke-WebRequest -Uri $UrlB -OutFile $FileB
}
Get-Job | Wait-Job
```

Once the jobs are complete, then you can continue with your script. You can also get the return information from the jobs by using the `Receive-Job` cmdlet.

Since we will be checking multiple services for multiple conditions, it is good to create a custom PowerShell object to keep track of the status and startup type for every service. Then when you can create a while loop to check that the services have stopped, you not do not check ones you know have stopped or were not found.

The while loop will need to run as long as services are running but should also contain a timer to terminate after a set amount of time, even if all the services do not stop. You will also want to add the ability to perform a hard kill of the running process if it does not stop on its own. You can do this but using the `Get-CimInstance` cmdlet to get the process ID of the service, then using the `Stop-Process` cmdlet to force it to stop. Since you do not want to run the `Stop-Process` repeatedly, you can add a property to the object to record that there was an attempt to stop it. Therefore, the custom PowerShell object will need the following properties.

- Service = Service Name
- Status = The status of the service
- Startup = The startup type of the service
- HardKill = A Boolean value set to true after the `Stop-Process` command

Once you put everything together, the process should look like figure 2 below.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

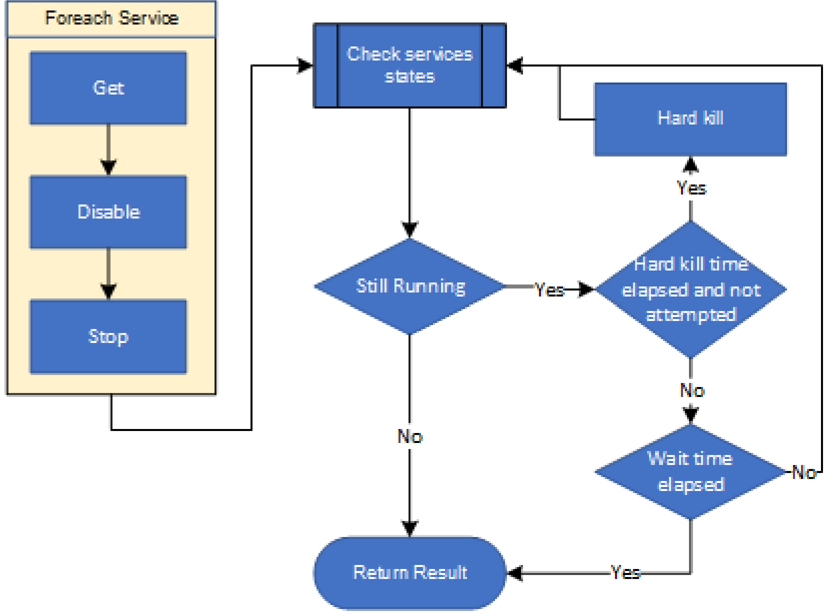


Figure 3 Service disable and stop function flow

Listing 3 Disable-WindowsService

```
Function Disable-WindowsService {
    [CmdletBinding()]
    [OutputType([object])]
    param(
        [Parameter(Mandatory = $true)]
        [string[]]$Services,
        [Parameter(Mandatory = $true)]
        [int]$HardKillSeconds,
        [Parameter(Mandatory = $true)]
        [int]$SecondsToWait
    )

    [System.Collections.Generic.List[PSObject]] $ServiceStatus = @()
    foreach ($Name in $Services) {
        $ServiceStatus.Add([pscustomobject]@{    #A
                Service  = $Name
                HardKill = $false
                Status   = $null
                Startup  = $null
            })
        try {
            $Get = @{    #B
                Name        = $Name
                ErrorAction = 'Stop'
            }
            $Service = Get-Service @Get
            $Set = @{
```


Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
                InputObject = $Service
                StartupType = 'Disabled'
            }
            Set-Service @Set
            $Stop = @{
                InputObject = $Service
                Force       = $true
                NoWait      = $true
                ErrorAction = 'SilentlyContinue'
            }
            Stop-Service @Stop
            Get-Service -Name $Name | ForEach-Object {
                $ServiceStatus[-1].Status = $_.Status.ToString()
                $ServiceStatus[-1].Startup = $_.StartType.ToString()
            }
        }
        catch {
            $msg = 'NoServiceFoundForGivenName,Microsoft.PowerShell' +
                '.Commands.GetServiceCommand'
            if ($_.FullyQualifiedErrorId -eq $msg) {
                $ServiceStatus[-1].Status = 'Stopped'    #C
            }
            else {
                Write-Error $_
            }
        }
    }

    $timer = [system.diagnostics.stopwatch]::StartNew()
    do {    #D
        $ServiceStatus | Where-Object { $_.Status -ne 'Stopped' } |
        ForEach-Object {
            $_.Status = (Get-Service $_.Service).Status.ToString()

            if ($_.HardKill -eq $false -and    #E
                $timer.Elapsed.TotalSeconds -gt $HardKillSeconds) {
                Write-Verbose "Attempting hard kill on $($_.Service)"
                $query = "SELECT * from Win32_Service WHERE name = '{0}'"
                $query = $query -f $_.Service
                $svcProcess = Get-CimInstance -Query $query
                $Process = @{
                    Id          = $svcProcess.ProcessId
                    Force       = $true
                    ErrorAction = 'SilentlyContinue'
                }
                Stop-Process @Process
                $_.HardKill = $true
            }
        }
        $Running = $ServiceStatus | Where-Object { $_.Status -ne 'Stopped' }
    } while ( $Running -and $timer.Elapsed.TotalSeconds -lt $SecondsToWait )
    $ServiceStatus |    #F
        Where-Object { $_.Status -ne 'Stopped' } |
        ForEach-Object { $_.Status = 'Reboot Required' }

    $ServiceStatus    #G
}
```

#A Create a custom PowerShell object to track the status of each service

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

#B Attempt to find the service, then disable and stop it
#C If the service doesn't exist, then there is nothing to stop, so consider that a success
#D Monitor the stopping of each service
#E If any services have not stopped in the predetermined amount of time, kill the process.
#F set reboot required if any services did not stop
#G return results

Like with many things in this book, event handling could be a book or at least several chapters on its own. This section was intended to give you an overview of some different ways that you can use it in your automations. There are many other ways to achieve event handling, some of which we will cover in subsequent chapters. I also encourage you to explore more resources on it because good event handling can really make your automations shine.

The next step in the automation will be checking and setting security baseline registry values. This will present a new concept of adaptable automation, which uses configuration data to control your script's execution.

6.2 Building data-driven functions

The Don't Repeat Yourself (DRY) principle is key to creating efficient and manageable automations. Even if you are not familiar with the term, you should be familiar with the concept of not repeating code over and over again in your scripts. This is the fundamental concept behind functions, modules, and the building blocks we talked about in the first half of this book. However, as you will see here, you can extend this concept beyond those and use external data to drive your automation.

The first and most crucial step in building a data-driven function is figuring out your data structure. Once you have your data structure figured out, you need to write the code to handle it and figure out how and where to store it. To demonstrate how to do this, we will build the step in the automation to configure the security baseline setting for your server. This will be done by checking and setting different registry keys.

You can easily return registry keys with the `Get-ItemProperty` cmdlet and change or add them with the `New-ItemProperty` cmdlet. If you build a couple of functions that can do this, then you only need one or two lines per registry key to check. But if you have ever looked at the list of registry keys required for security hardening, you will see this could quickly grow into a 500+ line script. In fact, if you do a quick search on GitHub or the PowerShell Gallery for security hardening scripts, you will find dozens of scripts over 1,000 lines long. Now imagine you have to maintain different copies for different operating system versions and different server roles. It would become a nightmare.

To prevent this from happening to you, we will create a function that can use external serialized data to provide the registry keys and values to your script. This data can be stored in external files that are both human-readable, easy to update, and easy to use in PowerShell.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

6.2.1 Determining your data structure

Checking and setting hundreds of registry keys may sound like an extremely tedious task that can have all sorts of different requirements. However, like with many things, the best approach is to break it up into different scenarios.

For example, I reviewed the Windows security baseline recommendations for an Azure virtual machine and found 135 registry settings. Out of those 135 different registry settings, there are only four different types of conditions the script needs to meet. Therefore, I can narrow my scope down to concentrate on those four. Because I know if the script can handle those, it can handle all 135.

I have selected an entry from each of these four to test with and listed them in table 1.

Table 1 Registry key values to check.

Key Path	Key Name	Expected Value
LanManServer\Parameters\	EnableSecuritySignature	= 1
EventLog\Security\	MaxSize	>= 32768
LanManServer\Parameters\	AutoDisconnect	Between 1-15
LanManServer\Parameters\	EnableForcedLogoff	= 1 or does not exist
* All key paths start in HKLM:\SYSTEM\CurrentControlSet\Services\		

Based on this table, we know our function will have to check if the value of a key is equal to a number, greater than or equal to a number, between two numbers, or does not exist. Knowing this, we can start to build the data structure.

Table 1 shows that you will need the Key Path, the Key Name, the expected value, and how to evaluate that value. You can convert those evaluations to comparison operators in PowerShell.

The first two conditions you need to test for are pretty straightforward. Equal is `-eq` and greater than or equal to is `-ge`. The third one makes things a little more complicated because it checks between two numbers. But this can be achieved by creating an array then using the `-in` operator. So, just considering these, your data structure might look something like figure 3 below.

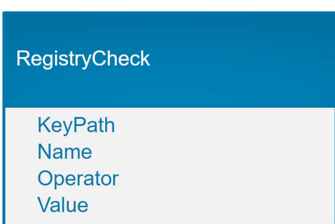


Figure 4 Initial data structure for the Registry Check

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

However, things get a little more complicated with the fourth one because there are two conditions it can be, and one of those conditions is it does not exist. However, when you think about it, "does not exist" is the equivalent of saying, "is equal to null." So now the only tricky part is handling the two different conditions. So, you can combine Value and Operator and make it an array named Test. Then you can have your script evaluate as many as you need. As long as one of them evaluates to true, the check will pass.

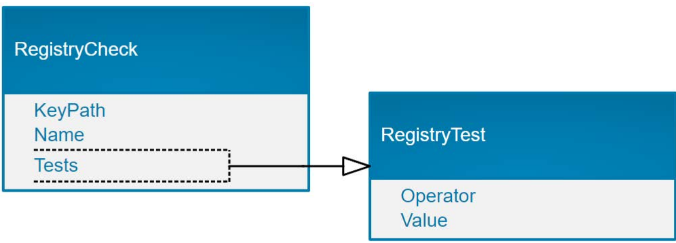


Figure 5 Updated data structure for the Registry Check to include the test operator and value as a nested array

If you were to build this out as a hashtable in PowerShell, it would be something like the snippet below.

```
@{
    KeyPath = 'HKLM:\SYSTEM\Path\Example'
    Name    = 'SecurityKey'
    Tests   = @(
        @{operator = 'eq'; value = '1' }
        @{operator = 'eq'; value = $null }
    )
}
```

6.2.2 Storing your data

Once you have your basic structure, it is time to think about how to store that data. There are numerous data-serialization formats out there, and many have native support in PowerShell. These include XML, CSV, JSON, and PowerShell Data Files. The format you choose for your automations is dependent on your needs, but my recommendation is to use JSON unless you have a specific reason not to. JSON consists of key-value pairs and supports strings, numbers, dates, Boolean, arrays, and nested objects. It is versatile and human-readable. You can also convert any JSON string to a PowerShell object using the `ConvertFrom-Json` cmdlet and back into JSON using `ConvertTo-Json` cmdlet.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
@{
    KeyPath = 'HKLM:\SYSTEM\RegPath\'
    Name    = 'EnableForcedLogoff'
    Tests   = @(
        @{
            operator = 'eq'
            value    = '1'
        }
        @{
            operator = 'eq'
            value    = $null
        }
    )
}
```

```
{
    "KeyPath": "HKLM:\\SYSTEM\\RegPath\\",
    "Name": "EnableForcedLogoff",
    "Tests": [
        {
            "operator": "eq",
            "value": "1"
        },
        {
            "operator": "eq",
            "value": null
        }
    ]
}
```

Figure 6 Side by Side comparison of PowerShell hashtable and the hashtable converted to JSON

By taking the data structure we just determined, you can build a PowerShell object with your checks, convert it to a JSON string, then export the JSON to a file.

Listing 4 Creating JSON

```
[System.Collections.Generic.List[PSObject]] $JsonBuilder = @()
$JsonBuilder.Add(@{    #A
    KeyPath =
    'HKLM:\SYSTEM\CurrentControlSet\Services\LanManServer\Parameters'
    Name    = 'EnableSecuritySignature'
    Tests   = @(
        @{operator = 'eq'; value = '1' }
    )
})
$JsonBuilder.Add(@{
    KeyPath =
    'HKLM:\SYSTEM\CurrentControlSet\Services\EventLog\Security'
    Name    = 'MaxSize'
    Tests   = @(
        @{operator = 'ge'; value = '32768' }
    )
})
$JsonBuilder.Add(@{
    KeyPath =
    'HKLM:\SYSTEM\CurrentControlSet\Services\LanManServer\Parameters'
    Name    = 'AutoDisconnect'
    Tests   = @(
        @{operator = 'in'; value = '1..15' }
    )
})
$JsonBuilder.Add(@{
    KeyPath =
    'HKLM:\SYSTEM\CurrentControlSet\Services\LanManServer\Parameters'
    Name    = 'EnableForcedLogoff'
    Tests   = @(
        @{operator = 'eq'; value = '1' }
```


Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
        @{operator = 'eq'; value = '$null' }
    )
})

$JsonBuilder |    #B
    ConvertTo-Json -Depth 3 |
    Out-File .\RegistryChecks.json -Encoding UTF8
```

#A add an entry for each registry key to check
#B converts the PowerShell object to JSON and export it to a file

As you can see in listing 4, the `ConvertTo-Json` only creates a JSON string. Similarly, the `ConvertFrom-Json` cmdlet only accepts strings. Since neither of these cmdlets can read or write to a file or any other external source, you need to use them in conjunction with other cmdlets.

The fact that the JSON cmdlets accept strings is actually a huge advantage. It means you can get your string from anywhere and convert it from JSON. For our purposes, we are going to read and write to the local file system. For this, we are going to use the `Out-File` cmdlet to write to a file and the `Get-Content` cmdlet to read from it. But in other situations, you could receive JSON from a web request, a SQL database, or even passed in as parameters. Anywhere you can get a string value from, you can use JSON.

That is not to say that JSON is the end-all-be-all for automations and PowerShell. Other formats also have their pros and cons.

XML is a tried and true format that has been around for over 20 years and is used by many applications. It is exceptionally versatile, like JSON, but has the advantage of schemas to aid in data validation. When converting from JSON, PowerShell just makes its best guess on the data type based on how it is structured in the string. This gives XML an advantage when it needs to be transferred between different applications. However, JSON is much easier to read and define. For example, in the figure below, I took the same PowerShell object for a single registry check and exported it to JSON (left) and XML (right).

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
{
  "KeyPath": "HKLM:\\SYSTEM\\CurrentControlSet\\Services\\LanManServ
  "Name": "EnableForcedLogoff",
  "Tests": [
    {
      "operator": "eq",
      "value": "1"
    },
    {
      "operator": "eq",
      "value": null
    }
  ]
}

```

```
<Objs Version="1.1.0.1" xmlns="http://schemas.microsoft.com/powershell/200
  <Obj RefId="0">
    <TN RefId="0">
      <T>System.Collections.Specialized.OrderedDictionary</T>
      <T>System.Object</T>
    </TN>
    <DCT>
      <En>
        <S N="Key">KeyPath</S>
        <S N="Value">HKLM:\SYSTEM\CurrentControlSet\Services\LanManServer\
      </En>
      <En>
        <S N="Key">Name</S>
        <S N="Value">EnableForcedLogoff</S>
      </En>
      <En>
        <S N="Key">Tests</S>
        <Obj N="Value" RefId="1">
          <TN RefId="1">
            <T>System.Object[]</T>
            <T>System.Array</T>
            <T>System.Object</T>
          </TN>
          <LST>
            <Obj RefId="2">
              <TN RefId="2">
                <T>System.Collections.Hashtable</T>
                <T>System.Object</T>
              </TN>
              <DCT>
                <En>
                  <S N="Key">operator</S>
                  <S N="Value">eq</S>
                </En>
                <En>
                  <S N="Key">value</S>
                  <S N="Value">1</S>
                </En>
              </DCT>
            </Obj>
            <Obj RefId="3">
              <TNRef RefId="2" />
              <DCT>
                <En>
                  <S N="Key">operator</S>
                  <S N="Value">eq</S>
                </En>
                <En>
                  <S N="Key">value</S>
                  <Nil N="Value" />
                </En>
              </DCT>
            </Obj>
          </LST>
        </Obj>
      </En>
    </DCT>
  </Obj>
</Objs>
```

Figure 7 The same PowerShell object converted to JSON and XML

CSV is excellent when you need to share the information with someone else. Especially non-technical people since you can use Excel to read and edit it. However, CSVs are flat files, so you can't have nest objects or arrays. Plus, PowerShell natively treats every item in a CSV as a string.

A PowerShell Data File (PSD1) contains key-value pairs very similar to JSON, except PowerShell treats them as hashtables instead of PowerShell objects. You are most likely familiar with these as module manifest files, but they can also be used to store data that you import into your script. They look very similar to JSON and support many of the same data types. However, one considerable disadvantage to PSD1 files is they require a physical file, whereas JSON is converted from a string variable. Also, as the name implies, PowerShell Data Files are unique to PowerShell and cannot be used in other applications. Therefore, PSD1 files are best left for use inside modules with relatively static data.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

JSON Validation

Unless you are familiar enough with JSON to know which characters to escape and which are not supported, it is always a good idea to use PowerShell or some other JSON editor to update your JSON files. Also, if you ever have problems importing your JSON, you can use the website jsonlint.com to evaluate it and let you know precisely where the issues are. There are also numerous JSON validation extensions for Visual Studio Code.

6.2.3 Updating your data structure

Now that you have your test data defined and exported to JSON, it is time to consider the data you need to resolve the failed checks. Since the tests are not always a one-for-one type comparison (i.e. EnableForcedLogoff can either be one or null), you cannot use the test value as the value to set. Therefore, you will need to add a new property to the JSON. This new property will tell the script what value to set when a check fails. You will also want to note the type of value it should be (e.g., DWORD, String, Binary, etc.).

To add these new fields to your JSON, you have two options. You can open the JSON in Visual Studio Code and manually copy and paste the fields into every entry in the file. All the while hoping you don't miss one or accidentally enter an illegal character. Or use the preferred method of having PowerShell to help you update all the entries.

You can quickly and easily add a new property to a PowerShell object using the `Select-Object` cmdlet. This cmdlet lets you specify the properties you want to return from a PowerShell object and allows you to create custom properties on the fly by passing a hashtable as a property. The hashtable only needs two key pairs. `Label` for the name of the property and `Expression` for the expression to evaluate. You will often see these written simply as `l` and `e` for short in scripts.

So, let's add some new properties to the JSON file. The first one we want is the data type named `Type`. We know most of them are of the type DWORD, so we can just hardcode DWORD into the property. You can then go and manually change any you need to. The second property we'll name `Data` and default its value to the first value in our test array. Again, you can manually update afterward, but this gives us a good head start instead of just writing blank values.

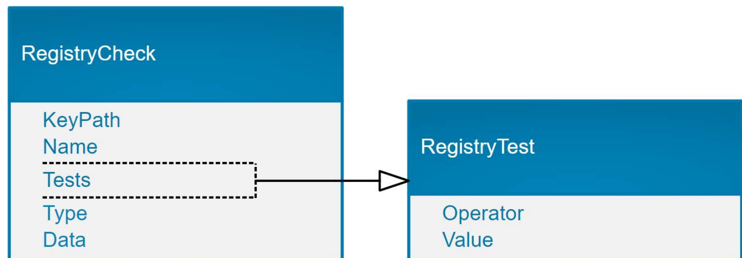


Figure 8 Final data structure for the Registry Check JSON with added values to set if the check fails

You can use the snippet below to add these fields to your JSON and export it as a new file.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

Listing 5 Add new data to JSON using PowerShell

```
$checks = Get-Content .\RegistryChecks.json -Raw |    #A
    ConvertFrom-Json

$updated = $checks |    #B
    Select-Object -Property *, @{l='Type';e={'DWORD'}},
        @{l='Data';e={$_.Tests[0].Value}}

ConvertTo-Json -InputObject $updated -Depth 3 |    #C
    Out-File -FilePath .\RegistryChecksAndResolves.json -Encoding utf8
```

#A Import the JSON file and convert it to a PowerShell object
#B Use the Select-Object to add new properties to the object
#C Convert the updated object with the new properties back to JSON and export

Now that we have the data structure, it is time to look at how to import it into the automation. To save you the trouble of filling out the JSON file yourself, I have included a copy in the Helper Scripts for this chapter.

6.2.4 Creating classes

One of the great things about using serialized data is it can be dynamic. For example, when you query a REST API that returns JSON, you do not need to know the data structure beforehand. PowerShell will just automatically convert it to a PowerShell object. However, this can also be a bad thing when a script is expecting a specifically formatted object. In these cases, your best option is to create a custom class, to define the properties your object needs.

As we see in our example here, we know what properties need to exist for the registry check. So, to prevent unexpected data from causing problems in your function, you will want to create a class. In fact, we will need to create two classes because our JSON has a nested object with the Tests property. Classes can exist inside their own files, or you can declare them in the psm1 for the module. For our purposes, we will just create these in the `PoshAutomate-ServerConfig.psm1` file.

Starting with the class for the Tests objects, we will need two string properties; the operator and value. When you build a class, you can create custom constructors inside of it. These allow you to convert objects, perform data validations, or perform other types of data manipulation to ensure your object has the correct values. Since the JSON import creates a generic object, we will create a constructor to accept a single object. Then have it assign the appropriate properties. We will also include a constructor with no parameters that will allow you to create an empty version of this call.

To define a class, you need to include the class keyword followed by the name of the class and curly brackets. Inside the brackets, you will define the properties of the class, followed by the constructors. The constructors must have the same name as the class.

Licensed to Tausif Khan <tsfahmed2@gmail.com>

Listing 6 Registry Test Class

```
class RegistryTest {
    [string]$operator
    [string]$Value
    RegistryTest(){    #A
    }
    RegistryTest(    #B
        [object]$object
    ){
        $this.operator = $object.operator
        $this.Value = $object.Value
    }
}
```

#A Method to create a blank instance of this class
#B Method to create an instance of this class populated with data from a generic PowerShell object

Now we can create the class for the main registry check object. It will be similar to the other class, except for the `Tests` property. In this case, we want to make it an array by adding square brackets inside the data type declaration. Then in our constructor, we will add a `foreach` to loop through each test and add it to the array.

Finally, we can add two additional properties to help with debugging and confirming your script is getting the correct data. First, a blank object of `SetValue` so you can record the value that the script is checking. It is set to object because you do not know what type of data may be returned from the different registry keys. And a Boolean value of `Success` set to `false`, you will have the script flip this to `true` if the check passes. These do not need to be in the JSON because their values are not predefined.

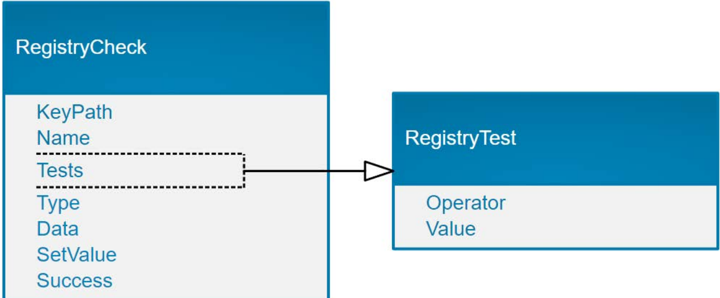


Figure 9 Final data structure for the Registry Check class with added values recording results

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

Listing 7 Registry Check Class

```
class RegistryCheck {
    [string]$KeyPath
    [string]$Name
    [string]$Type
    [string]$Data
    [string]$SetValue
    [Boolean]$Success
    [RegistryTest[]]$Tests
    RegistryCheck(){    #A
        $this.Tests += [RegistryTest]::new()
        $this.Success = $false
    }
    RegistryCheck(    #B
        [object]$object
    ){
        $this.KeyPath = $object.KeyPath
        $this.Name = $object.Name
        $this.Type = $object.Type
        $this.Data = $object.Data
        $this.Success = $false
        $this.SetValue = $object.SetValue

        $object.Tests | Foreach-Object {
            $this.Tests += [RegistryTest]::new($_)
        }
    }
}
```

#A Method to create a blank instance of this class
#B Method to create an instance of this class populated with data from a generic PowerShell object

6.2.5 Building the function

Since we only have four conditions to check for, it may be tempting to create an `if/else` conditional statement to test against each. However, there is the possibility you could have different conditions in the future. In that case, you could build an `if/else` that will handle all 14 different comparison operators in PowerShell, but that would make a huge nested mess of `if/else` statements, which would be a nightmare to troubleshoot. Plus, it goes against the DRY principle because you are repeating the same thing. So, instead, we will look at how we can build a function to accept dynamic conditions. However, first, we need to get the value to check.

As mentioned previously, the `Get-ItemProperty` cmdlet can return the value of a registry key. However, if the key or the key path do not exist, it will throw an error. Since a key not existing can be an expected result, you do not want this. You also will not want to use a `try/catch` here because there could be other reasons a value is not returned. For example, if access is denied, you could end up with a false positive using a `try/catch`. Instead, you can use the `Test-Path` cmdlet to test that the path exists. Then if it does, use the `Get-Item` cmdlet to return all the sub-keys and confirm the one you want is present. If both of these conditions are met, you can be ensured the key exists, and you can get the value from it.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

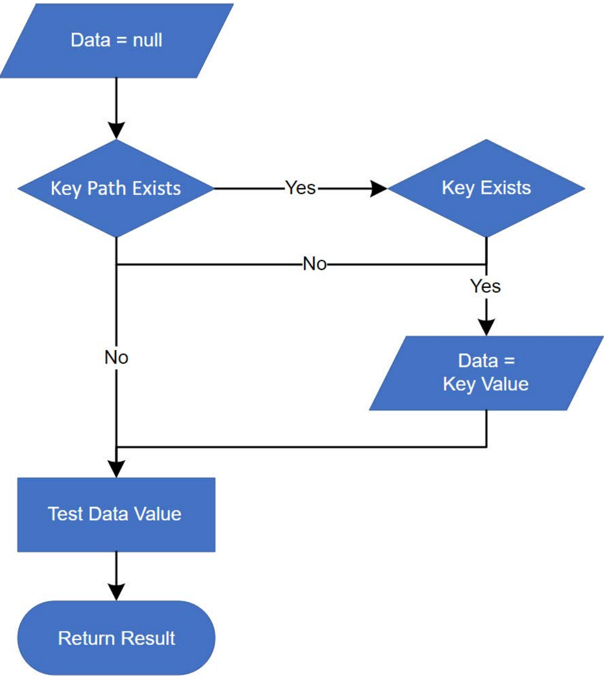


Figure 10 Confirming that the path to the registry key exists and that the key itself exists before attempting to get the value from it

Now that you have the value, it is time to build the logic to confirm it matches the expected value. The `Invoke-Express` cmdlet allows you to take a string and execute it as PowerShell code. Let's take, for example, our first registry key that should equal 1. A simple test for this may look something like this:

```
if($Data -eq 1){
    $true
}
```

You can quickly turn this into a string to swap the operator and the value on using some simple string formatting and pass it to the `Invoke-Express` cmdlet.

```
'if($Data -{0} {1}){{$true}}' -f 'eq', 1
```

The best thing about this is it treats everything just like you typed it out in your script. This means it will easily be able to handle arrays for checking between values. For example, in the snippet below, the value passed in is a string set to `1..15`. If you type `1..15` into PowerShell, it will create an array from 1 to 15. When this string is passed to the `Invoke-Expression`, it will be evaluated as an array. Thus, making it easy for you to determine if a value is between two numbers. When you run the snippet below, it should output true. You can then experiment with switching around the values for the first three variables to see how it works.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
$Data = 3
$Operator = 'in'
$Expected = '1..15'
$cmd = 'if($Data -{0} {1}){{$true}}' -f $Operator, $Expected
Invoke-Expression $cmd
```

Now all you have to do is loop through each test from your JSON, and if any of them return `true`, you know the value correct.

Another advantage to this is you can output the expression string to the Verbose stream to aid in testing and troubleshooting.

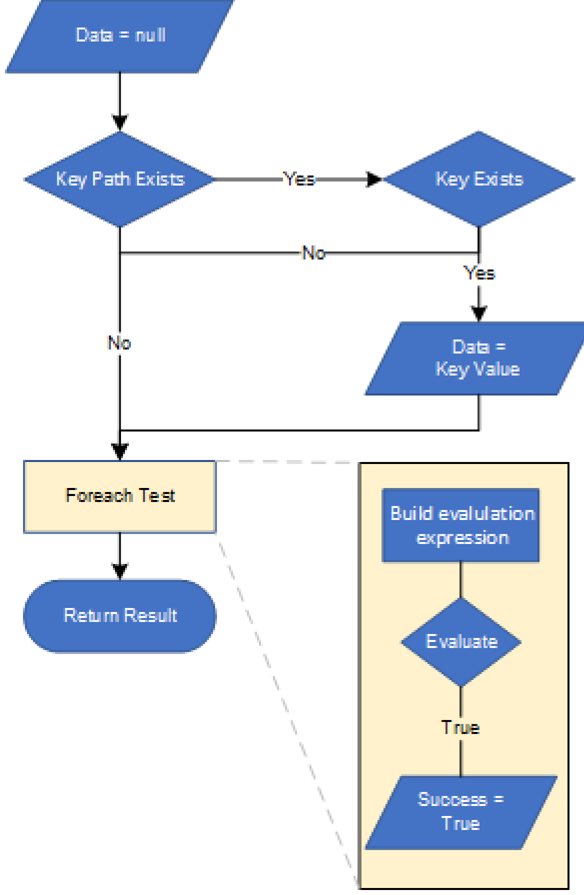


Figure 11 Full registry check workflow

One more thing to consider when building the logic of this function is how you want to pass the parameters. You can pass an object and reference the properties or list each property as a separate parameter. Passing the individual properties can make things easier for reusability and testing when you cannot guarantee you will always have your data formatted in a

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

consistent manner. However, in this case, since we have built a custom class object, you can guarantee that. Also, regardless of which method you choose, the tests will need to be a specifically formatted object. So, since you have already declared the class and can create the required object whenever you need to, you can simply have it pass the object. So, let's make the `Test-SecurityBaseline` function.

Listing 8 Test-SecurityBaseline

```
Function Test-SecurityBaseline {
    [CmdletBinding()]
    [OutputType([object])]
    param(
        [Parameter(Mandatory = $true)]
        [RegistryCheck]$Check
    )
    $Data = $null    #A
    if (-not (Test-Path -Path $Check.KeyPath)) {
        Write-Verbose "Path not found"    #B
    }
    else {
        $SubKeys = Get-Item -LiteralPath $Check.KeyPath    #C
        if ($SubKeys.Property -notcontains $Check.Name) {
            Write-Verbose "Name not found"    #D
        }
        else {
            try {
                $ItemProperty = @{    #E
                    Path = $Check.KeyPath
                    Name = $Check.Name
                }
                $Data = Get-ItemProperty @ItemProperty |
                    Select-Object -ExpandProperty $Check.Name
            }
            catch {
                $Data = $null
            }
        }
    }

    foreach ($test in $Check.Tests) {    #F
        $filter = 'if($Data -{0} {1}){{$true}}'    #G
        $filter = $filter -f $test.operator, $test.Value
        Write-Verbose $filter
        if (Invoke-Expression $filter) {
            $Check.Success = $true    #H
        }
    }

    $Check.SetValue = $Data    #I
    $Check
}
```

#A Set the initial value of $Data to null
#B If the path is not found, there is nothing to do because $Data is already set to null.
#C Get the keys that exist in the key path and confirm that the key you want is present.
#D If the key is not found, there is nothing to do because $Data is already set to null.
#E If the key is found, get the value and update the $Data variable with the value.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

#F Run through each test for this registry key.
#G Build the string to create the If statement to test the value of the $Data variable.
#H If the statement returns true, you know a test passed, so update the Success property.
#I Add the value of the key for your records and debugging

To handle the updating of the failed checks, you will want to create a separate function. This will allow you to run and test the checks and the updates separately.

This function will be named `Set-SecurityBaseline` and can use the same object from the `Test-SecurityBaseline` function to update the failed checks. This function will be pretty standard PowerShell. It just needs to ensure that the key path exists and create it if it doesn't. Then set the key to the value defined in the JSON. We will also force the Error Action to Continue so one failed entry does not stop the processing of the others.

Listing 9 Set-SecurityBaseline

```
Function Set-SecurityBaseline{
    [CmdletBinding()]
    [OutputType([object])]
    param(
        [Parameter(Mandatory = $true)]
        [RegistryCheck]$Check
    )
    if(-not (Test-Path -Path $Check.KeyPath)){    #A
        New-Item -Path $Check.KeyPath -Force -ErrorAction Stop
    }

    $ItemProperty = @{    #B
        Path         = $Check.KeyPath
        Name         = $Check.Name
        Value        = $Check.Data
        PropertyType = $Check.Type
        Force        = $true
        ErrorAction  = 'Continue'
    }
    New-ItemProperty @ItemProperty
}
```

#A Create the registry key path if it does not exist
#B Create or Update the registry key with the predetermined value

6.3 Controlling scripts with configuration data

In the previous sections, we saw how you can use event handling and serialized data to control the actions of a function. Now we can take it a step further and build a genuinely dynamic automation that can stitch together all of these functions. This is where the fundamentals of the building blocks concept from chapter 1 comes into its own.

We started with a function to stop and disable services. Then we built another function to check and registry values based on a JSON input and a third one to automatically set the registry value for those that did not meet the requirements. Finally, let's finish our automation with a few more simple examples that will help demonstrate the concepts here.

First, we will create a function to pass a string array to install Windows Features.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

Listing 10 Install-RequiredFeatures

```
Function Install-RequiredFeatures {
    [CmdletBinding()]
    [OutputType([object])]
    param(
        [Parameter(Mandatory = $true)]
        [string[]]$Features
    )
    [System.Collections.Generic.List[PSObject]] $FeatureInstalls = @()
    foreach ($Name in $Features) {    #A
        Install-WindowsFeature -Name $Name -ErrorAction SilentlyContinue |
            Select-Object -Property @{l='Name';e={$Name}}, * |
            ForEach-Object{ $FeatureInstalls.Add($_) }
    }

    $FeatureInstalls
}
```

#A Loop through each feature and install it

Then one final one to configure the internal firewall logging.

Listing 11 Set-FirewallDefaults

```
Function Set-FirewallDefaults {
    [CmdletBinding()]
    [OutputType([object])]
    param(
        [Parameter(Mandatory = $true)]
        [UInt64]$LogSize
    )
    $FirewallSettings = [pscustomobject]@{    #A
        Enabled       = $false
        PublicBlocked = $false
        LogFileSet    = $false
        Errors        = $null
    }

    try {
        $NetFirewallProfile = @{    #B
            Profile     = 'Domain', 'Public', 'Private'
            Enabled     = 'True'
            ErrorAction = 'Stop'
        }
        Set-NetFirewallProfile @NetFirewallProfile
        $FirewallSettings.Enabled = $true

        $NetFirewallProfile = @{    #C
            Name                  = 'Public'
            DefaultInboundAction = 'Block'
            ErrorAction           = 'Stop'
        }
        Set-NetFirewallProfile @NetFirewallProfile
        $FirewallSettings.PublicBlocked = $true

        $log = '%windir%\system32\logfiles\firewall\pfirewall.log'
        $NetFirewallProfile = @{    #D
            Name                  = 'Domain', 'Public', 'Private'
```

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
            LogFileName          = $log
            LogBlocked           = 'True'
            LogMaxSizeKilobytes  = $LogSize
            ErrorAction          = 'Stop'
        }
        Set-NetFirewallProfile @NetFirewallProfile
        $FirewallSettings.LogFileSet = $true
    }
    catch {
        $FeatureInstalls[-1].Errors = $_
    }

    $FirewallSettings
}
```

#A Create a custom object to record and output the results of the commands.
#B Enable all firewall profiles
#C Block all inbound public traffic
#D Set the firewall log settings, including the size.

These functions can go on and on as you think of other things to add or your requirements change.

Now comes the next challenge. Determining how to string all of these functions together and provide the correct parameters for them. This is where dynamic-based configuration data comes into play.

Similar to how you controlled the conditional statements in the previous example, you can perform similar tasks at the script level. For instance, you can generate a configuration file for each of the steps listed above based on the parameters of the functions. Then have one single script to call each function with the appropriate parameters.

6.3.1 Organizing your data

We now have five separate functions for this automation, each with its own parameters. And the values provided to those parameters can change between operating system versions and server roles. Which once again leaves us with one of those fine balancing acts of automation. Do I put everything into one massive JSON file and have the script parse it? Do I create a separate JSON file for each operating system version? Then would I need separate ones for each role that a server would need for each operating system? As you can see, it is easy to end up with a massive mess of files as it is to end up with a few gigantic unmanageable files. The best thing to do in these situations is to write it out.

Look at each step of your automation and think about the parameter requirements across versions and roles. Doing this will help you determine the best way to structure your data.

- Step one of installing roles and features is relatively consistent across operating system versions but wildly different between server roles.
- Step two will be stopping and disabling services. These can change slightly between operating system versions and based on the server role.
- Step three of setting security baselines will remain fairly consistent across roles but will differ between operating system versions.
- Step four of setting the internal firewall settings is consistent across operating systems but not server roles.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

As you can see, there is no clear winner between operating system or server role. Which again, is there the beauty of this model comes into play. Technically, you don't have to choose. What you can do is combine them.

For instance, you can create a Windows Server 2019 baseline configuration file with the settings all servers will have regardless of their role. Then you can make smaller role-specific configurations just to apply the deltas. You could even create new functions that could, for example, turn services back on.

For our purposes, we will build a simple configuration file that can be used on any Windows Server 2016 or later operating system. Taking a look at the different steps, we know we will need the following parameters:

1. Feature = Default features and roles to install
2. Service = List of services to stop and disable
3. SecurityBaseline = Security baseline registry keys
4. FirewallLogSize = Firewall log size

The tricky one here is the security baseline registry keys. There could be hundreds of entries in this list. To keep things cleaner, you can keep these in a separate file and just reference them in the control JSON. But you risk inadvertently causing issues if you don't remain vigilant about ensuring those references are not broken. The safe approach would be to combine them into one JSON.

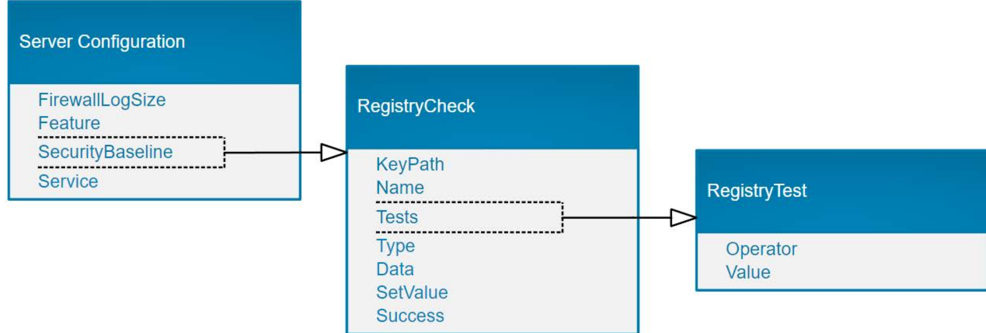


Figure 12 Server Configuration Class

And once again, you can create a class in the `PoshAutomate-ServerConfig.psm1`. This class will consist of the fours parameters identified above.

Listing 12 Server Config Class

```
class ServerConfig {
    [string[]]$Features
    [string[]]$Services
    [RegistryCheck[]]$SecurityBaseline
    [UInt64]$FirewallLogSize
    ServerConfig(){    #A
```

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
        $this.SecurityBaseline += [RegistryCheck]::new()
    }
    ServerConfig(    #B
        [object]$object
    ){
        $this.Features = $object.Features
        $this.Services = $object.Services
        $this.FirewallLogSize = $object.FirewallLogSize
        $object.SecurityBaseline | Foreach-Object {
            $this.SecurityBaseline += [RegistryCheck]::new($_)
        }
    }
}
```

#A Method to create a blank instance of this class
#B Method to create an instance of this class populated with data from a generic PowerShell object

Another significant advantage of using classes is that you can quickly and easily create your configuration JSON. You can add another function to the `PoshAutomate-ServerConfig.psm1` named `New-ServerConfig` and have it create a blank version of the `ServerConfig` class.

Listing 13 New-ServerConfig

```
Function New-ServerConfig{
    [ServerConfig]::new()
}
```

You can take this a step further and use this function to create a JSON template for you.

```
PS P:\> Import-Module .\PoshAutomate-ServerConfig.psd1 -Force
PS P:\> New-ServerConfig | ConvertTo-Json -Depth 4

{
    "Features": null,
    "Service": null,
    "SecurityBaseline": [
        {
            "KeyPath": null,
            "Name": null,
            "Type": null,
            "Data": null,
            "SetValue": null,
            "Tests": [
                {
                    "operator": null,
                    "Value": null
                }
            ]
        }
    ],
    "FirewallLogSize": 0
}
```

Now we have all our data defined; it is time to build the final part of the automation.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

6.3.2 Using your configuration data

The final step in this process is to tie everything together nicely with one script. This last script will get the required configuration information from the JSON file. Then use that data to run each of the functions we have created.

It is also a good idea for a script like this to add some logging output to record what was done. In this case, we can just create a function that will output the returned data to a table view and write it to a text file on the local machine. This means the script will need a parameter for the JSON file and one for the log file.

You can use the combination of the `Get-Content` and `ConvertFrom-Json` cmdlets that we used before to import your configuration data. Then convert that object to the `ServerConfig` class you defined. Now all you need to do is call each function in the order you want to execute them. Since each function has error handling built-in, you do not have to worry about it here. Just run the function and write the results to the log. After the execution completes, you can review the log for any potential errors or issues.

The only exception here is the Security Baseline functions. Since there are two separate functions, you can have the script run the check once. Then fix any none compliant registry keys. Then run the check once more to confirm everything is compliant.

Putting it all together, the script will follow the flow shown in figure 11.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

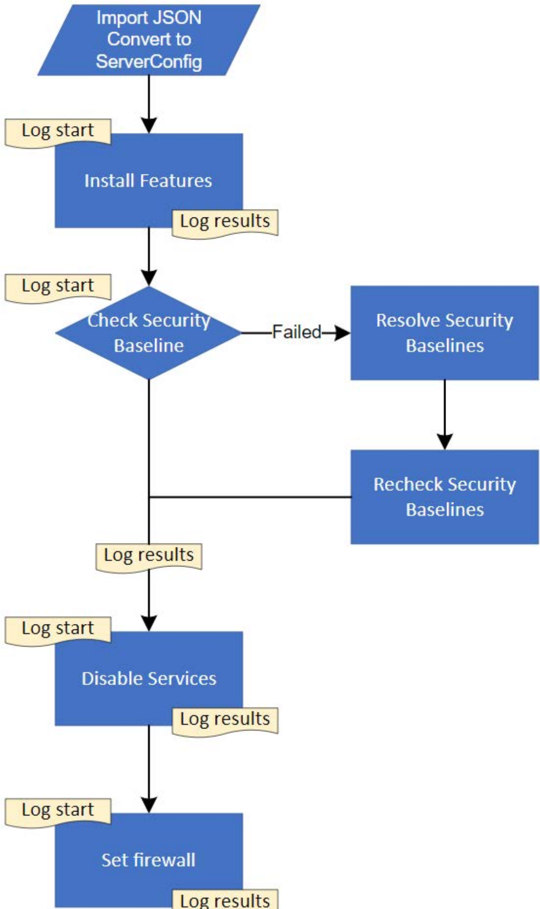


Figure 13 Workflow to set server configurations based on a JSON template

To include this function as part of the module, add a new file under the Public folder named `Set-ServerConfig.ps1` and enter the code from listing ##.

Listing 14 Set-ServerConfig

```
Function Set-ServerConfig {
    [CmdletBinding()]
    [OutputType([object])]
    param(
        [Parameter(Mandatory = $true)]
        [object]$ConfigJson,
        [Parameter(Mandatory = $true)]
        [object]$LogFile
    )
    $JsonObject = Get-Content $ConfigJson -Raw |     #A
```


Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
        ConvertFrom-Json
    $Config = [ServerConfig]::new($JsonObject)    #B

    Function Write-StartLog {    #C
        param(
            $Message
        )
        "`n$('#' * 50)`n# $($Message)`n" | Out-File $LogFile -Append
        Write-Host $Message
    }

    Function Write-OutputLog {    #D
        param(
            $Object
        )
        $output = $Object | Format-Table | Out-String
        if ([string]::IsNullOrEmpty($output)) {
            $output = 'No data'
        }
        "$($output.Trim())`n$('#' * 50)" | Out-File $LogFile -Append
        Write-Host $output
    }
    $msg = "Start Server Setup - $(Get-Date)`nFrom JSON $($ConfigJson)"
    Write-StartLog -Message $msg

    Write-StartLog -Message "Set Features"    #E
    $Features = Install-RequiredFeatures -Features $Config.Features
    Write-OutputLog -Object $Features

    Write-StartLog -Message "Set Services"    #F
    $WindowsService = @{
        Services        = $Config.Services
        HardKillSeconds = 60
        SecondsToWait   = 90
    }
    $Services = Disable-WindowsService @WindowsService
    Write-OutputLog -Object $Services

    Write-StartLog -Message "Set Security Baseline"
    foreach ($sbl in $Config.SecurityBaseline) {    #G
        $sbl = Test-SecurityBaseline $sbl
    }

    foreach ($sbl in $Config.SecurityBaseline |    #H
        Where-Object { $_.Success -ne $true }) {
        Set-SecurityBaseline $sbl
        $sbl = Test-SecurityBaseline $sbl
    }
    $SecLog = $SecBaseline |
        Select-Object -Property KeyPath, Name, Data, Result, SetValue
    Write-OutputLog -Object $SecLog

    Write-StartLog -Message "Set Firewall"    #I
    $Firewall = Set-FirewallDefaults -LogSize $Config.FirewallLogSize
    Write-OutputLog -Object $Firewall

    Write-Host "Server configuration is complete."
    Write-Host "All logs written to $($LogFile)"
}
```

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

#A Import the configuration data from the JSON file
#B Convert the JSON data to the class you defined
#C A small function to ensure consistent logs are written for an activity starting
#D A small function to ensure consistent logs are written for an activity completing
#E Set Windows Features first
#F Set the services
#G Check each registry key in the Security baseline
#H Fix any that did not pass the test
#I Set the firewall

6.3.3 Storing your configuration data

Where you store your configuration data can change from automation to automation, but it makes sense to store the configuration data within the module in most cases. Doing this will not only ensure your data is there when you need it, but if you implement version control on your modules, you will be able to track changes to the configuration files as well.

Another bonus to storing your configuration files within the module is you can create a wrapper function to help you select the files. This way, you do not have to look up the full path to the JSON file every time you want to run it. You can do this in a similar fashion to how you load the module functions from the different ps1 files.

To set this up, add a folder named `Configurations` to the module directory and place your configuration JSON files in there.

Listing 15 Create Server Config JSON

```
Import-Module .\PoshAutomate-ServerConfig.psd1 -Force    #A

$Config = New-ServerConfig    #B

$Content = @{    #C
    Path = '.\RegistryChecksAndResolves.json'
    Raw  = $true
}
$Data = (Get-Content @Content | ConvertFrom-Json)
$Config.SecurityBaseline = $Data

$Config.FirewallLogSize = 4096    #D

$Config.Features = @(    #E
    "RSAT-AD-PowerShell"
    "RSAT-AD-AdminCenter"
    "RSAT-ADDS-Toolsf"
)

$Config.Services = @(    #F
    "PrintNotify",
    "Spooler",
    "lltdsvc",
    "SharedAccess",
    "wisvc"
)

if(-not (Test-Path ".\Configurations")){    #G
    New-Item -Path ".\Configurations" -ItemType Directory
}
```

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
$Config | ConvertTo-Json -Depth 4 |     #H
    Out-File ".\Configurations\SecurityBaseline.json" -Encoding UTF8
```

#A Import the module
#B Create a blank configuration item
#C Import security baseline registry keys
#D Set default firewall log size
#E Set roles and features to install
#F Set services to disable
#G Create the Configurations folder
#H Export the security baseline

Then in the `PoshAutomate-ServerConfig.psm1` file, add a command to query this folder for the different JSON files. Then add a function named `Invoke-ServerConfig` that will display the JSON files for you to select. Once you make your selection, it will automatically execute the `Set-ServerConfig` function for you.

You can even use the `Out-GridView` to make a pop-up to make selections, or you can just pass in the name if you know it. Also, it can have multiple selections allowing you to run the operating system and role-based configurations one after the other.

Listing 16 Invoke-ServerConfig

```
Function Invoke-ServerConfig{
    [CmdletBinding()]
    [OutputType([object])]
    param(
        [string[]]$Config = $null
    )
    [System.Collections.Generic.List[PSObject]]$selection = @()
    $Path = @{     #A
        Path      = $PSScriptRoot
        ChildPath = 'Configurations'
    }
    $ConfigPath = Join-Path @Path

    $ChildItem = @{     #B
        Path   = $ConfigPath
        Filter = '*.JSON'
    }
    $Configurations = Get-ChildItem @ChildItem

    if(-not [string]::IsNullOrEmpty($Config)){     #C
        foreach($c in $Config){
            $Configurations | Where-Object{ $_.BaseName -eq $Config } |
                ForEach-Object { $selection.Add($_) }
        }
    }

    if($selection.Count -eq 0){     #D
        $Configurations | Select-Object BaseName, FullName |
            Out-GridView -PassThru | ForEach-Object { $selection.Add($_) }
    }

    $Log = "$($env:COMPUTERNAME)-Config.log"     #E
    $LogFile = Join-Path -Path $($env:SystemDrive) -ChildPath $Log
```

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
    foreach($json in $selection){    #F
        Set-ServerConfig -ConfigJson $json.FullName -LogFile $LogFile
    }
}
```

#A Get all the Configurations folder
#B Get all the JSON files in the Configurations folder
#C If a config name is passed, attempt to find the file
#D If config name is not passed or name is not found, prompt for a file to use
#E Set the default log file path
#F Run the Set-ServerConfig for each JSON file

Once you have everything put together, you can copy the module files over to your new server, import it, and run the configuration.

```
Import-Module .\PoshAutomate-ServerConfig.psd1 -Force
Invoke-ServerConfig
```

6.3.4 Do not put cmdlets into your configuration data.

One final thing to remember when using configuration data is that you never want to put any actual commands in your data. The values in your data should be static. Putting commands in your configuration data will not only make troubleshooting and testing a nightmare but can cause unexpected conditions in your code that could have dire consequences. While this may seem contradictory to the last sections, it is not. In that section, we passed in value and conditional operators but not actual commands. Those values and operators will return the same result every single time you run them.

An excellent example of this is dealing with dates. Say you need to check that a date is over X number of years from now. The best way to handle that would be to have your script create the DateTime object. Then you can include a property in your configuration that holds the number of years. Then, this property can be passed to the `AddYears()` method of the DateTime to have the script set the date value for you.

```
$AddYears = 1
$Data = Get-Date 1/21/2035
$DateFromConfig = (Get-Date).AddYears($AddYears)
$cmd = 'if($Data -{0} {1}){{$true}}' -f 'gt', '$DateFromConfig'
Invoke-Expression $cmd
```

A bad example would be to pass in a string set to `(Get-Date).AddYears(1)` and use the `Invoke-Expression` to evaluate it. While it would have the results as the previous example, it is also more prone to error. More difficult to troubleshoot and opens you up to potential injection attacks.

```
$Data = Get-Date 1/21/2035
$cmd = 'if($Data -{0} {1}){{$true}}' -f 'gt', '(Get-Date).AddYears(1)'
Invoke-Expression $cmd
```

The critical thing to remember here is that you should test all of your functions independently without running any external commands. The configuration data import is just a way for you to provide a list of parameters to your scripts.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

6.4 Summary

- You can use `try/catch` blocks to capture specific errors and take actions based on them.
- Remember the DRY principle. Don't repeat the same code over and over. Instead, use your data to help drive your scripts.
- JSON is a versatile data-serialization format that supports most of the same data types you find natively in PowerShell.
- If you have a defined data structure, you should create a class to help maintain its integrity.
- You can use external data to create data-driven functions and to control the execution of your scripts.
- Data files should be stored with the scripts when they are directly related to each other.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

7
Working with SQL

This chapter covers

- **Building SQL databases and tables**
- **Inserting and updating data**
- **Retrieving data**
- **Using data validation before writing to SQL**

Anyone who has worked in IT long enough has inevitably received a call for a panic department head that the Excel spreadsheet or Access database the entire department is reliant on has broken. As you investigate, you discover a mess of spaghetti code macros that have been cobbled together over the years. As you are racking your brain trying to reverse engineer it, you keep thinking to yourself, how did this happen.

I have found that these are often the result of one person taking some initiative to try and improve their job. In much the same way an Automator thinks. However, before they realized what was happening, the entire department became dependent on what they threw together in their spare time. I am here to tell you this is not just a problem in "other" departments. It happens with IT and with automations as well. And I am also here to show you how not to fall into this trap by learning to use a proper database.

In the last chapter, we saw how you can use data to help drive your scripts and automations. And the data we used was stored in local JSON files. This is fine when the data is relatively static, and you have tight control over who can update it. However, a local or even shared file will not cut when you have data that needs to be shared with multiple people who can update it. In these cases, you will want to use a relational database.

There are multiple different database engines available that you can use with PowerShell, but we will work with Microsoft SQL Server for this chapter. However, many of the items discussed here are database agnostic, so you could easily implement the same automations using your database of choice.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

At the end of chapter 5, I told the story of how I used use PowerShell remoting across various systems in different environments to resolve an issue with a bad definition update. What I failed to mention was how I knew where all the systems resided. With today's hybrid environments, it is increasingly difficult to track whether a server is physical, a VMware VM, a Hyper-V VM, an Azure or AWS VM, etc.

To help solve this, we will be creating a PowerShell module you and your team can use to track your server assets across all your different environments. This module will store the data about your servers in a SQL database. And you will build the functionality to:

1. Add a server to the database
2. Search the database for servers
3. Update the information for one or more servers

You can use Microsoft SQL Server Express for this automation, which you can download and use for free. If you already installed SQL Express in chapter 4 for the database health checks, you can use the same one here. Otherwise, I've included a script in the Helper Scripts folder for this chapter to install and set up a SQL Express instance.

We will also be using the `dbatools` PowerShell module to perform all the interactions with SQL. Including creating the database objects. Which we will start with right now by using the `New-DbaDatabase` cmdlet.

To create the database, all you need to do is provide the SQL instance and the database name. We will also set the Recovery Model to Simple. Without deviling deep into DBA territory, a Simple recovery model is acceptable unless you are designing mission-critical, highly available, zero data loss systems. You can still create full and differential backups with simple logs, so there is no need to require the resources that full logs will take.

If you are running PowerShell on the same machine as the SQL Express install, you can run the listing below, as written, to create the database. However, if you are on a remote machine or using your own SQL instance, then be sure to update the `$SqlInstance` variable for your environment.

Listing 7.1 Create PoshAssetMgmt database

```
$SqlInstance = "$($env:COMPUTERNAME)\SQLEXPRESS"
$DatabaseName = 'PoshAssetMgmt'
$DbaDatabase = @{
    SqlInstance   = $SqlInstance
    Name          = $DatabaseName
    RecoveryModel = 'Simple'
}
New-DbaDatabase @DbaDatabase
```

One last note before we get started, this chapter does not expect you to be a DBA, nor will it teach you how to be one. We will only use simple queries in the functions to provide examples of interacting with SQL. There are plenty of resources out there to help you learn how to create complex queries, backup your databases, and perform preventive maintenance. For now, we are just going to focus on getting data in and out of a database.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

7.1 Setting your schema

As we have now seen with almost every automation, the most crucial step in the automation process is defining the data you will need. In this case, you will need to create a table to hold your server information.

For the SQL table, you can start with some fairly standard columns like:

1. Name – the name of the asset
2. Operating System Type – Linux or Windows
3. Operating System Version – The name of the operating system version
4. Status – whether it is in service, being repaired, or retired
5. Remote Method – The method of remote connection to use for this server. (SSH, WSMan, Power CLI, etc.)

Next, you will need to add some additional columns that will allow you to create references to the external systems. When creating references to external systems, it is best to avoid using values that are subject to change, like display names. Most hypervisors have a unique internal identifier for their virtual machines. For instance, VMware has the Managed Object Reference ID for every virtual machine. Azure virtual machines all have a universally unique identifier (UUID). No matter what systems you use, you should be able to find a way to uniquely identify the different servers.

1. UUID – The unique identifier from the source systems.
2. Source – The system that you are referencing. (Hyper-V, VMware, Azure, AWS, etc.)
3. Source Instance – The instance of the source environment. This can be the vSphere cluster, Azure subscription, etc. Anything to let you know where that data came from.

Along with the items listed above, you will want to create an identity column. An identity column is a column that is automatically populated by the database. This will allow you to automatically assign an ID to every entry without having to write any code to do it. This will come in handy when you need to reference items between tables. Also, when updating items, you can use the ID instead of trying to match on other fields.

You can add additional fields for tracking cost centers, IP address, subnet, or whatever would make your job easier. Keep in mind that the ultimate goal here is to create one place to see all servers and quickly identify where they live. Be careful you are not just duplicating the data from other systems.

7.1.1 Data types

Much like when building a PowerShell function, you need to consider the data types when determining your table schema. However, it is not always as simple as an int equals an int. This is especially true when it comes to string.

In SQL, and other database engines, there are multiple different types of strings, and we could spend the entire rest of the chapter discussing the different types and when and why to use each. But the most common one used is the `nvarchar` type.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

A `nvarchar` column can hold 1 to 4000 byte-pairs, and most importantly, it can support Unicode characters. Since there is such a range, when you declare a `nvarchar`, you also need to set a maximum character length.

> **NVARCHAR MAX:** There is a nvarchar max in which you can store approximately 1 billion characters. However, using this is very inefficient on the SQL backend and, in most cases, is just unnecessary.

For most other fields, the decisions between SQL and PowerShell data types are straightforward. Based on the size of a number you need, you can choose an int, float, double, real, decimal, etc. And there are some data types with different names. For example, a GUID in SQL is a `uniqueidentifier`, and a Boolean is a `bit`.

The last thing you need to consider is whether or not to allow null values. In our example, we will want to ensure that all fields are populated because the data would not be helpful if any data is missing. However, if you added a column for the cost center, you could allow it to be null. This is because there could be servers without a cost center, so requiring it could prevent you from being able to add a server. On the other hand, having a blank UUID would make the entry worthless because you cannot reference it back to the source system.

So now we can map our data needed to their SQL data types.

Table 7.1 Servers

Name	Type	MaxLength	Nullable	Identity
ID	int	N/A	No	Yes
Name	nvarchar	50	No	No
OSType	nvarchar	15	No	No
OSVersion	nvarchar	50	No	No
Status	nvarchar	15	No	No
RemoteMethod	nvarchar	25	No	No
UUID	nvarchar	255	No	No
Source	nvarchar	15	No	No
SourceInstance	nvarchar	255	No	No

Now that you have the data defined, you can use the `New-DbaDbTable` cmdlet to create the table. Start by defining each column in a hashtable, then adding the different hashtables into an array for each table. This array is then passed to the `-ColumnMap` parameter to set schema information for each column. Finally, you can translate the information from the table directly into the hashtables.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
$ID = @{
    Name = 'ID';
    Type = 'int';
    MaxLength = $null;
    Nullable = $false;
    Identity = $true;
}
```

Now that you have your table schema defined, you can create the table by supplying the SQL instance, the database to create it, and a name for the table.

Listing 7.2 Create Servers table in SQL

```
$SqlInstance = "$($env:COMPUTERNAME)\SQLEXPRESS"
$DatabaseName = 'PoshAssetMgmt'
$ServersTable = 'Servers'
$ServersColumns = @(
    @{Name = 'ID';    #A
        Type = 'int'; MaxLength = $null;
        Nullable = $false; Identity = $true;
    }
    @{Name = 'Name';    #B
        Type = 'nvarchar'; MaxLength = 50;
        Nullable = $false; Identity = $false;
    }
    @{Name = 'OSType';    #C
        Type = 'nvarchar'; MaxLength = 15;
        Nullable = $false; Identity = $false;
    }
    @{Name = 'OSVersion';    #D
        Type = 'nvarchar'; MaxLength = 50;
        Nullable = $false; Identity = $false;
    }
    @{Name = 'Status';    #E
        Type = 'nvarchar'; MaxLength = 15;
        Nullable = $false; Identity = $false;
    }
    @{Name = 'RemoteMethod';    #F
        Type = 'nvarchar'; MaxLength = 25;
        Nullable = $false; Identity = $false;
    }
    @{Name = 'UUID';    #G
        Type = 'nvarchar'; MaxLength = 255;
        Nullable = $false; Identity = $false;
    }
    @{Name = 'Source';    #H
        Type = 'nvarchar'; MaxLength = 15;
        Nullable = $false; Identity = $false;
    }
    @{Name = 'SourceInstance';    #I
        Type = 'nvarchar'; MaxLength = 255;
        Nullable = $false; Identity = $false;
    }
)
$DbaDbTable = @{
    SqlInstance = $SqlInstance
    Database    = $DatabaseName
    Name        = $ServersTable
```

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
    ColumnMap    = $ServersColumns
}
New-DbaDbTable @DbaDbTable
```

#A Create ID column as an identity column
#B Create Name column as a string with a max length of 50 characters
#C Create OSType column as a string with a max length of 15 characters
#D Create OSVersion column as a string with a max length of 50 characters
#E Create a Status column as a string with a max length of 15 characters
#F Create RemoteMethod column as a string with a max length of 25 characters
#G Create UUID column as a string with a max length of 255 characters
#H Create Source column as a string with a max length of 15 characters
#I Create SourceInstance column as a string with a max length of 255 characters

Once you have your table created, it is time to create the module and functions to interact with them.

7.2 Connecting to SQL

Throughout this chapter, you will be making calls to a single SQL instance and database. To keep from having to pass these as parameters to your functions each and every time, you can set them as variables in the module's psm1 file. Then just reference these variables in your functions.

When you do this, it is always good to make the variables with a name that will be unique to your module. I tend to also include an underscore at the beginning of the variable name to help identify it as a module variable.

For things like connection information, it makes sense to create it as a single PowerShell object, with properties for the individual values, like the SQL server instance, the database, and any table name. This can make things cleaner and more manageable by only having one variable. But before you can do that, you need to create the module files.

We will be putting all the code created into a module named `PoshAssetMgmt`. You can quickly generate the base structure you will need using the `New-ModuleTemplate` function from chapter 2.

Listing 7.3 Creating the PoshAssetMgmt module

```
Function New-ModuleTemplate {    #A
    [CmdletBinding()]
    [OutputType()]
    param(
        [Parameter(Mandatory = $true)]
        [string]$ModuleName,
        [Parameter(Mandatory = $true)]
        [string]$ModuleVersion,
        [Parameter(Mandatory = $true)]
        [string]$Author,
        [Parameter(Mandatory = $true)]
        [string]$PSVersion,
        [Parameter(Mandatory = $false)]
        [string[]]$Functions
    )
    $ModulePath = Join-Path .\ "$($ModuleName)\$($ModuleVersion)"
```

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
    New-Item -Path $ModulePath -ItemType Directory
    Set-Location $ModulePath
    New-Item -Path .\Public -ItemType Directory

    $ManifestParameters = @{
        ModuleVersion     = $ModuleVersion
        Author            = $Author
        Path              = ".\$($ModuleName).psd1"
        RootModule        = ".\$($ModuleName).psm1"
        PowerShellVersion = $PSVersion
    }
    New-ModuleManifest @ManifestParameters

    $File = @{
        Path     = ".\$($ModuleName).psm1"
        Encoding = 'utf8'
    }
    Out-File @File

    $Functions | ForEach-Object {
        Out-File -Path ".\Public\$($_).ps1" -Encoding utf8
    }
}

$module = @{    #B
    ModuleName    = 'PoshAssetMgmt'    #C
    ModuleVersion = "1.0.0.0"    #D
    Author        = "YourNameHere"    #E
    PSVersion     = '7.1'    #F
    Functions     = 'Connect-PoshAssetMgmt',    #G
    'New-PoshServer', 'Get-PoshServer', 'Set-PoshServer'
}
New-ModuleTemplate @module    #H
```

#A This is the same function as Listing 5 in chapter 2
#B Set the parameters to pass to the function
#C The name of your module
#D The version of your module
#E Your name
#F The minimum PowerShell version this module supports
#G The functions to create blank files for in the Public folder
#H Execute the function to create the new module

Once the files have been created, open the `PoshAssetMgmt.psm1` and create a variable named `$_PoshAssetMgmt` to hold the connection information about your database. Since this variable is declared inside the psm1 file, it will automatically be scoped so all functions inside the module can access it. Therefore, you don't need to add it as a parameter or set it globally.

You will also add the same functionality we used in previous modules to import ps1 files and check for the `dbatools` module.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

Listing 7.4 PoshAutomate-AssetMgmt

```
$_PoshAssetMgmt = [pscustomobject]@{
    SqlInstance  = 'YourSqlSrv\SQLEXPRESS'    #A
    Database     = 'PoshAssetMgmt'
    ServerTable  = 'Servers'
}

$Path = Join-Path $PSScriptRoot 'Public'
$Functions = Get-ChildItem -Path $Path -Filter '*.ps1'    #B

Foreach ($import in $Functions) {    #C
    Try {
        Write-Verbose "dot-sourcing file '$($import.fullname)'"
        . $import.fullname    #D
    }
    Catch {
        Write-Error -Message "Failed to import function $($import.name)"
    }
}

[System.Collections.Generic.List[PSObject]]$RequiredModules = @()
$RequiredModules.Add([pscustomobject]@{    #E
    Name = 'dbatools'
    Version = '1.1.5'
})

foreach($module in $RequiredModules){    #F
    $Check = Get-Module $module.Name -ListAvailable

    if(-not $check){
        throw "Module $($module.Name) not found"
    }

    $VersionCheck = $Check |
        Where-Object{ $_.Version -ge $module.Version }

    if(-not $VersionCheck){
        Write-Error "Module $($module.Name) running older version"
    }

    Import-Module -Name $module.Name
}
```

#A Update SqlInstance to match your server name
#B Get all the ps1 files in the Public folder
#C Loop through each ps1 file
#D Execute each ps1 file to load the function into memory
#E Create an object for each module to check
#F Check if the module is installed on the local machine

7.2.1 Permissions

One of the best advantages of using SQL is the built-in permission handling. Microsoft SQL has so many different levels of permissions it is sure to fit your needs. It goes way deeper than simple read and write permissions. SQL can even support permissions down to the row and column levels.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

You may have noticed that the first listing in the chapter where you created the database did not include a credential parameter. That is because the `dbatools` module will use the logged-in user if no other credentials are supplied. This makes things super simple to implement in an Active Directory domain environment. However, not everyone is running in a domain, and there are times where you may want to run as a different user. To account for these situations, you can build a connection function using the `Connect-DbaInstance` cmdlet. This will allow you to set a default connection that all the other functions can use.

To do this, you create the `Connect-PoshAssetMgmt` function. The parameters of this function will allow you to pass in a SQL instance, database, and credentials. If the SQL instance or database is not provided, it can use the default values set in the `$_PoshAssetMgmt` variable.

Creating a connection function like this is pointless if you still have to pass it to every function. Therefore, you can save the connection information to a variable that the other functions can reference. Similar to what you just did with the `$_PoshAssetMgmt` variable in the psm1. The only difference here is this variable is inside of a function and not in the psm1.

When a variable is set inside the psm1, it is automatically scoped to the script level. This allows all the other functions in the module to read that variable. However, when you set a variable inside a function, it only exists in the scope of that function. That is unless you scope to the script level by adding `$script`: to the variable name. As you can see in the snippet below, the variable for the connection information is set using `$script:_SqlInstance` to ensure that it is scoped to the script level.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

Listing 7.5 Connect-PoshAssetMgmt

```
Function Connect-PoshAssetMgmt {
    [CmdletBinding()]
    [OutputType([object])]
    param(
        [Parameter(Mandatory = $false)]
        [string]$SqlInstance = $_PoshAssetMgmt.SqlInstance,

        [Parameter(Mandatory = $false)]
        [string]$Database = $_PoshAssetMgmt.Database,

        [Parameter(Mandatory = $false)]
        [PSCredential]$Credential
    )

    $connection = @{    #A
        SqlInstance = $SqlInstance
        Database    = $Database
    }

    if ($Credential) {    #B
        $connection.Add('SqlCredential', $Credential)
    }

    $Script:_SqlInstance = Connect-DbaInstance @connection

    $Script:_SqlInstance    #C
}
```

#A Set default connection parameters
#B Add credential object if passed
#C Output the result, so the person running it can confirm the connection information

You can test it by importing the module and running the `Connect-PoshAssetMgmt` function.

```
PS P:\> Import-Module '.\PoshAssetMgmt.psd1' -Force
PS P:\> Connect-PoshAssetMgmt

ComputerName Name             ConnectedAs
------------ ----             -----------
SRV01        SRV01\SQLEXPRESS SRV01\Administrator
```

7.3 Adding data to a table

Now that you have the connection defined and the database created, the next step will be to import the data into the tables. To do this, we will create a new function named `New-PoshServer`. This function will use the `Write-DbaDataTable` cmdlet to add an entry to the `Servers` table in SQL.

You can define the data to import by mapping the parameters to the table columns, thus ensuring you receive the correct data. However, that is just a small part of the equation. To ensure that you are inserting properly formatted data, you need to validate it before inserting it.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

7.3.1 String validation

To ensure that the `New-PoshServer` function can insert the data into the table, you must ensure that you have the correct data and that it matches the data types and lengths set in the table. Luckily for us, a lot of this validation can be done using the PowerShell parameter validation functionality.

For example, you want to ensure that you do not pass any null or blank values. You can quickly achieve this by setting the `Mandatory` parameter attribute to `True` for all the parameters. By default, PowerShell sets them to not required.

You will also need to check the length for all string parameters to ensure they do not exceed the max length for the column. To confirm a string does not exceed the max length, you can use the parameter validation attribute `ValidateScript`. The `ValidateScript` attribute allows you to define a script block that can validate a parameter's value. If your script block returns true, then the value is considered valid. For instance, to confirm the name is under the 50-character limit, you can have a simple conditional statement that checks the length of the string.

```
Function New-PoshServer {
    param(
        [Parameter(Mandatory=$true)]
        [ValidateScript({$_.Length -le 50 })]
        [string]$Name
    )
    $PSBoundParameters
}
New-PoshServer -Name 'Srv01'
New-PoshServer -Name 'ThisIsAReallyLongServerNameThatWillCertainlyExceed50Characters'
```

You can then repeat this same pattern with the `OSVersion`, `UUID`, and `SourceInstance` parameters, ensuring you enter the maximum length for each.

You will want to take a different approach for the `Status`, `OSType`, `RemoteMethod`, and `Source` parameters because these columns will have predefined values. For these, you can use the `ValidateSet` parameter attribute to control which values can be passed to this parameter.

```
[Parameter(Mandatory=$true)]
[ValidateSet('Active','Depot','Retired')]
[string]$Status,

[Parameter(Mandatory=$true)]
[ValidateSet('Windows','Linux')]
[string]$OSType,

[Parameter(Mandatory=$true)]
[ValidateSet('WSMan','SSH','PowerCLI','HyperV','AzureRemote')]
[string]$RemoteMethod,

[Parameter(Mandatory=$true)]
[ValidateSet('Physical','VMware','Hyper-V','Azure','AWS')]
[string]$Source,
```

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

You can update these validation sets to fit your individual needs. The great thing about them is they prevent situations where you can have misspellings or different abbreviations skewing your data. For instance, I've seen situations where half the VMs are listed as being in HyperV and the other half being in Hyper-V, making searching for all Hyper-V VMs a tedious and error-prone task.

In more advanced applications, you can use enumeration sets that contain a list of named constants with an underlying integral value. These are invaluable when building APIs or bringing data in from multiple locations. But for a simple PowerShell function, they can be overkill.

7.3.2 Insert data to a table

Once you have all your data validated, it is time to write it to the SQL table. To do this, you can write a T-SQL Insert statement and have it passed to a query, but the `dbatools` module has a cmdlet to make it even easier for you. You can perform single or bulk data uploads to a SQL table using the `Write-DbaDataTable` cmdlet, with no SQL commands required.

To use the `Write-DbaDataTable` cmdlet, you need to provide a PowerShell object where the properties match the columns in the table, then provide the SQL instance and the table, and it will handle the rest of the insert process for you.

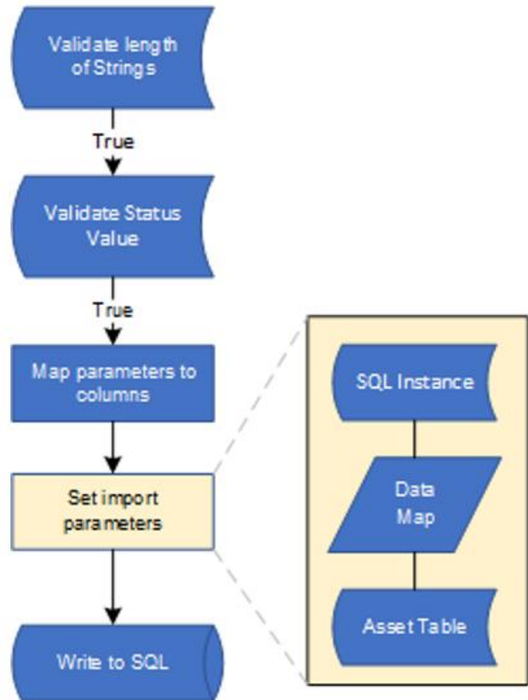


Figure 7.1 New-PoshServer function with validation and SQL instance information

Licensed to Tausif Khan <tsfahmed2@gmail.com>

Putting this all together, we can create our first function, `New-PoshServer`.

Listing 7.6 New-PoshServer

```
Function New-PoshServer {
    [CmdletBinding()]
    [OutputType([object])]
    param(
        [Parameter(Mandatory = $true)]    #A
        [ValidateScript( { $_.Length -le 50 })]
        [string]$Name,

        [Parameter(Mandatory = $true)]    #B
        [ValidateSet('Windows', 'Linux')]
        [string]$OSType,

        [Parameter(Mandatory = $true)]    #C
        [ValidateScript( { $_.Length -le 50 })]
        [string]$OSVersion,

        [Parameter(Mandatory = $true)]    #D
        [ValidateSet('Active', 'Depot', 'Retired')]
        [string]$Status,

        [Parameter(Mandatory = $true)]    #E
        [ValidateSet('WSMan', 'SSH', 'PowerCLI', 'HyperV', 'AzureRemote')]
        [string]$RemoteMethod,

        [Parameter(Mandatory = $false)]    #F
        [ValidateScript( { $_.Length -le 255 })]
        [string]$UUID,

        [Parameter(Mandatory = $true)]    #G
        [ValidateSet('Physical', 'VMware', 'Hyper-V', 'Azure', 'AWS')]
        [string]$Source,

        [Parameter(Mandatory = $false)]    #H
        [ValidateScript( { $_.Length -le 255 })]
        [string]$SourceInstance
    )

    $Data = [pscustomobject]@{    #I
        Name           = $Name
        OSType         = $OSType
        OSVersion      = $OSVersion
        Status         = $Status
        RemoteMethod   = $RemoteMethod
        UUID           = $UUID
        Source         = $Source
        SourceInstance = $SourceInstance
    }

    $DbaDataTable = @{    #J
        SqlInstance = $_SqlInstance
        Database    = $_PoshAssetMgmt.Database
        InputObject = $Data
        Table       = $_PoshAssetMgmt.ServerTable
    }
```

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
    Write-DbaDataTable @DbaDataTable

    Write-Output $Data    #K
}
```

#A Validate server name is less than or equal to 50 characters.
#B Validate that the OSType is one of the predefined values.
#C Validate OSVersion is less than or equal to 50 characters.
#D Validate that the Status is one of the predefined values.
#E Validate that the RemoteMethod is one of the predefined values.
#F Validate the UUID is less than or equal to 255 characters
#G Validate that the Source is one of the predefined values.
#H Validate the SourceInstance is less than or equal to 255 characters
#I Build the data mapping for the SQL columns
#J Write the data to the table
#K Since Write-DbaDataTable doesn't have any output the data object so you know which ones where added.

Once you have the function created, you can test it by creating a few test servers.

```
Import-Module '.\PoshAssetMgmt.psd1' -Force
Connect-PoshAssetMgmt | Out-Null

$testData = @{
    OSType         = 'Windows'
    Status         = 'Active'
    RemoteMethod   = 'WSMan'
    Source         = 'VMware'
    OSVersion      = 'Microsoft Windows Server 2019 Standard'
    SourceInstance = 'Cluster1'
}

New-PoshServer -Name 'Srv01' -UUID '001' @testData
New-PoshServer -Name 'Srv02' -UUID '002' @testData
New-PoshServer -Name 'Srv03' -UUID '003' @testData
```

Now that there is data in the database, let’s look at how you can retrieve it.

7.4 Getting data from a table

The `Invoke-DbaQuery` cmdlet from the `dbatools` module returns the results of a T-SQL query to PowerShell. All you need to do is pass the information about the connection (i.e., SQL Instance, Database, Credentials, etc.) along with the query. So, for example, running the snippet below will return all the servers you just added to the table.

```
$DbaQuery = @{
    SqlInstance = "$($env:COMPUTERNAME)\SQLEXPRESS"
    Database = 'PoshAssetMgmt'
    Query = 'SELECT * FROM Servers'
}
Invoke-DbaQuery @DbaQuery
```

This query works fine for now because you only have a few entries in your table. But as you add more servers to it, it will quickly become very resource-intensive. This is because the query will return every single record from the table to PowerShell. This will require you to

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

use the `Where-Object` cmdlet to filter the data if you only want specific servers. And doing that is horribly inefficient.

By filtering your results before they get returned to PowerShell, you will not only save on memory consumption by not having unneeded records sitting there waiting to be filtered out, but it will be exponentially faster. This is due to SQL's ability to optimize data retrieval. Through the use of query execution plans, indexes, and statistics SQL can makes notes the queries you run, that it can use in the future to retrieve those results again, much faster. Some more modern versions even have automatic indexing and optimization.

When you use the `Where-Object` cmdlet, PowerShell loops through each record one at a time to determine if it needs to be filtered or not. This occurs at a much slower and more resource-intensive rate. So, let's look at how to filter the data before it ever gets to PowerShell.

7.4.1 SQL where clause

To filter your data in a SQL query, you can add a where clause to it. Where clauses in SQL allow you to filter on any column in the table. For example, if I want to get the server named Srv01, I can use the SQL query below.

```
SELECT * FROM Servers WHERE Name = 'Srv01'
```

However, like with most things we deal with, you will want to make it dynamic. You can do this by replacing the value Srv01 with a SQL variable. A SQL variable is declared using the at symbol (@) in your query. You can then create a hashtable mapping the actual values to the variables. You then pass this hashtable to the `-SqlParameter` argument in the `Invoke-DbaQuery` cmdlet. This will then swap your variables for the values when it executes in SQL.

```
$DbaQuery = @{
    SqlInstance = "$($env:COMPUTERNAME)\SQLEXPRESS"
    Database = 'PoshAssetMgmt'
    Query = 'SELECT * FROM Servers WHERE Name = @name'
    SqlParameter = @{name = 'Srv01'}
}
Invoke-DbaQuery @DbaQuery
```

While swapping the values for variables, then creating a hashtable with the values may seem like a lot of extra steps, there are several very good reasons to do it this way.

1. It checks that the correct data types are passed.
2. It will automatically escape characters in your values that may cause issues in a standard SQL string.
3. Automatically converts data types, like DateTimes, from PowerShell to SQL format.
4. It allows SQL to use the same query plan even with different values making things faster.
5. It can prevent against SQL injection attacks.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

SQL Injection Attacks

A SQL injection attack is when an attacker is able to insert unwanted code into your SQL queries. For example, you could dynamically build the SQL where clause by joining strings together. For example:

```
$query = "SELECT * FROM Servers WHERE Name = '$($Server)'"
```

If the value of the $Server variable is "Srv01", then it would create the query string.

```
SELECT * FROM Servers WHERE Name = 'Srv01'
```

The problem with this method is it leave you vulnerable. A person could inject any value they wanted into the $Server variable, including malicious code, and you SQL server would not know any different. Take for instance if someone set the value of $Server to "';Truncate table xyz;select '". In this case your automation would execute the query below erasing all data from the table XYZ.

```
SELECT * FROM Servers WHERE Name = '';Truncate table xyz;select ''
```

By parameterizing the values you prevent against this. So, if someone happened to pass the value "';Truncate table xyz;select '" as the name of a server, SQL would only recognize it as a string with that value and not actually execute the malicious code.

Since there are eight columns in the table, there many be dozens of different different where clause combinations you could want. So, in lieu of trying to guess the different where clauses a person could want, you can use PowerShell to build it dynamically.

Just like with the insert function, you need to start by determining your parameters. And once again, you can create a one-for-one mapping between the columns and the parameters.

We can also make a couple of assumptions about the behavior of the where clause. For instance, we will want to use "and" and not "or" if multiple parameters are passed. You will also want to allow it to be run without a where clause.

Now that you know your parameters and how you want them to act, you can build the logic to create and append your where clause to the SQL query. One of the best ways to do this is by creating a string array that you populate based on the parameters passed. The trick here is to only add the items to the where clause if a value is passed to the parameter. You can do this by using the `$PSBoundParameters` variable.

The `$PSBoundParameters` variable is populated inside of every function by PowerShell and is a hashtable of the parameters and the values you passed. If a parameter is not used, then there is no entry in the `$PSBoundParameters` variable. Therefore, you can loop through the items in the `$PSBoundParameters` variable to build the where clause.

However, there is one catch to using the `$PSBoundParameters` variable in this manner. When you add the `CmdletBinding` to your function, it adds the common parameters to it. So, for instance, if you added the `Verbose` switch to your command, `$PSBoundParameters` would contain a key with the name Verbose. This would cause your code to add an entry in

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

the where clause for verbose. To prevent this, you can filter out any parameters added by the `CmdletBinding` by filtering names listed in the `[System.Management.Automation.Cmdlet]::CommonParameters` property list. This will ensure that only your defined parameters are used.

By using the `GetEnumerator()` method on the `$PSBoundParameters`, you can loop through each key pair in the hashtable and add the text for that column's filter. The text will be in the format "column operator variable" and added to a string array.

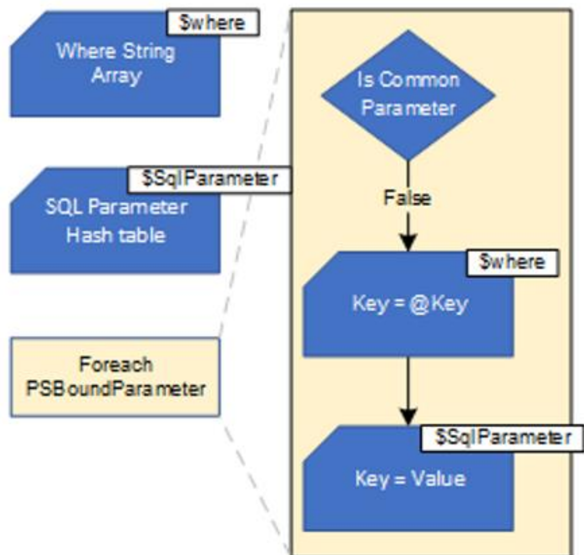


Figure 7.2 Create a string array for the where clause and a hashtable to the parameter values

You do not need to add the "and" between the different clauses at this point. That will be added at the end by using a `join` on the string array.

As you loop through each item, you will also need to build a hashtable with the key and the value for each parameter. For example, if you passed the parameter `Name` with the value of `Srv02`, the where argument would look like `Name = @Name`. Then the hashtable entry would have the key `Name` with the value `Srv02`.

Once you have the where clause string built, it is time to add it to the query, keeping in mind that you could run it without any filters. Therefore, you'll need to check if there are any values to filter. Again this can be done with a simple if condition to check the count of the where array. If it is greater than zero, then add the where clause. And finally, you can execute the `Invoke-DbaQuery` cmdlet to run the query and return the results.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

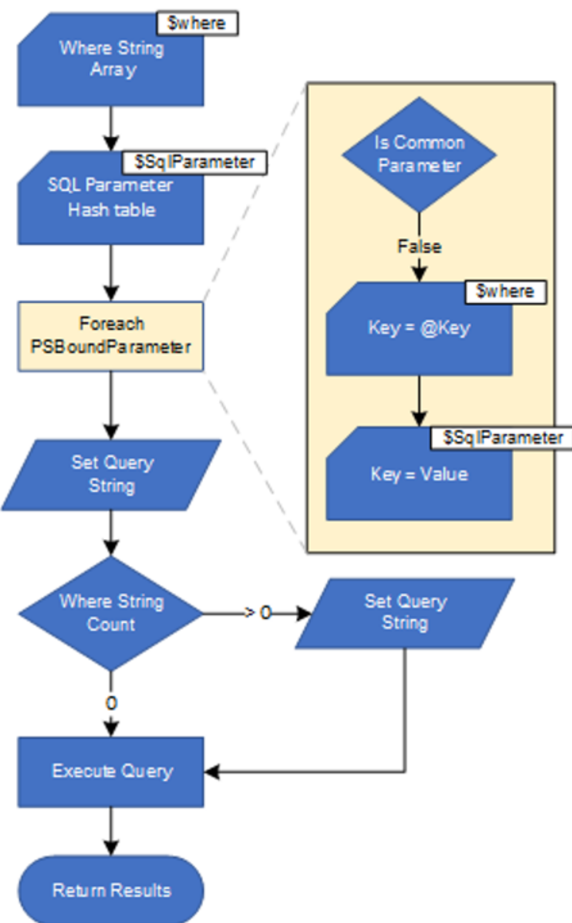


Figure 7.3 Get-PoshServer function with dynamic where clause for SQL side filtering

Listing 7.7 Get-PoshServer

```
Function Get-PoshServer {
    [CmdletBinding()]
    [OutputType([object])]
    param(
        [Parameter(Mandatory = $false)]
        [int]$ID,

        [Parameter(Mandatory = $false)]
        [string]$Name,

        [Parameter(Mandatory = $false)]
        [string]$OSType,
```


Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
        [Parameter(Mandatory = $false)]
        [string]$OSVersion,

        [Parameter(Mandatory = $false)]
        [string]$Status,

        [Parameter(Mandatory = $false)]
        [string]$RemoteMethod,

        [Parameter(Mandatory = $false)]
        [string]$UUID,

        [Parameter(Mandatory = $false)]
        [string]$Source,

        [Parameter(Mandatory = $false)]
        [string]$SourceInstance
    )

    [System.Collections.Generic.List[string]] $where = @()
    $SqlParameter = @{}
    $PSBoundParameters.GetEnumerator() |    #A
    Where-Object { $_.Key -notin
        [System.Management.Automation.Cmdlet]::CommonParameters } |
    ForEach-Object {
        $where.Add("$($_.Key) = @$($_.Key)")
        $SqlParameter.Add($_.Key, $_.Value)
    }

    $Query = "SELECT * FROM " +    #B
        $_PoshAssetMgmt.ServerTable

    if ($where.Count -gt 0) {    #C
        $Query += " Where " + ($where -join (' and '))
    }

    Write-Verbose $Query

    $DbaQuery = @{
        SqlInstance  = $_SqlInstance
        Database     = $_PoshAssetMgmt.Database
        Query        = $Query
        SqlParameter = $SqlParameter
    }

    Invoke-DbaQuery @DbaQuery    #D
}
```

#A Loop through each item in the $PSBoundParameters to create the where clause while filtering out common parameters
#B Set the default query
#C If where clause is needed, add it to the query
#D Execute the query and output the results

Go ahead and test it out with a few different combinations.

```
Import-Module '.\PoshAssetMgmt.psd1' -Force
Connect-PoshAssetMgmt
Get-PoshServer | Format-Table
```


Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
Get-PoshServer -Id 1 | Format-Table
Get-PoshServer -Name 'Srv02' | Format-Table
Get-PoshServer -Source 'VMware' -Status 'Active' | Format-Table
```

7.5 Updating records

Now that you can insert and retrieve values, it is time to look at updating records. You can update data using the `Invoke-DbaQuery` cmdlet, but unlike with retrieving data, you will use an Update SQL statement. This new function, `Set-PoshServer`, will be a combination of the Get and the New functions.

To start with, you already built all the data validation into the `New-PoshServer` function, all of which you can reuse in the `Set-PoshServer` function. Expect this time, you are going to add two additional parameters `InputObject` and `ID`. The `ID` parameter is for the ID of an existing server entry, and the `InputObject` parameter will use pipeline values from the `Get-PoshServer` function.

Since the function will need either the `ID` or the `InputObject` to update a record, you will need to ensure that at least one of them is included, but not both at the same time. To keep PowerShell from requiring both, you can use the `ParameterSetName` attribute and assign different names for the `InputObject` and `ID` parameters. This way, when the pipeline is used, it will not see the `ID` parameter as mandatory and vice-a-versa. It will also keep someone from using both at the same time.

```
[Parameter(ValueFromPipeline = $true,ParameterSetName="Pipeline")]
[object]$InputObject,
[Parameter(Mandatory = $true,ParameterSetName="Id")]
[int]$ID,
```

All other parameters not defined with a parameter set can be used regardless of whether the pipeline or the ID parameter is used.

7.5.1 Passing pipeline data

Allow values from pipelines is a great way to allow you to update multiple servers at once. For example, say you changed the name of your VMware cluster. You can use the `Get-PoshServer` function to get all the entries with the old cluster name and update them to the new name in a single line. However, to ensure you properly process the pipeline values, there is more required than just adding `ValueFromPipeline` to the parameter.

When using pipeline data, you must add the `begin`, `process`, and `end` blocks to your function. When you pass values from the pipeline, `begin` and `end` block execute once, but the `process` block will execute once for each value passed. Without it, only the last value passed to the pipeline will process.

When the function executes, the `begin` block executes once, regardless of how many items are in the pipeline. So, you use the `begin` block to set any variables or logic that only needs to run once. Then the `process` block is then executed once for each item in the pipeline. Then once all the values in the process `block` finish, the `end` block runs once. This can be used to create return data, close connections, etc.

Licensed to Tausif Khan <tsfahmed2@gmail.com>

Starting with the `begin` block, you can declare an array to store the results from each update and set the string for the query. Just like with the `Get-PoshServer` function, you will build your query dynamically. Expect this time, instead of building the where clause, you are building the Set values. You can also include a second string array to build an OUTPUT clause on the update query to return the values and compare the field before and after. For example, to update the Source column for a server, your query should look like the snippet below.

```
UPDATE [dbo].[Server]
SET Source = @Source
OUTPUT @ID AS ID, deleted.Source AS Prev_Source,
    inserted.Source AS Source
WHERE ID = @ID
```

When building the query, you will also want to ensure you exclude the `ID` and `InputObjects` parameters from being added to the query and hashtable, just like you do with the common parameters.

Next, you can set the parameters for the `Invoke-DbaQuery` command since these will be the same for each update. Finally, if the `ID` parameter is passed and not the `InputObject`, you will want to check that it is a valid ID number before trying to update it.

There are a few ways to test if the invoking command included the `ID` parameter or the pipeline. But the best way is by check the value of the `ParameterSetName` property on the `$PSCmdlet` variable. The `$PSCmdlet` variable is automatically created when a function is executed and contains the information about the invocation. If the property is equal to `ID`, you can guarantee the `ID` parameter was used. If you simply check for the existence of the `$ID` variable, you could run into a false positive if someone had set a global variable with that same name.

Next, in the process block, you can execute the update query for each `InputObject`. Here all you need to do is add the value for the ID to the SQL parameters, then add the SQL parameters to the `Invoke-DbaQuery` parameters.

Now that you have all the values set for the update statement, you can pass it to the `Invoke-DbaQuery` cmdlet and add the output to the `$Return` array. And finally, in the `end` block, you can write the `$Return` to the output stream.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

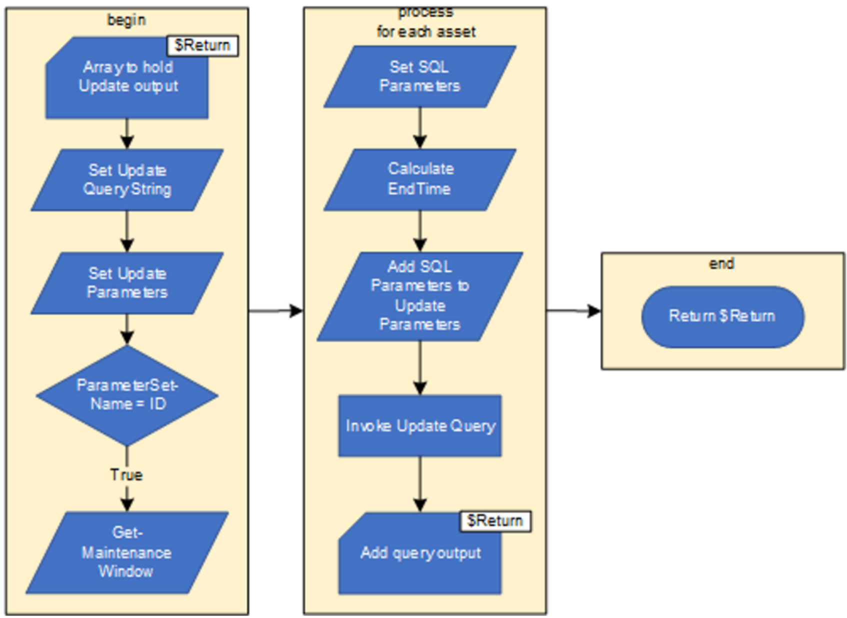


Figure 7.4 Set-PoshServer function with the ability to use pipeline or an ID number to identify which item to update

Listing 7.8 Set-PoshServer

```
Function Set-PoshServer {
    [CmdletBinding()]
    [OutputType([object])]
    param
    (
        [Parameter(ValueFromPipeline = $true,
            ParameterSetName = "Pipeline")]
        [object]$InputObject,
        [Parameter(Mandatory = $true,
            ParameterSetName = "ID")]
        [int]$ID,

        [Parameter(Mandatory = $false)]
        [ValidateScript( { $_.Length -le 50 })]
        [string]$Name,

        [Parameter(Mandatory = $false)]
        [ValidateSet('Windows', 'Linux')]
        [string]$OSType,

        [Parameter(Mandatory = $false)]
        [ValidateScript( { $_.Length -le 50 })]
        [string]$OSVersion,

        [Parameter(Mandatory = $false)]
```

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
        [ValidateSet('Active', 'Depot', 'Retired')]
        [string]$Status,

        [Parameter(Mandatory = $false)]
        [ValidateSet('WSMan', 'SSH', 'PowerCLI', 'HyperV', 'AzureRemote')]
        [string]$RemoteMethod,

        [Parameter(Mandatory = $false)]
        [ValidateScript( { $_.Length -le 255 })]
        [string]$UUID,

        [Parameter(Mandatory = $false)]
        [ValidateSet('Physical', 'VMware', 'Hyper-V', 'Azure', 'AWS')]
        [string]$Source,

        [Parameter(Mandatory = $false)]
        [ValidateScript( { $_.Length -le 255 })]
        [string]$SourceInstance
    )
    begin {
        [System.Collections.Generic.List[object]] $Return = @()
        [System.Collections.Generic.List[string]] $Set = @()
        [System.Collections.Generic.List[string]] $Output = @()
        $SqlParameter = @{ID = $null}    #A

        $PSBoundParameters.GetEnumerator() |    #B
        Where-Object { $_.Key -notin @('ID', 'InputObject') +
            [System.Management.Automation.Cmdlet]::CommonParameters  } |
        ForEach-Object {
            $set.Add("$($_.Key) = @$($_.Key)")    #C
            $Output.Add("deleted.$($_.Key) AS Prev_$($_.Key),
                inserted.$($_.Key) AS $($_.Key)")
            $SqlParameter.Add($_.Key, $_.Value)
        }

        $query = 'UPDATE [dbo].' +    #D
        "[$($_PoshAssetMgmt.ServerTable)] " +
        'SET ' +
        ($set -join (', ')) +
        ' OUTPUT @ID AS ID, ' +
        ($Output -join (', ')) +
        ' WHERE ID = @ID'

        Write-Verbose $query

        $Parameters = @{    #E
            SqlInstance  = $_SqlInstance
            Database     = $_PoshAssetMgmt.Database
            Query        = $query
            SqlParameter = @{}
        }

        if ($PSCmdlet.ParameterSetName -eq 'ID') {    #F
            $InputObject = Get-PoshServer -Id $Id
            if (-not $InputObject) {
                throw "No server object was found for id '$Id'"
            }
        }
    }
```

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
    process {
        $SqlParameter['ID'] = $InputObject.ID    #G

        $Parameters['SqlParameter'] = $SqlParameter    #H
        Invoke-DbaQuery @Parameters | ForEach-Object { $Return.Add($_) }
    }
    end {
        $Return    #I
    }
}
```

#A Create the SQL Parameters hashtable to hold the values for the SQL variables, starting with a null value for the ID
#B Loop through each item in the $PSBoundParameters to create the where clause while filtering out common parameters and the ID and InputObject parameters.
#C Add parameters other than the ID or InputObject to the Set clause array and SqlParameters.
#D Set the query with the output of the changed items
#E Set the parameters for the database update command
#F If the ID was passed, check that it matches an existing server
#G Update the ID for this InputObject
#H Update SQL parameters and execute the update
#I Return the changes

Now you can test using both an ID and the pipeline.

```
Import-Module '.\PoshAssetMgmt.psd1' -Force
Connect-PoshAssetMgmt
Set-PoshServer -Id 1 -Status 'Retired' -Verbose
Get-PoshServer -SourceInstance 'Cluster1' | Set-PoshServer -SourceInstance 'Cluster2'
```

7.6 Keeping data in sync

One of the biggest challenges with any list, spreadsheet, or database is keeping data up to date. Unfortunately, since every environment is different, I cannot give you one full-proof way to keep all your data update to date, but I can provide some tips.

The first and most important tip is to create a schedule for syncing your data. The more often you do this, the better your data will be, and it will be much easier to spot errors or omissions. If you can, the best way would be to set up a scheduled job to run daily to check and confirm your data.

The second tip is, only remove data when absolutely necessary. You may have noticed that we did not create a `Remove-PoshServer` function. This is because the removal of asset information should be carefully controlled. This is why there is a retired status for servers. You should only remove items after a predetermined amount of time or if bad data gets entered.

Right up there with the others is to watch out for duplicate data. The way the `New-PoshServer` function is written, it will allow you to and the same server in the system over and over. Again, this is by design because you could very well have two servers with the same name or even the same UUID. (Trust me, I’ve seen enough cloned VMs in my days not to doubt this). So, when you are building your automation to sync the data, you will want to build in checks to ensure data is unique to your environment.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

7.6.1 Getting server data

The way you retrieve your server information will depend very much on the type of servers and hypervisors you have.

For example, VMware PowerCLI will return the guest OS operating system from the `Get-VM` cmdlet. However, the `Get-VM` cmdlet for Hyper-V does not, but you can get the VM ID within the running VM. Again, this is another topic that could be a chapter on its own. But to get you started, I have included a couple of samples in the Helper folder for this chapter for different hypervisors and clouds.

Another option is to gather the data from external sources and export it to a CSV or JSON file. Then import that file into PowerShell and run the update sync. This method works well for disconnected networks.

You can test this method out for yourself using the listing below along with the SampleData.CSV file from the Helper folder.

Listing 7.9 Sync from external CSV

```
$ServerData = Import-Csv ".\SampleData.CSV"    #A

$ServerData | ForEach-Object {    #B
    $values = @{    #C
        Name           = $_.Name
        OSType         = $_.OSType
        OSVersion      = $_.OSVersion
        Status         = 'Active'
        RemoteMethod   = 'PowerCLI'
        UUID           = $_.UUID
        Source         = 'VMware'
        SourceInstance = $_.SourceInstance
    }

    $record = Get-PoshServer -UUID $_.UUID    #D

    if($record){    #E
        $record | Set-PoshServer @values
    }
    else{
        New-PoshServer @values
    }
}
```

#A Import the data from the CSV
#B Get all the Virtual Machines
#C Get the values for all items and mapped to the parameters for the Set-PoshServer and New-PoshServer functions.
#D Run the Get-PoshServer to see if a record exists with a matching UUID
#E If the record exists, update it; otherwise, add a new record

7.7 Setting a solid foundation

Building your solutions using a relational database has several advantages. Not only is it setting up with better backup and recovery, speed and reliability, but it also makes things easier on you.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

For example, you do not need to worry about building anything into your code to deal with permissions. Just let SQL handle it.

Also, your functions will run much smoother because you can filter results before they ever get to PowerShell. Imagine if this was all in a CSV file. You would need to import all the data into PowerShell, convert everything that is not a string back to its indented data type, and rely on pipeline filtering.

Finally, as you will see in the next few chapters, having data stored in a relational database will set you up nicely for future growth by allowing you to create interactions with it outside of a standard module like this.

7.8 Summary

- Using a relational database is more reliable than shared files.
- Most relational databases can handle permissions on their own without you needing to code anything specific for it.
- You always want to validate your data before writing it to a database.
- Setting variables as parameters in a SQL script helps with data type conversions, escape characters, and preventing SQL injection attacks.
- When using pipelines, include the process block, so each item in the pipeline is processed.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

8 Cloud-based automation

This chapter covers

- **Setting up Azure Automation**
- **Creating PowerShell runbooks in Azure Automation**
- **Executing runbooks from Azure to On-Premises environments**

With most companies embracing, at the very least, a hybrid approach to cloud-based computing, a lot of IT professionals have had to quickly adapt to keep up. However, at the same time, it has opened up an entirely new set of tools that you can use to perform your automations.

When you hear the words cloud-based automation, that can refer to two different things. One is the automation of cloud-based assets such as virtual machines or PaaS services. The other, and the one we will be focusing on, is using a cloud-based tool to perform automations. And as you will learn in this chapter, those automations are not limited to just cloud-based resources.

You may remember way back in chapter 2, we created a script to clean up log files by adding them to a ZIP archive. I mentioned that you could use then copy those archives to a cloud-based storage container. You also saw in chapter 3 how you can use tools like Task Scheduler and Jenkins to schedule jobs like this. In this chapter, we will take it a step further and use Azure Automation to copy those archive files from your on-premises server to Azure Blob storage.

Azure Automation is a cloud-based automation platform that allows you to run PowerShell scripts serverless directly in Azure. However, it is not just limited to Azure-based executions. It can use ***hybrid runbook workers*** to execute these scripts, known as runbooks, on individual servers or groups of servers. This allows you to store and schedule scripts in Azure to run on servers in your on-premises environment or even in other clouds.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

8.1 Chapter resources

This chapter is in three parts. First, you will create the Azure resources required to support the automation. Then set up a local server as a hybrid runbook worker. And the final part will be the creation of the automation in Azure.

If you do not have an Azure subscription, you can sign up for a free 30-day trial. Since Azure Automation is a cloud-based service, there are fees associated with job run times. At the time of this writing, and ever since Azure Automation was introduced, it provides 500 minutes a month for free. After you use the 500 minutes, the cost is a fraction of a cent per minute. In September of 2021, it is $0.002 USD, which comes out to just under $90 a month if you have an automation that runs 24/7. For the purposes of this chapter, we will stay well under 500 minutes.

In addition to the Azure subscription, you will also need a server running Windows Server 2012 R2 or greater with TCP port 443 access to the internet. This server will execute scripts from Azure in your on-premises environment.

Since this chapter deals with hybrid scenarios, there will be different places you will need to execute different code snippets and scripts. Also, some of it will require PowerShell 5.1 as Azure Automation does not fully PowerShell 7 yet.

To help keep everything straight, I will include callouts for which environment and version of PowerShell to use. For this chapter, there will be three different environments.

1. Any device with PowerShell 7 installed. Most likely, the one you have used throughout this book.
2. The server that you will connect to Azure for the hybrid scenario. All snippets for this will use PowerShell 5.1 and ISE.
3. The script editor in the Azure Automation portal.

The first two can be the same device, but make sure to pay attention to when you need to use Windows PowerShell 5.1 versus PowerShell 7.

8.2 Setting up Azure Automation

Setting up an Azure Automation account can be as simple as going to the Azure portal, click Create Resource, then running through the wizard to create a new automation account. This approach will work if you plan on only doing cloud-based automations. However, if you want to create hybrid runbook workers, you need to do a few more things.

The basic organizational structure for those not familiar with Azure is a single Azure Active Directory (Azure AD) Tenant, with subscriptions and resources underneath it. The tenant is where all your user accounts, groups, and service principal are stored.

Subscriptions are where you create Azure resources. All Azure consumption is billed to the Subscription level. You can have multiple subscriptions under one tenant.

Next are the Resource Groups. Resources like Azure Automation accounts or Virtual Machines cannot be added directly to a subscription. Instead, they are added to Resource Groups. Resource Groups allow you to group your individual resources into logical collections. They also help with setting permissions since all Azure resources inherit permissions down, and there is no way to block inheritance or create deny permissions.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

For this chapter, we will create a brand-new Resource Group so you can keep everything separate from any other Azure resources you may have. Then you will create the required Azure resources. For this automation, we will create the following resources.

- **Azure Automation Account** – This is the automation platform that will hold and execute your scripts.
- **Log Analytics Workspace** – Provides the agent for creating a hybrid runbook worker for executing Azure Automation scripts in your on-premises environment.
- **Storage Account** – This Will be used to store the files uploaded from the automation.

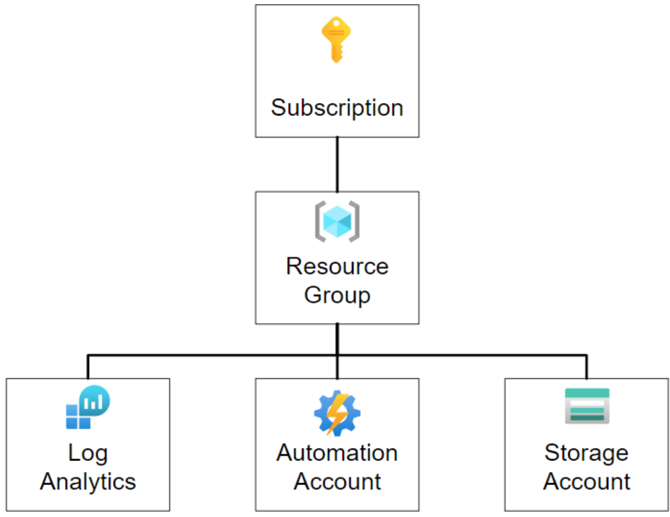


Figure 8.1 Azure resources for the storage upload automation

8.2.1 Azure Automation

As mentioned previously, Azure Automation is a cloud-based automation platform with the ability to run automations in the cloud, on-premises, or other clouds. The automations are created as runbooks. A runbook can be a script written in PowerShell 5.1, PowerShell Workflow, Python 2, or Python 3. In addition, there are also graphical runbooks that allow you to drag and drop different actions. For our automation, we will be creating a PowerShell runbook.

PowerShell 7

As of now (September 2021), all PowerShell runbooks are running with PowerShell 5.1, but PowerShell 7 support is in development. Unfortunately, it is not yet in public or even private preview, so all runbooks referenced in this chapter are Windows PowerShell 5.1.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

When you execute a runbook in Azure Automation, you can choose to execute it in Azure or on a hybrid runbook worker. The hybrid runbook worker is what allows Azure automation to access resources in your on-premises environment. When you execute a runbook on a hybrid runbook worker, the runbook is delivered to the machine for execution. This allows you one central place to store and maintain your scripts, regardless of where they need to run.

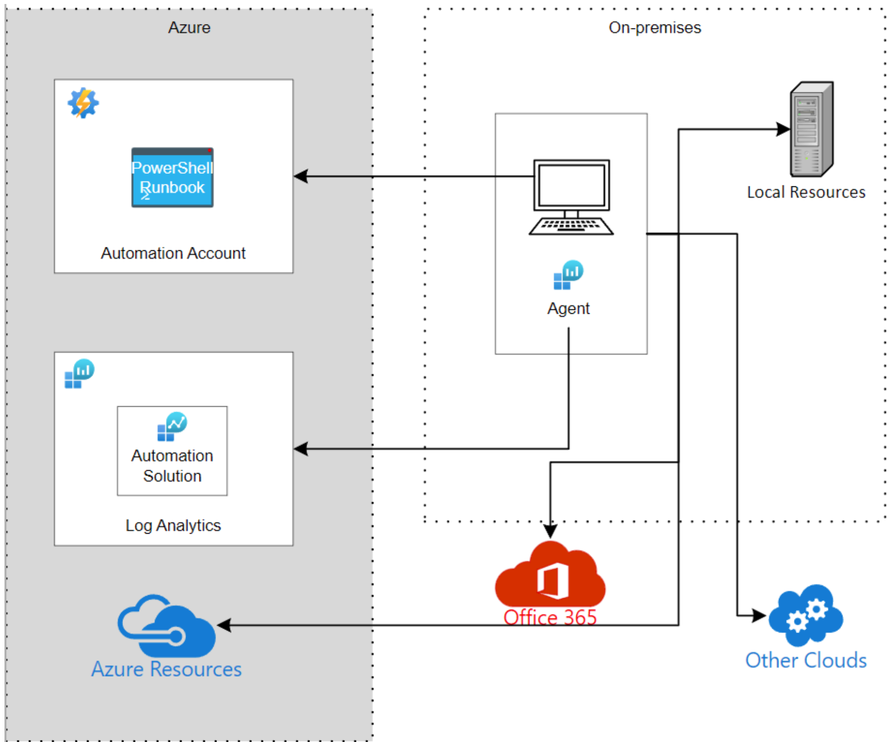


Figure 8.2 Automation account hybrid runbook worker automation flow

Azure Automation also has the ability to store different assets for your runbooks to use. These include modules, variables, credentials, and certificates.

8.2.2 Log Analytics

A lot of the functionality in Azure Automation is tightly integrated with Log Analytics, including the hybrid runbook workers. To make a server a hybrid runbook worker, the Microsoft Monitoring Agent (MMA) must be installed and set to a Log Analytics workspace with the Azure Automation solution added.

Log Analytics is a monitoring and data collection tool in Azure. Most services in Azure have the ability to send diagnostic and metric logs to Log Analytics. You can then search

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

these logs using Kusto queries and create alerts based on query results. You can also send logs from on-premises or VMs in other clouds via the MMA. When MMA communicates with a Log Analytics workspace with the Automation solution added, it will download the files you need to make the server a hybrid runbook worker.

The process to set up Azure Automation and add a server a hybrid runbook worker is as follows:

1. Create Azure Automation account
2. Create Log Analytics workspace
3. Add the Automation Solution to the workspace
4. Install the MMA on an on-premises server
5. Connect MMA to the Log Analytics workspace
6. MMA will download the hybrid runbook worker files and PowerShell module
7. Use the hybrid runbook worker module to connect to the Azure Automation account

Even if you are not planning to use hybrid runbook workers, setting a Log Analytics workspace is a good idea because you can forward the logs from your automations to it. This will give you the ability to search your job executions, outputs, and errors easily. Plus, it will allow you to create alerts for failed jobs or errors.

8.2.3 Creating Azure resources

Before you can do anything, you must first install the Azure PowerShell modules and connect to your Azure subscription.

There are several methods for connecting to Azure via PowerShell. You can have it prompt you to authenticate. You can pass a credential object to the command or use a service principal and certificate. For now, we just want to install the Azure modules and connect by having it prompt you.

To support this automation, you will need to create a Resource Group, Azure Automation account, Log Analytics workspace, and an Azure storage account.

Different resources have different naming requirements. For most Azure resources, the name can contain only letters, numbers, and hyphens. In addition, the name must start with a letter, and it must end with a letter or a number. And most have a length limit. For example, Log Analytics must be between 4-63 characters, and Azure Automation is 6-50. However, storage accounts present their own unique naming requirements.

A storage account's name must be 3 to 24 characters long and contain **only lowercase** letters and numbers. And most importantly, the name must be unique across all existing storage account names in Azure. Not your Azure but all of Azure. So, a little trick I've developed is to add a timestamp to the end of the name. Also, since all of these resources are being used for the same automation, you can give them all the same name. That is as long as the name meets the requirements of the resource with the most strict standard.

You will also need to choose a region when setting up Azure resources. Some resources are only available in certain regions, and some have regional dependencies when interacting with other resources. For example, to link your Azure Automation account to a Log Analytics workspace, they need to be in compatible regions. Generally, this means that they are in the same region, but there are exceptions to this rule. For instance, you can only link Log

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

Analytics in EastUS to Azure Automation in EastUS2. You can see in the table below a listing of compatible regions.

Table 8.1 Supported regions for linked Automation accounts and Log Analytics workspace

Log Analytics workspace region	Azure Automation region
AustraliaEast	AustraliaEast
AustraliaSoutheast	AustraliaSoutheast
BrazilSouth	BrazilSouth
CanadaCentral	CanadaCentral
CentralIndia	CentralIndia
CentralUS	CentralUS
ChinaEast2	ChinaEast2
EastAsia	EastAsia
EastUS2	EastUS
EastUS	EastUS2
FranceCentral	FranceCentral
JapanEast	JapanEast
KoreaCentral	KoreaCentral
NorthCentralUS	NorthCentralUS
NorthEurope	NorthEurope
NorwayEast	NorwayEast
SouthCentralUS	SouthCentralUS
SoutheastAsia	SoutheastAsia
SwitzerlandNorth	SwitzerlandNorth
UAENorth	UAENorth
UKSouth	UKSouth
USGovArizona	USGovArizona
USGovVirginia	USGovVirginia
WestCentralUS	WestCentralUS
WestEurope	WestEurope
WestUS	WestUS
WestUS2	WestUS2

For the complete list of supported mappings, refer to the Microsoft docs.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

ENVIRONMENT All code snippets in this section are using PowerShell 7.

Go ahead and open a PowerShell console or VS Code and install the required Azure PowerShell modules and import them to your local session.

```
Install-Module -Name Az
Install-Module -Name Az.MonitoringSolutions
Import-Module -Name Az,Az.MonitoringSolutions
```

Next set the variables to use throughout this section. Feel free to change the values to whatever naming conventions you would like to use.

```
$SubscriptionId = 'The GUID of your Azure subscription'
$DateString = (Get-Date).ToString('yyMMddHHmm')
$ResourceGroupName = 'PoshAutomate'
$WorkspaceName = 'poshauto' + $DateString
$AutomationAccountName = 'poshauto' + $DateString
$StorageAccountName = 'poshauto' + $DateString
$AutomationLocation = 'SouthCentralUS'
$WorkspaceLocation = 'SouthCentralUS'
```

Then connect to your subscription.

```
Connect-AzAccount -Subscription $SubscriptionId
```

Now that you are connected to your Azure subscription, you can create your resources, starting with the resource group.

```
New-AzResourceGroup -Name $ResourceGroupName -Location $AutomationLocation
```

Then you can create the Log Analytics workspace, Azure Automation account, and Storage account inside the resource group.

```
$WorkspaceParams = @{
       ResourceGroupName = $ResourceGroupName
       Name              = $WorkspaceName
       Location          = $WorkspaceLocation
}
New-AzOperationalInsightsWorkspace @WorkspaceParams

$AzAutomationAccount = @{
       ResourceGroupName = $ResourceGroupName
       Name              = $AutomationAccountName
       Location          = $AutomationLocation
       Plan              = 'Basic'
}
New-AzAutomationAccount @AzAutomationAccount

$AzStorageAccount = @{
       ResourceGroupName = $ResourceGroupName
       AccountName       = $StorageAccountName
       Location          = $AutomationLocation
       SkuName           = 'Standard_LRS'
       AccessTier        = 'Cool'
}
New-AzStorageAccount @AzStorageAccount
```

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

You will also need to add the Azure Automation solution to the Log Analytics workspace. This is what will allow you to create hybrid runbook workers.

```
$WorkspaceParams = @{
       ResourceGroupName = $ResourceGroupName
       Name              = $WorkspaceName
}
$workspace = Get-AzOperationalInsightsWorkspace @WorkspaceParams

$AzMonitorLogAnalyticsSolution = @{
       Type                = 'AzureAutomation'
       ResourceGroupName   = $ResourceGroupName
       Location            = $workspace.Location
       WorkspaceResourceId = $workspace.ResourceId
}
New-AzMonitorLogAnalyticsSolution @AzMonitorLogAnalyticsSolution
```

8.2.4 Authentication from automation runbooks

When you execute a runbook in Azure Automation it does not by default have access to any other Azure resources. When it executes it on a hybrid runbook worker, it has access to the local system, but that is it. If you need your runbook to connect to different Azure resources or other local systems, you must set it up to do so.

To access Azure-based resources, you can create a managed identity. A managed identity is an Azure AD object that you can assign to the automation account. Then, when your automation executes, it can run under the context of this managed identity. You can think of this as the equivalent of setting the run as account in a scheduled task or cron job from chapter 3.

You can then give this identity permissions to any Azure resources just as you would with any user account. The best part about it is there are no passwords or secrets required. You just assign it to the automation account, and your runbooks can use it. We will discuss the security implications of this later in this chapter.

We will create a system-assigned managed identity for our automation and give it contributor access to the storage account.

```
$AzStorageAccount = @{
       ResourceGroupName = $ResourceGroupName
       AccountName       = $StorageAccountName
}
$storage = Get-AzStorageAccount @AzStorageAccount

$AzAutomationAccount = @{
       ResourceGroupName     = $ResourceGroupName
       AutomationAccountName = $AutomationAccountName
       AssignSystemIdentity  = $true
}
$Identity = Set-AzAutomationAccount @AzAutomationAccount

$AzRoleAssignment = @{
       ObjectId           = $Identity.Identity.PrincipalId
       Scope              = $storage.Id
       RoleDefinitionName = "Contributor"
}
New-AzRoleAssignment @AzRoleAssignment
```

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

8.2.5 Resource keys

Now that you have all the resources created, you need to get the variable and keys required to connect your local server to the Log Analytics workspace and Azure Automation account to make it a hybrid runbook worker. You will need the Log Analytics workspace ID and key and the Automation account's URL and key. Luckily, this is something we can get via PowerShell. After running this command, save the output, as you will need it to set up the hybrid runbook worker in the next section.

```
$InsightsWorkspace = @{
       ResourceGroupName = $ResourceGroupName
       Name              = $WorkspaceName
}
$Workspace = Get-AzOperationalInsightsWorkspace @InsightsWorkspace

$WorkspaceSharedKey = @{
       ResourceGroupName = $ResourceGroupName
       Name              = $WorkspaceName
}
$WorspaceKeys = Get-AzOperationalInsightsWorkspaceSharedKey @WorkspaceSharedKey

$AzAutomationRegistrationInfo = @{
       ResourceGroupName     = $ResourceGroupName
       AutomationAccountName = $AutomationAccountName
}
$AutomationReg = Get-AzAutomationRegistrationInfo @AzAutomationRegistrationInfo
@"
`$WorkspaceID = '$($Workspace.CustomerId)'
`$WorkSpaceKey = '$($WorspaceKeys.PrimarySharedKey)'
`$AutoURL = '$($AutomationReg.Endpoint)'
`$AutoKey = '$($AutomationReg.PrimaryKey)'
"@
```

8.3 Creating a hybrid runbook worker

Now that you have all the Azure resources created, you can create a hybrid runbook worker that will allow you to execute runbooks locally. To do this, you need to install the Microsoft Monitoring Agent (MMA), link it to the Log Analytics workspace, and then link it to the Automation account.

Hybrid runbook workers can be Linux or Windows servers, but the instructions below are for a Windows-based server to not overcomplicate things. For Linux, the process is the same but with different commands.

> **ENVIRONMENT** All code listings in this section must be run on the server you are making a hybrid runbook worker and will use Windows PowerShell 5.1.

On the server, you want to make a hybrid runbook worker, open PowerShell ISE as an administrator. Then, add the workspace ID and key to the listing below to install the Log Analytics agent using PowerShell.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

Listing 8.1 Install Microsoft Monitoring Agent

```
$WorkspaceID = 'YourId'    #A
$WorkSpaceKey = 'YourKey'

$agentURL = 'https://download.microsoft.com/download' +    #B
    '/3/c/d/3cd6f5b3-3fbe-43c0-88e0-8256d02db5b7/MMASetup-AMD64.exe'

$FileName = Split-Path $agentURL -Leaf    #C
$MMAFile = Join-Path -Path $env:Temp -ChildPath $FileName
Invoke-WebRequest -Uri $agentURL -OutFile $MMAFile | Out-Null

$ArgumentList = '/C:"setup.exe /qn ' +    #D
    'ADD_OPINSIGHTS_WORKSPACE=0 ' +
    'AcceptEndUserLicenseAgreement=1"'
$Install = @{
    FilePath     = $MMAFile
    ArgumentList = $ArgumentList
    ErrorAction  = 'Stop'
}
Start-Process @Install -Wait | Out-Null

$Object = @{    #E
    ComObject = 'AgentConfigManager.MgmtSvcCfg'
}
$AgentCfg = New-Object @Object

$AgentCfg.AddCloudWorkspace($WorkspaceID,    #F
    $WorkspaceKey)

Restart-Service HealthService    #G
```

#A Set the parameters for your workspace
#B URL for the agent installer
#C Download the agent
#D Install the agent
#E Load the agent config com object
#F Set the workspace ID and key
#G Restart the agent for the changes to take effect

Once the install completes, you will need to wait a couple of minutes for it to perform the initial sync. This initial sync will download the hybrid runbook worker files, allowing you to connect the local server to Azure Automation.

When creating hybrid runbook workers, you need to include a group name. The group allows you to create a pool of workers for load-balancing and high availability. However, you have no control over which individual server in that group will execute a script. Therefore, if you need to execute a script on a specific server, you can use PowerShell remoting or put it in a hybrid worker group on its own.

In our scenario where we are copying a file from the local system to the cloud, it would make sense to create a single hybrid runbook worker, so it can just go from the local system to the cloud. However, suppose the files are saved to a network share. In that case, you could create multiple Hybrid Runbooks Workers in a single group where anyone could execute the runbook.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

To create your hybrid runbook worker, add the Automation URL and key you retrieved earlier and run the listing below to link the local server with your Automation account.

Listing 8.2 Create Hybrid Runbook Worker

```
$AutoUrl = ''    #A
$AutoKey = ''
$Group   = $env:COMPUTERNAME

$Path = 'HKLM:\SOFTWARE\Microsoft\System Center ' +    #B
    'Operations Manager\12\Setup\Agent'
$installPath = Get-ItemProperty -Path $Path |
    Select-Object -ExpandProperty InstallDirectory
$AutomationFolder = Join-Path $installPath 'AzureAutomation'

$ChildItem = @{    #C
    Path    = $AutomationFolder
    Recurse = $true
    Include = 'HybridRegistration.psd1'
}
Get-ChildItem @ChildItem | Select-Object -ExpandProperty FullName

Import-Module $modulePath    #D

$HybridRunbookWorker = @{    #E
    Url       = $AutoUrl
    key       = $AutoKey
    GroupName = $Group
}
Add-HybridRunbookWorker @HybridRunbookWorker
```

#A Set the parameters for your Automation Account
#B Find the directory the agent was installed in
#C Search the folder for the HybridRegistration module
#D Import the HybridRegistration module
#E Register the local machine with the automation account

8.3.1 PowerShell modules on hybrid runbook workers

When building your runbooks for use in Azure Automation, you have to be mindful of your module dependencies, especially when executing on a hybrid runbook worker.

You can import modules to your automation account directly from the PowerShell gallery and even upload your own custom ones. These modules are automatically imported when your runbook executes in Azure. However, these modules do not transfer to your hybrid runbook workers. Therefore, you will need to manually ensure that you have the correct module versions installed on the hybrid runbook workers.

For this automation, you will be uploading a file from the local file system to Azure Blob. Therefore, you will need the `Az.Storage` module installed on the hybrid runbook worker. This module is part of the default Az PowerShell module installation.

Also, when installing modules on the hybrid runbook worker, you must scope them to install for All Users. Otherwise, your automation may not be able to find them.

```
Install-Module -Name Az -Scope AllUsers
```

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

8.4 Creating a PowerShell runbook

When creating a PowerShell runbook, you have two options. You can write the script locally using an IDE like VS Code or ISE and import it to the automation account, or you can write it directly in the portal using the runbook editor. The advantage of the local development is testing is much easier and quicker. The disadvantage is there are some commands unique to Azure Automation for importing assets like credentials and variables. For this reason, I recommend using a hybrid approach. You can develop the script locally to get all the functionality working, then transfer it to Azure Automation, and update it to use any assets you need.

To see how this works, you will build the script for uploading the archive files to Azure Blob storage locally. Then import it to the automation account and update it to use the managed identity.

Since the purpose of this chapter is to show you how to use Azure Automation and not all the ins and outs of Azure Blob storage, we are going to keep the script relatively simple. It will check a folder for ZIP files, upload them to the Azure Blob, then delete them off the local system. I have also included the script **New-TestArchiveFile.ps1** in the *Helper Scripts* for this chapter that you can use to make some test ZIP files.

As with any cloud-based automation, the first thing we need to do is connect to the account. For testing, you can use the `Connect-AzAccount` cmdlet without any credentials or other authentication parameters. This will cause it to prompt you to authenticate manually. You will change this to automate the authentication once it is imported as a runbook.

To upload to Azure storage, you must create a context object using a storage account key. The Azure Storage cmdlets use this context to authenticate with the storage account. You can create all sorts of different keys with different permissions. By default, there will be two keys with full permissions. We will use one of these for this automation.

Next, you need a container to put the files in, similar to a folder on the local file system. You can use different containers for organizing files and also for controlling access for different files. Then finally, it will check if the file already exists in Azure, and if not, it will upload it, then delete it.

You may notice here we are not doing any checks before deleting the file. That is because the `Set-AzStorageBlobContent` cmdlet has built-in hash checks. Therefore, if something goes wrong during the upload process, it will return an error. You can then stop the removal by having the execution terminate if there is an error.

> **ENVIRONMENT** The listing below should be created and tested on the hybrid runbook worker using Windows PowerShell 5.1.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

Listing 8.3 Upload ZIP files to Azure Blob

```
$FolderPath = 'L:\Archives'    #A
$Container = 'devtest'

$ResourceGroupName = 'PoshAutomate'    #B
$StorageAccountName = ''
$SubscriptionID = ''

Connect-AzAccount    #C
Set-AzContext -Subscription $SubscriptionID

$ChildItem = @{    #D
    Path   = $FolderPath
    Filter = '*.zip'
}
$ZipFiles = Get-ChildItem @ChildItem

$AzStorageAccountKey = @{    #E
    ResourceGroupName = $ResourceGroupName
    Name              = $StorageAccountName
}
$Keys = Get-AzStorageAccountKey @AzStorageAccountKey
$AzStorageContext = @{
    StorageAccountName = $StorageAccountName
    StorageAccountKey  = $Keys[0].Value
}
$Context = New-AzStorageContext @AzStorageContext

$AzStorageContainer = @{    #F
    Name        = $Container
    Context     = $Context
    ErrorAction = 'SilentlyContinue'
}
$containerCheck = Get-AzStorageContainer @AzStorageContainer
if(-not $containerCheck){
    $AzStorageContainer = @{
        Name        = $Container
        Context     = $Context
        ErrorAction = 'Stop'
    }
    New-AzStorageContainer @AzStorageContainer| Out-Null
}

foreach($file in $ZipFiles){
    $AzStorageBlob = @{    #G
        Container   = $container
        Blob        = $file.Name
        Context     = $Context
        ErrorAction = 'SilentlyContinue'
    }
    $blobCheck = Get-AzStorageBlob @AzStorageBlob
    if (-not $blobCheck) {
        $AzStorageBlobContent = @{    #H
            File        = $file.FullName
            Container   = $Container
            Blob        = $file.Name
            Context     = $Context
            Force       = $true
```

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
            ErrorAction = 'Stop'
        }
        Set-AzStorageBlobContent @AzStorageBlobContent
        Remove-Item -Path $file.FullName -Force
    }
}
```

#A Set the local variables
#B Set the Azure Storage Variables
#C Connect to Azure
#D Get all the ZIP files in the folder
#E Get the storage keys and create a context object that will be used to authenticate with the storage account
#F Check to see if the container exists. If it does not, create it.
#G Check if the file already exists in the container. If not, upload it, then delete it from the local server.
#H Upload the file to the Azure storage

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

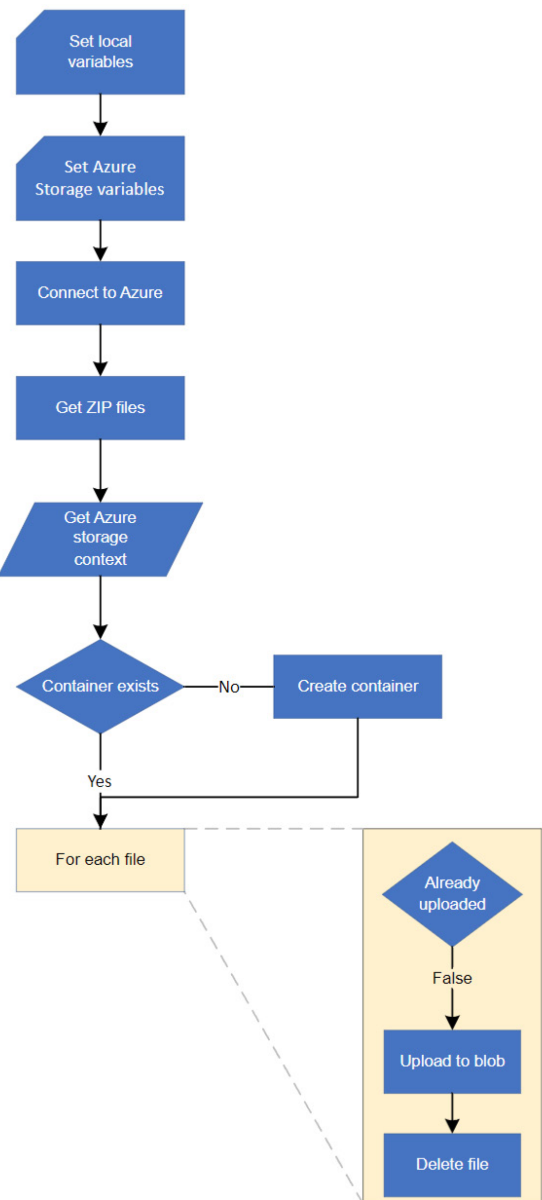


Figure 8.3 ZIP file upload process when running locally

After successfully testing the script, you can save it as a ps1 file and upload it to Azure Automation.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
$AzAutomationRunbook = @{
       Path                  = 'C:\Path\Upload-ZipToBlob.ps1'
       ResourceGroupName     = $ResourceGroupName
       AutomationAccountName = $AutomationAccountName
       Type                  = 'PowerShell'
       Name                  = 'Upload-ZipToBlob'
       Force                 = $true
}
$import = Import-AzAutomationRunbook @AzAutomationRunbook
```

Go ahead and run the **New-TestArchiveFile.ps1** script again on the hybrid runbook worker to make a new ZIP file for testing through Azure.

8.4.1 Automation assets

When testing the zip file upload script, we hardcoded the values required for connecting to the storage account. Instead of hardcoding those values into the script, you can convert them to variables in the Azure Automation account.

Making them variables gives you the ability to share them amongst other runbooks in the same account. This is great for things like the storage account and subscription IDs where you could have multiple different automations using the same resource. Then, if you later change the storage account or want to reuse this same runbook in another subscription, all you need to do is update the variables. But at the same time, this can be a double-edged sword because changing one variable can affect multiple different runbooks.

Also, don't be afraid of using descriptive names for your variable. It will save you a lot of guessing in the long run. For example, for this automation, instead of making a variable named ResourceGroup, make it descriptive, like ZipStorage_ResourceGroup. This way, you know it is the resource group for the Zip Storage automation.

Azure Automation also provides the ability to encrypt your variables. You can do this in the portal or through PowerShell by setting the Encrypted argument to true. As we will discuss later with the security considerations, there are different permission levels in Azure Automation. Therefore, if someone has read access to the automation account, they can see the value of unencrypted variables in plain text. While variables may not be as sensitive as passwords, they can hold sensitive data like connection strings or API keys. So, to prevent accidentally having sensitive data in plain text, I recommend encrypted all variables.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

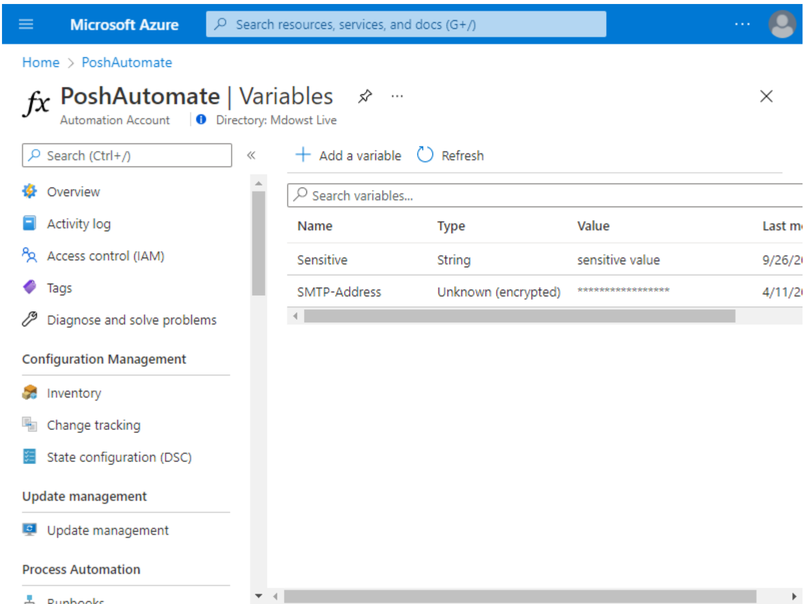


Figure 8.4 Encrypted and Unencrypted automation variables

Go ahead and create variables for the subscription, resource group, and storage account.

```
$AutoAcct = @{
       ResourceGroupName     = $ResourceGroupName
       AutomationAccountName = $AutomationAccountName
       Encrypted             = $true
}
$Variable = @{
       Name  = 'ZipStorage_AccountName'
       Value = $StorageAccountName
}
New-AzAutomationVariable @AutoAcct @Variable

$Variable = @{
    Name  = 'ZipStorage_Subscription'
    Value = $SubscriptionID
}
New-AzAutomationVariable @AutoAcct @Variable

$Variable = @{
    Name  = 'ZipStorage_ResourceGroup'
    Value = $ResourceGroupName
}
New-AzAutomationVariable @AutoAcct @Variable
```

8.4.2 Runbook editor

Now that you have your variables created, you need to tell your script about them. To do this, navigate to the Azure portal and your automation account. Select Runbooks from the

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

left menu, then click on the runbook named **Upload-ZipToBlob**. When the runbook loads, click the *Edit* button to open the runbook editor.

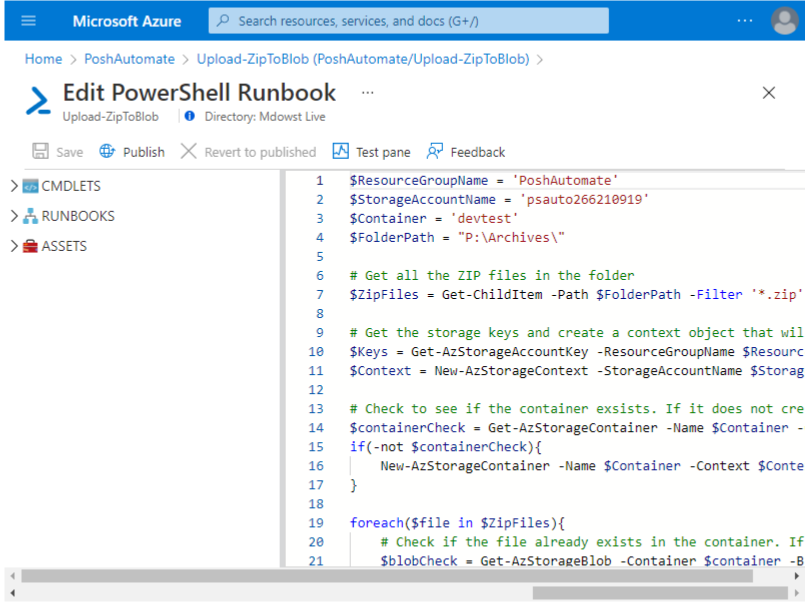


Figure 8.5 Azure Automation runbook editor

This runbook editor works just like any other IDE or development editor. It has auto-complete suggestions and syntax highlighting. But on top of that, it can insert assets stored in the automation account. If you expand *Assets* > *Variables* in the left menu, you will see the variables you just created.

> **ENVIRONMENT** **All remaining code snippets and listings are using the Runbook Editor in Azure Automation.**

Starting with the `$SubscriptionID` variable, remove the value set in the script so that the line is just: `$SubscriptionID =`

Place the cursor after the equals sign, click on the ellipsis next to `ZipStorage_SubscriptionID`, and select *Add "Get Variable" to canvas*. The line should now look like this:

```
$SubscriptionID = Get-AutomationVariable -Name 'ZipStorage_SubscriptionID'
```

Repeat this same process with the resource group and storage account name variable. Next, you can convert the `$FolderPath` and `$Container` variables to parameters.

One final thing to do is to have the script authenticate with Azure. Since you are using a managed identity, there is no need to add credential assets or pass secrets to the runbook.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

Instead, you can connect to Azure as the managed identity by adding the `-Identity` switch to the `Connect-AzAccount` cmdlet.

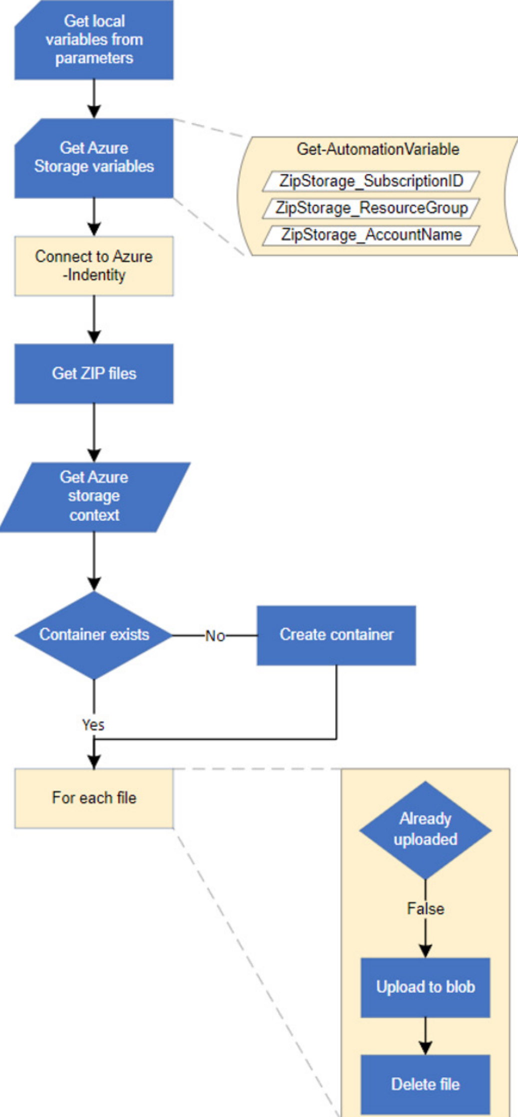


Figure 8.6 ZIP file upload process when running through automation hybrid worker runbook

Now the final version of the runbook should look like the listing below.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

Listing 8.4 Upload-ZipToBlob

```
param(
    [Parameter(Mandatory = $true)]
    [string]$FolderPath,
    [Parameter(Mandatory = $true)]
    [string]$Container
)

$SubscriptionID = Get-AutomationVariable `    #A
    -Name 'ZipStorage_SubscriptionID'
$ResourceGroupName = Get-AutomationVariable -Name 'ZipStorage_ResourceGroup'
$StorageAccountName = Get-AutomationVariable -Name 'ZipStorage_AccountName'

Connect-AzAccount -Identity    #B
Set-AzContext -Subscription $SubscriptionID

$ChildItem = @{    #C
    Path   = $FolderPath
    Filter = '*.zip'
}
$ZipFiles = Get-ChildItem @ChildItem

$AzStorageAccountKey = @{    #D
    ResourceGroupName = $ResourceGroupName
    Name              = $StorageAccountName
}
$Keys = Get-AzStorageAccountKey @AzStorageAccountKey
$AzStorageContext = @{
    StorageAccountName = $StorageAccountName
    StorageAccountKey  = $Keys[0].Value
}
$Context = New-AzStorageContext @AzStorageContext

$AzStorageContainer = @{    #E
    Name        = $Container
    Context     = $Context
    ErrorAction = 'SilentlyContinue'
}
$containerCheck = Get-AzStorageContainer @AzStorageContainer
if(-not $containerCheck){
    $AzStorageContainer = @{
        Name        = $Container
        Context     = $Context
        ErrorAction = 'Stop'
    }
    New-AzStorageContainer @AzStorageContainer| Out-Null
}

foreach($file in $ZipFiles){
    $AzStorageBlob = @{    #F
        Container   = $container
        Blob        = $file.Name
        Context     = $Context
        ErrorAction = 'SilentlyContinue'
    }
    $blobCheck = Get-AzStorageBlob @AzStorageBlob
    if (-not $blobCheck) {
        $AzStorageBlobContent = @{    #G
```

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

```
            File        = $file.FullName
            Container   = $Container
            Blob        = $file.Name
            Context     = $Context
            Force       = $true
            ErrorAction = 'Stop'
        }
        Set-AzStorageBlobContent @AzStorageBlobContent
        Remove-Item -Path $file.FullName -Force
    }
}
```

#A Get the Azure Storage Variables
#B Connect to Azure
#C Get all the ZIP files in the folder
#D Get the storage keys and create a context object that will be used to authenticate with the storage account
#E Check to see if the container exists. If it does not, create it.
#F Check if the file already exists in the container. If not, upload it, then delete it from the local server.
#G Upload the file to the Azure storage

Once you have the updates done to the script, it is time to test it. Click on the *Test Pane* button. In the test pane, you can test the execution of the script. First, enter the values for the folder path and container name parameters. Then under the *Run Settings,* select Hybrid Worker and select the server you set up earlier. Then click *Start*.

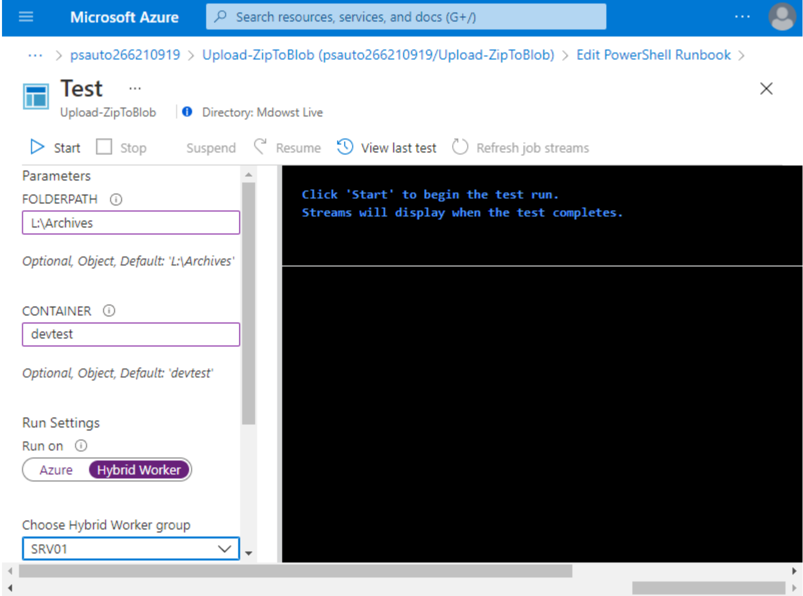


Figure 8.7 Runbook editor test pane

Since you selected the hybrid runbook worker, it will execute on that server and should behave just as it did when you ran it earlier.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

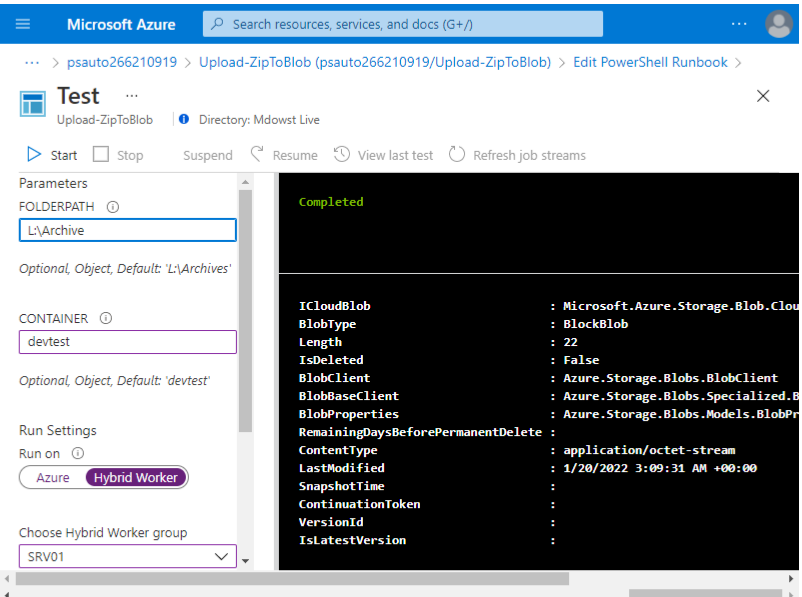


Figure 8.8 Runbook test results

If the test runs successfully, check the storage account to confirm the file was uploaded.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

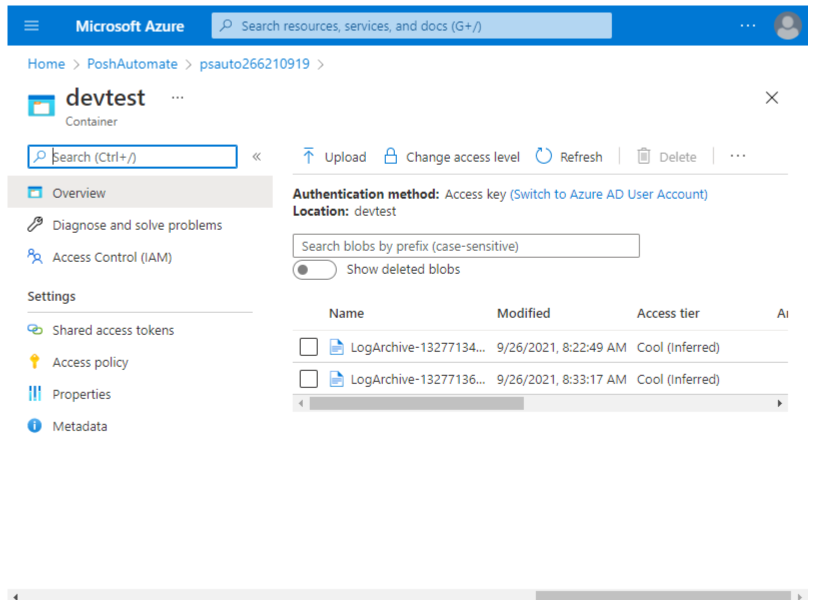


Figure 8.9 ZIP in Azure Storage Blob

Then back in the runbook editor, close the test pane, and click Publish. When you do this, your runbook is available for anyone with permission to execute it. From this main runbook screen, you can manually execute a runbook or create a recurring schedule for it. You can also make additional edits and use the tester without affecting the published version until you hit the Publish button again.

Go ahead and run the ***New-TestArchiveFile.ps1*** on the hybrid runbook worker server to make a new test file. Then from the *Upload-ZipToBlob* runbook in the Azure portal, click *Start*. Then enter the same parameter values, select your hybrid worker, and click *OK*.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

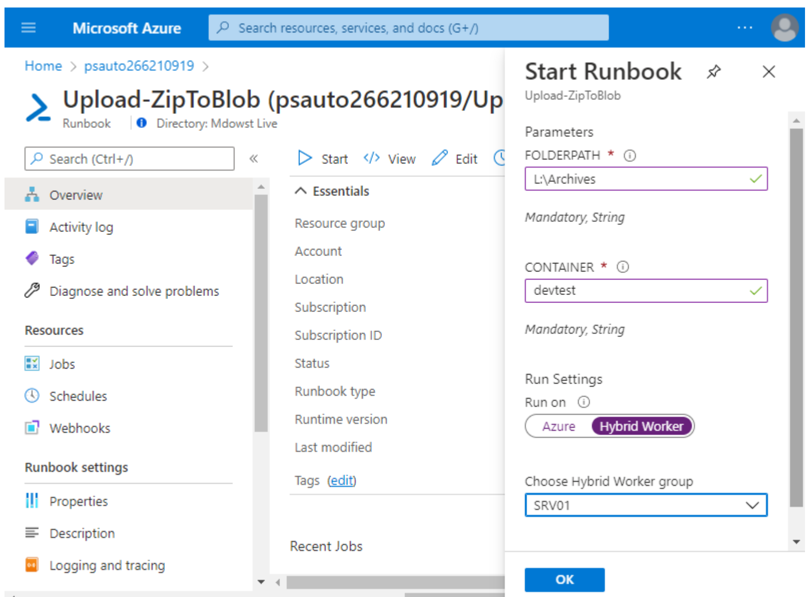


Figure 8.10 Starting an Azure Automation Runbook with parameters and on a hybrid runbook worker

This will execute the runbook and open the job screen. From the job screen, you can view all the execution details, including the hybrid worker used, the parameters, the output, and any errors or warnings. This data is retained in the portal for 30 days after execution.

Licensed to Tausif Khan <tsfahmed2@gmail.com>

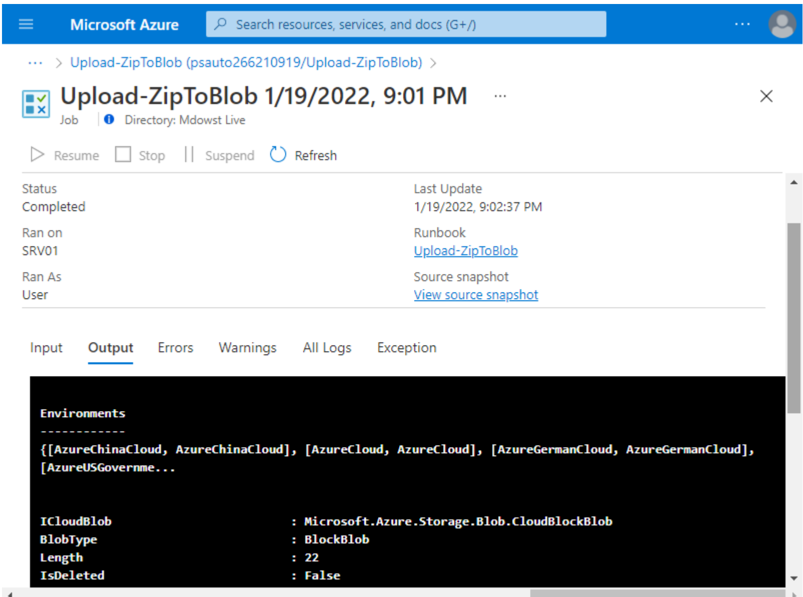


Figure 8.11 Azure Automation runbook output

8.4.3 Runbook output

I mentioned earlier that you can take any PowerShell script and import it to Azure Automation as a runbook. While this is true, there are a few commands that do not work in Azure Automation runbooks. Two of the most common ones are `Write-Host` and `Write-Progress`.

You can still include `Write-Host` and `Write-Progress` in your scripts; just know since there is no console, they will not produce any output. To show data in the Output section of the runbook, it must be written to the output stream using `Write-Output`. Therefore, it is usually best to convert any `Write-Host` commands to `Write-Output` when making a runbook. Since there is no other equivalent to `Write-Progress`, it would be best to remove them from the script.

Also, verbose logging is off by default. So, if you have any `Write-Verbose` commands in your script, it will not display in the logs unless you specifically turn on verbose logging in the runbook settings.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

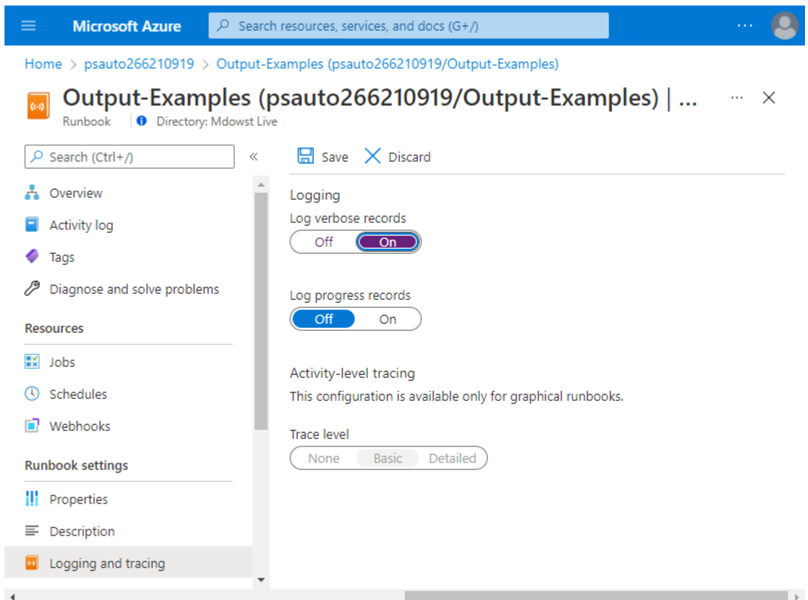


Figure 8.12 Enable verbose logging for an Azure Automation runbook

Once you do this, the verbose logs will show in the *All Logs* section of the runbook job. But be careful because verbose logging can slow down the execution of your runbooks, so only use it when needed, then turn it back off.

You can see how the different outputs work by importing the **Output-Examples.ps1** script, from the Helper Scripts folder, as a runbook and executing it.

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

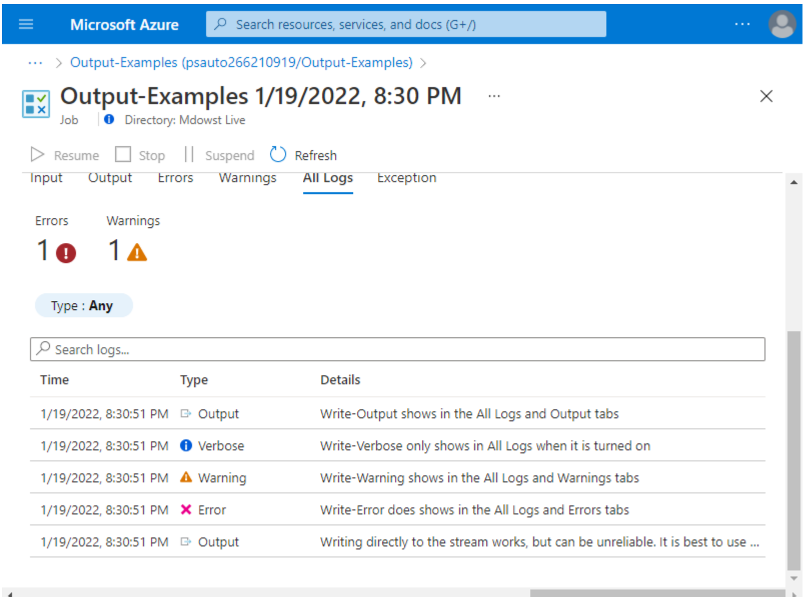


Figure 8.13 Azure Automation runbook output example

8.4.4 Interactive Cmdlets

As with most automation situations, you cannot have cmdlets that require interactive input in your runbooks. For example, `Read-Host` will not work because there is no way to interact with the runbook other than entering parameters.

You can also run into issues with cmdlets that have interactive logins. For example, the `Add-AzAccount` cmdlet will prompt for credentials if none are provided. The runbooks can often detect these and terminate the execution rather than hang and wait for input that will never come. However, this does not work all the time. So, to be safe, if you have a cmdlet that could prompt for user interaction, it is best to test and confirm that the way you are using it will not cause a user prompt to appear.

8.5 Security considerations

Like with any other automation platform, you need to be aware of the security implication of both what the automations can do and what data is stored in it. For instance, you just created a managed identity and assigned it permissions to a storage account. Now any person with permissions to create runbooks in that automation account can use that managed identity in their scripts. The same goes for any variables, credentials, or certificates you store in that account. However, there are a few things you can do to help protect your environment.

In Azure Automation, there are multiple different levels of permissions that you can assign to users. One of these is the Automation Operator. This role will allow someone to

©Manning Publications Co. To comment go to liveBook

Licensed to Tausif Khan <tsfahmed2@gmail.com>

execute a runbook, but they cannot create, edit, or delete one. A good use case for this is a runbook that performs maintenance tasks on a VM that requires elevated permissions. You can assign those permissions to the managed identity, then give the people who need to execute it the Automation Operator role. They will now only be able to perform what your runbook can perform and not edit or create any custom automations. Also, they don't even need permissions to the individual VMs as it will run as the managed identity.

The cloud offers the awesome ability to create new instances of items in seconds. Therefore, you should create a separate development Automation Account you can point to other development resources. Have your team build and test their runbooks there. Then once you are ready, you can move them to your production account with my stricter controls and permissions.

8.6 Summary

- Azure Automation can execute PowerShell scripts serverless in Azure or on a hybrid runbook worker in your local environment.
- Log Analytics with the Automation solution is required to support Hybrid runbook workers.
- You can securely store variables, credentials, and certificates in Azure Automation.
- You can develop and test your PowerShell scripts outside of Azure Automation, then import the ps1 file as a Runbook.
- If someone has permission to create a runbook in an Automation account, they have access to all the assets, including credentials and manage identities stored in that account.

©Manning Publications Co. To comment go to liveBook