Getting Started with Pester 5

A Beginner's Guide

Owen Heaume

Apress®

Getting Started with Pester 5

A Beginner's Guide

Owen Heaume

Apress®

Getting Started with Pester 5: A Beginner's Guide

Owen Heaume
West Sussex, UK

ISBN-13 (pbk): 979-8-8688-0305-5 ISBN-13 (electronic): 979-8-8688-0306-2
https://doi.org/10.1007/979-8-8688-0306-2

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Smriti Srivastava
Development Editor: Laura Berendson
Editorial Assistant: Kripa Joseph

Cover designed by eStudioCalamar
Cover image designed by Cover image from Pixabay

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, Suite 4600, New York, NY 10004-1562, USA. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub. For more detailed information, please visit https://www.apress.com/gp/services/source-code.

If disposing of this product, please recycle the paper

Table of Contents

About the Author

Owen Heaume is a seasoned PowerShell programmer with a passion for crafting efficient solutions in the dynamic landscapes of Intune and Azure. Having recently embarked on a professional journey in PowerShell programming for a prominent company within their automation team, Owen is dedicated to mastering the intricacies of Pester, Azure DevOps, and adhering to best practices.

Owen has published books on deploying applications in Intune using PowerShell, deploying applications in ConfigMgr using PowerShell, and deploying language and regional settings using ConfigMgr. In this book, Owen shares insights gained from real-world experiences, providing readers with practical knowledge and a glimpse into the mind of a multifaceted professional thriving in the realms of technology.

About the Technical Reviewer

Kasam Shaikh is a prominent figure in India's artificial intelligence (AI) landscape, holding the distinction of being one of the country's first four Microsoft Most Valuable Professionals (MVPs) in AI. Currently he serves as a senior architect. Kasam boasts an impressive track record as an author, having authored five best-selling books dedicated to Azure and AI technologies. Beyond his writing endeavors, Kasam is recognized as a Microsoft Certified Trainer (MCT) and an influential tech YouTuber (@mekasamshaikh). He also leads the largest online Azure AI community, known as DearAzure | Azure INDIA, and is a globally renowned AI speaker. His commitment to knowledge sharing extends to contributions to Microsoft Learn, where he plays a pivotal role.

Within the realm of AI, Kasam is a respected subject matter expert (SME) in generative AI for the cloud, complementing his role as a senior cloud architect. He actively promotes the adoption of No Code and Azure OpenAI solutions and possesses a strong foundation in Hybrid and Cross-Cloud practices. Kasam Shaikh's versatility and expertise make him an invaluable asset in the rapidly evolving landscape of technology, contributing significantly to the advancement of Azure and AI.

In summary, Kasam Shaikh is a multifaceted professional who excels in both technical expertise and knowledge dissemination. His contributions span writing, training, community leadership, public speaking, and architecture, establishing him as a true luminary in the world of Azure and AI. Kasam was recently awarded as top voice in AI by LinkedIn, making him the sole exclusive Indian professional acknowledged by both Microsoft and LinkedIn for his contributions to the world of artificial intelligence!

CHAPTER 1

Unveiling the Power of Pester

Welcome to the exciting world of Pester! In this chapter, we will embark on a journey to demystify the art of testing in PowerShell using the powerful tool called Pester. If you are new to PowerShell or testing frameworks, fear not; this guide is designed especially for you. Pester is not just any testing framework – it's your key to ensuring your PowerShell scripts are robust, reliable, and perform exactly as you intend.

In the upcoming sections, we will unravel the fundamentals of Pester. We'll begin by understanding what Pester is and why it holds a crucial place in the toolkit of every PowerShell developer. You'll learn about the compelling reasons to incorporate Pester into your scripting workflow, empowering you to write code with confidence.

We'll guide you through the installation process, ensuring you have the latest version of Pester ready to use.

But our exploration doesn't conclude there. In this chapter, we'll introduce you to the diverse spectrum of testing: unit tests, acceptance tests, and integration tests. These tests, much like different roles in a theatrical script, serve distinct purposes. Just as a director carefully selects actors for different scenes, you'll learn to choose the right test type for various scenarios in your coding journey. Whether you're validating individual components, interacting with real systems, or staging integrated functions, you'll have the tools to script tests that match your specific needs.

O. Heaume, *Getting Started with Pester 5*, https://doi.org/10.1007/979-8-8688-0306-2_1

We'll then navigate the intricate landscape of Pester files and their structure. By understanding the anatomy of these files, you'll lay a sturdy foundation for crafting well-organized and potent tests.

Whether you're a novice stepping into the PowerShell realm or a seasoned developer eager to refine your testing expertise, this chapter promises to furnish you with vital insights to kickstart your Pester journey.

What Is Pester?

Pester, in the realm of PowerShell, is not just a testing framework; it's a game-changer. Imagine having a reliable assistant by your side, carefully checking your PowerShell scripts for errors, ensuring they perform as expected, and providing you with the peace of mind that your code is robust. That assistant is Pester.

At its core, Pester is a testing framework specifically tailored for PowerShell. It's designed to simplify the process of writing and running tests for your PowerShell scripts and functions. But it's not just about spotting bugs; Pester encourages a mindset of proactive development. With Pester, you can validate your code's functionality, ensure it handles various scenarios, and confirm it responds correctly to different inputs.

Pester operates on a simple yet profound principle: **automated testing is your safety net**. It allows you to write tests that mimic real-world interactions with your scripts. By simulating different usage scenarios, input variations, and edge cases, you can be confident that your PowerShell code behaves as intended under diverse conditions.

Why Use Pester?

In the early days of my PowerShell journey, the notion of incorporating Pester into my workflow seemed like an unnecessary complication. I had already carefully crafted my scripts, ensuring they worked flawlessly on my

system. The prospect of investing more time in writing additional code for testing felt daunting. I questioned, "Why add this layer of complexity when my scripts were already running smoothly?"

What I didn't realize then was that Pester isn't just about finding bugs or ensuring basic functionality. It's a powerful ally that elevates your scripts from functional to exceptional. Here's why taking that initial step to embrace Pester can transform your PowerShell experience:

1. **Automated assurance:** Pester acts as your automated quality control mechanism. It tirelessly validates your code, freeing you from the burden of manual testing. With Pester, you can be confident that your scripts are always in top-notch condition, no matter how many times you modify them.

2. **Confidence in code changes:** As your scripts evolve, ensuring they remain stable becomes vital. Pester empowers you to confidently refactor and enhance your code. By running a suite of tests, you can instantly identify if any recent changes have unintended consequences, enabling you to catch issues before they escalate.

3. **Effective collaboration:** Imagine sharing your scripts with team members or contributors. Pester ensures that your code behaves consistently across different environments. It becomes the common language that bridges the gap between developers, fostering collaboration and collective progress.

4. **Saves time in the long run:** Initially, writing tests might seem like an additional effort, but it's an investment that pays off over time. Detecting and fixing issues early prevents potential disasters down

the line. The time saved by avoiding manual bug hunting far outweighs the time spent crafting tests.

5. **Quality documentation:** Pester tests serve as living documentation for your code. They provide clear examples of how your functions and modules are meant to be used. This documentation becomes invaluable, especially when revisiting your own code after a considerable time gap or when onboarding new team members.

6. **Proactive problem solving:** Pester doesn't just find problems; it anticipates them. By simulating various scenarios and inputs, you can proactively identify potential weaknesses in your code. Addressing these vulnerabilities before they manifest in real-world usage enhances the resilience of your scripts.

In essence, Pester isn't merely about testing; it's about empowering your scripts to reach their full potential. The beauty of Pester lies in its seamless integration with PowerShell. If you're already familiar with PowerShell, learning Pester is a natural next step in enhancing your scripting arsenal. Embracing Pester equips you with the tools to create robust, reliable, and maintainable PowerShell solutions. So, take the plunge, invest a bit of time now, and watch your scripts shine in the long run. Pester isn't just a testing framework; it's your ticket to PowerShell excellence.

Installing Pester

Installing Pester is a straightforward process, ensuring you have a robust testing environment for your PowerShell scripts. Pester can be installed on any Windows computer with PowerShell version 3 or higher, although it's advisable to use PowerShell version 5 or 7 for the best experience.

Microsoft recognized the power of Pester and included it by default in Windows 10 and 11. However, the default versions might not always be the latest, and updating them can be tricky. Here's the recommended method to ensure you have the most up-to-date and easily maintainable version of Pester installed.

1. **Removing the Preinstalled Version:** If you're dealing with an older version of Pester, it's best to remove it completely. Run the following script shown in Listing 1-1 in an administrative PowerShell window.

Listing 1-1. Uninstalling legacy Pester

```
   $module = "C:\Program Files
\WindowsPowerShell\Modules\Pester"
   takeown /F $module /A /R
   icacls $module /reset
   icacls $module /grant "*S-1-5-32-544:F"
/inheritance:d /T
```

   This script ensures the clean removal of the preinstalled Pester version, leaving your system ready for the latest installation.

2. **Installing the Latest Version:** With the old version removed, installing the latest Pester version is a breeze. Execute the following command in an administrative PowerShell window:

   ```
   Install-Module Pester -Force
   ```

This command fetches and installs the latest version of Pester, ensuring you have the most advanced features at your fingertips.

3. **Easy Updates:** Managing updates is now hassle-free. To update Pester, simply run the following command in an administrative PowerShell window:

   ```
   Update-Module Pester
   ```

   This one-liner keeps your Pester framework current, incorporating any improvements or bug fixes seamlessly.

By following these steps, you ensure that Pester is not just installed on your system, but it's the latest, most potent version, and ready to empower your PowerShell testing endeavors. Now, let's dive into understanding Pester's file structure and its core components.

Navigating the Testing Landscape in PowerShell: Test Types

Welcome to the realm of testing in PowerShell, where scripts transform into robust and dependable solutions. Developers fine-tune their scripts through a series of carefully planned tests, much like orchestrating a captivating theater performance. Understanding the diverse landscape of testing methodologies is akin to exploring the varied techniques in the world of theater, each designed to bring out the best in a production.

In this section, we embark on a journey through the fundamental pillars of testing: unit tests, acceptance tests, and integration tests. Each test type is a unique lens through which developers can scrutinize their code, ensuring it not only meets its functional requirements but also weathers the challenges of real-world execution.

I'll employ a theater analogy (throughout this book) alongside conventional explanations to simplify the topic. Understanding these test types can be a bit overwhelming at first, so likening them to elements in a theater production might make the concepts more digestible.

Unit Tests

Unit tests focus on individual components or "units" of your code, typically functions or cmdlets. A unit test evaluates a specific piece of functionality in isolation. For example, if you have a function that converts lowercase text to uppercase, a unit test for this function would provide it with specific input and check if the output matches the expected result. The goal is to validate that the function performs as intended in various scenarios. Unit tests are isolated from the broader system and do not rely on external resources or dependencies.

The Analogy: Precision on Stage

Unit tests are the fundamental building blocks, akin to rehearsing scenes with precision in a theater production. Just as each actor and prop must be scrutinized for readiness and quality, unit tests meticulously examine individual components of your PowerShell script. These tests isolate functions, methods, or cmdlets, subjecting them to rigorous evaluations to ensure they perform their designated tasks flawlessly.

Integration Tests

Integration tests assess the interactions between different components or systems within your script. In the context of PowerShell, integration tests frequently involve connecting to external resources, databases, APIs, or other modules.

Unlike unit tests, integration tests focus on how these components collaborate and whether they work correctly when integrated. For example, if your script communicates with a database, an integration test would verify that the script can successfully connect, retrieve data, and handle responses. Mocking, where certain components are simulated to mimic real behavior, is often used in integration testing to isolate different parts of the system.

This book centers on both unit tests and integration tests, with a dedicated exploration of mocking for our integration tests, a topic that will be covered in a later chapter.

The Analogy: Orchestrating Script Performances

Integration tests resonate with the orchestration of diverse roles in a theater production. In a play, actors collaborate to bring characters to life, each contributing a unique essence to the overall performance. Similarly, integration tests explore how different components of your script interact. Whether it's connecting with databases, APIs, or external services, these tests ensure that the script functions seamlessly in a connected environment, much like the synergy required among actors for a compelling stage production.

Acceptance Tests

Acceptance tests evaluate the overall behavior of your script within the real system, simulating user interactions or system operations. Unlike unit tests, acceptance tests are less concerned with the internal logic of individual functions and more focused on the script's end-to-end functionality. These tests often deal with real-world scenarios, covering the entire application workflow. However, acceptance tests might not cover all edge cases, as their purpose is to validate general system behavior rather than specific conditions.

The Analogy: The Grand Stage Performance

Acceptance tests are the final act on stage, equivalent to presenting the complete play. Here, the entire script is performed, mirroring the way a theater production reaches the audience. These tests focus on the end-to-end functionality, mimicking real-world scenarios and user interactions. Just as the audience evaluates a play based on its presentation, engagement, and impact, acceptance tests gauge your script's performance, ensuring it satisfies user requirements and expectations.

Each type of test serves a specific purpose in ensuring the reliability of your PowerShell code. By employing a combination of unit tests to validate individual functions, acceptance tests to assess overall system behavior, and integration tests to test component interactions, you can create a robust testing strategy that thoroughly evaluates your scripts' functionality and performance.

As we delve into the intricacies of each test type, remember that testing is not merely a quality assurance task; it's a theatrical journey where developers refine their craft. So, prepare for a captivating exploration of PowerShell testing, where precision, completeness, and harmony are the guiding principles.

Pester Test Naming Convention and File Structure

In the world of Pester, naming conventions and file structures are your allies. Pester operates under the assumption that any file ending with *.tests.ps1* is a test file – a convention we highly recommend adhering to. While it's theoretically possible to alter this behavior, diving into such complexity is beyond the scope of this beginner's guide.

Why does Pester have this preference? Because Pester adores functions, and it challenges you to become a better coder by crafting functions that perform singular tasks. Writing functions this way not only

aligns with PowerShell best practices but also makes your code highly testable. Consider a function like *Get-User*. It deserves its own file, aptly named *Get-User.ps1*. Correspondingly, your Pester test for this function should be named *Get-User.tests.ps1*. This clean separation ensures clarity in your project structure.

However, the naming standard can be further enhanced for differentiation. Adding a descriptor about the type of test being conducted is a common practice. For instance, if you're writing unit tests for *Get-User*, your test file could be named *Get-User.unit.tests.ps1*. Similarly, for integration tests, it could be *Get-User.integration.tests.ps1*, and for acceptance tests, *Get-User.acceptance.tests.ps1*.

This categorization not only clarifies the test type but also enables selective test execution – a topic we'll explore later in this book.

Keeping your .tests files in the same directory as the code they are testing is a wise choice. This organization fosters coherence and ensures that your tests are always in sync with your code. For example,

```
Get-User\
      Get-User.ps1
      Get-User.tests.ps1
```

If you're working with modules, a structured approach within module-related directories is advisable:

```
Get-User\
      Get-User\Public\
            Get-User.ps1
      Get-User\Tests\
            Get-User.tests.ps1
```

This modular arrangement facilitates a seamless workflow, making your tests as integral a part of your project as the code they validate. As you progress through this guide, you'll gain deeper insights into how such meticulous structuring can optimize your PowerShell projects.

Summary

In this introductory chapter, we delved into the fascinating world of Pester, the testing framework designed to empower PowerShell developers. We embarked on a journey to demystify testing in PowerShell, exploring the essential concepts that underpin Pester's functionality.

We began by understanding what Pester is and why it's indispensable for every PowerShell developer. Pester emerged as more than just a testing framework; it became the key to ensuring that PowerShell scripts are robust, reliable, and precisely execute their intended tasks. By embracing Pester, you unlock the potential to write code with confidence, fostering a culture of reliability and efficiency.

The chapter continued by addressing various types of tests, each serving a distinct purpose in the realm of software testing. We deciphered the nuances between unit tests, acceptance tests, and integration tests, employing a theatrical analogy to simplify these concepts.

Next, we navigated the intricacies of Pester files and their structures. Understanding the anatomy of these files laid the groundwork for crafting well-organized and potent tests. We also explored Pester's naming conventions and file structures, emphasizing the importance of clear categorization for different test types. This structuring ensures that tests are always in harmony with the code they validate.

This chapter provided a comprehensive foundation for the Pester journey ahead. By comprehending the core concepts of Pester, including its purpose, types of tests, and file structures, you are equipped with the foundational knowledge required to harness the full potential of this powerful testing framework.

In Chapter 2, we will focus on the key building blocks of Pester: **Describe**, **Context**, and **It**. These constructs provide the framework for structuring tests effectively, enabling us to validate different aspects of our scripts with precision. Let's get cracking!

CHAPTER 2

Mastering Pester Fundamentals

Welcome to the heart of Pester! In this chapter, we will delve deep into the fundamental building blocks that empower Pester to work its magic in the world of PowerShell testing. Understanding these core elements – **Describe**, **Context**, **It**, **BeforeAll**, **AfterAll**, **BeforeEach**, and **AfterEach** – is akin to mastering the essential chords in music or the basic strokes in painting. They form the foundation upon which your Pester tests will stand strong and resilient.

We will demystify the structure and purpose of these elements, guiding you through their application. By the end of this chapter, you'll not only comprehend the syntax and usage of Pester's fundamental components but also grasp their significance in crafting robust and reliable tests for your PowerShell scripts. So, let's embark on this journey of mastering Pester fundamentals, where you will gain the skills needed to wield the testing power of Pester effectively.

Understanding Blocks in Pester

In the realm of Pester, every test script revolves around a fundamental structure comprising various special script blocks. Each of these blocks plays a specific role, some obligatory and others optional. Let's start by unraveling the very cornerstone of a Pester test: the **Describe** block.

O. Heaume, *Getting Started with Pester 5*, https://doi.org/10.1007/979-8-8688-0306-2_2