Wstęp do PowerShell

PowerShell to nowa powłoka konsolowa w systemach z rodziny Windows. Nowa jest tutaj pewnym uproszczeniem – pierwsza jej wersja dostępna była już bowiem w Windows XP SP2, Windows Server 2003 SP1 oraz w Windows Vista (data wydania – listopad 2006). Pierwsze wzmianki na temat PowerShell pojawiły się już w roku 2003 - wtedy to została zaprezentowana wczesna wersja nowego narzędzia zarządzania systemem Windows o nazwie Monad/Microsoft Shell (MSH).

Nazwanie PowerShell konsolą jest z kolei zbyt dużym uproszczeniem – jest on bowiem narzędziem automatyzacji i konfiguracji procesów systemowych, zaś konsola i język skryptowy stanowią trzon komunikacyjny.

Idea powstania nowego narzędzia do zarządzania systemem Windows nie jest dla Microsoft świeża. Od samego początku istnienia okienek wszelkie polecenia wymagające konsoli wykonywane były w linii poleceń command.com (podstawowa powłoka systemów DOS, w tym Microsoft DOS). Było to naturalne zwłaszcza, że system Windows stanowił graficzną powłokę dla systemu DOS (sytuacja ta była prawdziwa dla wszystkich systemów klienckich do wersji Windows Millenium Edition). Chociaż systemy Windows od wersji XP nie wykorzystują podsystemu DOS (korzystają z „nowego" jądra NT), Microsoft zachował pewną zgodność z poprzednimi wersjami systemów włączając w system interpreter poleceń cmd.exe (nowa wersja command.com). Niestety sam w sobie posiadał on zbyt małą funkcjonalność uwzględniając możliwości nowych wersji systemu. Ponadto posiadał pewne polecenia, które mogły być potencjalnym wektorem ataku (np. polecenie at).

Niedobory konfiguracji systemowej z poziomu linii poleceń Microsoft załatwił poprzez dodanie WMI (Windows Module Instrument), pozwalającego na dostęp do systemowych informacji i powiadomień (przykładowo dotyczących kont użytkowników). To jednak nie załatwiało sprawy – poprzez naleciałości z poprzednich wersji systemów konsola odstaje zarówno pod względem wygody użytku jak i efektywności (polecenia proceduralne muszą obsługiwać dane obiektowe COM, czyli tłumaczyć dane proste na rozbudowane).

PowerShell jest odpowiedzią na wszystkie opisane powyżej bolączki linii poleceń. Po pierwsze w całości został napisany w technologii .NET (obecnie używa wersji Core). Oznacza to, że w pełni wspiera obiektowość. Po drugie wszystkie polecenia konsoli zostały przepisane na nowo w taki sposób, by były łatwiejsze do nauki i codziennego użytkowania. Z pozoru dłuższe (np. zamiast pobrania zawartości folderu poleceniem dir PowerShell używa polecenia Get-ChildItem), praktycznie łatwiejsze do zapamiętania (Weź-DziedziczącyElement).

Przytoczone powyżej polecenie w języku PowerShell nazywa się cmdlet (command let). Polecenie zawsze składa się z pary czasownik-rzeczownik. Dzięki temu jest łatwiejsze do zapamiętania. Każde polecenie zwraca OBIEKT (nie zaś dane proste). Ponadto polecenia te, przy użyciu bibliotek .NET, programiści mogą wykorzystać we własnych aplikacjach celem odczytu bądź konfiguracji wskazanych elementów systemu, uwzględniając w tym część graficzną.

INFORMACJA: Celem ułatwienia pracy zaawansowanym użytkownikom systemu Windows wiele poleceń PowerShell powiada tzw. aliasy (referencje, inne nazwy, skróty), które pozwalają wywoływać je jako odpowiedniki poleceń cmd. Nie zmienia to faktu, że w PowerShell nawet wywołanie polecenia ping, dir, cd powoduje zwrócenie obiektu, który można wyświetlić lub przechwycić do skryptu/programu.

Kolejną ważną częścią PowerShell są potoki. Standardowa linia poleceń w danej chwili może operować jednym poleceniem, które zwróci nam jakąś wartość. Wartość tę sami musimy przekazać

nowemu (następnemu) poleceniu celem obrobienia jej. Często w tym celu pisane były skrypty batch. W przypadku PowerShell, podobnie jak to ma miejsce w bash systemów Unix/Unix-like, możemy wyjście jednego polecenia obrobić bezpośrednio kolejnym poleceniem „w linii”. Przykładem takiego polecenia może być np.:

Get-ChiltItem | grep .ods

co spowoduje wywołanie polecenia dir (ekwiwalent), którego wynik zostanie oczyszczony (wyświetlą się wszystkie pliki z rozszerzeniem ods).

Same skrypty też są znacznie czytelniejsze dla osób programujących bądź korzystających chociażby z Bash. PowerShell pozwala na pisanie własnych funkcji, wyrażeń lambda, tworzenia obiektów oraz zmiennych. Dodatkowo możliwe jest korzystania z warunku, pętli i obsługi wyjątków. Sam język skryptowy wzorowany jest na takich językach jak C# i C.

Dodatkowo PowerShell pozwala w pełni zarządzać komputerem bądź grupą komputerów w lokalizacjach odległych/zdalnych. Całość działa prze protokół HTTP/HTTPS (przy HTTPS wymagany jest certyfikat, działa podobnie do SSH dla systemu Linux).

Obecna wersja 6 wydana jest na licencji MIT. Dostępna jest na różne platformy systemowe (w tym na system Linux oraz OS X). Dzięki temu skrypty działające na systemie Windows mogą działać na systemach z rodziny Unix/Unix-like. Ponadto możliwe jest zdalne zarządzanie komputerami Windows z linii poleceń systemów Unix/Unix-like.

1. Podstawowa składnia poleceń.

Jak zostało to wcześniej wspomniane, polecenia w PowerShell mają następującą składnię:

czasownik-rzeczownik [parametry_polecenia] [parametry_wspolne]

Czasownikiem może być słowo Get (weź/pobierz), Set (ustaw), Invoke (wywołaj) czy Enter (wejdź).

Reczownikiem będzie natomiast zadanie, które chcemy wykonać. Przykładowo z czasownikiem Get można podłączyć ChildItem, co poskutkuje wyświetleniem wszystkich elementów zależnych bieżącej lokalizacji (wyświetlą się wszystkie foldery i pliki, które znajdują się w aktualnie wybranym katalogu bądź magazynie danych). Z Invoke można z kolei połączyć rzeczownik WebRequest (żądanie sieciowe), które pozwala na pobieranie treści z dowolnej lokalizacji zdalnej (przykładowo pliki z ftp/http(s), w tym pliki html po wpisaniu adresu strony WWW).

Parametry polecenia mogą być różne. Przykładowo dla polecenia (cmdlet) Get-Help parametrem polecenia będzie nazwa innego polecenia (cmdlet). W przypadku innych poleceń parametry należy poprzedzić ich nazwą, np. -ComputerName, po której to nazwie trzeba podać nazwę komputera, na którym chcemy wykonać polecenie:

Get-Proccess -ComputerName moj_pc -Name explor* -FileVersionInfo

Dodatkowo wskazujemy, że interesuje nas plik (bądź pliki), których nazwa rozpoczyna się frazą explor* (Name) oraz chcemy dowiedzieć się szczegółów wersji o wskazanym pliku (FileVersionInfo).

Parametry wspólne to takie, które można zastosować z dowolnym cmdlet. Wspólnymi parametrami są między innymi:

- debug – wyświetla szczegółowe informacje wykonywania danego procesu (o ile polecenie wspiera wyświetlanie takich informacji)
- OutVariable – wynik polecenia zachowywany jest w zmiennej o podanej nazwie,

Get-Proccess -ComputerName moj_pc -Name explor* -FileVersionInfo -outVariable name

spowoduje powstanie zmiennej $name. Po wywołaniu jej nazwy wyświetli się zapisana w niej wartość

- ErrorVariable – przechowuje ewentualny ciąg znakowy z błędami w podanej zmiennej (jak wyżej)

Wszystkie parametry wspólne można znaleźć w dokumentacji na stronie podanej w materiałach (punkt 2)) lub w PowerShell można wykonać następujące polecenie:

Get-Help about_commonParameters

Dodatkowo PowerShell może bezpośrednio korzystać ze wszystkich klas dostępnych w bibliotekach .NET. Przykładowo jeżeli chcielibyśmy odwołać się do metody konsoli systemowej można dokonać tego w następujący sposób:

[System.Console]::Write(„Tekst do wyświetlenia w konsoli!”)

Powyższe polecenie pokazuje jak można odwoływać się do systemowych obiektów klas .NET. Obiekt zawsze podawany jest w kwadratowym nawiasie, natomiast jego składowe dostępne są po łączniku :: (składnia podobna do Visual C++ i odwołań do składowych statycznych).

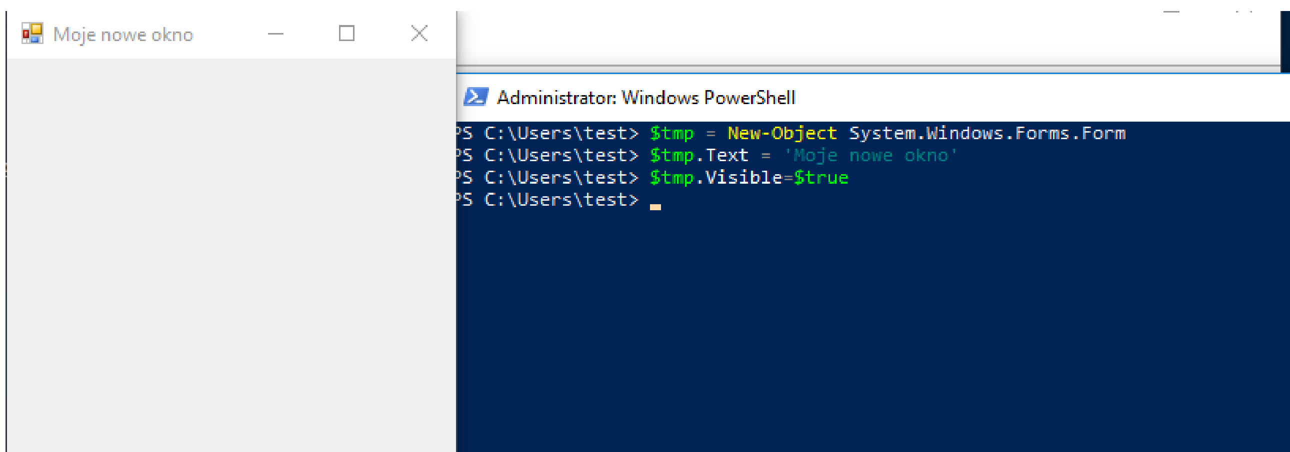
Obiekty można tworzyć także samemu poprzez cmdlet New-Object:

$tmp = New-Object System.Windows.Forms.Form

powyższe polecenie spowoduje utworzenie obiektu nowej formatki (graficznej). Zostanie ona przypisana do zmiennej $tmp. Od tego momentu można komunikować się z tak utworzoną formatką poprzez złączenie kropką, np.:

$tmp.Text = ‘Moje nowe okno’

W ten sposób zmienimy tytuł wyświetlanego okna. Przykład:



Oczywiście okno będzie wyglądało na „zawieszone”. Powodem jest fakt, że pokazanie formatki odbywa się w tym samym procesie, w którym działa konsola. Ponieważ polecenie pokazania formatki się kończy (zostaje wykonane i otrzymujemy znak zachęty w nowej linii) toteż wywołane okno „mrozi się” (ważniejsza jest linia poleceń PowerShell niż ono samo). Aby uruchomić utworzony obiekt tak, by było możliwe np. kliknięcie przycisku X u góry okna należy zamienić ostatnią linię na:

```
$tmp.ShowDialog()
```

Nie spowoduje to utworzenie jako takiego niezależnego procesu (nadal będzie to konsola PowerShell), jednak metoda ta jest tzw. blokującą. Nie będziemy mogli nic napisać w konsoli gdyż jako główny element zostanie ustawiona wywołana formatka (za to nie będziemy mogli nic napisać w konsoli).

2. Podsumowanie

Jak widać konsola PowerShell jest świetnym narzędziem systemowym. Dzięki bezpośredniemu połączeniu z bibliotekami .NET/.NET Core możliwe staje się pisanie narzędzi, które w pełni pozwalają na zmiany i odczyt wszystkich ustawień systemowych. Ponadto system Windows staje się w pełni konfigurowalny także przez linię poleceń, w której możemy zarządzać całymi grupami komputerów (brak ograniczeń zarządzania poprzez Pulpit Zdalny). Brak też ograniczeń platformy – skrypty kompatybilne z .NET Core mogą być uruchamiane na dowolnym systemie operacyjnym posiadającym .NET Core co tylko zwiększa możliwości pracy i zarządzania systemami z rodziny Windows.

MATERIAŁY:

1) https://en.wikipedia.org/wiki/PowerShell
2) https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_commonparameters?view=powershell-6
3) https://docs.microsoft.com/pl-pl/powershell/scripting/getting-started/cookbooks/creating-a-custom-input-box?view=powershell-6