Skrypty PowerShell: struktura i dobre praktyki

Poziom: średniozaawansowany **Dotyczy: Windows PowerShell 5.1, PowerShell 7+**

Czas: ~35 min

Od pliku `.ps1` z trzema poleceniami do skryptu, który bezpiecznie uruchomi ktoś inny, o trzeciej w nocy, bez nadzoru — parametry, tryb ścisły, obsługa błędów i szkielet gotowy do skopiowania.

Skróty użyte na tej stronie (1)

Skrót	Co znaczy
TCP	Transmission Control Protocol: protokół sieciowy z potwierdzaniem dostarczenia

Fiszka

TYLKO ODCZYT

Zdefiniowanie parametrów skryptu: param()

```
param(
    [string]$NazwaSerwera,
    [int]$NumerPortu = 443
)
```

> **Pojęcie: param().** Blok na samym początku pliku `.ps1` (przed jakąkolwiek inną instrukcją wykonywalną, komentarze dozwolone) deklarujący, jakie argumenty skrypt przyjmuje z linii wywołania — `.\skrypt.ps1 -NazwaSerwera wsus01 -NumerPortu 8443`. Bez `param()` skrypt wciąż może odwoływać się do argumentów przez `$args`, ale traci nazwane parametry, walidację typu i podpowiadanie składni w edytorze.

🔒 TYLKO ODCZYT

Wymuszenie podania parametru: [Parameter(Mandatory)]

```
param(
    [Parameter(Mandatory)]
    [string]$NazwaSerwera
)
```

Skrypt uruchomiony bez `-NazwaSerwera` nie wystartuje z pustą albo domyślną wartością — PowerShell sam zapyta interaktywnie o brakujący parametr, a w sesji bez możliwości interakcji (na przykład w zadaniu zaplanowanym) zwróci błąd zamiast ciągnąć dalej z niekompletnymi danymi.

Przykład:

```
cmdlet skrypt.ps1 at command pipeline position 1
Supply values for the following parameters:
NazwaSerwera:
```

🔒 TYLKO ODCZYT

Ograniczenie parametru do zamkniętej listy wartości: ValidateSet

```
param(
    [ValidateSet("Restart", "Stop", "Start")]
    [string]$Akcja = "Start"
)
```

Skrypt odrzuca dowolną wartość spoza podanej listy jeszcze przed wykonaniem pierwszej linii kodu — literówka w nazwie akcji (`"Restrat"`) kończy się jasnym komunikatem błędu walidacji, zamiast cichym przejściem do gałęzi `else`, której nikt nie napisał na taki przypadek.

Przykład:

```
Cannot validate argument on parameter 'Akcja'. The argument "Restrat" does not belong to
the set
"Restart,Stop,Start" specified by the ValidateSet attribute.
```

TYLKO ODCZYT

Walidacja zakresu wartości liczbowej: ValidateRange

```
param(
    [ValidateRange(1, 65535)]
    [int]$NumerPortu
)
```

Odrzucamy wartości spoza sensownego zakresu numeru portu TCP/UDP na etapie parsowania parametrów, zamiast dowiadywać się o błędzie dopiero przy próbie faktycznego połączenia się na port `0` albo `99999`.

TCP = Transmission Control Protocol: protokół sieciowy z potwierdzaniem dostarczenia

TYLKO ODCZYT

Walidacja przez własny blok skryptu: ValidateScript

```
param(
    [ValidateScript({ Test-Path $_ })]
    [string]$SciezkaDoPliku
)
```

Sprawdzamy dowolny, własny warunek na wartości parametru — tutaj: czy podana ścieżka w ogóle istnieje na dysku, zanim skrypt spróbuje z niej cokolwiek odczytać.

TYLKO ODCZYT

Deklaracja typu parametru wymuszająca konwersję

```
param(
    [datetime]$DataOd
)
```

Wpisanie `-DataOd "2026-01-15"` na wejściu skryptu automatycznie konwertuje tekst na obiekt `DateTime` — dalszy kod operuje na prawdziwej dacie z dostępnymi metodami i porównaniami, nie na tekście wymagającym ręcznego parsowania.

🔍 TYLKO ODCZYT

Przekazanie zestawu parametrów przez tablicę haszującą: splatting

```
$parametry = @{
    ComputerName = "NazwaKomputera"
    Credential   = $poswiadczenia
    ScriptBlock  = { Get-Service }
}
Invoke-Command @parametry
```

> **Pojęcie: splatting.** Zamiast wypisywać każdy parametr wywołania osobno w jednej długiej linii, składamy je w tablicę haszującą i przekazujemy całość znakiem `@` zamiast `$` przed nazwą zmiennej — czytelniejsze przy poleceniach z wieloma parametrami i wygodne, gdy część parametrów jest warunkowa (dopisywana do tablicy tylko czasem, przed wywołaniem).

⚠️ ZMIENIA SYSTEM

Włączenie trybu ścisłego: Set-StrictMode

```
Set-StrictMode -Version Latest
```

> **Pojęcie: tryb ścisły w PowerShellu.** Odpowiednik `set -euo pipefail` z basha, opisanego na stronie Tryb ścisły: set -euo pipefail — z tą różnicą, że `Set-StrictMode` skupia się na innym zestawie pułapek niż bashowy `set -u`. Włączony na `-Version Latest`, wymusza błąd przy odwołaniu do nieistniejącej właściwości obiektu, przy użyciu niezainicjalizowanej zmiennej i przy niepoprawnym wywołaniu metody — bez niego PowerShell po cichu zwraca `$null` w wielu z tych sytuacji, co maskuje literówkę w nazwie właściwości równie skutecznie jak brak `set -u` maskuje literówkę w nazwie zmiennej basha.

Przykład błędu wyłapanego przez tryb ścisły:

```
InvalidOperation: The property 'StatuS' cannot be found on this object.
```

🔍 TYLKO ODCZYT

Ustawienie zachowania przy błędach niekrytycznych: $ErrorActionPreference

```
$ErrorActionPreference = "Stop"
```

> **Pojęcie: błąd przerywający i nieprzerywający.** PowerShell rozróżnia dwa rodzaje błędów — przerywające (`terminating`), które natychmiast zatrzymują bieżący blok kodu, i nieprzerywające (`non-terminating`), które domyślnie tylko wypisują czerwony komunikat na ekran i pozwalają skryptowi jechać dalej, dokładnie tak jak bash bez `set -e` leci dalej po niepowodzeniu komendy. Zdecydowana większość błędów zwracanych przez wbudowane cmdlety (na przykład `Get-Service` na nieistniejącej usłudze) jest domyślnie nieprzerywająca. `$ErrorActionPreference = "Stop"` zmienia to zachowanie globalnie dla bieżącej sesji albo skryptu, zamieniając każdy nieprzerywający błąd w przerywający — bez tego `try/catch` opisany niżej w ogóle nie przechwyci większości błędów cmdletów, bo `catch` reaguje wyłącznie na błędy przerywające.

□□ TYLKO ODCZYT

Wymuszenie błędu przerywającego dla pojedynczego polecenia: -ErrorAction Stop

```
Get-Service -Name NazwaUslugi -ErrorAction Stop
```

Zamiast zmieniać zachowanie całej sesji przez `$ErrorActionPreference`, wymuszamy błąd przerywający tylko dla jednego, konkretnego wywołania — bardziej precyzyjne podejście, gdy reszta skryptu ma normalnie tolerować drobne, nieprzerywające błędy innych poleceń.

□□ TYLKO ODCZYT

Obsługa błędów: try / catch / finally

```
try {
    Get-Service -Name NazwaUslugi -ErrorAction Stop
}
catch {
    Write-Error "Nie udało się pobrać usługi: $($_.Exception.Message)"
}
```

```
finally {
    Write-Output "Sekcja zakończona, niezależnie od wyniku."
}
```

`try` obejmuje kod, który może zawieść, `catch` przechwytuje wyłącznie błędy przerywające (stąd `-ErrorAction Stop` w bloku `try` wyżej — bez niego `catch` nigdy by się nie uruchomił dla tego konkretnego polecenia), a `finally` wykonuje się zawsze, niezależnie od tego, czy `try` się powiodło, czy wpadło w `catch` — dokładnie ta sama gwarancja sprzątania, co w skrypcie zmieniającym plik `hosts` na stronie WinRM: zarządzanie zdalne.

WinRM = Windows Remote Management: zdalne zarządzanie Windowsem, podstawa PowerShell Remoting

TYLKO ODCZYT

Przechwycenie konkretnego typu wyjątku

```
try {
    Get-ADUser -Identity "NazwaUzytkownika" -ErrorAction Stop
}
catch [Microsoft.ActiveDirectory.Management.ADIdentityNotFoundException] {
    Write-Warning "Użytkownik nie istnieje w katalogu."
}
catch {
    Write-Error "Nieoczekiwany błąd: $($_.Exception.Message)"
}
```

Rozróżniamy konkretny, spodziewany typ wyjątku od wszystkich pozostałych — pierwszy pasujący blok `catch` wygrywa, więc kolejność od najbardziej szczegółowego do ogólnego ma znaczenie.

ZMIENIA SYSTEM

Sprawdzenie historii błędów bieżącej sesji: $Error

```
$Error[0] | Format-List * -Force
```

> **Pojęcie: $Error.** Automatyczna tablica przechowująca wszystkie błędy, jakie wystąpiły w bieżącej sesji, od najnowszego (`$Error[0]`) do najstarszego — przydatna

przy diagnozowaniu tuż po awarii, zanim sesja się zamknie, bez konieczności odtwarzania błędu jeszcze raz.

□□ TYLKO ODCZYT

Wygenerowanie własnego błędu: throw

```
if (-not (Test-Path $SciezkaDoPliku)) {
    throw "Plik konfiguracyjny nie istnieje: $SciezkaDoPliku"
}
```

`throw` generuje błąd przerywający z własnym komunikatem — zatrzymuje bieżący zakres wykonania natychmiast, chyba że stoi wewnątrz `try`, wtedy trafia do odpowiadającego `catch`.

□□ TYLKO ODCZYT

Strumienie wyjścia: Write-Output, Write-Host, Write-Verbose, Write-Error

Polecenie	Trafia do potoku	Kiedy używamy
`Write-Output`	tak	zwrócenie wyniku dalej w potoku — to, co funkcja albo skrypt faktycznie „zwraca"
`Write-Host`	nie	komunikat wyłącznie na ekran dla operatora, nigdy nie powinien być dalej przetwarzany przez inny skrypt
`Write-Verbose`	nie, chyba że `-Verbose`	szczegóły diagnostyczne, widoczne tylko po jawnym włączeniu trybu gadatliwego
`Write-Warning`	nie	ostrzeżenie, sytuacja niepokojąca, ale niewymagająca przerwania
`Write-Error`	nie (błąd, nie dane)	zgłoszenie błędu nieprzerywającego z poziomu własnego kodu

Przykład różnicy w praktyce:

```
function Test-Wyjscie {
    Write-Output "to trafi do zmiennej"
    Write-Host "to tylko na ekran"
}
$wynik = Test-Wyjscie
$wynik
```

```
to tylko na ekran
to trafi do zmiennej
```

`Write-Host` wypisało się od razu na ekran w trakcie wykonania funkcji, natomiast `$wynik` po przypisaniu zawiera wyłącznie tekst z `Write-Output` — to jest najczęstsza przyczyna błędów w funkcjach, których wynik ktoś próbuje dalej przetwarzać w potoku, a które przez pomyłkę używają `Write-Host` zamiast `Write-Output`.

◫ OSTROŻNIE · NIEODWRACALNE

Zapowiedź akcji bez jej wykonania: -WhatIf

```
function Remove-StareLogi {
    [CmdletBinding(SupportsShouldProcess)]
    param(
        [string]$Katalog
    )
    Get-ChildItem $Katalog -Filter *.log | ForEach-Object {
        if ($PSCmdlet.ShouldProcess($_.FullName, "Usunięcie pliku")) {
            Remove-Item $_.FullName
        }
    }
}

Remove-StareLogi -Katalog C:\Logs -WhatIf
```

> **Pojęcie: SupportsShouldProcess.** Atrybut `[CmdletBinding(SupportsShouldProcess)]` dodaje funkcji wbudowane wsparcie dla `-WhatIf` (pokaż, co by się stało, nic nie rób) i `-Confirm` (zapytaj przed każdą operacją) — wewnątrz funkcji `$PSCmdlet.ShouldProcess()` sam sprawdza, czy któryś z tych parametrów jest aktywny, i zwraca `$false` zamiast wykonać właściwą akcję, jeśli tak.

Przykład:

```
What if: Performing the operation "Usunięcie pliku" on target "C:\Logs\stary.log".
What if: Performing the operation "Usunięcie pliku" on target "C:\Logs\archiwalny.log".
```

⚠ OSTROŻNIE · NIEODWRACALNE

Wymuszenie potwierdzenia przed każdą operacją: -Confirm

```
Remove-StareLogi -Katalog C:\Logs -Confirm
```

Ta sama funkcja z `-Confirm` pyta o potwierdzenie osobno dla każdego pliku, zanim go usunie — przydatne przy pierwszym uruchomieniu nowego skryptu na produkcji, gdy `-WhatIf` już potwierdziło, że lista celów wygląda poprawnie, ale wciąż chcemy kontrolować tempo wykonania ręcznie.

⚠ ZMIENIA SYSTEM

Rozpoczęcie zapisu całej sesji do pliku: Start-Transcript

```
Start-Transcript -Path "C:\Logi\skrypt-$(Get-Date -Format 'yyyyMMdd-HHmmss').log"
```

> **Pojęcie: transkrypcja.** `Start-Transcript` zapisuje do pliku dosłownie wszystko, co pojawia się w konsoli — wpisywane polecenia i cały ich wynik, łącznie z `Write-Host` — aż do wywołania `Stop-Transcript`. To inny mechanizm niż logowanie własnymi `Write-Output` / `Add-Content`, bo nie wymaga świadomego dopisywania każdej linijki logu ręcznie, kosztem mniejszej kontroli nad formatem zapisanego pliku.

⚠ ZMIENIA SYSTEM

Zapisanie własnego logu do pliku

```
function Write-Log {
    param([string]$Wiadomosc, [string]$Poziom = "INFO")
    $wpis = "{0} [{1}] {2}" -f (Get-Date -Format "yyyy-MM-dd HH:mm:ss"), $Poziom,
$Wiadomosc
    Add-Content -Path "C:\Logi\skrypt.log" -Value $wpis
```

```
}


Write-Log "Rozpoczęto przetwarzanie" -Poziom "INFO"
```

Budujemy własny format logu, gdy potrzebujemy większej kontroli niż daje `Start-Transcript` — jednolity znacznik czasu, poziom ważności i wiadomość w jednej linii, łatwe do dalszego przeszukiwania narzędziem typu `Select-String`.

> **Uważaj**
>
> - `catch` bez `-ErrorAction Stop` na poprzedzającym poleceniu nigdy się nie uruchomi dla większości cmdletów, bo ich domyślne błędy są nieprzerywające.
> - `Write-Host` w funkcji, której wynik ktoś przekazuje dalej w potoku, jest jednym z najczęstszych źródeł „zniknięcia" danych — trafiają na ekran, nie do zmiennej.
> - `Set-StrictMode -Version Latest` na starym, nigdy wcześniej pod nim nie testowanym skrypcie potrafi ujawnić od razu kilkanaście utajonych literówek — warto włączać go od początku pisania nowego kodu, nie doklejać na koniec do gotowego skryptu bez planu na przetestowanie każdego wyjątku.

Z życia wzięte

> **Z ŻYCIA WZIĘTE**
>
> Tomek pisze skrypt restartujący usługę Optimy na sql01 i owija go w `try/catch`, przekonany że jest bezpiecznie. W środę usługa faktycznie nie istnieje pod tą nazwą po literówce w parametrze, `Get-Service` wypisuje czerwony błąd na ekranie, a `catch` mimo to się nie uruchamia. Dopiero dopisanie `-ErrorAction Stop` do `Get-Service` sprawia, że ten sam błąd trafia tam, gdzie powinien.

Po co to istnieje

Skrypt napisany „na szybko" do jednorazowego użycia i skrypt, który ma działać samodzielnie z harmonogramu zadań przez kolejne lata, różnią się przede wszystkim tym, ile założeń można w nich bezpiecznie pominąć. Jednorazowy skrypt piszemy i uruchamiamy sami, na oczach, więc błędną literówkę widzimy natychmiast na ekranie. Skrypt produkcyjny uruchomi ktoś inny, o innej porze, bez patrzenia na ekran — musi sam wykryć, że coś poszło nie tak, zatrzymać się w

bezpiecznym miejscu i zostawić ślad w logu tłumaczący, co konkretnie zawiodło.

> **Pojęcie: skrypt, funkcja, moduł.** Skrypt to pojedynczy plik `.ps1`, uruchamiany bezpośrednio z linii poleceń albo z harmonogramu — ma swój własny zakres zmiennych i kończy działanie po wykonaniu ostatniej linii. Funkcja to nazwany blok kodu wewnątrz skryptu (albo modułu), wywoływany wielokrotnie z różnymi argumentami — istnieje po to, żeby nie powtarzać tego samego fragmentu kodu w kilku miejscach. Moduł to zbiór funkcji, zmiennych i aliasów spakowanych razem i importowanych poleceniem `Import-Module` do dowolnej sesji, która ich potrzebuje — to jest jednostka dystrybucji i ponownego użycia kodu między różnymi skryptami i różnymi osobami, w odróżnieniu od funkcji, która żyje w obrębie jednego pliku, dopóki jej świadomie nie wydzielimy do modułu.

Jak to działa

`Set-StrictMode` i `$ErrorActionPreference` odpowiadają na dwa różne pytania, mylone ze sobą równie często jak `set -e` i `set -u` w bashu. `Set-StrictMode` pilnuje poprawności samego kodu — nieistniejące zmienne, nieistniejące właściwości, złe wywołania metod — niezależnie od tego, czy jakiekolwiek polecenie w ogóle zwróciło błąd. `$ErrorActionPreference` decyduje, co się dzieje, gdy polecenie (cmdlet, funkcja, wywołanie zewnętrznego programu) faktycznie zgłosi błąd — czy skrypt jedzie dalej, czy się zatrzymuje. Oba razem dają pokrycie zbliżone do bashowego `set -euo pipefail`, ale są dwoma osobnymi ustawieniami, które trzeba włączyć osobno, i o czym łatwo zapomnieć, testując skrypt wyłącznie na przypadku, gdzie wszystko się powiodło.

`try/catch` bez `-ErrorAction Stop` na krytycznym poleceniu to pułapka analogiczna do czterech sytuacji, w których bashowe `set -e` nie działa, opisanych na stronie o trybie ścisłym basha — mechanizm istnieje i wygląda na aktywny, ale w praktyce nie łapie niczego, bo błąd, który miał go uruchomić, nigdy nie stał się przerywający.



Bez -ErrorAction Stop blad nieprzerywajacy omija catch, z nim trafia do niego poprawnie (kliknięcie otwiera pełny rozmiar)

Częste błędy

Poleganie na samym `try/catch` bez `$ErrorActionPreference` albo `-ErrorAction Stop`. Skrypt „przechodzi" przez `catch` bez wejścia do niego, mimo widocznego czerwonego komunikatu błędu na ekranie chwilę wcześniej — bo ten konkretny błąd był nieprzerywający i `try/catch` nigdy go nie zobaczył.

Zwracanie wyniku funkcji przez Write-Host zamiast Write-Output. Kod działa poprawnie przy ręcznym uruchomieniu (bo widać wynik na ekranie), a przestaje działać, gdy ktoś spróbuje przypisać go do zmiennej albo puścić dalej w potoku — `$wynik = Moja-Funkcja` zwraca `$null` albo pusty ciąg, mimo że na ekranie coś się wyświetliło.

Brak `-WhatIf` przy pierwszym uruchomieniu skryptu usuwającego dane na produkcji. `SupportsShouldProcess` istnieje właśnie po to, żeby dało się bezpiecznie zweryfikować listę celów przed wykonaniem nieodwracalnej operacji — pominięcie tego kroku i uruchomienie od razu bez `-WhatIf` jest samo w sobie błędem proceduralnym, niezależnie od tego, czy skrypt akurat zawiera błąd w logice.

Szkielet skryptu produkcyjnego

```
#Requires -Version 5.1
<#
.SYNOPSIS
    Krótki opis tego, co robi skrypt.
.DESCRIPTION
    Dłuższy opis, parametry, przykład wywołania.
.EXAMPLE
    .\SzkieletSkryptu.ps1 -NazwaSerwera wsus01 -Akcja Restart
#>

[CmdletBinding(SupportsShouldProcess)]
param(
    [Parameter(Mandatory)]
    [string]$NazwaSerwera,

    [ValidateSet("Restart", "Stop", "Start")]
```

```
    [string]$Akcja = "Start",

    [string]$SciezkaLogu = "C:\Logi\$($MyInvocation.MyCommand.Name)-$(Get-Date -Format
'yyyyMMdd').log"
)

Set-StrictMode -Version Latest
$ErrorActionPreference = "Stop"

function Write-Log {
    param([string]$Wiadomosc, [string]$Poziom = "INFO")
    $wpis = "{0} [{1}] {2}" -f (Get-Date -Format "yyyy-MM-dd HH:mm:ss"), $Poziom, $Wiadomosc
    Add-Content -Path $SciezkaLogu -Value $wpis
    Write-Verbose $wpis
}

try {
    Write-Log "Rozpoczęto: akcja=$Akcja, serwer=$NazwaSerwera"

    if ($PSCmdlet.ShouldProcess($NazwaSerwera, "Wykonanie akcji $Akcja")) {
        switch ($Akcja) {
            "Restart" { Restart-Service -InputObject (Get-Service -ComputerName $NazwaSerwera
-Name NazwaUslugi) }
            "Stop"    { Stop-Service -InputObject (Get-Service -ComputerName $NazwaSerwera -
Name NazwaUslugi) }
            "Start"   { Start-Service -InputObject (Get-Service -ComputerName $NazwaSerwera -
Name NazwaUslugi) }
        }
    }

    Write-Log "Zakończono pomyślnie"
}
catch {
    Write-Log "BŁĄD: $($_.Exception.Message)" -Poziom "ERROR"
    throw
}
finally {
    Write-Log "Koniec wykonania skryptu"
}
```

Ten szkielet łączy wszystkie elementy z tej strony naraz: nagłówek `.SYNOPSIS`/`.DESCRIPTION` czytany przez `Get-Help`, parametry z walidacją, tryb ścisły, jednolite logowanie, wsparcie `-WhatIf`/`-Confirm` i `try/catch/finally` z ponownym rzuceniem błędu (`throw` bez argumentu w bloku `catch` przekazuje oryginalny wyjątek dalej, zamiast go tłumić) po zapisaniu do logu.

□□ Ćwiczenia

> Wszystko poniżej wykonujemy we własnej sesji PowerShell, w katalogu `$env:TEMP`, bez uprawnień administratora.

Ćwiczenie 1 — Write-Host kontra Write-Output

```
function Test-Funkcja1 { Write-Host "wynik A" }
function Test-Funkcja2 { Write-Output "wynik B" }

$a = Test-Funkcja1
$b = Test-Funkcja2

"a = [$a]"
"b = [$b]"
```

Przewidujemy, co znajdzie się w `$a` i `$b`, zanim uruchomimy kod.

Odpowiedź

`$a` będzie puste (`a = []`), bo `Write-Host` wypisało tekst bezpośrednio na ekran i nigdy nie trafiło do potoku, więc przypisanie do zmiennej nie miało czego przechwycić. `$b` będzie zawierać `wynik B`, bo `Write-Output` przekazuje wartość dalej w potoku, skąd przypisanie do zmiennej ją odbiera. To jest praktyczna demonstracja tabeli strumieni wyjścia z fiszki wyżej.

Ćwiczenie 2 — tryb ścisły ujawnia literówkę

```
function Sprawdz-Usluge {
    param($Nazwa)
    $usluga = Get-Service -Name $Nazwa
    if ($usluga.StatuS -eq "Running") {
        "Usługa działa"
    } else {
        "Usługa nie działa"
    }
}

Sprawdz-Usluge -Nazwa "Spooler"

Set-StrictMode -Version Latest
Sprawdz-Usluge -Nazwa "Spooler"
```

Zauważamy literówkę `StatuS` zamiast `Status` w kodzie funkcji — sprawdzamy, jak zachowuje się bez i z trybem ścisłym.

Odpowiedź

Bez `Set-StrictMode` odwołanie do nieistniejącej właściwości `StatuS` po cichu zwraca `$null`, porównanie `$null -eq "Running"` daje `$false`, więc funkcja zawsze zgłasza „Usługa nie działa" — nawet jeśli usługa faktycznie działa. Po włączeniu `Set-StrictMode -Version Latest` to samo wywołanie rzuca jawny błąd `PropertyNotFoundException`, natychmiast pokazując literówkę zamiast cichej, błędnej odpowiedzi. To jest dokładnie ten sam mechanizm, co `set -u` w bashu wyłapujące niezdefiniowaną zmienną.

Pytania kontrolne

1. Dlaczego `try { Get-Service NieistniejacaUsluga } catch { "złapano" }` nie wypisuje „złapano", mimo że usługa naprawdę nie istnieje?

Odpowiedź

Domyślny błąd `Get-Service` dla nieistniejącej usługi jest nieprzerywający — wypisuje czerwony komunikat na ekran i pozwala wykonaniu jechać dalej, więc blok `catch` nigdy się nie aktywuje, bo nie doszło do błędu przerywającego. Naprawą jest dodanie `-ErrorAction Stop` do wywołania `Get-Service` wewnątrz `try`, dokładnie jak w fiszce „Obsługa błędów" wyżej.

2. Funkcja ma `[CmdletBinding(SupportsShouldProcess)]`, ale w jej treści nigdzie nie pojawia się `$PSCmdlet.ShouldProcess()`. Co się stanie po wywołaniu jej z `-WhatIf`?

Odpowiedź

Sam atrybut `SupportsShouldProcess` tylko udostępnia parametry `-WhatIf` i `-Confirm` na poziomie wywołania funkcji — nie robi nic automatycznie z samą logiką wewnątrz. Bez jawnego wywołania `$PSCmdlet.ShouldProcess()` w miejscu, gdzie faktycznie wykonuje się zmieniająca stan operacja, funkcja przyjmie `-WhatIf` bez błędu składni, ale i tak wykona wszystkie operacje normalnie — `-WhatIf` bez odpowiadającego mu `ShouldProcess()` w kodzie jest atrapą, nie realną ochroną.

Powiązane: Tryb ścisły: set -euo pipefail · Podstawy składni PowerShella · Polityka wykonywania i podpisywanie skryptów · Poświadczenia i bezpieczne hasła w skryptach · Wysyłanie poczty ze skryptu

Źródła: `Get-Help about_Functions_Advanced` · `Get-Help about_Try_Catch_Finally` · `Get-Help about_CommonParameters` · dokumentacja Microsoft Learn, „about Preference Variables"

Revision #39
Created 2026-09-25 06:44:37 UTC by Tomasz Konieczny
Updated 2026-10-26 14:06:48 UTC by Tomasz Konieczny