Dvě knihy v jedné

Patrik Malina

Jak vyzrát na

Microsoft Windows

PowerShell 2.0

Rychlý zdroj informací pro zaneprázdněné administrátory

Nové cmdlety, proměnné a operátory, spouštění procesů na pozadí
Práce s příkazy a skripty, správa Active Directory
Shrnutí, opakování na koncích témat, otázky a odpovědi

DVD obsahuje:

Další knihu o základech PowerShellu ve formátu PDF

Všechny skripty uvedené v knize
PowerShell Pack, PowerShell 2 SDK, LogParser
PowerGUI, AD Cmdlets a další užitečné nástroje

CPRESS

Patrik Malina

Jak vyzrát na Windows PowerShell 2.0

Computer Press, a. s.
Brno
2010

Jak vyzrát na Windows PowerShell 2.0

Patrik Malina

Computer Press, a. s., 2010. Vydání první.

Jazyková korektura: Alena Láníčková
Vnitřní úprava: Petr Klíma
Sazba: Ctibor Foltýn
Rejstřík: Daniel Štreit
Obálka: Martin Sodomka
Komentář na zadní straně obálky: Libor Pácl

Technická spolupráce: Jiří Matoušek, Zuzana Šindlerová, Dagmar Hajdajová
Odpovědný redaktor: Libor Pácl
Technický redaktor: Jiří Matoušek
Produkce: Petr Baláš

Computer Press, a. s.,
Holandská 8, 639 00 Brno

Objednávky knih:
http://knihy.cpress.cz
distribuce@cpress.cz
tel.: 800 555 513

ISBN 978-80-251-2732-2
Prodejní kód: K1742
Vydalo nakladatelství Computer Press, a. s., jako svou 3583. publikaci.

Obsah

Úvod

Je tomu už téměř pět let, kdy jsem si poprvé opatřil veřejnou testovací verzi aplikace, které se říkalo Monad, a začal zkoumat, co že to vlastně Microsoft pouští mezi dělný počítačový lid. Dostupnost dokumentace na Internetu a tehdejší informační zázemí dávalo tušit, že se jedná spíše o undergroundovou zábavu než o stabilní produkt v produkci softwarového gigantu. A byl tu ještě jeden spolehlivý příznak toho, že Monad je něco krajně „podezřelého" – před započetím prvních pokusů mi už bylo z doslechu známo, že Monad má být pouze a jenom příkazové rozhraní, tedy shell, ale zdráhal jsem se tomu uvěřit, dokud si to sám neprověřím. A po chvíli, když jsem Monad nainstaloval, bylo jasné jen to, že Monad je opravdu funkční shell.

Vše ostatní ovšem bylo naprosto záhadné. Šířily se odvážné zkazky o tom, že Monad nahradí starý dobrý příkazový řádek – inu, už bylo načase, však on ten starý dobrý Cmd.exe nikdy nebyl žádná velká hvězda, pokud jsme si připustili srovnání s jeho ekvivalenty v Unixu. Šířily se však ještě bláznivější drby, a těm mohl tehdy uvěřit opravdu jen málokdo: Monad by jednou měl nabýt vrchu nad grafickým uživatelským rozhraním a stát se hlavním nástrojem pro správu! Kdo by na jaře roku 2005 takové báchorce uvěřil! Na druhou stranu, Monad přišel s alespoň nějakou dokumentací, nápověda fungovala a jeho základní kvality bylo možné rozpoznat a prověřit. Upřímně řečeno, byl to pramen živé vody pro někoho, kdo nikdy neztratil pocit, že ovládání operačního systému primárně pomocí klikání je často nejen nepraktické, ale v zásadě též kacířské a nactiutrhačné. Nakonec přišla na podzim roku 2005 i příležitost podělit se o své nadšení s ostatními aktivisty „microsoftího undergroundu" – počítačová škola Gopas mi nabídla v dramaturgii své konference TechEd jednu přednášku, v níž jsem mohl dobré zprávy o Monadu zvěstovat i ostatním. Po jejím skončení jsem během započaté diskuze měl pocit, že někteří posluchači čekali na podobnou událost jako na návrat spasitele. Bylo opravdu znát, že ani léta grafického drilu ve Windows nevykořenila unixovou tradici a sílu myšlenky příkazové konzoly.

Podobné pocity zřejmě měli i samotní tvůrci Monadu, neboť věci poté nabraly rychlý spád. O rok později již byla blízko první finální verze produktu, jehož jméno se změnilo přes Microsoft Shell na výsledné PowerShell, a bylo jasné, že tvůrčí tým není parta zapomenutých zoufalců v nitru redmondského bludiště. Chystal se nový operační systém Windows Vista a PowerShell se málem stal jeho výchozí součástí, nakonec byl uvolněn jako instalační balíček pro několik verzí Windows a doprovázela jej dobrá dokumentace a podpora na Internetu. V té době jsem již natolik věřil v budoucnost této technologie, že jsem nabídl své počítačové škole sestavení kurzu na téma PowerShell s ambiciózním rozsahem pěti dnů. A spolu s tímto úmyslem jsem přistoupil na závazek, že ke kurzu připravím i další nezbytnou součást, o kterou byla tehdy velká nouze: učebnici. Tak se zrodila první původní česká kniha o PowerShellu, která nakonec v rozšířené podobě vyšla v nakladatelství, jež stojí i za tímto svazkem. A je to ona kniha, kterou najdete jako přílohu na doprovodném disku k této učebnici, která je jejím nástupcem.

Od uvedení první knihy uběhly více než dva roky a PowerShell neuvěřitelně dozrál. Jeho vývoj je stejně zajímavý, jako je bouřlivý, a především uvedení finální verze PowerShellu 2 znamená zásadní krok vpřed. Ze zajímavé rarity se stal de facto standard pro správu operačních systémů a aplikací společnosti Microsoft a v současné době je znalost PowerShellu v podstatě nezbytná, pokud chceme i nadále provádět jejich administraci, především pak u nových a přicházejících verzí. Na počátku předchozí knihy jsem spekuloval, že správci, kteří včas nevezmou PowerShell na vědomí, se mohou časem ocitnout mimo „první ligu". Čas dozrál pro tuto knihu, jež se zabývá PowerShellem po dalších třech letech vývoje, a já již nemusím spekulovat: ten, kdo bude PowerShell ignorovat, se nejpozději s příští verzí „svého" systému či aplikace ocitne jako správce mimo hru. I to byl důvod, proč jsem se pustil do tvorby této publikace, která v mnohých tématech navazuje na dílko první, která však především přináší řadu zcela nových témat, jež se v PowerShellu objevila spolu s vývojem verze 2.

Publikace, kterou čtenář dostává do ruky, tedy nespadla z čistého nebe. Jak jsem již naznačil, navazuje na první knihu, kterou jsem PowerShellu věnoval a která vysvětlovala základy jeho použití i úvodní problematiku skriptovacího jazyka. Následující kapitoly pak navazují na látku, která již byla jednou vysvětlena – PowerShell 2 sice přidává řadu vylepšení, avšak základy práce jsou stejné a stejně tak výchozí skriptovací postupy se nezměnily. Čtenář nemusí litovat, pokud předchozí svazek nevlastní, neboť jeho kompletní verze je umístěna na doprovodném disku této publikace a má tak sloužit jako referenční materiál i úvodní kurz pro toho, komu se látka v této knize zdá dosti pokročilá.

Tento svazek navazuje na knihu předchozí také svou strukturou. První a druhá kapitola jsou jednak úvodem, shrnujícím momentální postavení PowerShellu, jednak základním roztříděním existujících variant a popisem jejich zprovoznění. Třetí kapitolu jsem věnoval popisu spouštění příkazů a skriptů v PowerShellu za jakýchkoliv podmínek – interaktivně či automaticky, lokálně či na vzdáleném počítači. Čtvrtá část je zaměřena na jazyk PowerShell jako takový a jeho zajímavé možnosti, zatímco pátá část je přesným opakem, neboť se zaměřuje na konkrétní oblasti správy systému Windows. Kapitola šestá je úplně zasvěcena adresářové službě Active Directory, kde PowerShell zaznamenal velký rozmach, a část sedmá je zaměřena na správu sítí v užším slova smyslu. V osmé kapitole jsem podal stručný přehled současných aplikovaných variant PowerShellu a rozšíření, část devátá (příloha) je pak referenčním přehledem. Jsem si jako autor plně vědom, že ani druhý svazek na dané téma nemůže zdaleka vyčerpat nabízené možnosti, a látka celé knihy tak nutně musí být vzorkem nabízených možností. Pevně doufám, že tento vzorek je alespoň dostatečně reprezentativní a zajímavý.

Na tomto místě knihy se sluší poděkovat. Můj vděk patří řadě účastníků mých kurzů, kteří neváhali a chtěli se s novou technologií co nejdříve seznámit, čímž podněcovali i mé úsilí o co nejlepších poznání možností PowerShellu a mou snahu je co nejlepším způsobem vysvětlit ostatním. Můj vděk určitě patří též sdružení WUG, které mi umožnilo opakovaně o PowerShellu přednášet a dopřát tak předchozí publikaci větší popularitu. A mám-li na závěr někoho výslovně zmínit, pak děkuji Martinu Trnkovi, který trpělivě přečetl rukopis a poskytl cenné připomínky, a především rodině, která to trpělivě vydržela.

Patrik Malina

KAPITOLA 1

PowerShell a jeho svět

PowerShell se pomalu stal neodmyslitelnou součástí operačních systémů Windows a dalšího aplikačního softwaru společnosti Microsoft. Nebylo by na tom nic divného, kdyby se nejednalo o technologii, která byla ještě před pár lety pro Windows poměrně netypická: textové příkazové rozhraní, správcovská textová konzola, zkrátka shell. Tradiční správci Windows sledovali nástup PowerShellu před několika lety tu se shovívavým úsměvem, tu s mírnou zvědavostí, tu s naprostým nepochopením. Úloha a postavení grafického rozhraní při správě se zdály tak neotřesitelné, že jen málokdo spatřoval v PowerShellu opravdovou budoucnost – proč by Microsoft opouštěl zavedenou a osvědčenou koncepci a navíc riskoval, že jeho rozmazlení správci se budou muset učit příkazový shell?

V průběhu několika posledních let však došlo k řadě překvapení. PowerShell jako technologie ukazoval působivé kvality a získával si řady příznivců. Autorský tým připravil před několika lety jeho první oficiální verzi a přes některé chybějící možnosti jsme tak získali hodnotný nástroj pro správu a automatizaci. Řada příznivců i softwarových firem rozpoznali potenciál nové technologie a využili její otevřenost k tomu, aby připravili různá vylepšení a rozšíření, která naznačovala cestu vpřed a dále za obzor původních představ. A společnost Microsoft se rozhodla k razantnímu kroku, který řada skalních uživatelů produktů této firmy považovala za „nůž do zad“: produkt MS Exchange se stal svého druhu průkopníkem a jeho správci pokusnými králíky, když ve verzi 2007 se stal rozšířený PowerShell primárním administrátorským rozhraním a grafická konzola zdaleka nepokrývala všechnu potřebnou funkcionalitu. Toto byl důležitý signál, že PowerShell není hříčka jakéhosi bohémského tvůrčího týmu, ale že jde o zásadní koncepci, jejíž nástup máme jednou okusit všichni.

PowerShell se však zdaleka nestal jen novým konzolovým rozhraním v oné podobě, jakou připravil vývojový tým společnosti Microsoft. PowerShell je již dnes ve skutečnosti opravdovou základnou – platformou pro vývoj dalších součástí, které nadále rozšiřují možnosti správy operačního systému a aplikací. Vývoj přitom postupuje hned v několika proudech a využívá PowerShell různými způsoby. Třeba tvůrci produktu PowerGUI naplnili původní myšlenku autorů PowerShellu a „dotáhli do konce“ originální návrh: PowerShell bude prostředníkem pro ovládání operačních systémů a aplikací a grafická rozhraní budou jeho nadstavbou. Právě toto provádí PowerGUI, stejně jako to dnes dělá třeba konzola pro správu produktu MS Exchange nebo nové grafické rozhraní pro správu Active Directory v serverech Windows 2008. Řada vývojářů sleduje jiný směr a využívá PowerShell jako ovládací nástroj při přístupu k dalším a dalším datovým zdrojům a rozhraním. Dobré podpory se dočkal MS SQL Server a databáze vůbec, zcela samostatnou kategorií je pak třeba služba Active Directory. V těchto případech tvůrci těží především z klíčových vymožeností PowerShellu, kterými jsou objektové zpracování dat a především objektová roura. Do třetice můžeme zmínit nastoupený trend, který prosazují sami tvůrci operačního systému Windows. Systém je již dodáván s PowerShellem a sadou rozšiřujících modulů, které poskytují rozhraní pro správu dalších a dalších součástí Windows.

Poslední tři léta byla pro příznivce PowerShellu obzvláště plodná a zajímavá, neboť tvůrčí tým pracoval na verzi 2, která byla nakonec uvolněna ve druhé polovině roku 2009 postupně pro všechny operační systémy Windows počínaje verzí XP. Druhá oficiální verze představuje velmi výrazný posun vpřed a přináší vylepšení v mnoha směrech –

nacházíme zde zásadní koncepční rozšíření (vzdálený přístup, úlohy na pozadí, zpracování událostí, moduly atd.), rozšiřuje se rodina cmdletů a také některé struktury jazyka nabízejí nové, významné možnosti (kupříkladu funkce jako vlastní cmdlety). Verze 2 je již automaticky distribuována jako součást nejnovějších verzí Windows (7 a 2008 R2) a konečně též pracuje – jak bychom koneckonců očekávali – i ve variantě Core serverového operačního systému. Stal se tak opravdu univerzálním rozhraním pro správu, na němž se bude do budoucna stavět. Svou roli PowerShell ve verzi 2 potvrdil i tam, kde už způsobil jeden šok: produkt MS Exchange 2010 je opět průkopníkem, neboť jeho administrátorské konzoly jsou postaveny právě na nové verzi a grafická nadstavba je pak závislá právě na implementaci PowerShellu.

Na tomto místě je potřeba zmínit, že správce prozatím nemůže pomocí PowerShellu ovládat úplně vše, co by v operačním systému či aplikacích chtěl. Některé části Windows a řada nastavení prozatím nemají vhodné rozhraní, které by umožnilo přístup pomocí prostředků PowerShellu, byť zprostředkovaně. Týká se to kupříkladu síťových funkcí a řady služeb, jež se sítěmi souvisejí, ale i řady jiných systémových služeb, pro jejichž ovládání potřebujeme speciální rozhraní, často reprezentované specifickým programem pro textovou konzolu. Řada úloh je také pomocí PowerShellu hůře zvladatelná než prostřednictvím jiného, dosud tradičně používaného nástroje (třeba některá nastavení oprávnění na souborový systém či objekty Active Directory). Tato situace však bude pouze dočasná, neboť PowerShell se dravě vyvíjí a funkce, které nezvládne připravit samotný autorský tým, se třeba objevují v příspěvcích autorů z komunity volného softwaru. Tato zmínka nás přivádí k další důležité skutečnosti: PowerShell dokázal vzbudit velký zájem tvůrců a v současné době najdeme řadu projektů, které doplňují a rozšiřují jeho možnosti.

Ani v případě, že jsme k prosazování PowerShellu stále mírně skeptičtí, bychom neměli přehlížet právě skutečnost, že Microsoft jej využívá stále více jako nosnou technologii pro správu svých produktů. Vedle Exchange, o kterém již padla zmínka, se PowerShell objevuje v dalším klíčovém produktu SharePoint, kde plní roli nosného správcovského rozhraní, a třeba služba Active Directory již nabízí ve verzi Server 2008 R2 některé funkce, které jinak než v PowerShellu nemůžeme ovládat. Tento trend se zdá zřejmý a nezvratný.

PowerShell ve verzi 2 se tedy zdá být onou „pravou“ chvílí, kdy bychom se nejpozději měli začít s novou koncepcí správy seznamovat. Jistě, i do budoucna zde budou grafické konzoly, a jejich pravověrní příznivci se na ně budou moci spolehnout. Ale třeba již zmíněný Exchange jasně ukázal, jaké pravidlo bude do budoucna platit: dobře ovládnout znamená dobře ovládnout pomocí PowerShellu. Neznalost této technologie bude znamenat hendikep, jehož velikost se bude zvyšovat úměrně s pronikáním PowerShellu do všech klíčových produktů Microsoftu. A je docela možné, že ona zlomová chvíle již nastala: možná že již dnes neznalost PowerShellu znamená dobrovolný vstup do „druhé ligy“ správců platformy Windows a dalších aplikací jejich výrobce.

V následujících kapitolách čtenář nalezne řadu úloh z různých zákoutí používání PowerShellu a také z všemožných oblastí správy Windows. V mnoha případech si ukážeme, jak dravým způsobem se PowerShell prosazuje – tu a tam se zatím dotahuje na pozice jiných nástrojů pro správu, místy ale již výrazně vede a představuje nejlepší cestu k pro-

vedení té či oné práce. V každém případě ale představíme čtenáři konzistentní, funkční a velmi užitečné prostředí pro moderní administraci, na jehož základních kamenech lze spolehlivě vybudovat rozsáhlý aparát pro pokročilou správu sítí se systémy Windows prakticky jakékoliv velikosti.

KAPITOLA 2

PowerShell jako software

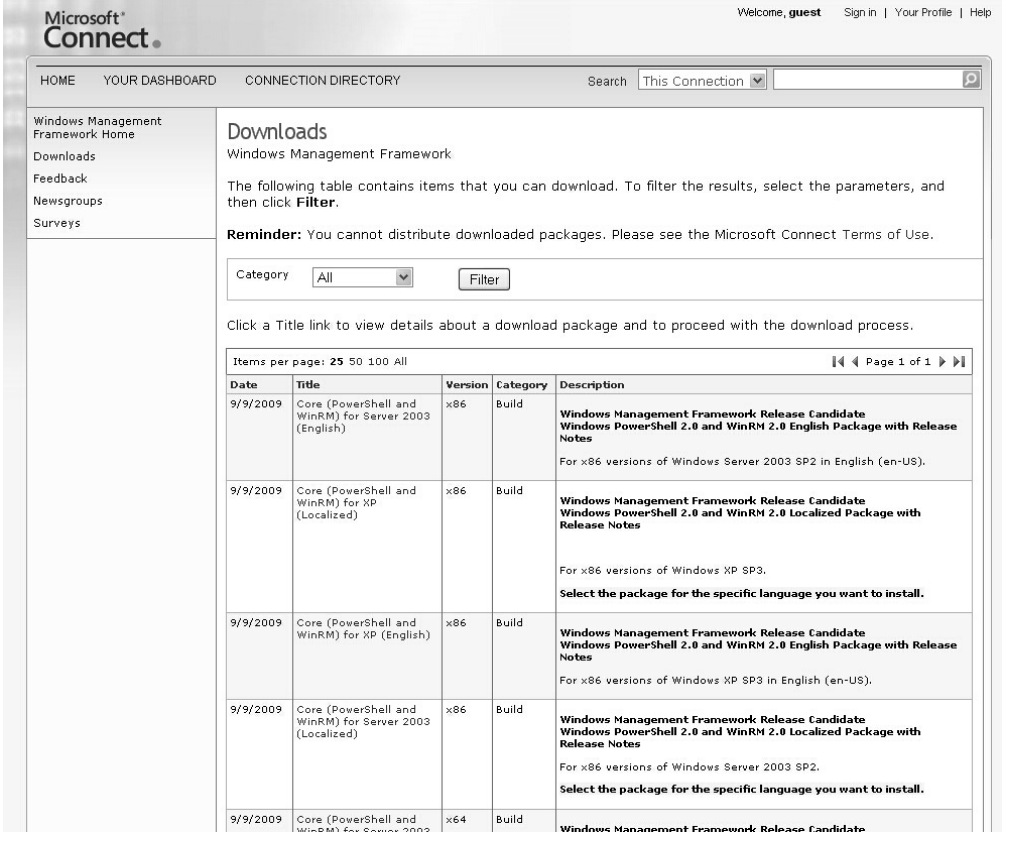
V této kapitole:

- Verze a varianty PowerShellu
- Instalace PowerShellu
- Rozšíření a doplňky PowerShellu

Existující implementace a verze PowerShellu

PowerShell se dosud dočkal dvou oficiálních samostatných verzí, které byly uvedeny do světa a označeny jako „finální“. Jeho vývoj však byl a je bouřlivý a usilovný, což se projevuje vnášením zmatku do množství dostupných verzí a variant, často dodávaných s „mateřským“ produktem jako jeho správcovské rozhraní. V této části se tedy budeme blíže věnovat verzím PowerShellu a různým odrůdám, jež se postupně rodí s uváděním nových verzí různých produktů Microsoftu.

Na počátku zkusme vytyčit základní orientační body – distribuce PowerShellu si hrubě rozdělíme na dvě velké skupiny. V jedné z nich budou zařazeny základní distribuce (PowerShell v úzkém slova smyslu), tedy samotné „jádro“ PowerShellu (core). Druhá skupina naší klasifikace pak zahrnuje odvozené varianty (odrůdy), které přidávají k jádru další možnosti či rozšíření. Tato bývají typicky poplatná samotnému produktu, se kterým je takovýto PowerShell dodáván (zmiňme první z nich, Exchange 2007 a jeho EMS – Exchange Management Shell).



Obrázek 2.1: Stránky Microsoft Connect jsou výchozím místem jak pro pořízení nejnovější verze PowerShellu, tak pro zaslání připomínek a námětů na další vylepšení

První skupina – jádra PowerShellu – dnes zahrnuje dvě finální verze, označované jako 1 a 2. Obě jsou plně funkční a použitelné, verze 2 je pak vybavena velkým množstvím nových možností oproti o tři roky starší verzi 1. V současné době platí, že obě tyto verze jsou dostupné jako samostatný instalační balík pro Windows od verze XP až po současnost (dělící čarou do minulosti jsou Windows 2000, pro něž není PowerShell distribuován). Samostatný instalační balík není potřeba shánět u verzí Windows, které obsahují PowerShell jako svoji výchozí součást. Tabulka nám ukazuje tyto systémy a verzi jádra PowerShellu.

Tabulka 2.1: Verze jádra PowerShellu a operační systémy, v nichž je předinstalován

Verze Windows	Verze jádra PowerShellu
Windows 2008 Server (ne varianty „Core“)	PowerShell 1.0
Windows 7	PowerShell 2.0
Windows 2008 Server R2	PowerShell 2.0

Ostatní systémy, jež nejsou v tabulce zmíněny, neobsahují PowerShell ve své základní instalaci a tento musí být přidán dodatečně (Windows XP, 2003, 2003 R2, Vista). Dodejme, že v současné době je jádro PowerShellu (tedy ona základní distribuce PowerShellu 2 a těsně souvisejících knihoven) označováno jako Windows Management Framework.

Tip: Čtenář může najít oficiální informace o uvedení tohoto balíku v následujícím článku, který je součástí oficiální podpory společnosti Microsoft:

http://support.microsoft.com/kb/968929

Na této stránce jsou i odkazy na stahování instalačních balíků.

Druhá skupina, kterou jsme si výše vymezili, pak zahrnuje rozšířené varianty PowerShellu, jež jsou doplněny o různé doplňky, moduly apod. Základem je vždy jádro (core), k němuž jsou přidána rozšíření buďto ve formě snap-inu (především ve verzi 1), nebo modulu (ve verzi 2). Takovéto varianty jsou dodávány s aplikacemi Microsoftu, a jsou tedy poskytovány jako rozhraní pro aplikační správu. Těmto rozšířeným variantám se budeme blíže věnovat v jedné z dalších kapitol.

Tabulka 2.2: Významné aplikované varianty PowerShellu v produktech firmy Microsoft

Produkt	Rozšíření PowerShellu	Funkcionalita
MS Exchange 2007	Exchange Management Shell	Správa služby Exchange
MS Exchange 2010	Exchange Management Shell	Správa služby Exchange
Rodina MS System Center	Management Shell	Správa aplikací
MS SQL Server 2008	SQL Server PowerShell	Správa databázového serveru a databází

Verze CTP a související potíže

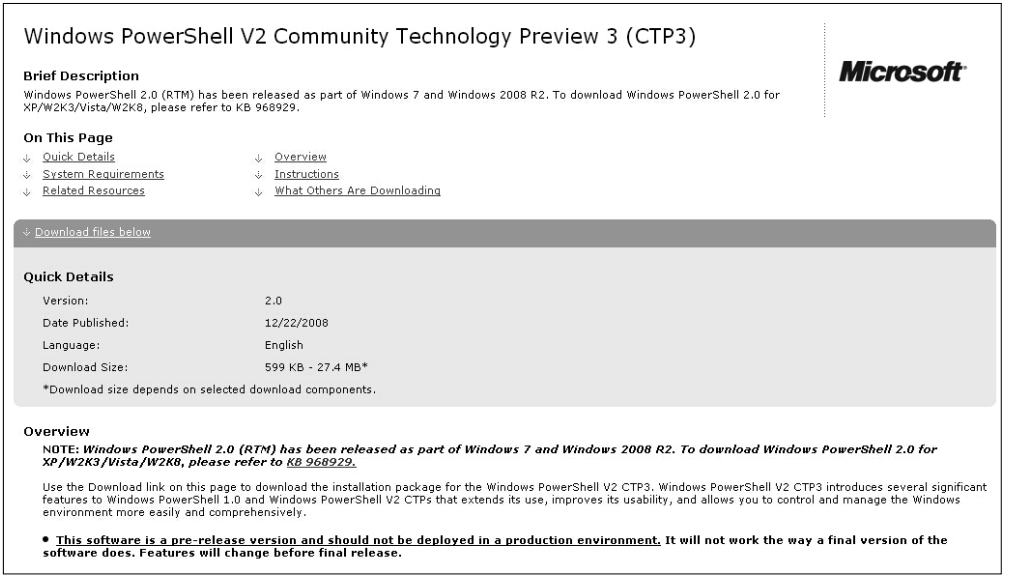
Jak jsme již zmínili, vývoj jádra PowerShellu byl a je bouřlivý, což se mimo jiné projevilo postupným uvolňováním testovacích verzí. Mezi verzemi PowerShell 1 a PowerShell 2 byly postupně uvolněny tři testovací verze, jež byly veřejně dostupné a pro jejichž

označení se vžily zkratky CTP (community technology preview – tedy něco ve smyslu ukázkové verze pro nedočkavé příznivce). Tyto distribuce měly a mají pro vývoj PowerShellu obrovský význam, neboť vývojový tým bedlivě naslouchá připomínkám uživatelů a průběžně je zapracovává do dalších verzí.

Tip: Zájemcům o vylepšení PowerShellu doporučujeme navštívit následující stránky, kde lze přispět do diskuze a oznámit případné chyby:

https://connect.microsoft.com/windowsmanagement

Testovací verze CTP však s sebou nesou jisté riziko. Vyznačují se totiž odlišnostmi (někdy dosti výraznými) od závěrečné, oficiální uvolněné verze PowerShellu 2. To, co funguje v některé z verzí CTP (PowerShell 2 měl postupně takové verze tři), nemusí fungovat ve finální podobě, případně to může fungovat jinak, pod jiným jménem apod. Jinak řečeno, během vývoje testovacích verzí se mění jména cmdletů, jména jejich přepínačů, někdy cmdlety i jejich přepínače zcela mizí a objevují se jiné. PowerShell se tedy může vyvíjet dosti dramaticky a my to musíme mít na zřeteli.



Obrázek 2.2: Vývojové verze PowerShellu jsou stále dostupné ke stahování. K jejich použití již ale není žádný důvod – finální verze jsou k dispozici.

S vývojem verzí CTP souvisí i druhotný problém. Vývoj PowerShellu 2, který trval několik let, poskytl zájemcům verze CTP, na jejichž základě vznikla spousta dokumentace, ale také článků na Internetu s řadou příkladů a řešení. Tato „powershellová“ literatura však trpí tím, že se řada věcí v PowerShellu změnila a třeba příklady nemusí fungovat v uvedené podobě. Pak musíme sáhnout k jejich úpravě dle aktuálního stavu. Tyto skutečnosti musí uživatel PowerShellu stále respektovat a ke zdrojům na Internetu musí přistupovat s odpovídající rezervou.

Tip: Tobias Weltner, přispěvatel blogu PowerShell.com a autor pěkné, zdarma dostupné knihy o PowerShellu, napsal krásný článek o problematice kontroly verzí PowerShellu. Na následující stránce je krom výkladu také ukázka skriptů, které nám mohou pomoci.

http://powershell.com/cs/blogs/tobias/archive/2010/01/24/are-you-using-the-correct-powershell-version.aspx

Tip: Jiný autor zajímavých stránek, Oisin Grehan, se také věnuje důslednému sledování vývoje verzí PowerShellu a jejich proměn. Řadu informací naleznete na jeho blogu, především v těchto článcích:

http://www.nivot.org/2009/05/22/PowerShellV20DifferencesBetweenCTP3Win7BetaAndWin7RC.aspx

http://www.nivot.org/2008/12/23/PowerShell20CTP3HasArrived.aspx

http://blogs.msdn.com/powershell/archive/2008/12/24/windows-powershell-ctp2-to-ctp3-conversion-guide.aspx

Správa služby Active Directory

Adresářová služba Active Directory patří mezi nejdůležitější služby v sítích s Windows, a proto asi vypadá zvláštně, že její správě se tvůrci PowerShellu začali věnovat až v době vývoje verze 2. V první verzi vlastně neexistuje žádná speciální podpora pro práci s Active Directory (myšleny cmdlety, provider pro PSDrive apod.) a veškerá správa se musí provádět nepřímo, pomocí existujících rozhraní (starší ADSI, nověji pomocí tříd .NET Frameworku). Teprve uvedení verze 2 a nových verzí operačního systému Windows s sebou konečně přineslo také odvozenou implementaci PowerShellu s modulem pro Active Directory.

Obrázek 2.3: Rozšíření Active Directory PowerShell od Microsoftu je podporováno i prostřednictvím blogu jeho autorů

Active Directory PowerShell můžeme používat právě jen v omezeném počtu nových operačních systémů, takže jeho masové rozšíření a využití teprve nastane (tabulka shrnuje jeho dostupnost). Na serverových variantách je tento odvozený PowerShell instalován automaticky spolu s dalšími nástroji na správu Active Directory, do Windows 7 jej můžeme přidat jako součást balíku RSAT (Remote Server Administration Tools). Bohužel na starších verzích Windows tato odvozená implementace správně nepracuje, a proto řada uživatelů jistě zůstane u další logické volby.

Tabulka 2.3: Zbrusu nový MS Active Directory PowerShell a jeho možnosti použití

Verze Windows	Active Directory PowerShell
Windows 2008 R2	Ano
Windows 2008	Ne
Windows 2003	Ne
Windows 7	Ano
Windows Vista	Ne
Windows XP	Ne

Tip: Podrobné informace o práci s rozšířením Active Directory PowerShell nalezne čtenář na následujících stránkách:

http://technet.microsoft.com/en-us/library/dd378937%28WS.10%29.aspx

http://blogs.msdn.com/adpowershell/default.aspx

http://blogs.msdn.com/photos/adpowershell/images/9453590/original.aspx

Pro doplnění dodejme, že omezené možnosti instalace Active Directory PowerShellu nevylučují v žádném případě správu starších řadičů domény a domén vůbec. Domény provozované na verzi serveru 2003 je možno spravovat, pokud máme v síti zapojen alespoň jeden řadič ve verzi 2008 R2.

Tip: Čtenář nalezne výchozí informace pro výše popsaný scénář na této stránce:

http://blogs.msdn.com/adpowershell/archive/2009/09/18/active-directory-management-gateway-service-released-to-web-manage-your-windows-2003-2008-dcs-using-ad-powershell.aspx

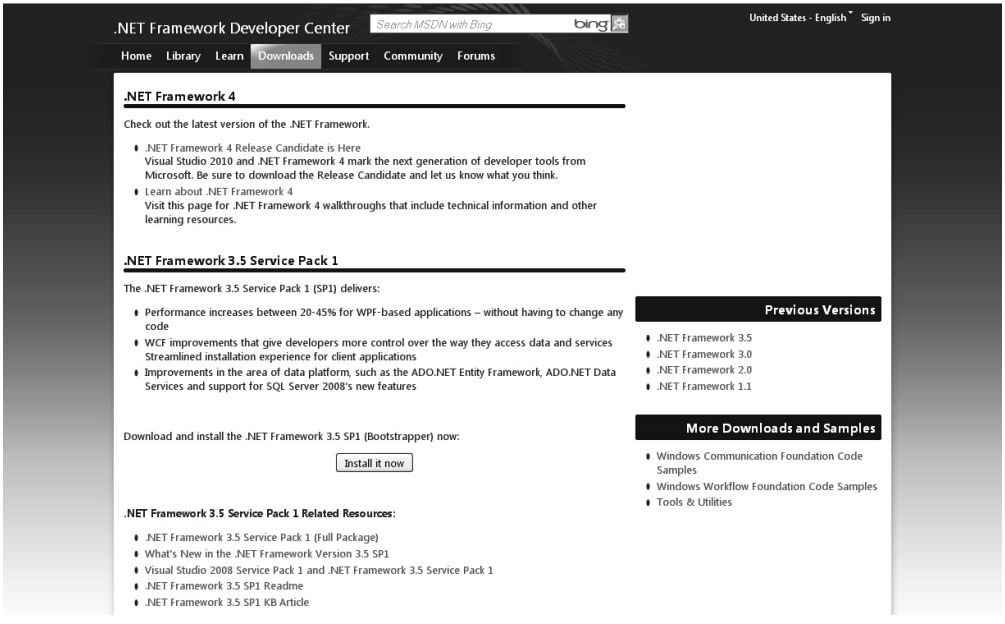
Instalace PowerShellu

Úvod

V některých verzích Windows není PowerShell nainstalován jako výchozí součást, takže musíme instalaci provést sami. Stejně tak budeme instalaci provádět u všech systémů, kde nemáme novější verzi 2.0. Celý proces je poměrně jednoduchý, a pokud splníme výchozí požadavky, tak i bezproblémový.

Instalace

PowerShell, či přesněji Windows Management Framework, můžeme nainstalovat na jakýkoliv systém Windows od verze Windows XP k novějším (s výjimkou nejnovějších verzí 7 a 2008, kde už nainstalován je).



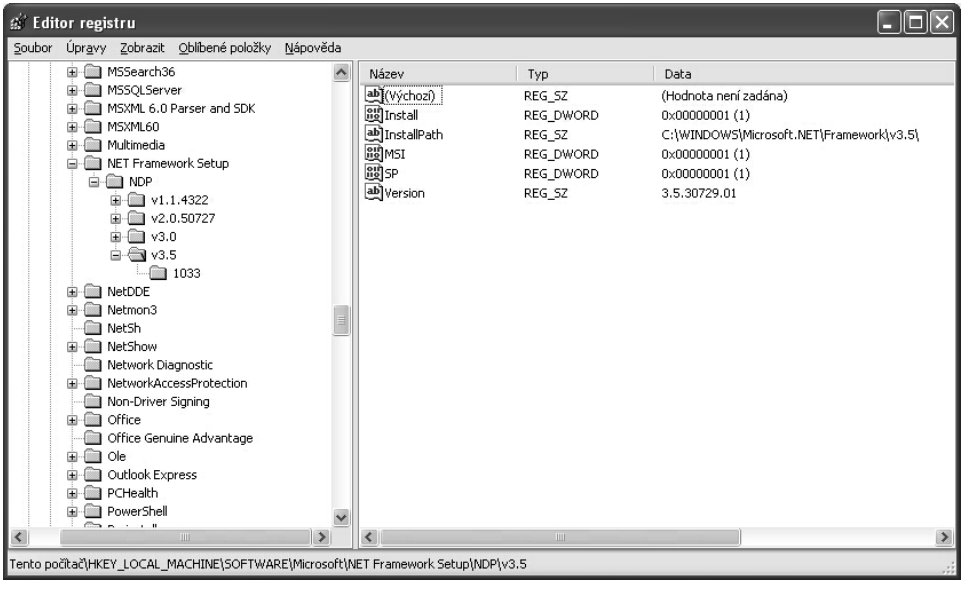
Obrázek 2.4: Před instalací se ujistíme, zda máme správnou verzi .NET Frameworku, případně ji stáhneme a nainstalujeme

1. Prověříme na cílovém počítači verzi platformy .NET Framework. V případě potřeby stáhneme a nainstalujeme nejnovější verzi. Instalované verze můžeme prověřit několika způsoby: jednak prohledáním adresáře s instalacemi, jednak prozkoumáním klíčů v databázi Registry.

```
C:\>dir %windir%\Microsoft.NET\Framework\
 Svazek v jednotce C je 120GB.
 Sériové číslo svazku je E86A-5C34.

 Výpis adresáře C:\WINDOWS\Microsoft.NET\Framework

09. 08. 2009  22:14    <DIR>          .
09. 08. 2009  22:14    <DIR>          ..
...
09. 08. 2009  22:11    <DIR>          v1.0.3705
17. 10. 2009  09:58    <DIR>          v1.1.4322
16. 09. 2006  19:37    <DIR>          v2.0.40607
01. 02. 2010  22:41    <DIR>          v2.0.50727
01. 02. 2009  15:02    <DIR>          v3.0
09. 08. 2009  22:14    <DIR>          v3.5
...
```

V Registry prozkoumáme následující klíč:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\NET Framework Setup\NDP
```



Obrázek 2.5: Příslušný klíč v databázi Registry nese informace o instalovaných verzích .NET Frameworku.

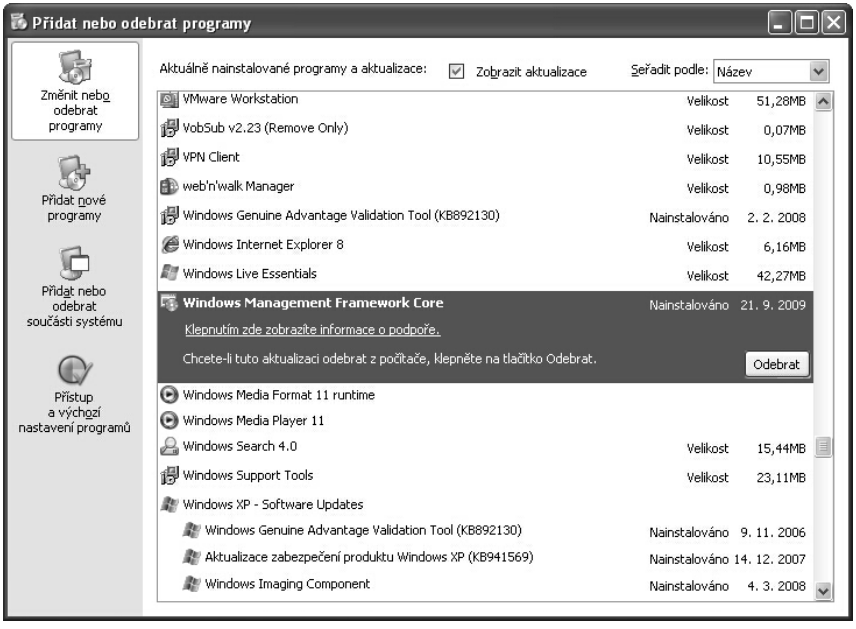
Tip: Aktuální verze .NET Frameworku je k dispozici na těchto stránkách:
http://msdn.microsoft.com/en-us/netframework/default.aspx
Podrobné informace o označení verzí pak najdeme zde:
http://msdn.microsoft.com/en-us/kb/kb00318785.aspx

Balík po stažení nainstalujeme pomocí spuštění jednoduchého průvodce.

2. Pokud pracujeme se systémem Windows XP, prověříme, že je nainstalován balík Service Pack 3, případně jej stáhneme a nainstalujeme.

Tip: Distribuční varianta SP3 pro Windows XP je ke stažení zde:
http://www.microsoft.com/downloads/details.aspx?FamilyId=5B33B5A8-5E76-401F-BE08-1E1555D4F3D4&displaylang=en

3. Přejdeme na stránku podpory PowerShellu a vyhledáme stažení potřebné verze shellu: *http://support.microsoft.com/kb/968929*
4. Spustíme instalační balíček a prostě dokončíme grafický průvodce.



Obrázek 2.6: Nezapomínejme, že PowerShell v nejnovější verzi je instalován jako Windows Management Framework a pod tímto jménem také figuruje u služby Windows Installer

Shrnutí

PowerShell je distribuován v dobře popsaných instalačních balících, jež obsahují nejdůležitější součásti, včetně potřebné verze služby Windows Remote Management pro vzdálenou správu. Základem stoprocentní funkcionality PowerShellu je ovšem .NET Framework, a proto bychom neměli váhat s instalací jeho poslední verze všude tam, kde je to možné.

Důležité k zapamatování

- Operační systémy Windows Server 2008 R2 a Windows 7 mají PowerShell 2 jako svou součást a nevyžadují žádnou další instalaci.
- Operační systém Windows Server 2008 obsahuje verzi PowerShell 1. Novější verzi je potřeba doinstalovat.
- Operační systém Windows Server 2008 Core nemá oficiálně podporován PowerShell. Jeho instalaci můžeme provést způsobem, která je popsán na následující internetové stránce. Tento postup není podporován společností Microsoft. *http://dmitrysotnikov.wordpress.com/2008/05/15/powershell-on-server-core/*
- Systém Windows XP vyžaduje instalaci SP3 pro PowerShell 2.

Rozšíření a doplňky

Úvod

PowerShell je prozatím určitě „mladý produkt“, myšleno ve srovnání s dobou vývoje jiného osvědčeného softwaru. Jeho nízký věk však výrazně kontrastuje s počtem uživatelů, vývojářů a softwarových firem, kteří byli vyprovokováni a chytili hozenou rukavici. Díky tomu dnes můžeme směle instalovat pro PowerShell řadu zajímavých doplňků a rozšíření.

Windows System Modules

Nástup PowerShellu ve verzi 2 znamenal také masivní přechod k distribuci různých rozšíření prostřednictvím tzv. modulů (zmíníme se o nich podrobněji v následující kapitole). Nejnovější verze Windows (Server 2008 R2) tak již přichází s řadou zajímavých modulů, které jsou připraveny ve výchozím adresáři PowerShellu a můžeme je rovnou „natáhnout“ a začít používat. Zmiňme třeba Active Directory PowerShell, podporu pro přenos souborů BITS, ale také třeba modul pro ovládání služby IIS. Grafické rozhraní Serveru 2008 R2 též nabízí přímý odkaz na spuštění konzoly PowerShellu, která automaticky tyto moduly natahuje a dává k dipozici. Operaci můžeme provést také sami kdykoliv později následujícím způsobem:

```
=>powershell -importsystemmodules
```

Windows 7 PowerShell Pack

http://code.msdn.microsoft.com/PowerShellPack

http://blogs.msdn.com/powershell/archive/2009/10/15/introducing-the-windows-7-resource-kit-powershell-pack.aspx

Tento balík je vlastně sbírkou zajímavých skriptů, modulů a funkcí pro PowerShell, které se hodí pro správu a také pro některé další experimenty s PowerShellem. Najdeme zde skripty pro tvorbu grafického rozhraní, ale také rozhraní pro práci se službou Plánování úloh či inventuru operačního systému. Balík vznikl postupným sbíráním řešení, jež poskytl vývojový tým a především autoři stránek podpory PowerShellu (populární rubriky The Scripting Guy).

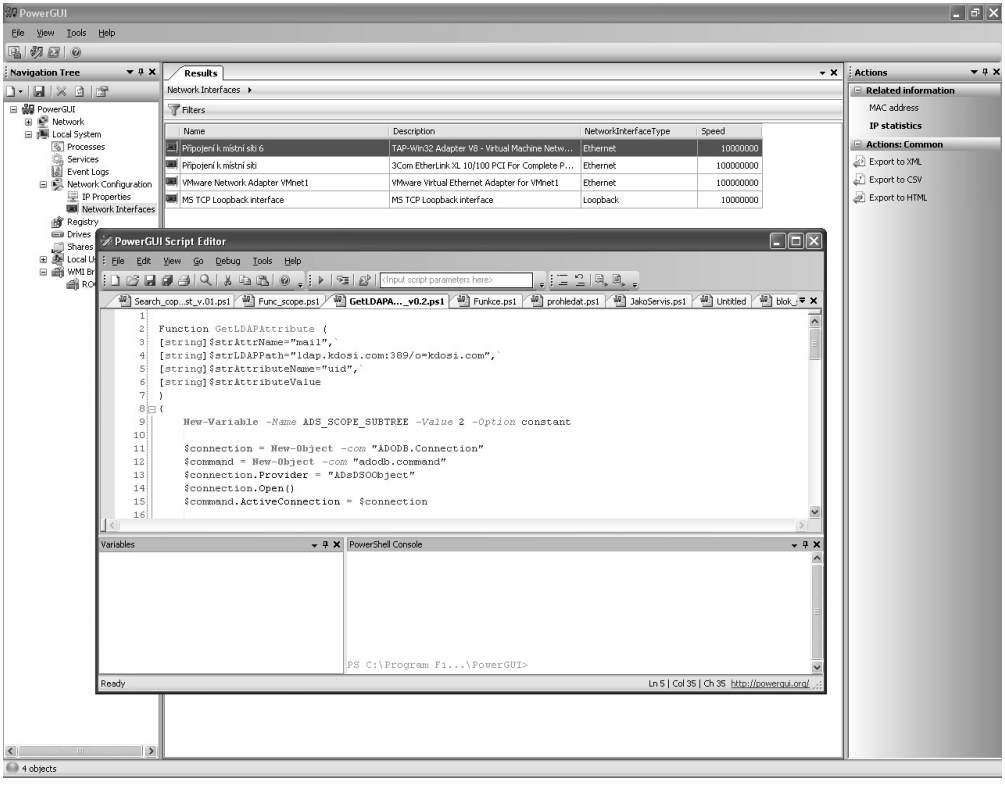
PowerGUI

www.powergui.org

PowerGUI bezesporu patří mezi nejlepší rozšíření, jež byla na bázi PowerShellu vyvinuta. Zahrnuje v sobě dva klíčové nástroje, jež poslouží stejně dobře správcům jako všem tvůrcům skriptů v jazyce PowerShell. Software je vyvíjen a distribuován jako volný – jednak na něm pracují dobrovolní přispěvatelé, jednak je podporován softwarovou firmou Quest Software.

Balík zahrnuje dvě klíčové části. Samotné PowerGUI je grafické rozhraní pro správu Windows a případných dalších aplikací – ano, stále mluvíme o PowerShellu a zároveň

o grafickém správcovském rozhraní. PowerGUI je zcela postaveno na PowerShellu a všechny operace, které „naklikáme“, jsou převedeny na kód v PowerShellu a v něm také provedeny. Druhá část balíku se jmenuje PowerGUI Script Editor a jde o integrované vývojové rozhraní (IDE), v němž můžeme psát skripty a ladit jejich činnost pomocí ladicího rozhraní.



Obrázek 2.7: PowerGUI patří rozhodně mezi to nejlepší, co kdy pro PowerShell vzniklo.

PowerShell Commands for Active Directory

http://www.quest.com/powershell/activeroles-server.aspx

Tento doplněk je jedním z nejlepších a vůbec nejpopulárnějších rozšíření nového správcovského rozhraní. Objevil se v dobách PowerShellu 1, který nenabízel žádnou vlastní přímočarou variantu pro správu Active Directory, a znamenal bez nadsázky revoluci v používání PowerShellu. Ta se projevila mimo jiné tím, že řada správců začala používat PowerShell právě kvůli Active Directory a tomuto rozšíření, a nejeden uživatel dokonce ani nepostřehl, že tato sada cmdletů nepohází z původní dílny Microsoftu! Balík se stal standardem de facto pro práci s Active Directory a my mu budeme v příslušné kapitole věnovat odpovídající pozornost. Je k dispozici zdarma a spolu s ním lze stáhnout též špičkovou doprovodnou dokumentaci.

Windows 7 Troubleshooting Platform

http://msdn.microsoft.com/en-us/library/dd776530.aspx

http://technet.microsoft.com/en-us/library/ee424304%28WS.10%29.aspx

http://www.withinwindows.com/2009/01/12/crash-course-on-authoring-windows-7-troubleshooting-packs/

http://blogs.msdn.com/powershell/archive/2009/06/14/kudos-to-the-win7-diagnostics-team.aspx

Operační systém Windows 7 obsahuje zajímavou novinku pod výše uvedeným názvem. Jedná se o sbírku skriptů v PowerShellu a doplňující soubory XML s popisy, které slouží k řešení různých typů problémů s operačním systémem. Určitě nejde o nahodilý počin, neboť můžeme vyhledat dokumentaci a začít sami tvořit další skripty a balíčky, které budou toto „ladicí centrum" doplňovat a rozšiřovat. Jde o velmi zajímavou a bezprostřední ukázku využití nové verze PowerShellu.

Windows PowerShell 2.0 SDK

http://go.microsoft.com/fwlink/?LinkID=180421

Tento balík je určen pokročilým zájemcům o PowerShell a programátorům aplikací na platformě .NET, kteří chtějí s PowerShellem pracovat. Jde o podrobnou dokumentaci jádra PowerShellu a popis jeho architektury. Najdete zde i řadu příkladů, ukázkových knihoven a skriptů, které mimo jiné ukazují, jak můžeme PowerShell rozšiřovat pomocí vlastních cmdletů.

Ačkoliv tento balík vypadá jako dosti specializovaný doplněk, neměli by jej přehlížet zájemci o tvorbu cmdletů pomocí samotného PowerShellu naleznou v něm popis potřebných objektů a ukázky obecných možností cmdletů, které lze využívat i ve funkcích v jazyce PowerShell.

PowerShell Community Extensions

http://www.codeplex.com/Pscx

Velmi populární balík rozšíření je jedním z nejlepších příspěvků komunity dobrovolných tvůrců a přispěvatelů, která se vytvořila kolem PowerShellu. Je velmi populární již od svého prvního uvedení a přinesl řadu vylepšení, kdy především po uvolnění PowerShellu verze 1 měli uživatelé pocit, že autoři mnoho věcí nestihli. Balík mimo jiné obsahuje implementaci Active Directory do podoby PSDrive, ale také řadu zajímavých funkcí a cmdletů, třeba pro práci s texty.

Shrnutí

PowerShell je již sám o sobě velmi silným nástrojem, avšak některá dostupná rozšíření jej proměňují v ještě všestrannější platformu pro ovládání aplikací a služeb. Doplňky jsou k dispozici jak ve starším provedení (snap-iny), tak především v novějších variantách jako moduly. Některá rozšíření se již stala v podstatě nezbytnou součástí shellu pro správu pokročilého prostředí na platformě Windows.

Důležité k zapamatování

- PowerGUI je jedním z nejlepších doplňků, který si můžeme nainstalovat – nabízí jak zajímavou grafickou nadstavbu, tak kvalitní editor pro tvorbu a ladění skriptů v PowerShellu.
- Quest PowerShell Commands for Active Directory je výborný balík pro všechny verze PowerShellu na správu Active Directory. Jedná se o prakticky nezbytné vybavení každého správce této služby.

Otázky a odpovědi

Otázka:

Můžeme nainstalovat PowerShell na všechny dostupné verze operačního systému Windows?

Odpověď:

Ne, nemůžeme. PowerShell závisí na dalších technologiích (především běhovém prostředí .NET Framework) a pracuje na Windows XP a novějších. PowerShell nelze použít na verze 2000 a starší.

Otázka:

Co jsou verze PowerShell 2 CTP (1, 2 nebo 3)? Mohu je bezpečně používat?

Odpověď:

Takto označené verze byly určeny k testování a není doporučeno jich využívat v produkčním prostředí. Ačkoliv jsou normálně funkční, liší se od konečné verze a některé příkazy nemusí pracovat stoprocentně.

Otázka:

Jakou verzi .NET Frameworku PowerShell vyžaduje?

Odpověď:

PowerShell běží i na starších verzích frameworku od verze 2. Avšak aby byl PowerShell plně funkční (všechny cmdlety), je potřeba nainstalovat framework ve verzi 3.5 SP1.

KAPITOLA 3

Práce v prostředí PowerShellu

V této kapitole:

- Skriptovací bloky a ovládání PowerShellu
- Vzdálené připojení v PowerShellu
- PowerShell jako webová služba
- Úlohy na pozadí a hromadné spouštění
- Automatizované úlohy
- Moduly v PowerShellu

Interaktivní práce

Úvod

Uživatelé PowerShellu, kteří nejsou úplnými začátečníky, jsou zřejmě zvyklí na dvě hlavní cesty používání PowerShellu: interaktivní provádění akcí v konzole a spouštění hotových skriptů v přímém či automatizovaném režimu. Při práci s konzolou jsme uvykli tomu, že konzolové prostředí je vlastně hřištěm a my jsme upoutání uvnitř – pokud chceme uchovat data, ukládáme je do souborů, chceme-li uchovat nastavení, rovněž provádíme export. .NET Framework nám však umožňuje mnohem více, neboť spouštění kódu v PowerShellu lze řídit takřka totálně: jakýkoliv kus kódu lze proměnit v objektovou proměnnou stejně tak jako vše, co nastane při jeho spuštění. PowerShell ve verzi 2 přináší nové možnosti, jak spouštět kód a přitom důsledně kontrolovat vše jakoby „zvenčí". Nezapomínejme tedy na to, že vše v PowerShellu je objekt.

Skriptovací blok a tělo skriptu

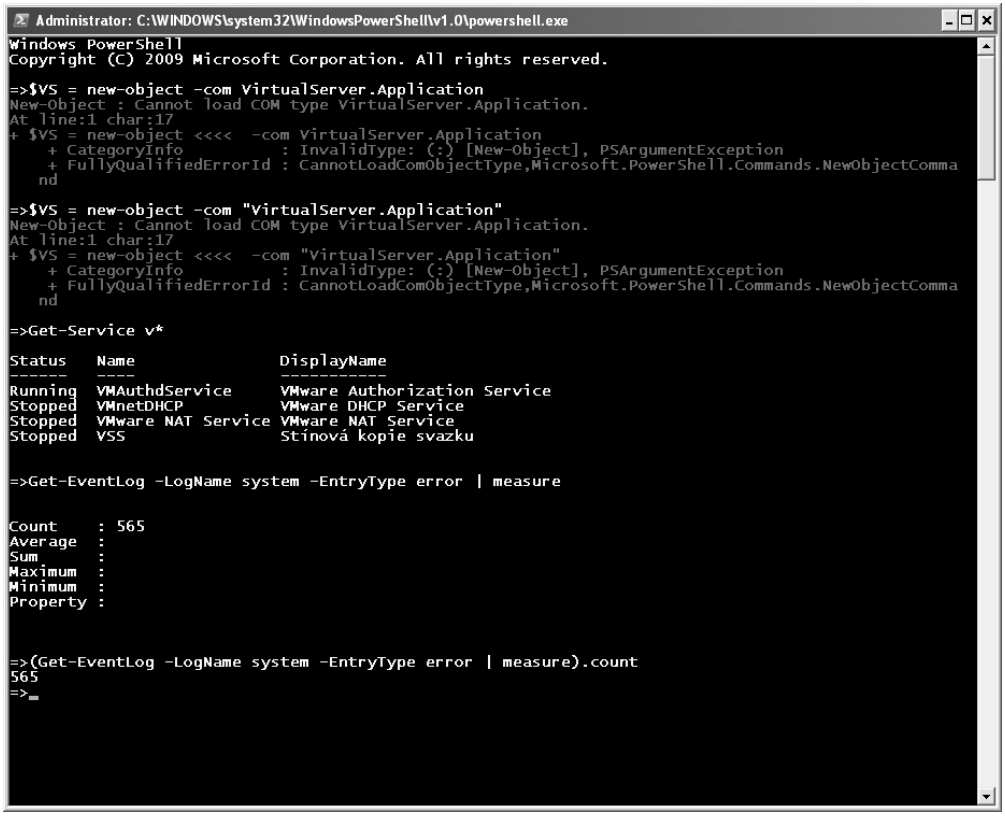
Jakýkoliv kus kódu či příkazová sekvence v PowerShellu jsou při svém běhu důsledně řízeny a kontrolovány, stejně jako jakýkoliv jiný kód na platformě .NET, na níž PowerShell běží. Skriptovací blok (blok skriptu, prováděcí blok) je základní jednotkou, do níž patří určitá sekvence příkazů a dalších struktur jazyka PowerShell. Takovýto blok a jeho ohraničení má zcela zásadní význam, neboť při jeho spuštění (*Invocation*) vzniká jakési „hřiště", které mu .NET Framework vymezí. Hranice tohoto hřiště jsou velmi důležité – představují bariéru pro řadu důležitých struktur, které v našich skriptech používáme. Skriptovací blok tedy vymezuje výchozí ohraničení pro platnost proměnných, které vzniknou uvnitř, a také třeba určuje rozsah platnosti chyb, které nastanou a mohou být zachyceny.

Hranice skriptovacího bloku vznikají buďto automaticky (spustíme konzolu, spustíme skript), nebo jsou určeny výslovně (nadefinujeme funkci či skriptovací blok jako takový). Při výslovném určování hranic používáme složené závorky, které se v praxi objevují v mnoha souvislostech.

```
=>{Write-Output "Zevnitř"}
Write-Output "Zevnitř"

=>Function Blok {"Zevnitř funkce"}

ForEach-Object {"Zevnitř roury"}
```

Všechny tyto konstrukce jsou pouze obměnou skriptovacího bloku, zasazeného do různých souvislostí. K jeho provedení (spuštění, invokaci) může dojít různými cestami.

```
=>& {"Z nepojmenovaného bloku"}
Z nepojmenovaného bloku
=>cat .\blok_skript.ps1
Write-Output "Zevnitř skriptu"
=>.\blok_skript.ps1
Zevnitř skriptu
=>
=>Function Blok {"Zevnitř funkce"}
```



Obrázek 3.1: Spuštěná konzola PowerShellu představuje samostatný „prováděcí blok" – automaticky zakládá hranice pro platnost proměnných, funkcí a dalších objektů


```
=>Blok
Zevnitř funkce
=>
=>1..3 | ForEach-Object {"Zevnitř roury"}
Zevnitř roury
Zevnitř roury
Zevnitř roury
=>if (1 -eq 1) {"Z podmínky"}
Z podmínky
```

Řada případů ukazuje skriptovací blok, který je vázaný v nějaké jiné konstrukci (třeba funkci) a tím je i pojmenován a uložen. My však můžeme blok skriptu uložit i přímo do proměnné.

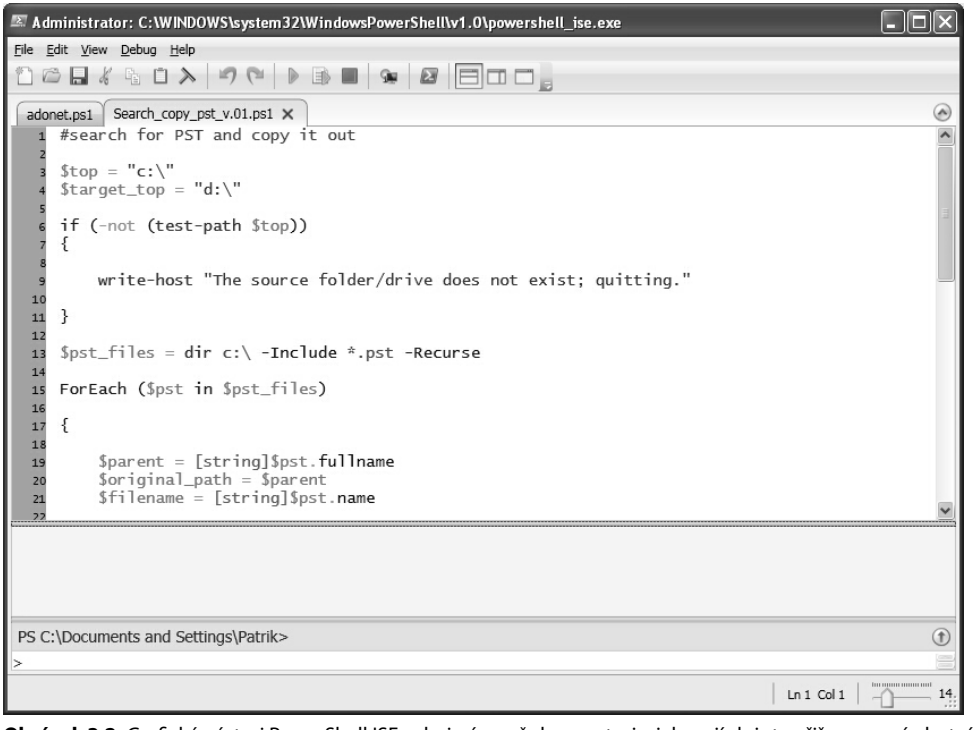
```
=>[ScriptBlock]$blok = ({"Z ulozeneho bloku"})
=>$blok
"Z ulozeneho bloku"

=>$blok.Invoke()
Z ulozeneho bloku
=>
```

```
=>$blok | Get-Member

   TypeName: System.Management.Automation.ScriptBlock
...
```

Ve všech těchto případech dochází při spuštění k vytvoření onoho „hřiště“, o němž jsme se zmínili výše – jde o pracovní prostor, v němž jsou platné proměnné, sbírány chyby atd. Tento prostor označujeme jako *Runspace.* Pokud chceme, aby nějaká data překročila hranice vymezeného hřiště, musíme je určitým způsobem „prostrčit“ přes hranice – typickou možností jsou proměnné, kterým nastavíme větší rozsah platnosti, než je ten výchozí.

blok_skript.ps1

```
=>cat .\blok_skript.ps1
$Global:vzkaz = "Zevnitř skriptu"
Write-Host $vzkaz

=>.\blok_skript.ps1
Zevnitř skriptu

=>$vzkaz
Zevnitř skriptu
=>
```



Obrázek 3.2: Grafický nástroj PowerShell ISE velmi názorně demonstruje, jak mají skripty přiřazeny své vlastní „hřiště“ – každá záložka představuje samostatný *Runspace*

V našem příkladu tedy proměnná překročila původní hranice – jako globální entita je dostupná i v nadřízeném „hřišti“, odkud byl skript (podřízené „hřiště“) zavolán. Stejným způsobem pracují třeba funkce a filtry.

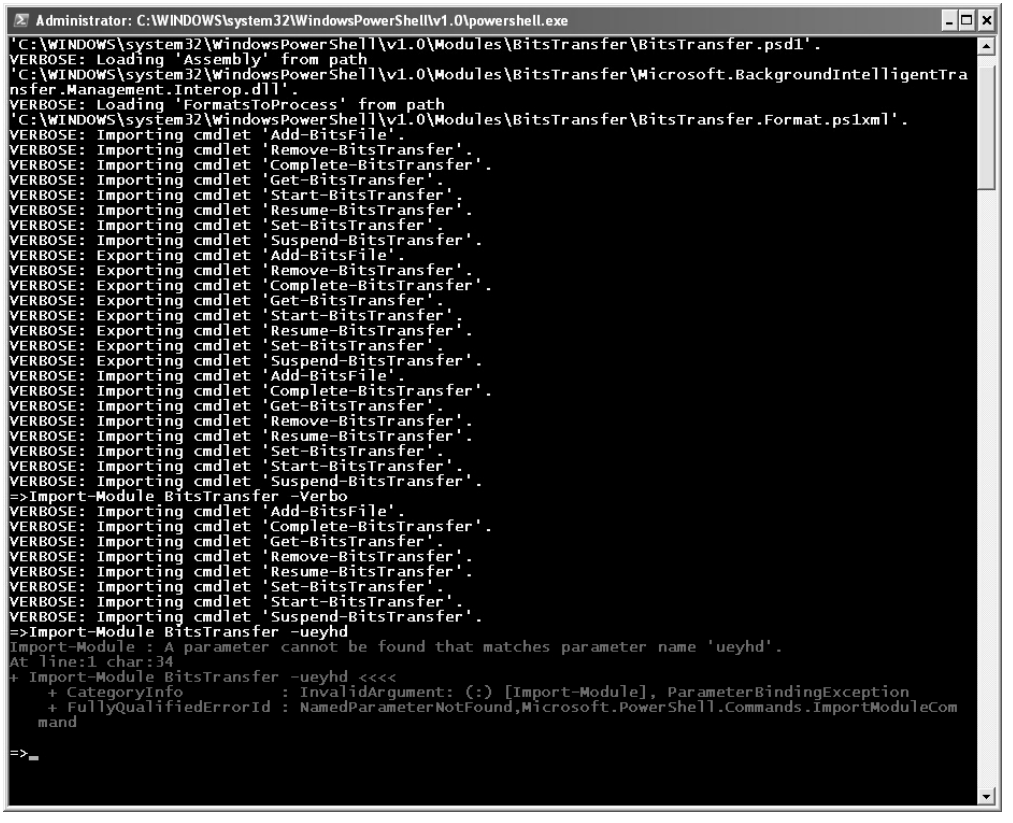
Všechny roury a kanály PowerShellu

Čtenář třeba není úplným začátečníkem v oblasti skriptování, takže si možná přinesl určité znalosti o použití příkazové konzoly a programu Cmd.exe. Příkazy (programy) v konzole i cmdlety PowerShellu pracují na první pohled stejně při předávání dat. Načítají data z tzv. standardního vstupu a vysílají je na standardní výstup. Krom toho většinou posílají chybová hlášení do tzv. chybového kanálu, který proudí nezávisle na datech, pokud neřekneme jinak. Postaru tedy pracujeme v zásadě se třemi kanály: vstupním, výstupním a chybovým. PowerShell vnáší do této ustálené situace určitý zajímavý pokrok. Jeho cmdlety totiž dokáží zasílat data do více směrů, které jsou přesně pojmenovány a určeny, a využívají tak více výstupních kanálů, s nimiž lze dále pracovat. Vstupní a výstupní datový kanál zůstávají nejdůležitějšími, ale jsou zde i další, které se mohou hodit.

Prozkoumejme několik situací. Zkusíme restartovat nějakou službu a uvidíme, jak se příkaz zachová.

```
=>Restart-Service spooler
WARNING: Waiting for service 'Zařazování tisku (spooler)' to finish starting...
=>

=>Restart-Service spooler -PassThru
WARNING: Waiting for service 'Zařazování tisku (spooler)' to finish starting...

Status   Name               DisplayName
------   ----               -----------
Running  spooler            Zařazování tisku

=>Restart-Service spooler | Out-Null
WARNING: Waiting for service 'Zařazování tisku (spooler)' to finish starting...
=>

=>Restart-Service spooler -PassThru | out-null
WARNING: Waiting for service 'Zařazování tisku (spooler)' to finish starting...
=>
```

PowerShell se na první pohled chová podivně. Ačkoliv jsme opakovaně zakončili tok dat rourou pomocí příkazu *Out-Null*, varování je nepřekonatelné a stále se objevuje. Znamená to, že varování sice končí v naší textové konzole, ale neprošlo rourou jako ostatní data – konzola je tedy „konečnou stanicí“ pro varovnou informaci, avšak ta zjevně do konzoly přišla nezávislým kanálem.



Obrázek 3.3: Textová konzola je výchozím výstupním „zobrazovačem" pro všechny kanály, jež PowerShell ovládá. Pokud neřekneme jinak, všechna data směřují sem.

Vyzkoušejme zase jiný pokus. Zkusíme nahrát modul PowerShellu pomocí standardního příkazu a přimějeme jej, aby vypsal podrobnější informaci o celé akci.

```
=>Import-Module BitsTransfer -Verbose
VERBOSE: Importing cmdlet 'Add-BitsFile'.
...
VERBOSE: Importing cmdlet 'Suspend-BitsTransfer'.

=>Import-Module BitsTransfer -Verbose | Out-Null
VERBOSE: Importing cmdlet 'Add-BitsFile'.
...
VERBOSE: Importing cmdlet 'Suspend-BitsTransfer'.
```

Ani v tomto případě jsme přesměrováním roury do „černé díry" ničeho nedosáhli, neboť podrobný výpis stále končí v naší konzole. I tento kanál tedy směruje svůj výpis do textové konzoly, avšak i v tomto případě je to pouze „závěrečný zobrazovač" a zpráva přišla jinudy, než kudy běžně proudí data.

Předchozí příklady ukazují, že cmdlety PowerShellu mohou komunikovat paralelně pomocí několika různých datových proudů (kanálů). To však neznamená, že těmito

kanály vždy posílají data – jde o teoretickou možnost a mnohé cmdlety zdaleka nevyužívají celou šíři těchto možností.

Poznámka: Na tomto místě se možná naskýtá otázka, proč se vlastně PowerShell chová při zasílání svých dat tak, jak nyní popisujeme. Proč máme hned několik výstupních kanálů namísto jednoho, kterým by klidně mohlo proudit všechno, obzvláště když to nakonec skončí v textové konzole? Jak jsme viděli, přináší to vlastně pouze problémy!

Inu, několik dobrých důvodů se jistě najde. V první řadě, několik kanálů máme k dispozici právě proto, aby se nám různé výstupy nepletly do jednoho guláše! PowerShell se snaží držet čistá „výstupní" data – požadovaná informace je zaslána hlavním výstupním datovým kanálem (tedy tím, co označujeme jako roura). Pokud cmdlety produkují dodatečné informace, jsou tyto schválně izolovány, aby nám požadovaná data „neznečišťovaly".

S tím souvisí další možnost, a to nasměrovat tyto dodatečné kanály odděleně do vlastních proměnných. Můžeme tak získat specifická data v čisté, oddělené podobě. Další výhodou je, že můžeme řídit bezprostřední výstup pro každý kanál odděleně, a tak ovládat, jaká data nakonec skončí ve kterém výstupním proudu (a třeba v textové konzole). Na závěr uveďme zdaleka ne poslední vymoženost: do jednotlivých kanálů můžeme odděleně zapisovat vlastní data pomocí sady odpovídajících cmdletů.

S výstupními kanály lze různě pracovat a možnosti jsou shrnuty v tabulce 3.1. Do výstupních kanálu lze zapisovat data dle našeho přání, a obráceně: namísto do konzoly můžeme různé výstupní kanály přesměrovat do proměnných. Povšimněme si také sloupců, jež uvádějí Řídicí proměnnou a Jednotlivé řízení – hlavní proměnná určuje chování výstupního kanálu pro aktuální rozsah platnosti (relaci v konzole, skript...), kdežto přepínače u jednotlivých příkazů vynucují chování právě pro daný cmdlet bez ohledu na globální nastavení (a mají tedy přednost).

Tabulka 3.1: Přehled datových kanálů PowerShellu a odpovídajících proměnných, cmdletů a přepínačů

Kanál	Zápis	Řídicí proměnná	Do proměnné	Jednotlivé řízení
StdOut	Write-Host		-OutVariable	
	Write-Output			
StdIn	Read-Host			
Error	Write-Error	ErrorActionPreference	-ErrorVariable	-ErrorAction
Debug	Write-Debug	DebugPreference		-Debug
Verbose	Write-Verbose	VerbosePreference		-Verbose
Warning	Write-Warning	WarningPreference	-WarningVariable	-WarningAction
Progress	Write-Progress	ProgressPreference		

Ukažme si prakticky, jaký je vztah mezi těmito nastaveními.

```
=>$VerbosePreference
SilentlyContinue

=>Import-Module BitsTransfer
```

```
=>$VerbosePreference = "Continue"

=>Import-Module BitsTransfer
VERBOSE: Importing cmdlet 'Add-BitsFile'.
VERBOSE: Importing cmdlet 'Complete-BitsTransfer'.
VERBOSE: Importing cmdlet 'Get-BitsTransfer'.
VERBOSE: Importing cmdlet 'Remove-BitsTransfer'.
VERBOSE: Importing cmdlet 'Resume-BitsTransfer'.
VERBOSE: Importing cmdlet 'Set-BitsTransfer'.
VERBOSE: Importing cmdlet 'Start-BitsTransfer'.
VERBOSE: Importing cmdlet 'Suspend-BitsTransfer'.
```

Na počátku jsme si prověřili nastavení řídicí proměnné, která určuje, zda se podrobný výpis (Verbose) má zasílat do konzoly, a její určující funkci jsme ověřili spuštěním příkazu (už víme, že přepínač *-Verbose* by vše změnil, neboť má přednost). Následně je vidět změnu chování spolu se změnou proměnné.

Tip: Přepínače uvedené v naší tabulce patří do kategorie „obecných" přepínačů, které se vyskytují univerzálně u řady příkazů. Podrobnosti se o nich dočteme ve vynikajícím tématu nápovědy PowerShellu: Get-Help about_CommonParameters.

Obrázek 3.4: Kanál *Progress* je schopen vykreslit do konzoly stavový pruh s průběžnou informací o probíhající operaci

Řídicí proměnné mají vliv nejen na cmdlety, jež různé kanály využívají, ale také na samotné příkazy zápisu do těchto kanálů, jak si můžeme ověřit.

```
=>$DebugPreference = "silentlycontinue"
=>$DebugPreference
SilentlyContinue
=>Write-Debug "Debug"
=>
=>$DebugPreference = "continue"
=>Write-Debug "Debug"
DEBUG: Debug
```

V tuto chvíli dodejme, že PowerShellu vlastně chybí jednoduchý mechanizmus, pomoci kterého bychom mohli snadno vybrat jednotlivé kanály, sloučit je v jeden proud dat a přesměrovat je třeba do výstupního souboru. Takovouto operaci můžeme udělat, ovšem pokročilejším způsobem, jak si ukážeme dále.

Tip: Zapáleným zájemcům k podrobnějšímu prozkoumání této problematiky autor doporučuje perfektní skript Oisina Grehana, který najdete na jeho blogu. Jde o modul, který právě umožňuje soustředit všechny kanály do jednoho výstupu. Jde o krásnou ukázku využití pokročilých nových možností PowerShellu 2.

http://www.nivot.org/Trackback.aspx?guid=21136a8a-636d-4665-8637-e93a24bbd61d

Uchopení příkazového bloku

V předchozích příkladech jsme si ukazovali skriptovací blok, jeho ohraničení a také význam pro vytvoření rozsahu (scopu) proměnných či třeba chybových stavů. V běžných situacích považujeme skriptovací blok prostě za automatickou součást našich řešení – ve smyčce *ForEach* prostě udělá opakovaně to, co potřebujeme, za podmínkou *If* se provede podmíněná část kódu, a pokud je skriptovací blok založen jako celý spustitelný skript, tak prostě předpokládáme, že po jeho spuštění se uvnitř odehraje vše na „připraveném hřišti".

Mnohokrát zdůrazníme v této knize, že PowerShell je postaven na objektovém přístupu, a to důsledně – proto i příkazový blok je vlastně založen jako objekt, a my se o o tom můžeme nejen přesvědčit, ale můžeme toho případně i využít. PowerShell 2 přinesl krom viditelných rozšíření i některá méně zjevná – a jedním z nich je i třída *PowerShell*, která je novou součástí knihoven tvořících PowerShell jako takový. Tato třída je určena ke tvorbě skriptovacích bloků, které jakoby uchopíme „zvenčí" – celý skriptovací blok založíme jako objektovou proměnnou a posléze budeme se skriptem pracovat pomocí vlastností a metod této proměnné. Začněme jednoduchým příkladem.

```
=>$blok1 = [powershell]::Create()
=>$blok1 | Get-Member

   TypeName: System.Management.Automation.PowerShell

Name                    MemberType Definition
----                    ---------- ----------
InvocationStateChanged Event      System.EventHandler`1[System.Management.Au
AddArgument             Method     powershell AddArgument(System.Object value
AddCommand              Method     powershell AddCommand(string cmdlet), powe
AddParameter            Method     powershell AddParameter(string parameterNa
AddParameters           Method     powershell AddParameters(System.Collection
AddScript               Method     powershell AddScript(string script), power
BeginInvoke             Method     System.IAsyncResult BeginInvoke(), System.
BeginStop               Method     System.IAsyncResult BeginStop(System.Async
CreateNestedPowerShell Method     powershell CreateNestedPowerShell()
Dispose                 Method     System.Void Dispose()
EndInvoke               Method     System.Management.Automation.PSDataCollect
EndStop                 Method     System.Void EndStop(System.IAsyncResult as
```

```
Equals                  Method     bool Equals(System.Object obj)
GetHashCode             Method     int GetHashCode()
GetType                 Method     type GetType()
Invoke                  Method     System.Collections.ObjectModel.Collection[
Stop                    Method     System.Void Stop()
ToString                Method     string ToString()
Commands                Property   System.Management.Automation.PSCommand Com
InstanceId              Property   System.Guid InstanceId {get;}
InvocationStateInfo     Property   System.Management.Automation.PSInvocationS
IsNested                Property   System.Boolean IsNested {get;}
Runspace                Property   System.Management.Automation.Runspaces.Run
RunspacePool            Property   System.Management.Automation.Runspaces.Run
Streams                 Property   System.Management.Automation.PSDataStreams
```

Ve výpisu členů objektu dokážeme rozpoznat přinejmenším některé „podezřelé“, takže je zkusíme prověřit. Třeba metoda *AddCommand* napovídá, že pomocí ní do bloku vložíme příkaz, a třeba metoda *AddScript* pak učiní totéž s větším celkem. Pojďme to vyzkoušet.

```
=>$blok1.AddCommand("Get-Process")

Commands            : System.Management.Automation.PSCommand
Streams             : System.Management.Automation.PSDataStreams
InstanceId          : 75ec5575-b205-45a9-85b9-01b26f6147b7
InvocationStateInfo : System.Management.Automation.PSInvocationStateInfo
IsNested            : False
Runspace            : System.Management.Automation.Runspaces.LocalRunspace
RunspacePool        :


=>$blok1.Commands

Commands
--------
{Get-Process}
```

V tuto chvíli tedy objektová proměnná reprezentující spustitelný blok v PowerShellu obsahuje jeden příkaz, který čeká na spuštění. Blok spustíme následujícím způsobem:

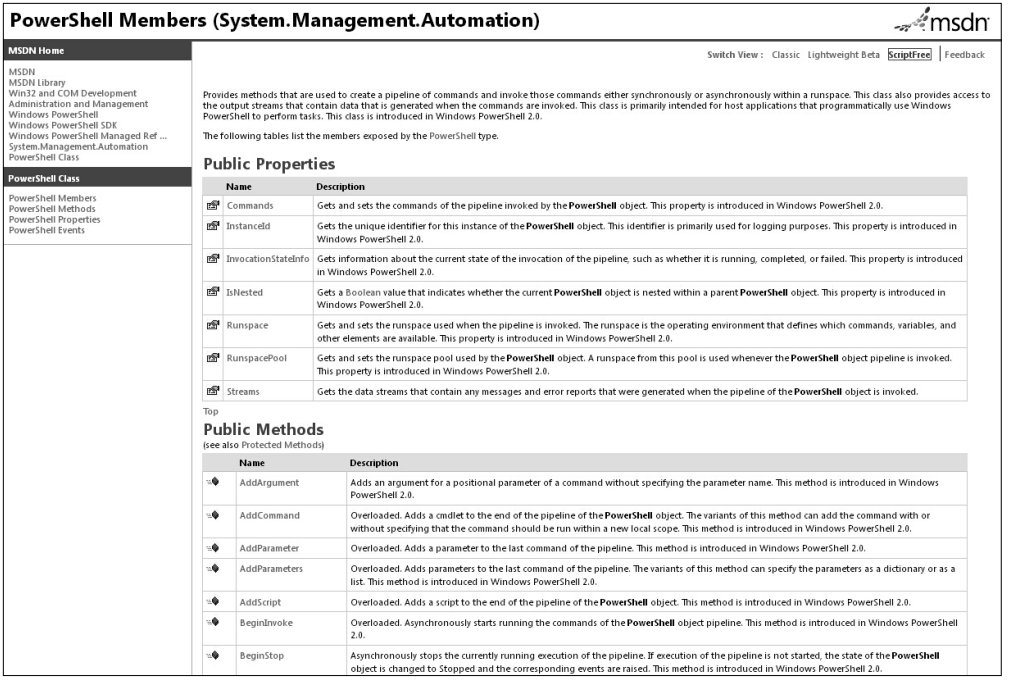
```
=>$blok1.Invoke()

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    111       5     1672        108    35     0,75   3344 alg
     52       2      584        452    18     1,72   1168 ati2evxx
     45       2      568        536    18     1,22   3784 ati2evxx
     99       3     3112        924    35     4,48   2712 atiptaxx
    342     126     4960       1352    56    11,84    288 BTStackServer
    147       3     1952       1040    50    19,05   3524 BTTray
```

To, co jsme vytvořili, je tedy vlastně skriptovací blok „na dálkové ovládání“ – máme jej v objektové proměnné a další řízení je v našich rukou. Skriptovací blok může samozřej-

mě obsahovat nejen jednotlivý příkaz, ale celou sekvenci. Založíme si zdrojový kód jako textový soubor a postupně jej do objektu vložíme.

script.txt

```
=>$blok1.Commands.Clear()

=>cat .\script.txt
$sluzba = Get-WmiObject win32_service -Filter "name='spooler'"
$sluzba.StopService()
$sluzba.StartService()
Write-Host "Restartovano"

=>$script = ((cat .\script.txt) -join "`n" )
=>$scriptobjekt = $executioncontext.InvokeCommand.NewScriptBlock($script)

=>$blok1.AddScript($scriptobjekt) | Out-Null
=>$blok1.Commands

Commands
--------
{{ $sluzba = Get-WmiObject win32_service -Filter "name='spooler'" $sluzba.StopSer-
vice() $sluzba....

=>$blok1.Invoke()

__GENUS          : 2
__CLASS          : __PARAMETERS
__SUPERCLASS     :
__DYNASTY        : __PARAMETERS
__RELPATH        :
__PROPERTY_COUNT : 1
__DERIVATION     : {}
__SERVER         :
__NAMESPACE      :
__PATH           :
ReturnValue      : 5

__GENUS          : 2
__CLASS          : __PARAMETERS
__SUPERCLASS     :
__DYNASTY        : __PARAMETERS
__RELPATH        :
__PROPERTY_COUNT : 1
__DERIVATION     : {}
__SERVER         :
__NAMESPACE      :
__PATH           :
ReturnValue      : 10
```



PowerShell Members (System.Management.Automation)

msdn

MSDN Home

MSDN
MSDN Library
Win32 and COM Development
Administration and Management
Windows PowerShell
Windows PowerShell SDK
Windows PowerShell Managed Ref ...
System.Management.Automation
PowerShell Class

PowerShell Class

PowerShell Members
PowerShell Methods
PowerShell Properties
PowerShell Events

Switch View : Classic Lightweight Beta ScriptFree | Feedback

Provides methods that are used to create a pipeline of commands and invoke those commands either synchronously or asynchronously within a runspace. This class also provides access to the output streams that contain data that is generated when the commands are invoked. This class is primarily intended for host applications that programmatically use Windows PowerShell to perform tasks. This class is introduced in Windows PowerShell 2.0.

The following tables list the members exposed by the PowerShell type.

Public Properties

Name	Description
Commands	Gets and sets the commands of the pipeline invoked by the **PowerShell** object. This property is introduced in Windows PowerShell 2.0.
InstanceId	Gets the unique identifier for this instance of the **PowerShell** object. This identifier is primarily used for logging purposes. This property is introduced in Windows PowerShell 2.0.
InvocationStateInfo	Gets information about the current state of the invocation of the pipeline, such as whether it is running, completed, or failed. This property is introduced in Windows PowerShell 2.0.
IsNested	Gets a Boolean value that indicates whether the current **PowerShell** object is nested within a parent **PowerShell** object. This property is introduced in Windows PowerShell 2.0.
Runspace	Gets and sets the runspace used when the pipeline is invoked. The runspace is the operating environment that defines which commands, variables, and other elements are available. This property is introduced in Windows PowerShell 2.0.
RunspacePool	Gets and sets the runspace pool used by the **PowerShell** object. A runspace from this pool is used whenever the **PowerShell** object pipeline is invoked. This property is introduced in Windows PowerShell 2.0.
Streams	Gets the data streams that contain any messages and error reports that were generated when the pipeline of the **PowerShell** object is invoked.

Top

Public Methods

(see also Protected Methods)

Name	Description
AddArgument	Adds an argument for a positional parameter of a command without specifying the parameter name. This method is introduced in Windows PowerShell 2.0.
AddCommand	Overloaded. Adds a cmdlet to the end of the pipeline of the **PowerShell** object. The variants of this method can add the command with or without specifying that the command should be run within a new local scope. This method is introduced in Windows PowerShell 2.0.
AddParameter	Overloaded. Adds a parameter to the last command of the pipeline. This method is introduced in Windows PowerShell 2.0.
AddParameters	Overloaded. Adds parameters to the last command of the pipeline. The variants of this method can specify the parameters as a dictionary or as a list. This method is introduced in Windows PowerShell 2.0.
AddScript	Overloaded. Adds a script to the end of the pipeline of the **PowerShell** object. This method is introduced in Windows PowerShell 2.0.
BeginInvoke	Overloaded. Asynchronously starts running the commands of the **PowerShell** object pipeline. This method is introduced in Windows PowerShell 2.0.
BeginStop	Asynchronously stops the currently running execution of the pipeline. If execution of the pipeline is not started, the state of the **PowerShell** object is changed to Stopped and the corresponding events are raised. This method is introduced in Windows PowerShell 2.0.

Obrázek 3.5: PowerShell a jeho architektura jsou výborně popsány v dokumentační knihovně MSDN. Není problém si najít detailní popis cmdletů a dalších potřebných tříd.

Za zmínku určitě stojí objektová proměnná *$executioncontext*, kterou jsme využili pro převod textového zdrojového souboru na prováděcí blok. Tato proměnná (znalci odpustí mírné zjednodušení) vlastně reprezentuje některé schopnosti interpretru PowerShellu, mezi něž patří i vyhodnocení skriptovací sekvence příkazů. Tím si tedy uložíme do proměnné *$scriptobjekt* funkční blok skriptu, který posléze můžeme spustit. Pro úplnost dodejme, že jeho provedení bychom mohli zařídit třeba i následujícím způsobem:

```
=>Invoke-Command -ScriptBlock $scriptobjekt | select returnvalue | ft -AutoSize

Restartovano
returnvalue
-----------
          0
          0
```

Ukažme si nyní jednu z nejzajímavějších možností, kvůli které stojí za to uchopit skriptovací blok „do vlastních rukou“. Naše objektová proměnná, která vznikla dle třídy *PowerShell*, totiž nabízí přístup ke všem datovým kanálům. Použijeme k tomu jiný zdrojový kód, který bude vypadat takto:

streams.ps1

```
=>cat .\streams.ps1
$VerbosePreference = "continue"
$DebugPreference = "continue"
```

```
$WarningPreference = "continue"
$ErrorActionPreference = "continue"

Write-Verbose "Start"
Write-Host "Startuji..."
Write-Verbose "Ctu data"
$verze = (Get-Host).version
Write-Verbose "Chystam vypis"
Write-Debug "Verze nactena"
Write-Warning "Pozor, zobrazime verzi PowerShellu!"
Write-Verbose "Vypis"
Write-Host $verze
Writte-Host $verze                        #zamerna chyba
Write-Verbose "Hotovo"
Write-Debug "Konec uspesne"
```

Náš testovací skript tedy bude posílat výstupní informace všemi možnými kanály, jak to jen bude možné.

streams.ps1

```
=>$blok = [powershell]::Create()
=>$script = ((cat .\streams.ps1) -join "`n" )
=>$scriptobjekt = $executioncontext.InvokeCommand.NewScriptBlock($script)
=>$blok.AddScript($scriptobjekt) | Out-Null
=>$blok.Commands

Commands
--------
{$VerbosePreference = "continue"...
```

Náš zdrojový kód jsme tedy připravili jako prováděcí kód do skriptovacího bloku v objektové proměnné. Pojďme se nyní podívat na stav datových kanálů a také na to, jak se jejich obsah bude měnit.

```
=>$blok.Streams

Error    : {}
Progress : {}
Verbose  : {}
Debug    : {}
Warning  : {}

=>$blok.Invoke()
=>$blok.Streams

Error    : {Cannot invoke this function because the current host does not imple-
           ment it., Cannot invoke this function because the current host does not
           implement it., The term 'Writte-Host
           ' is not recognized as the name of a cmdlet, function, script file, or
           operable program.
           Check the spelling of the name, or if a path was included, verify that
           the path is correct and try again.}
```

```
Progress : {}
Verbose  : {Start, Ctu data, Chystam vypis, Vypis...}
Debug    : {Verze nactena, Konec uspesne}
Warning  : {Pozor, zobrazime verzi PowerShellu!}
```

Výpis krásně ukazuje, že jednotlivé proudy jsou striktně odděleny a jsou v nich uložena data tak, jak jsme je posílali do toho kterého kanálu. Podívejme se podrobně třeba na chyby:

```
=>$blok.Streams.Error

Write-Host : Cannot invoke this function because the current host does not imple-
ment it.
At line:7 char:11
+ Write-Host <<<<  "Startuji..."
    + CategoryInfo          : NotImplemented: (:) [Write-Host], HostException
    + FullyQualifiedErrorId : HostFunctionNotImplemented,Microsoft.PowerShell.
Commands.WriteHostCommand
```

Chybové hlášení nám sděluje dosti důležitou informaci: cmdlet *Write-Host* nezapracoval, jak jsme očekávali, a textový výstup se skutečně neobjevil. Je to proto, že instance třídy *PowerShell* zasílá data do roury, kde se očekává další zpracování. Zkusme tedy náš výchozí kód upravit, znovu spustit a následně prozkoumat datové proudy.

streams2.ps1

```
=>cat .\streams2.ps1
$VerbosePreference = "continue"
$DebugPreference = "continue"
$WarningPreference = "continue"
$ErrorActionPreference = "continue"

Write-Verbose "Start"
Write-Output "Startuji..."
Write-Verbose "Ctu data"
$verze = (Get-Host).version
Write-Verbose "Chystam vypis"
Write-Debug "Verze nactena"
Write-Warning "Pozor, zobrazime verzi PowerShellu!"
Write-Verbose "Vypis"
Write-Output $verze
Writte-Host $verze #zamerna chyba
Write-Verbose "Hotovo"
Write-Debug "Konec uspesne"

=>$blok3 = [powershell]::Create()
=>$script = ((cat .\streams2.ps1) -join "`n" )
=>$scriptobjekt = $executioncontext.InvokeCommand.NewScriptBlock($script)
=>$blok3.AddScript($scriptobjekt) | Out-Null

=>$blok3.Streams

Error    : {}
```

```
Progress : {}
Verbose  : {}
Debug    : {}
Warning  : {}

=>$blok3.Invoke()
Startuji...

Major  Minor  Build  Revision
-----  -----  -----  --------
2      0      -1     -1

=>$blok3.Streams

Error    : {The term 'Writte-Host' is not recognized as the name of a cmdlet, func-
           tion, script file, or operable program. Check the spelling of the name,
           or if a path was included, verify
           that the path is correct and try again.}
Progress : {}
Verbose  : {Start, Ctu data, Chystam vypis, Vypis...}
Debug    : {Verze nactena, Konec uspesne}
Warning  : {Pozor, zobrazime verzi PowerShellu!}
```


Nyní již naše řešení pracuje perfektně, zcela dle předpokladů. Chybový kanál zachytil jednu chybu (záměrně chybně zapsaný příkaz) a ostatní kanály obsahují přesně informaci, kterou jsme jimi zaslali. Také je vidět, že provedení kódu vrátilo kýžená data – zaměnili jsme příkaz *Write-Host*, který pracuje s výslovným výpisem do konzoly, za příkaz *Write-Output*, který výslovně posílá data do roury. Zdá se to jako malý rozdíl, ale je to zcela zásadní odlišnost: z roury lze data dál číst a zpracovávat, v konzole však už máme jen výsledný text k zobrazení. Mohli bychom totiž pokračovat třeba takhle:

```
=>($blok3.Invoke() | measure).count
2
=>$blok3.Invoke() | Get-Member
        TypeName: System.String
…
        TypeName: System.Version
Name           MemberType Definition
----           ---------- ----------
...
Build          Property   System.Int32 Build {get;}
Major          Property   System.Int32 Major {get;}
MajorRevision Property   System.Int16 MajorRevision {get;}
Minor          Property   System.Int32 Minor {get;}
MinorRevision Property   System.Int16 MinorRevision {get;}
Revision       Property   System.Int32 Revision {get;}
```

Jinými slovy, náš skript poslal na výstup dva objekty. První z nich byl prostý text (hlášení „Startuji...“), druhý pak získaná data (verze PowerShellu). Další zpracování za rourou je jen na nás.

Okolnosti spuštění bloku – $MyInvocation

Předcházející příklady ukázaly, že blok skriptu v PowerShellu můžeme uchopit jako objekt, stejně jako ostatní data. Možná tuto vymoženost nevyužijeme příliš často, přesto se však nabízí jeden zajímavý důsledek architektury PowerShellu, který se nám může hodit. Po spuštění konzoly, skriptu, obecně každého bloku, který má svůj rozsah působnosti, se vytvářejí některé výchozí proměnné PowerShellu a ukládají se do nich určité hodnoty. Jedna z nich se jmenuje *$MyInvocation* a nese vlastně svědectví o tom, za jakých okolností byl skriptovací blok spuštěn. Toho můžeme v některých situacích využít.

Při tvorbě skriptů někdy potřebujeme získat odkaz na adresář, ze kterého byl původní skript spuštěn. Běžně se stává, že chceme z daného adresáře načítat další soubory, případně do něj ukládat nějaké exporty či záznamy událostí (logy). Při sestavování adresářových cest k takovým souborům se nám hodí, pokud víme, odkud jsme začali.

Prozkoumejme nejdříve první možnost – chceme v našem skriptu používat cestu místa, odkud jsme jej zavolali. Pak to provedeme tak, že sáhneme na jednu z automatických proměnných PowerShellu:

script.ps1

```
=>pwd
Path
----
C:\Patrik\Work\PowerShellBook2\temp\3\start

=>cat .\script.ps1

$MyInvocation.MyCommand | fl *

=>.\script.ps1

HelpUri          :
Path             : C:\Patrik\Work\PowerShellBook2\temp\3\start\script.ps1
Definition       : C:\Patrik\Work\PowerShellBook2\temp\3\start\script.ps1
Visibility       : Public
ScriptBlock      : $MyInvocation.MyCommand | fl *
OutputType       : {}
ScriptContents   : $MyInvocation.MyCommand | fl *
OriginalEncoding : System.Text.SBCSCodePageEncoding
Name             : script.ps1
CommandType      : ExternalScript
ModuleName       :
Module           :
Parameters       : {}
ParameterSets    : {}
```

Pouze kýženou cestu tedy získáme takto:

```
=>cat .\script.ps1
($MyInvocation.MyCommand).Path

=>.\script.ps1
C:\Patrik\Work\PowerShellBook2\temp\3\start\script.ps1
```

Ať už se tedy v našem skriptu vydáme kamkoliv, vždy budeme moci zjistit výchozí místo díky tomu, že okolnosti našeho startu jsou zaznamenány v automatické proměnné. Naše proměnná však nese ještě více informací, než bychom možná doufali. PowerShell a jeho bloky skriptů v sobě nesou také zděděnou informaci o svém prapůvodu.

Shrnutí

PowerShell 2 přinesl zajímavé novinky v oblasti „ovladatelnosti" při provádění kódu. Princip použití objektů platí v podstatě univerzálně, a umožňuje nám tedy uchopit do proměnných bloky skriptu, jeho výstupní data či různá nastavení prováděných relací. Přímo v běžícím skriptu či relaci je tedy možné zkoumat i nastavení „sebe sama", a úlohy v PowerShellu tedy mohou spoléhat na referenční informace o situaci, která nastane po jejich spuštění.

Důležité k zapamatování

- Základní funkční část skriptů v PowerShellu, kterou je blok skriptu, určuje rovněž výchozí hranice působnosti proměnných. Potřebujeme-li překročit automaticky vytvářené hranice, můžeme využít především globálních proměnných.
- Výstupní proudy v PowerShellu můžeme velmi efektivně využívat pomocí řídicích proměnných i příkazů pro přímý zápis dat. Požadujeme-li totální zpracování všech výstupů, můžeme skript uchopit do proměnné a jeho provádění pečlivě řídit.

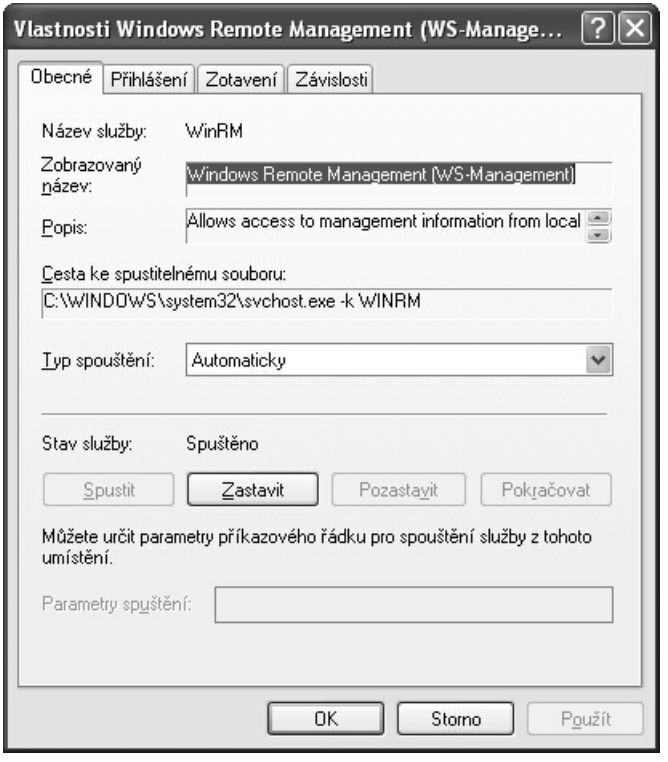
Připojení ke vzdáleným počítačům

Úvod

Možnost připojení ke vzdálenému počítači a jeho správa přímo prostředky PowerShellu je jednou z nejzásadnějších novinek verze 2. Tato toužebně očekávaná funkcionalita přišla po měsících testování a nadějných zkušebních verzích v podobě, jejíž možnosti jsou příjemným překvapením. Práce „vzdáleně" pomocí PowerShellu tak konečně splňuje jeden z nejdůležitějších předpokladů pro bezezbytku použitelné správcovské rozhraní.

Připojení ke vzdáleným počítačům zajišťuje služba Windows Remote Management (WinRM, též WS-Management), na kterou PowerShell v této věci zcela spoléhá. Tato služba (přesněji její instalace) je dostupná pro širokou škálu systémů Windows, od klientských (XP, Vista, 7) po serverové (2003, 2008), čímž je také dána použitelnost PowerShellu „na dálku". Příslušná služba i PowerShell verze 2 musejí být pro účel vzdáleného připojení nainstalovány na obou stranách spojení (tedy jak na volajícím, tak volaném počítači).

Služba WinRM není v systému Windows žádnou převratnou novinkou, neboť se objevila již před lety spolu s aktualizací SP1 na serverový systém Windows 2003. Před nástupem PowerShellu správce využívali její možnosti typicky prostřednictvím nástrojů příkazového rozhraní, kupříkladu *WinRM.cmd* či *WinRS.exe*. PowerShell využívá této služby jako mechanizmu pro samotné spojení a jeho zabezpečení, samotné ovládání se provádí pomocí nové sady příkazů přímo v PowerShellu samotném.



Obrázek 3.6: Služba WinRM je klíčovým prvkem při vzdáleném připojení PowerShellu k počítačům po síti. Musí být spuštěna na obou stranách spojení.

Služba WinRM zajišťuje spojení mezi ovládajícím a ovládaným počítačem prostřednictvím protokolu HTTP, případně jeho bezpečné zapouzdřené varianty SSL. Tento způsob spojení jednak dosahuje vysoké míry zabezpečení, jednak dovoluje překlenout tradiční obtíže – překážky v podobě firewallů či jiných filtrování síťového přenosu. Právě možnosti zabezpečení, včetně přísného vstupního ověření identity aktérů spojení, je klíčovou vlastností, díky níž je PowerShell pro vzdálenou správu vhodný i za nejpřísnějších bezpečnostních podmínek.

Protokol HTTP, resp. SSL, dává tušit, že služba WinRM pracuje v podstatě jako webový server a klient, a tomu také odpovídají možnosti (a nutnost) konfigurace. Službu můžeme nastavit v jakémsi „zjednodušeném“ režimu, jak uvidíme dále, ale v případě potřeby může správce velmi přísně stanovit podmínky, za nichž spojení mohou proběhnout. Mezi bezpečnostní opatření patří vymezení uživatelů, kteří budou moci spojení provádět, dále vyjmenování počítačů, mezi nimiž bude možné spojení navázat, a také lze vynutit určitý způsob ověření účastníků spojení. Tyto i další možnosti budou ukázány v postupech v této kapitole.

Již několikrát zmiňovaná služba WinRM zajišťuje pro PowerShell i jiné funkce než jen vzdálené připojení, a proto na ni narazíme i v jiných situacích. Její správná funkciona-

lita souvisí i se spouštěním úloh na pozadí (tzv. jobů) v PowerShellu, o nichž pojednáváme v jiné části této knihy.

Práci na vzdáleném počítači provádíme v PowerShellu v zásadě dvěma způsoby. Prvním z nich je interaktivní připojení, kdy „převezmeme" příkazový řádek vzdáleného počítače a přímo jej interaktivně ovládáme – konzolové rozhraní je tedy zobrazeno na našem počítači, akce však probíhají na vzdáleném stroji. Jde tedy o analogii spojení typu Telnet či použití populární pomůcky PSExec.exe z dílny Sysinternal ve světě Windows. Druhým způsobem je volání příkazů či celých skriptů (ucelených úloh) v režimu mimo interaktivní relaci – v takovém případě vizuálně nepřebíráme konzolové rozhraní vzdáleného počítače, ale spouštíme skriptovací blok s určením cílového počítače, na němž akce proběhne. Vystupující data následně můžeme zpracovat pomocí roury a vhodně je kombinovat s místními zdroji.

Průzkum nastavení služby WinRM (WSMan)

Službu WinRM (WS-Man) a její nastavení můžeme v PowerShellu prohlédnout a změnit prostřednictvím jednotky typu PSDrive, kterou vývojový tým za tímto účelem připravil jako novinku ve verzi 2. Kontrolu nastavení provedeme následujícím způsobem:

1. Přepneme se do kořene jednotky (PSdrive) se jménem Wsman.

```
=>cd wsman:
=>dir

   WSManConfig:

ComputerName                                Type
------------                                ----
localhost                                   Container

=>cd .\localhost

Start WinRM Service
WinRM service is not started currently. Running this command will start the
WinRM service.

Do you want to continue?
[Y] Yes  [N] No  [S] Suspend  [?] Help (default is "Y"):
```

2. Služba není v našem případě spuštěna, což je běžný výchozí stav. Pro další práci je potřeba službu spustit:

```
[Y] Yes  [N] No  [S] Suspend  [?] Help (default is "Y"): y
=>cd .\localhost

=>dir | ft -AutoSize

   WSManConfig: Microsoft.WSMan.Management\WSMan::localhost

Name                Value      Type
----                -----      ----
```

```
MaxEnvelopeSizekb   150        System.String
MaxTimeoutms        60000      System.String
MaxBatchItems       32000      System.String
MaxProviderRequests 4294967295 System.String
Client                         Container
Service                        Container
Shell                          Container
Listener                       Container
Plugin                         Container
ClientCertificate              Container
```

3. Z výpisu je dobře patrné, že konfigurační parametry jsou zanořeny ve více úrovních a konfigurace je uspořádána v hierarchické struktuře. V „podadresářích" nalezneme některá klíčová nastavení služby. Následujícím způsobem si ověříme, na kterých portech protokolu TCP/IP bude prováděno síťové připojení:

```
=>cd .\Client\DefaultPorts
=>dir | ft -AutoSize

   WSManConfig: Microsoft.WSMan.Management\WSMan::localhost\Client\DefaultPorts

Name  Value Type
----  ----- ----
HTTP  5985  System.String
HTTPS 5986  System.String
```

4. Podobně si kdykoliv můžeme ověřit, jaký způsob ověřování klientů služby je momentálně povolen:

```
=>cd .\Client\Auth
=>dir | ft -AutoSize

   WSManConfig: Microsoft.WSMan.Management\WSMan::localhost\Client\Auth

Name        Value Type
----        ----- ----
Basic       true  System.String
Digest      true  System.String
Kerberos    true  System.String
Negotiate   true  System.String
Certificate true  System.String
```

Rychlá konfigurace služby WinRM

PowerShell poskytuje správcům možnost provést rychlou konfiguraci služby WinRM tak, abychom bez složitých nastavení mohli využívat nové možnosti co možná nejsnáze. Služba je po instalaci či před prvním použitím vypnuta a výchozí konfigurace připojení vylučuje. Následující postup připraví službu pro okamžité využití. Spočívá vlastně ve spuštění automatizovaného skriptu, který se ukrývá za výchozím příkazem.

```
=>Enable-PSRemoting

WinRM Quick Configuration
```

```
Running command "Set-WSManQuickConfig" to enable this machine for remote management
through WinRM
service.
 This includes:
    1. Starting or restarting (if already started) the WinRM service
    2. Setting the WinRM service type to auto start
    3. Creating a listener to accept requests on any IP address
    4. Enabling firewall exception for WS-Management traffic (for http only).

Do you want to continue?
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is
"Y"): y

WinRM service type changed successfully.
WinRM service started.

Created a WinRM listener on HTTP://* to accept WS-Man requests to any IP on this
machine.

=>
```


Nastavení přístupu ke službě WinRM mimo Active Directory

Služba WinRM a s ní i vzdálené připojení PowerShellu jsou ve výchozím nastavení poměrně přísně zabezpečeny. Ověření připojujících se klientů služba provádí přednostně pomocí protokolu Kerberos, a pokud volající či volaný počítač nejsou pod správou služby Active Directory (jsou „mimo doménu Windows“), nelze spojení ihned po spuštění služby navázat. Službu je nutné nakonfigurovat tak, aby přijímala i ověření klientů prostřednictvím lokální databáze uživatelských účtů (stará dobrá lokální databáze SAM). Konfiguraci služby pro takovéto ověření provedeme na přijímajícím počítači následujícím způsobem.

Upozornění: Následující konfigurace výrazně snižuje zabezpečení vzdáleného přístupu a je vhodná především pro testování!

1. Přepneme se na PSDrive, který obsahuje nastavení služby WinRM, do části s nastavením klientu:

```
=>cd WSMan:\localhost\Client
=>dir

   WSManConfig: Microsoft.WSMan.Management\WSMan::localhost\Client

Name                    Value
----                    -----
NetworkDelayms          5000
URLPrefix               wsman
AllowUnencrypted        false
Auth
DefaultPorts
TrustedHosts
```

2. Přiřadíme položce *TrustedHosts* hodnotu, jež dovolí ověřování oproti lokální databázi účtů SAM.

```
=>Set-Item TrustedHosts "<local>"
=>dir

   WSManConfig: Microsoft.WSMan.Management\WSMan::localhost\Client

Name                        Value
----                        -----
NetworkDelayms              5000
URLPrefix                   wsman
AllowUnencrypted            false
Auth
DefaultPorts
TrustedHosts                <local>
```

Vymezení cílového počítače pro vzdálené připojení

Již jsme zmínili, že služba WinRM (WSMan) nabízí velmi precizní konfiguraci zabezpečení, mezi jejíž pilíře jistě náleží vymezení počítačů, jež mezi sebou při vzdálené správě mohou komunikovat. Jednorázové nastavení povolení pro určitý počítač provedeme následujícím způsobem:

1. Uložíme jméno počítače, jenž bude povolen pro vzdálenou komunikaci, do příslušného parametru služby WinRM:

```
set-item WSMan:\localhost\Client\TrustedHosts -Value "Patrik-d"
```

2. Potvrdíme náš záměr po bezpečnostní výzvě:

```
WinRM Security Configuration.
This command modifies the TrustedHosts list for the WinRM client. The computers
in the TrustedHosts
 list might not be authenticated. The client might send credential information
to these computers.
Are you sure that you want to modify this list?
[Y] Yes  [N] No  [S] Suspend  [?] Help (default is "Y"): y
```

3. Případně ihned vstoupíme do vzdálené relace:

```
=>Enter-PSSession -ComputerName patrik-d -Credential patrik
[patrik-d]: PS C:\Users\Patrik\Documents> dir
```

Upozornění: Parametr TrustedHost se nachází v konfigurační části Client, což může být zavádějící: nastavením příslušného jména serveru povolujeme obousměrné spojení, tedy jak roli klientu, tak roli serveru pro daný počítač.

Povolení vzdálené komunikace pro více počítačů

Služba WinRM pochopitelně dovoluje nastavit kontrolní parametr *TrustedHosts* tak, aby obsahoval seznam více počítačů, jimž bude dovolena vzdálená komunikace. Takovouto

konfiguraci můžeme připravit na základě seznamu jmen strojů, jež mají být zahrnuty mezi důvěryhodné.

seznam.txt, trusted.ps1

1. Připravíme si seznam počítačů, jež mají být zdůvěryhodněny pro vzdálené relace:

```
=>cat C:\seznam.txt
Server1
Server2
10.0.0.200
10.0.0.201
Patrik-D
Stanice5
Stanice6
*.mojedomena.cz
=>
```

Tip: V seznamu mohou vystupovat jak krátká jména, tak adresy IP, plná doménová jména, ale třeba i celé sady počítačů v dané doméně, jež jsou zastoupeny symbolem hvězdičky.

2. Obsah seznamu načteme do proměnné jako celek, abychom mohli pracovat se všemi položkami najednou.

```
$pocitace = (new-object system.io.streamreader "C:\seznam.txt").ReadToEnd()
```

3. Jednotlivé řádky převedeme na seznam s položkami oddělenými čárkami, jak to konfigurace parametru *TrustedHost* vyžaduje. Využíváme zde metody *Replace*, která je definována obecně na jakémkoliv řetězci – ta provede vyhledání konce řádku a jeho nahrazení čárkou.

```
$hosts = $pocitace.Replace("`n",",")
```

4. V dalším kroku načteme současný obsah parametru *TrustedHosts*, abychom o něj nepřišli – nové hodnoty ze seznamu budou ke stávajícím přidány. Je však potřeba je zapisovat najednou, a proto si momentální obsah uložíme:

```
$stavajici_trustedhosts = (get-item wsman:\localhost\Client\TrustedHosts).value
```

5. Novou hodnotu parametru nastavíme jako spojení zazálohovaného seznamu a seznamu nově připraveného:

```
Set-Item WSMan:\localhost\Client\TrustedHosts -value `
"$stavajici_trustedhosts,$hosts"
```

6. V případě potřeby se o výsledku přesvědčíme. Můžeme jej také uložit jako referenční soubor.

```
=>(dir WSMan:\localhost\Client\TrustedHosts).Value
Patrik-d,Server1,Server2,10.0.0.200,10.0.0.201,Patrik-D,Stanice5,Stanice6,*.
mojedomena.cz

=>((dir WSMan:\localhost\Client\TrustedHosts).Value).Split(",") | Set-Content `
c:\SetTrustedHosts.txt
```

Připojení ke vzdálené konzole PowerShellu

PowerShell dovoluje přímé použití příkazové konzoly „na dálku“ na vzdáleném počítači. Interaktivní přenos příkazů a dat zajišťuje služba WinRM na pozadí tak, že při samotné relaci je zcela transparentní, a příkazová konzola tak bezprostředně představuje přímé ovládací rozhraní vzdáleného počítače. Připojení do interaktivní konzoly provedeme takto:

1. Pomocí příslušného příkazu vstoupíme do relace vzdáleného přístupu:

```
=>Enter-PSSession -ComputerName patrik-d -Credential patrik
```

2. V dialogu, jenž byl vyvolán přepínačem *-Credential*, zadáme přihlašovací údaje.
3. PowerShell odpoví předáním konzolového rozhraní vzdáleného počítače. Na počátku řádku se objeví jméno vzdáleného počítače jako součást výzvy (promptu) konzoly.

```
[patrik-d]: PS C:\Users\Patrik\Documents> hostname
Patrik-D

[patrik-d]: PS C:\Users\Patrik\Documents> whoami
patrik-d\patrik

[patrik-d]: PS C:\Users\Patrik\Documents> ipconfig

Windows IP Configuration

Ethernet adapter Local Area Connection:

   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . : fe80::e546:2e41:cc78:cc3f%8
   IPv4 Address. . . . . . . . . . . : 10.0.0.140
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 10.0.0.138
...
```

Takovéto rozhraní je plnohodnotným PowerShellem běžícím na vzdáleném stroji.

Hostování služby WSMan jako aplikace IIS – scénář „fan-in“

Technologie vzdáleného přístupu pomocí PowerShellu je navržena jako univerzální mechanizmus, který by měl pokrýt všechny budoucí požadavky na vzdálenou správu operačních systémů a aplikací Microsoftu. Patří sem tedy i scénáře, kdy na jednom serveru běží služba či aplikace, kterou sdílí více uživatelů, a nezávislí správci potřebují současně přistupovat k serveru, aby zajistili údržbu (typickými příklady jsou třeba hostované služby na aplikacích Exchange či SQL Server). Již víme, že vzdálený přístup je prováděn vlastně pomocí webové technologie a že služba WSMan se chová jako webový server a klient zároveň, dle potřeby komunikace. Je tedy logické, že tento princip můžeme posunout na vyšší úroveň: předávání požadavků službě WSMan můžeme zajistit také tím, že ji budeme hostovat ve službě IIS. Stane se tedy vlastně aplikací IIS, jež bude realizována prostřednictvím modulu služby WSMan.

Poznámka: Hostování služby WSMan (WinRM) v podobě aplikace serveru IIS není samo o sobě nijak vázáno na PowerShell. Jde o obecnější možnost, jak službu volat a používat. PowerShell „pouze" využívá pro své potřeby vzdáleného přístupu (remotingu) touto cestou; mohli bychom ale přistoupit ke službě WinRM také pomocí jiného klientu, než je PowerShell. Podrobnější informace čtenář nalezne na následujících stránkách:

http://msdn.microsoft.com/en-us/library/ee309364(VS.85).aspx

http://msdn.microsoft.com/en-us/library/ee309370(VS.85).aspx

Tvůrci PowerShellu tedy nabízejí pro náročnější scénáře, které jsme naznačili, pokročilejší variantu přijímání spojení vzdáleného přístupu. Tvůrci označují tento scénář jako „fan-in" nebo též „many-to-one", tedy připojení z mnoha míst k jednom centrálnímu serveru, a právě přijímající operační systém musí zvládnout obsluhu současně přicházejících požadavků na provádění úloh v PowerShellu. Pro tento účel lze využít webovou službu IIS a službu WSMan, která vyřizuje požadavky vzdálených powershellových spojení, tak vlastně nastavíme jako webovou aplikaci IIS. Následující postup tedy popisuje konfiguraci označovanou jako „fan-in". Přijímání požadavků vzdálené správy převezme služba IIS a ta bude následně zařizovat spouštění a provádění příkazů v PowerShellu. Takováto forma připojení a správy je náročná na bezpečnost, a proto si server IIS nastavíme tak, aby vynucoval bezpečné spojení SSL zajištěné certifikátem.

Upozornění: Následující scénář a celý postup jsou náročné na důsledné provedení. Do hry vstupuje řada „proměnných", které mohou zhatit zdárné fungování. V celém postupu se budeme snažit upozorňovat na citlivá místa.

1. Na serveru přidáme potřebné role webových služeb IIS a hostování služby WinRM. Tuto akci můžeme provést několika způsoby: vedle grafického rozhraní se nabízí nástroj příkazového řádku Ocsetup.exe a správci na serverech Windows 2008 R2 mohou použít též modul PowerShellu, který nabízí příslušné cmdlety. V prostředí Cmd.exe to můžeme provést takto:

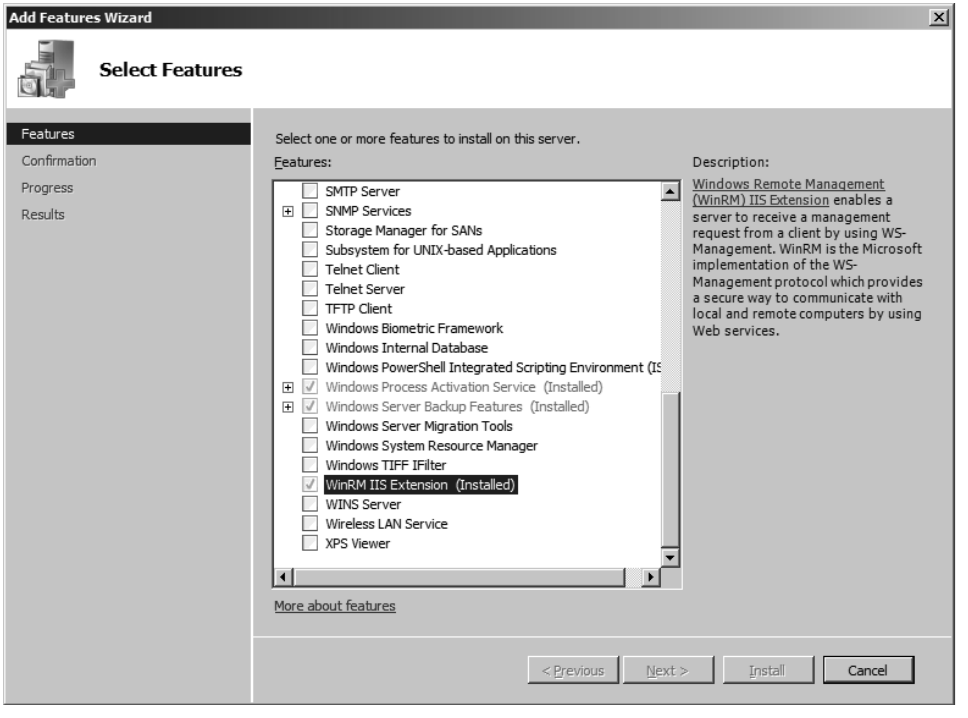
```
D:\Users\Administrator>start /w ocsetup.exe IIS-WebServerRole;WAS-WindowsActivat
ionService;IIS-WebServerManagementTools;IIS-Manage
mentConsole;WAS-ConfigurationAPI;IIS-BasicAuthentication;IIS-WindowsAuthenticati
on;IIS-DigestAuthentication /quiet /norestart
```

V PowerShellu pak můžeme postupovat takto:

iis_feature.txt

```
=> cat .\IIS_feature.txt
Web-WebServer
Web-Mgmt-Tools
WAS
WinRM-IIS-Ext

=> $role = cat .\IIS_feature.txt
=> $role
Web-WebServer
Web-Mgmt-Tools
```

```
WAS
WinRM-IIS-Ext

=> Add-WindowsFeature -IncludeAllSubFeature -Restart -Verbose -Name $role

VERBOSE: Checking if running in 'WhatIf' Mode.
VERBOSE: Start Installation...
VERBOSE: Performing operation "Add-WindowsFeature" on Target "[Windows Process
Activation Service] Configuration APIs".
VERBOSE: Performing operation "Add-WindowsFeature" on Target "[Web Server (IIS)]
Management Tools".
...
VERBOSE: [Installation] Succeeded: [Web Server (IIS)] ASP.
VERBOSE: [Installation] Succeeded: [Web Server (IIS)] ASP.NET.

Success Restart Needed Exit Code Feature Result
------- -------------- --------- --------------
True    No             Success   {Configuration APIs, .NET Environment, Pro...
```



Obrázek 3.7: Služba IIS musí být nainstalována se všemi potřebnými rozšířeními. Mezi klíčové součásti patří doplněk pro službu WinRM pro IIS.

Upozornění: Zavedení *Feature* jménem *WinRM IIS Extension* je naprosto zásadní předpoklad pro zdárné fungování požadované aplikace. Pokud bude chybět nějaká část webových služeb, jež přímo nesouvisí s naším scénářem, mnoho se nestane. Bez této komponenty to však jistojitě pracovat nebude.

2. Potřebné role jsme přidali, a pokud bychom se chtěli přesvědčit o jejich přítomnosti, můžeme to udělat takto:

```
=> Get-WindowsFeature -Name web*

Display Name                                            Name
------------                                            ----
[X] Web Server (IIS)                                    Web-Server
    [X] Web Server                                      Web-WebServer
        [X] Common HTTP Features                        Web-Common-Http
            [X] Static Content                          Web-Static-Content
            [X] Default Document                        Web-Default-Doc
            [X] Directory Browsing                      Web-Dir-Browsing
            [X] HTTP Errors                             Web-Http-Errors
...
```

3. Služba je připravena, a začneme se tedy věnovat aplikaci jako takové. Chceme náš vzdálený přístup nastavit jako zabezpečený, a proto bude naše aplikace pracovat se zabezpečením SSL; pro tyto účely budeme potřebovat serverový certifikát. Pro účely testování či interní sítě bude vyhovovat tzv. certifikát self-signed, který si vytvoříme bez služby certifikační autority. Použijeme k tomu nástroj Makecert.exe z balíku SDK.

Tip: Čtenář nalezne podrobné pojednání o certifikátech a nástroji Makecert.exe v kapitole 5.

Vyrobíme si tedy nejdříve kořenový certifikát, který bude reprezentovat certifikační autoritu, a posléze certifikát pro službu SSL.

Instalace Windows SDK, makecert.txt

```
=> $env:path += ";D:\Program Files\Microsoft SDKs\Windows\v6.1\bin"
=> makecert.exe -r -pe -n "CN=DC1.foundation.int" -eku 1.3.6.1.5.5.7.3.1 -ss my
-sr localMachine -sky exchange d:\data\winrm.cer
Succeeded

=> dir cert:\LocalMachine\My | where {$_.subject -like "*dc1.found*"} | fl

Subject      : CN=DC1.foundation.int
Issuer       : CN=DC1.foundation.int
Thumbprint   : BAC9E9FD9BE1441B902FD8FF84B4945CDF7C5761
FriendlyName :
NotBefore    : 11.2.2010 21:40:54
NotAfter     : 1.1.2040 0:59:59
Extensions   : {System.Security.Cryptography.Oid, System.Security.Cryptography.
Oid}
```

Ověřili jsme si, že certifikát je na místě, kde jej budeme potřebovat. Nesmíme však zapomenout na důležitou věc: aby byl náš certifikát serveru považován za důvěryhodný, musí jeho vydavatel být umístěn mezi důvěryhodnými kořenovými certifikáty. Proto si ověříme, zda je na pravém místě, a pokud ne, tak jej tam zařadíme.

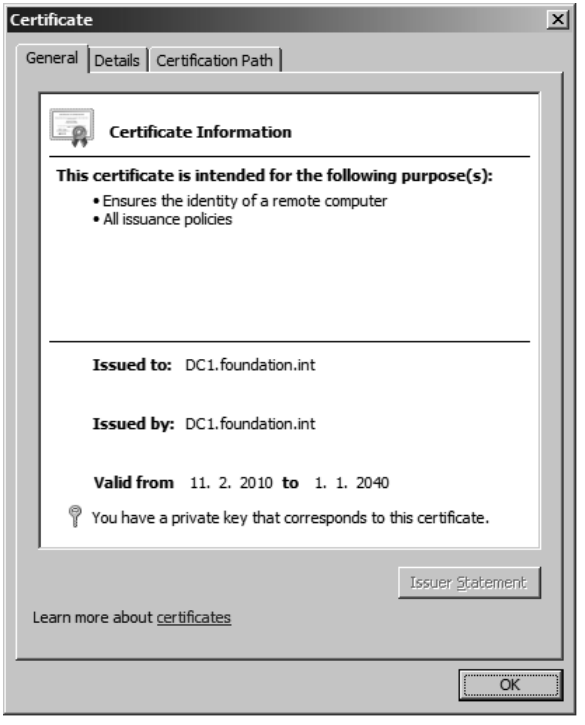
Instalace Windows SDK

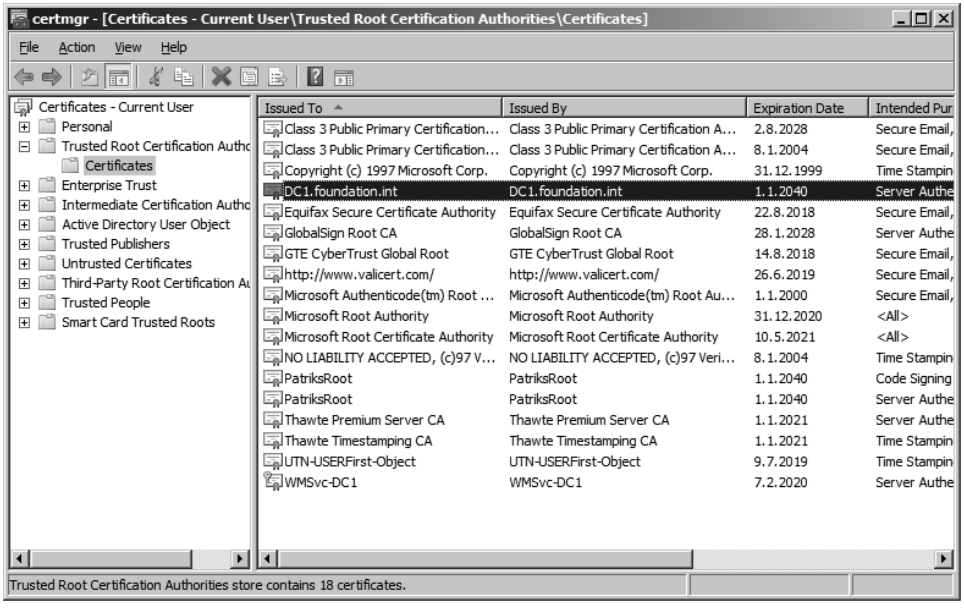
```
=> & "CertMgr.exe"  /Add winrm.cer /s /r localmachine root
CertMgr Succeeded
```

Certifikáty jsou připraveny a můžeme přistoupit ke konfiguraci webové služby.

Upozornění: Autor doporučuje čtenářům důsledně prověřit, zda certifikát je opravdu na svém místě a zda je operačním systémem považován za důvěryhodný. Pokud tomu tak není, webový server s ním bude mít potíže a může tím začít řetězec komplikací nevyhnutelně vedoucí k celkově nefunkční službě.

Obrázek 3.8: Služba IIS potřebuje svůj certifikát pro poskytování bezpečného spojení SSL. Takový certifikát může být podepsán i sám sebou a vytvořen bez certifikační autority.





Obrázek 3.9: Správná funkce certifikátu a návazné služby IIS bude zajištěna až poté, co certifikát umístíme do složky mezi důvěryhodné kořenové certifikáty

4. Vytvoříme si nový adresář pro hostovanou službu WSMan na našem webovém serveru, poté založíme novou aplikaci a aplikační pool pro její hostování a pak nastavíme základní parametry. Ukážeme si řešení pomocí modulu pro správu webových služeb (viz též kapitola 8).

```
=> Import-Module webadministration
=> cd D:\inetpub\wwwroot
D:\inetpub\wwwroot
=> mkdir WSMan_FanIn

    Directory: Microsoft.PowerShell.Core\FileSystem::D:\inetpub\wwwroot

Mode            LastWriteTime       Length Name
----            -------------       ------ ----
d----     31.1.2010     0:35        <DIR> WSMan_FanIn

=> New-WebAppPool WSMan_FanIn

Name                    State        Applications
----                    -----        ------------
WSMan_FanIn             Started

=> New-Website -Name WSMan_FanIn -PhysicalPath "D:\inetpub\wwwroot\WSMan_FanIn"
-Port 8080 -ApplicationPool WSMan_FanIn

Name             ID   State      Physical Path                  Bindings
----             --   -----      -------------                  --------
WSMan_FanIn      2    Started    D:\inetpub\wwwroot\WSMan_FanIn http *:8080:
```

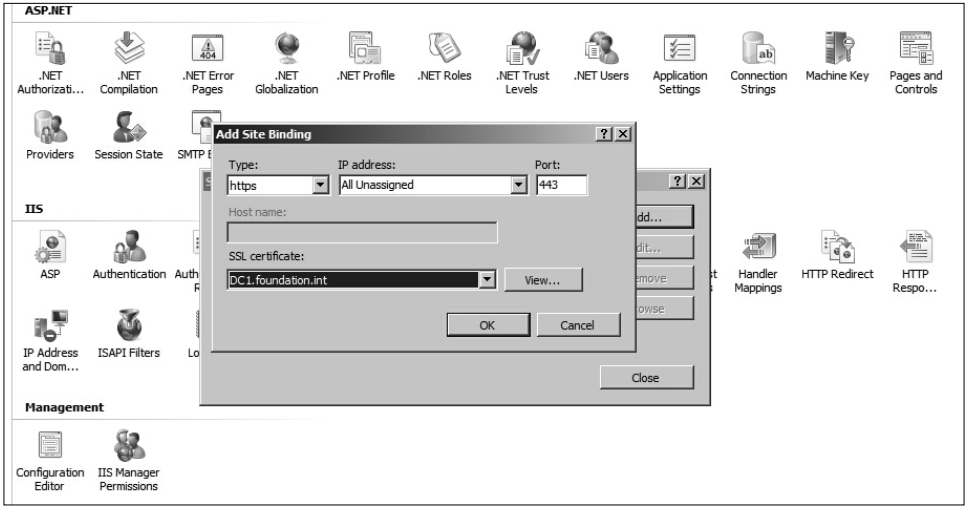
Vyrobili jsme si postupně novou složku jako kořen aplikace, poté založili aplikační pool služby IIS (do nějž izolujeme naši aplikaci) a pak vytvořili samotnou aplikaci. Prověříme si, že pracuje, a posléze postoupíme dále.

```
=> Get-WebURL -Url http://dc1.foundation.int:8080 -Content | fl

ResponseUri : http://dc1.foundation.int:8080/
Status      : OK
Description : OK
Content     : Testovaci stranka pro WSMan_FanIn

=>New-WebApplication -Site WSMan_FanIn -Name WSMan_FanIn -PhysicalPath D:\inet-
pub\wwwroot\WSMan_Fan
In -ApplicationPool WSMan_FanIn

Name             Application pool   Protocols     Physical Path
----             ----------------   ---------     -------------
WSMan_FanIn      WSMan_FanIn        http          D:\inetpub\wwwroot\WSMan_FanIn
```

5. Naše aplikace prozatím pracuje bez SSL a vrací namísto služby WSMan jen statickou stránku, pokud nějakou umístíme do kořenové složky (toto doporučujeme vyzkoušet). Nyní připojíme náš připravený certifikát k webové aplikaci a prověříme, zda server správně pracuje při zabezpečeném přenosu. Napojení (binding) můžeme provést třeba v grafické konzole IIS Manager, viz obrázek. Poté provedeme test:

```
=> Get-WebURL -Url https://dc1.foundation.int -Content | fl

ResponseUri : https://dc1.foundation.int/
Status      : OK
Description : OK
Content     : Testovaci stranka pro WSMan_FanIn
```

Obrázek 3.10: PowerShell a služba WinRM budou hostovány v IIS jako jakákoliv jiná aplikace



Obrázek 3.11: Nezbytně nutně musíme spojit náš certifikát s příslušným portem našeho serveru služby IIS

6. Nyní jsme si vyzkoušeli, že server správně navazuje spojení SSL, a můžeme konečně zapojit do naší aplikace modul WSMan. Okopírujeme konfigurační soubor pro službu WSMan z výchozího umístění do adresáře služeb IIS.

```
copy $env:windir\system32\wsmanconfig_schema.xml $env:windir\system32\inetsrv\
config\schema\wsmanconfig_schema.xml
```

7. Dále musíme upravit konfigurační soubor serveru IIS jménem *ApplicationHost.config*. Najdeme jej v adresáři *%windir%|System32|inetsrv|config*. Před úpravou jej okopírujeme jako zálohu a posléze najdeme část:

```
<sectionGroup name="system.webServer">
```

Do ní přidáme řádek:

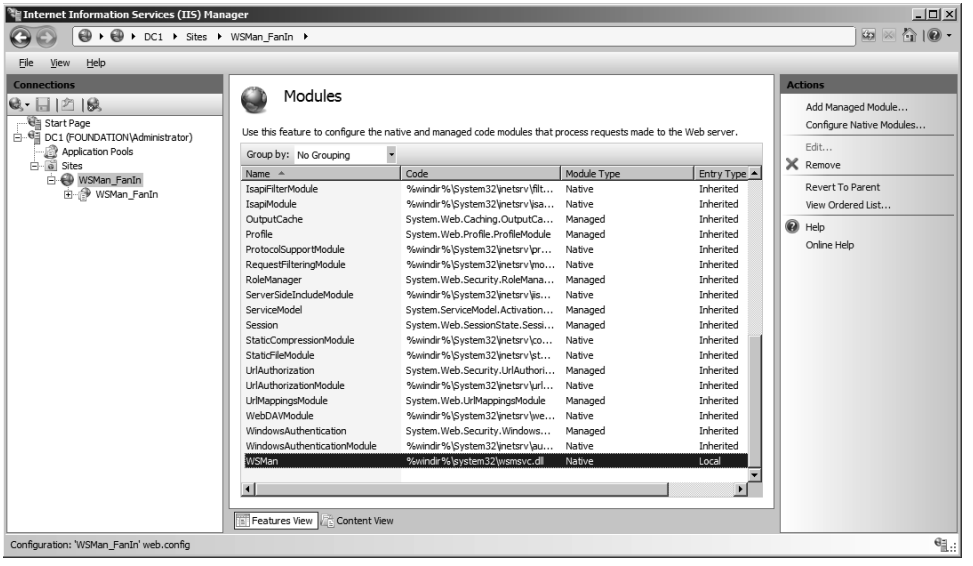
```
<section name="system.management.wsmanagement.config" overrideModeDefault="Allo
w" />
```

Naše aplikace prozatím obsahuje jen onu testovací statickou stránku (pokud jsme si ji vyrobili), a proto konečně zaregistrujeme knihovnu služby WSMan, která bude obsluhovat naše požadavky.

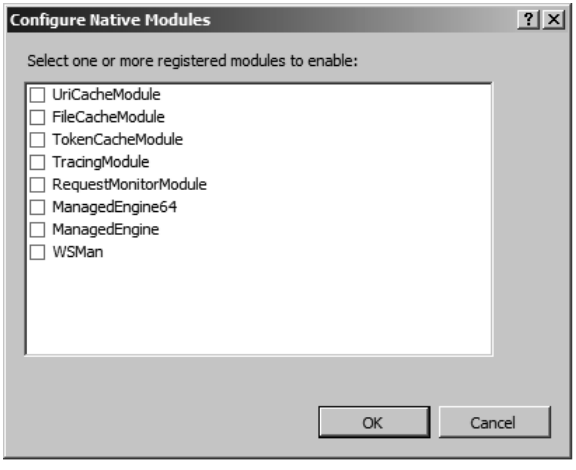
```
New-WebGlobalModule -Name WSMan -Image "$env:windir\system32\wsmsvc.dll
```

Tip: Pokud je čtenář zvyklý na konfiguraci pomocí aplikace Appcmd.exe, stejný krok by vypadal asi takto:

```
=> & "$env:windir\system32\inetsrv\appcmd.exe" add module /name:"WSMan" /
app.name:"WSMan_FanIn/WSMan_FanIn"
MODULE object "WSMan" added
```



Obrázek 3.12: Modul služby Winrm bude zajišťovat samotné provádění požadovaných akcí na cílovém spravovaném počítači. Tato knihovna bude vlastně naší aplikací na IIS.



Obrázek 3.13: Modul služby WSMan se objeví v nabídce globálních modulů v grafickém rozhraní IIS. Pokud se tak nestane, nezdařila se nám úprava souboru *ApplicationHost.Config*.

8. Nyní musíme připravit konfigurační soubor pro naši aplikaci. Za tím účelem můžeme použít následující vzor, který uložíme do adresáře s naší aplikací jako web.config.

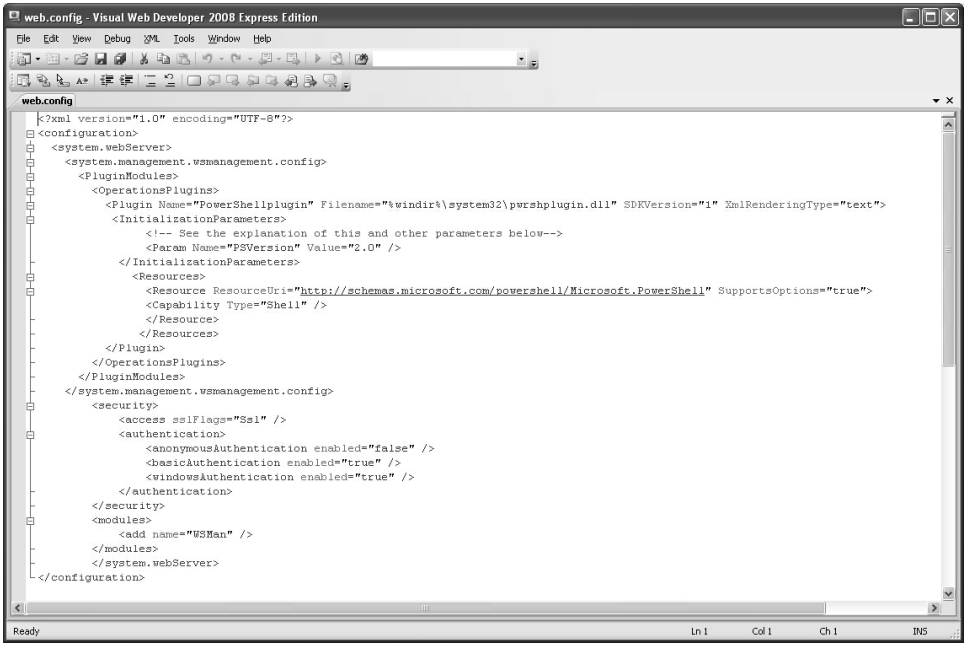
web.config

```
***obsah souboru***
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
```

```
  <system.webServer>
    <system.management.wsmanagement.config>
      <PluginModules>
        <OperationsPlugins>
          <Plugin Name="PowerShellplugin" Filename="%windir%\system32\
           pwrshplugin.dll" SDKVersion="1" XmlRenderingType="text">
           <InitializationParameters>
                <!-- Sem mohou byt umisteny další konfiguracni parametry-->
                <Param Name="PSVersion" Value="2.0" />
            </InitializationParameters>
              <Resources>
                <Resource ResourceUri="http://schemas.microsoft.com/powershell/
                 Microsoft.PowerShell" SupportsOptions="true">
                <Capability Type="Shell" />
                </Resource>
               </Resources>
          </Plugin>
        </OperationsPlugins>
      </PluginModules>
    </system.management.wsmanagement.config>
        <security>
            <access sslFlags="Ssl" />
            <authentication>
                <anonymousAuthentication enabled="false" />
                <basicAuthentication enabled="true" />
                <windowsAuthentication enabled="true" />
            </authentication>
        </security>
        <modules>
            <add name="WSMan" />
        </modules>
  </system.webServer>
</configuration>
***konec souboru***
```

V konfiguračním souboru stojí za povšimnutí několik míst. V první řadě je v horní části vidět, že konfigurační soubor souvisí s aplikací *wsmanagement*. Dále rozpoznáme, že služba bude spolupracovat s modulem pro PowerShell (*pwrshplugin.dll*), a především najdeme inicializační blok, do nějž mohou být umístěny určité konfigurační údaje služby.

9. V následujícím kroku provedeme tzv. odemčení některých částí globálních nastavení webové služby. Pokud bychom to neudělali, naše aplikace by nemohla uplatnit svá specifická nastavení, neboť globální konfigurace platná pro celou službu IIS by vše ovládala. Použijeme pro změnu aplikaci Appcmd.exe, která je právě určena ke komplexní správě webových aplikačních služeb.



Obrázek 3.14: Soubor *web.config* je vlastně jediným souborem, který leží v adresáři naší aplikace. Veškerou práci zastanou již připravené moduly v knihovnách DLL.

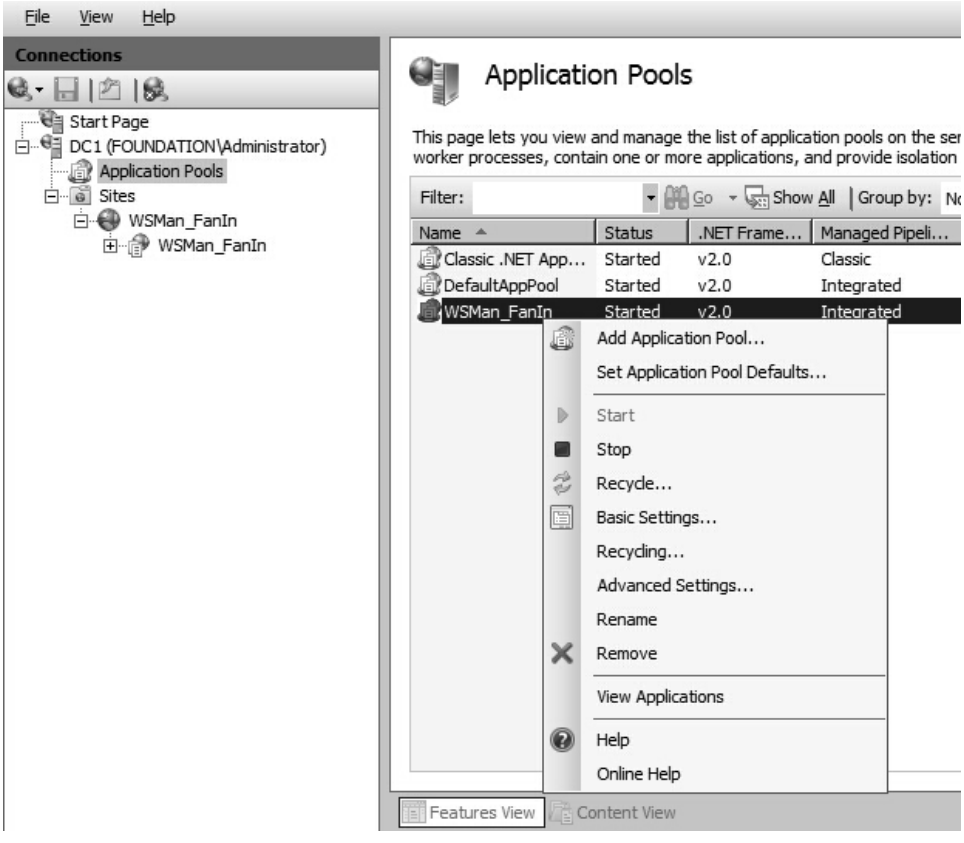
Unlock.ps1

```
=> & "$env:windir\system32\inetsrv\appcmd.exe" unlock config -section:access
Unlocked section "system.webServer/security/access" at configuration path
"MACHINE/WEBROOT/APPHOST".
=>
=>  & "$env:windir\system32\inetsrv\appcmd.exe" unlock config -section:anonymou-
sAuthentication
Unlocked section "system.webServer/security/authentication/anonymousAuthentica-
tion" at configuration path "MACHINE/WEBROOT/APPHOST".
=>
=> & "$env:windir\system32\inetsrv\appcmd.exe" unlock config -section:basicAu-
thentication
Unlocked section "system.webServer/security/authentication/basicAuthentication"
at configuration path "MACHINE/WEBROOT/APPHOST".
=>
=> & "$env:windir\system32\inetsrv\appcmd.exe" unlock config -section:windowsAu-
thentication
Unlocked section "system.webServer/security/authentication/windowsAuthenticati-
on" at configuration path "MACHINE/WEBROOT/APPHOST".
=>
```

Povšimneme si opět, jaké části jsou „odblokovány“ – týkají se ověření přistupujícího uživatele, protože tyto parametry si naše aplikace bude řídit samostatně.

10. Nyní můžeme pro jistotu restartovat aplikační pool, v němž je naše aplikace zařazena.

```
=> Stop-WebAppPool WSman_FanIn
=> Start-WebAppPool WSman_FanIn
```



Obrázek 3.15: Restart aplikačního poolu můžeme samozřejmě provést i ručně v grafické konzole IIS

11. Pro jistotu si prověříme všechna nastavení pomocí následujícího přehledu:

```
=> Get-WebApplication

Name             Application pool   Protocols    Physical Path
----             ----------------   ---------    -------------
WSMan_FanIn      WSMan_FanIn        http         D:\inetpub\wwwroot\WSMan_FanIn

=> Get-WebAppPoolState

Value
-----
Started
Started
Started
```

```
=> Get-WebBinding

protocol                                                    bindingInformation
--------                                                    ------------------
http                                                        *:80:
https                                                       *:443:

=> Get-WebGlobalModule -Name ws*

Name                    Image
----                    -----
WSMan                   %windir%\system32\wsmsvc.dll

PS D:\data> Get-Website

Name             ID   State      Physical Path                  Bindings
----             --   -----      -------------                  --------
WSMan_FanIn      1    Started    D:\inetpub\wwwroot\WSMan_FanIn http *:80:
                                                                https *:443:
```

Všechna nastavení se zdají být na svém místě: máme aplikační pool, v něm aplikaci, připraven je i potřebný globální modul a webová stránka běží a naslouchá na potřebném portu.

12. Naše aplikace je připravena na straně serveru, a proto můžeme přistoupit k připojení z klientské strany. Na straně klientu použijeme objekt *PSSession* pro uložení konfiguračních parametrů a posléze pomocí něj budeme relaci volat.

session.ps1

```
=>$options = new-object System.Management.Automation.Remoting.PSSessionOption
=>$options.SkipCACheck = $true
=>$options.SkipCNCheck = $true
=>$options.SkipRevocationCheck = $true
=>$options

MaximumConnectionRedirectionCount : 5
NoCompression                     : False
NoMachineProfile                  : False
ProxyAccessType                   : None
ProxyAuthentication               : Negotiate
ProxyCredential                   :
SkipCACheck                       : True
SkipCNCheck                       : True
SkipRevocationCheck               : True
OperationTimeout                  : 00:03:00
NoEncryption                      : False
UseUTF16                          : False
Culture                           :
UICulture                         :
MaximumReceivedDataSizePerCommand :
MaximumReceivedObjectSize         :
```

```
ApplicationArguments            :
OpenTimeout                     : 00:03:00
CancelTimeout                   : 00:01:00
IdleTimeout                     : 00:04:00
```

Dobře si povšimněme vlastností, které jsme nastavili. Všechny tři se týkají certifikátu a připojení SSL a zajistí nám, že u certifikátu serveru nebude striktně kontrolována vydávající autorita, seznam zneplatněných certifikátů ani přesné jméno serveru. Tato opatření činíme proto, že používáme certifikát podepsaný sebou samým, který takové kontrole těžko vyhoví.

Upozornění: V praxi bychom měli použít na serveru certifikát SSL od certifikační autority a všechna popsaná nastavení by měla striktně zůstat v původní podobě, aby ke kontrolám došlo.

13. Nyní spustíme samotnou relaci vzdáleného připojení. Uložíme si ji do proměnné, což nám dovolí snadno volat naše spojení v souvislosti s jakýmkoliv příkazem či příkazovým blokem.

```
=>$ses = New-PSSession -ConnectionUri https://dc1:443 `
-Credential foundation\administrator -SessionOption $options
```

14. Relace na vzdálený počítač je připravena a spojena, můžeme jí tedy začít používat.

```
=>Invoke-Command $ses {$env:computername}
DC1
```


Shrnutí

PowerShell 2 obsahuje kvalitní mechanizmus pro vzdálené připojení mezi spravovanými počítači. Služba WinRM, jež takováto spojení zajišťuje, poskytuje nejvyšší možnou míru zabezpečení, a proto je možno bez obav tuto funkcionalitu nasadit i v produkčním prostředí s vysokými nároky na zabezpečení.

Důležité k zapamatování

- Vzdálený přístup pomocí PowerShellu (remoting) je zcela závislý na službě WinRM. Tato musí být jednak zapnuta, jednak správně nastavena.
- Služba WinRM je velmi důsledná v nastavení zabezpečení. Správce může přesně vymezit, kteří uživatelé budou moci mezi kterými počítači komunikovat.
- PowerShell dovoluje vstoupit do interaktivní vzdálené relace pomocí příkazu *Enter-PSSession*.
- Služba WinRM komunikuje prostřednictvím protokolu HTTP, resp. SSL, takže její konfigurace a zabezpečení odpovídá nastavením webového serveru.

Hromadné provádění úloh

Úvod

První verze PowerShellu trpěla jednou zásadní nevýhodou, jež značně znesnadňovala práci správce při automatizaci administrátorských úloh. Spuštěná interaktivní relace, která je reprezentována textovou konzolou, dovoluje v první verzi provádět pouze jednu interaktivní úlohu, případně vykovávat jediný zdrojový soubor-skript v jazyce PowerShell. Veškeré úlohy, jež vyžadují delší čas na dokončení, tak „obsadily" relaci a nebylo možno pokračovat jinak než případným spuštěním relace další. Tato byla na původní zcela nezávislá a kupříkladu veškerá zamýšlená výměna dat se musela odehrávat prostřednictvím exportu do souborů a jejich následného opakovaného zpracování.

PowerShell ve druhé verzi podporuje nezávislé provádění různých úloh, jež mohou běžet zcela nezávisle bez vzájemné synchronizace a jejichž případné datové výstupy je možno zpracovat dodatečně. Nezávislé „pracovní činnosti" v PowerShellu označujeme jako *joby* (jobs) a jejich zavedení je částí mozaiky, kterou tvůrci nazvali zhruba jako „univerzální prováděcí model" (překlad z angličtiny zde poněkud pokulhává za výstižným pojmenováním v originále: *Universal Code Execution Model*). Joby jsou tedy nedílnou součástí širší koncepce, do níž zapadá také spouštění příkazů a skriptů na vzdálených počítačích – o tomto tématu hovoříme v jedné z přilehlých kapitol.

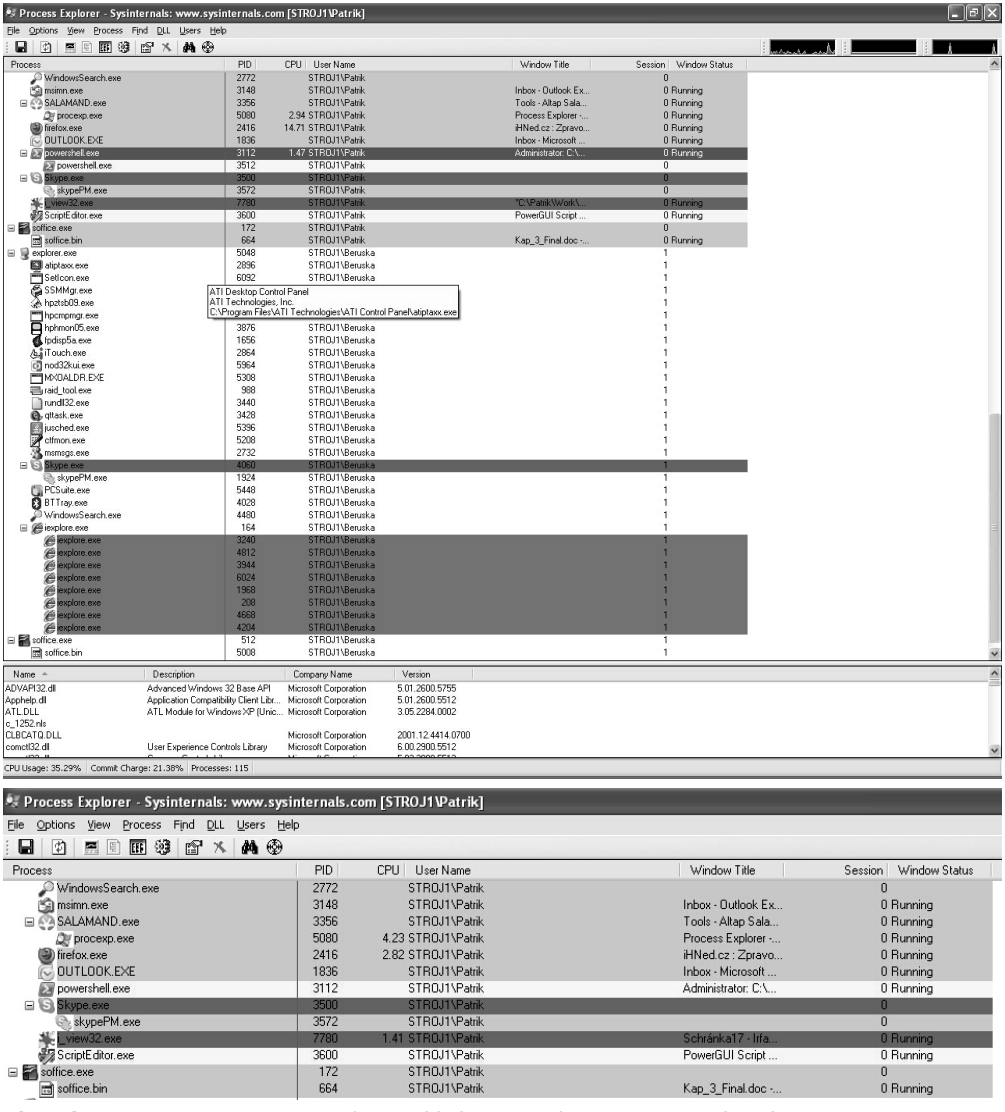
PowerShell je důsledně objektový a tomu také odpovídá způsob zavedení jobů samotných. Spustit jakoukoliv nezávislou úlohu – job – můžeme přímočaře, pomocí následujícího příkazu:

```
=>Start-Job {get-service}

Id              Name            State      HasMoreData     Location        Command
--              ----            -----      -----------     --------        -------
1               Job1            Running    True            localhost       get-service
```

Je patrné, že PowerShell na náš požadavek odpoví vytvořením jakési struktury, o níž nám podává základní přehled. Job je automaticky označen identifikátorem a jeho náležitosti jsou uloženy k dalšímu použití. Jinými slovy, PowerShell vytvořil objekt (instanci třídy) typu „job", který nadále bude nejen podávat o spuštěném úkolu informace, ale případně bude držet získaná data.

Joby, o jejichž stavu se chceme přesvědčit, mohou být snadno zobrazeny následujícím způsobem:

```
=>Get-Job

Id              Name            State      HasMoreData     Location        Command
--              ----            -----      -----------     --------        -------
1               Job1            Completed  True            localhost       get-service
```



Obrázek 3.16 a 3.17: Nově spuštěný *job* je z pohledu operačního systému prováděn jako samostatný proces, o čemž se přesvědčíme třeba pomocí skvělé aplikace Process Explorer. Na prvním obrázku je *job* v běhu, druhý obrázek ukazuje stav po jeho skončení.

Ve výpisu dobře vidíme, že na rozdíl od prvotní zprávy je nyní úloha dokončena (stav je označen jako „Completed“) a navíc nám PowerShell napovídá, že pro nás úloha připravila na svém výstupu nějaká data (vlastnost *HasMoreData* je nastavena na pravdivostní hodnotu *True*). Shromážděná data čekají do chvíle, kdy si je sami „vyzvedneme“, k čemuž použijeme v následujícím příkladu další z rodiny spřízněných cmdletů:

```
=>Receive-Job Job1
```

```
Status   Name               DisplayName
------   ----               -----------
Stopped  Adobe LM Service   Adobe LM Service
Stopped  Alerter            Výstrahy
Running  ALG                Služba brány aplikačního rozhraní
Stopped  AppMgmt            Správa aplikací
Stopped  aspnet_state       Stavová služba ASP.NET
Running  Ati HotKey Poller  Ati HotKey Poller
Stopped  ATI Smart          ATI Smart
Running  AudioSrv           Zvuk systému Windows
Running  BITS               Služba inteligentního přenosu na po...
...
=>
```

Obdrželi jsme na výstupu očekávanou kolekci objektů, na kterou jsme při použití dobře známého příkazu zvyklí. Nyní příkaz zopakujeme ještě jednou:

```
=>Receive-Job Job1
=>Get-Job Job1

Id              Name         State      HasMoreData     Location        Command
--              ----         -----      -----------     --------        -------
1               Job1         Completed  False           localhost       get-service

=>
```

Ukázka jasně naznačuje, že shromážděná data můžeme „vysypat“ jen jednou – job je poté sice stále přítomen v seznamu, ale výsledek jeho činnosti byl odčerpán a není více touto cestou k dispozici. Ideální cestou je tedy uložení dat do proměnné:

```
=>Start-Job {get-service}
=>Get-Job

Id              Name         State      HasMoreData     Location        Command
--              ----         -----      -----------     --------        -------
1               Job1         Completed  False           localhost       get-service
3               Job3         Completed  True            localhost       get-service

=>Get-Job

=>$services = Receive-Job 3
=>$services

Status   Name               DisplayName
------   ----               -----------
Stopped  Adobe LM Service   Adobe LM Service
Stopped  Alerter            Výstrahy
Running  ALG                Služba brány aplikačního rozhraní
Stopped  AppMgmt            Správa aplikací
Stopped  aspnet_state       Stavová služba ASP.NET
Running  Ati HotKey Poller  Ati HotKey Poller
Stopped  ATI Smart          ATI Smart
Running  AudioSrv           Zvuk systému Windows
```

```
Running  BITS               Služba inteligentního přenosu na po...
...
=>
=>Receive-Job 3
=>
```

Objektová podstata (či reprezentace) jednotlivých jobů je výborným prostředkem k jejich hromadné správě, jak nesměle naznačují následující příklady:

```
=>Get-Job | Where-Object {$_.state -eq "Completed"}

Id              Name         State      HasMoreData     Location        Command
--              ----         -----      -----------     --------        -------
1               Job1         Completed  False           localhost       get-service
3               Job3         Completed  False           localhost       get-service

=>Start-Job {get-process}

Id              Name         State      HasMoreData     Location        Command
--              ----         -----      -----------     --------        -------
5               Job5         Running    True            localhost       get-process

=>Get-Job | Where-Object {$_.HasMoreData -eq $true} | % {Receive-Job -job $_ `
-OutVariable $_.name}

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    112       5     1676        204    35     0,08   2928 alg
     52       2      584        732    18     0,19   1492 ati2evxx
     44       2      532        544    18     0,22   2628 ati2evxx
     44       2      516        688    18     0,17   3620 ati2evxx
    101       3     2912       1084    36     0,77   2644 atiptaxx
     99       3     3112        956    35     1,14   4000 atiptaxx
    203      28     4944       1412    55     0,73   4016 BTStackServer
...


=>$job5

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    112       5     1676        204    35     0,08   2928 alg
     52       2      584        732    18     0,19   1492 ati2evxx
     44       2      532        544    18     0,22   2628 ati2evxx
     44       2      516        688    18     0,17   3620 ati2evxx
    101       3     2912       1084    36     0,77   2644 atiptaxx
     99       3     3112        956    35     1,14   4000 atiptaxx
    203      28     4944       1412    55     0,73   4016 BTStackServer
...
```

Druhý příklad naznačuje, že můžeme využít přepínače příkazu *Receive-Job* pro určení výstupní proměnné, což oceníme obzvláště při zpracování více úloh najednou. Stejný

postup provádíme i v následujícím příkladu, kde je pro zjednodušení akce provedena bez čerpání dat z roury:

```
=>Start-Job {Get-Process}

Id              Name            State      HasMoreData     Location        Command
--              ----            -----      -----------     --------        -------
9               Job9            Running    True            localhost       Get-Process

=>Receive-Job 9 -OutVariable job9 | out-null
=>$job9

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    112       5     1676        212    35     0,08   2928 alg
     52       2      584        736    18     0,19   1492 ati2evxx
     44       2      532        548    18     0,22   2628 ati2evxx
     44       2      516        692    18     0,17   3620 ati2evxx
    101       3     2912       1088    36     0,80   2644 atiptaxx
     99       3     3112        960    35     1,14   4000 atiptaxx
    203      28     4944       1420    55     0,73   4016 BTStackServer
...
```

Potřebujeme-li, aby objekt reprezentující určitý job si výstupní datovou sadu podržel i přesto, že ji poprvé odčerpáme, lze výchozí chování cmdletu *Receive-Job* ovlivnit příslušným přepínačem, jak je vidět v této ukázce:

```
=>Start-Job {Get-WmiObject win32_useraccount}

Id         Name        State      HasMoreData     Location        Command
--         ----        -----      -----------     --------        -------
11         Job11       Running    True            localhost       Get-WmiObject wi...


=>Get-Job

Id         Name        State      HasMoreData     Location        Command
--         ----        -----      -----------     --------        -------
...
9          Job9        Completed  False           localhost       Get-Process
11         Job11       Completed  True            localhost       Get-WmiObject wi...

=>Receive-Job 11 -Keep

AccountType : 512
Caption     : STROJ1\Administrator
Domain      : STROJ1
SID         : S-1-5-21-527237240-1979792683-839522115-500
FullName    :
Name        : Administrator

AccountType : 512
Caption     : STROJ1\ASPNET
```

```
Domain      : STROJ1
SID         : S-1-5-21-527237240-1979792683-839522115-1005
FullName    : ASP.NET Machine Account
...

=>Get-Job 11

Id       Name        State      HasMoreData     Location       Command
--       ----        -----      -----------     --------       -------
11       Job11       Completed  True            localhost      Get-WmiObject wi...

=>Receive-Job 11

AccountType : 512
Caption     : STROJ1\Administrator
Domain      : STROJ1
SID         : S-1-5-21-527237240-1979792683-839522115-500
FullName    :
Name        : Administrator
...
```

Je dobře vidět, že původní objekt si datovou sadu podržel a při opětovném dotázání byla tato k dispozici.

Paralelní spuštění krátkých úloh na témže počítači

Mechanizmus jobů je velmi dobře použitelný pro spouštění dávek, jež zahrnují řadu jednoduchých skriptů či příkazů. Tento postup je výchozím řešením pro spuštění několika nezávislých příkazů, jež provedou inventuru operačního systému a výsledek uloží do výstupního souboru CSV. Nezávislé provádění úloh naznačuje možnosti zrychlení celé dávky, která může v praxi zahrnovat větší množství déle trvajících úkonů.

jobs.ps1

1. Nejdříve si sestavíme seznam úloh, jež budeme následně spouštět jako samostatné joby.

   ```
   #definujeme jednotlive ulohy

   $akce = ConvertFrom-StringData @"
   sluzby={Get-Service}
   procesy={Get-Process}
   ucty={Get-WmiObject win32_useraccount}
   "@
   ```

   Použili jsme elegantní formát hešovací tabulky, v níž budeme uchovávat jméno každé úlohy a jí přináležející příkaz (či obecně skriptovací blok). Hešovací tabulka vznikla převodem textového řetězce se zapsanými dvojicemi (konstrukce *Here-String*) pomocí příkazu *ConvertFrom-StringData*, který je novinkou verze 2.

2. Pokračujme nejdříve preventivním zahozením případných objektů reprezentujících joby stejného jména a posléze již můžeme naše nové joby nastartovat.

```
#preventivni odstraneni jobu stejneho jmena

$akce.GetEnumerator() | % { get-Job -name "Uloha_$($_.Name)"} -ea `
silentlycontinue | Remove-Job

#spoustime joby

$akce.GetEnumerator() | % { Start-Job -name "Uloha_$($_.Name)" -ScriptBlock `
(Invoke-expression $_.Value) -OutVariable $_.Name}
```

3. Poté, co úlohy začnou nezávisle získávat data, předáme příkaz k vyčkávání na jejich dokončení:

```
#cekame na dokonceni jobu

$akce.GetEnumerator() | % { Wait-Job -name "Uloha_$($_.Name)"}

4. Jakmile jsou joby ukončeny, můžeme zahájit výstup dat do exportních souborů:

#export dat do souboru CSV

$akce.GetEnumerator() | % { Receive-Job -name "Uloha_$($_.Name)" -keep | Export-
Csv -path "c:\$($_.Name).csv" -NoTypeInformation}
```

Paralelní spuštění větších skriptů na témže počítači

Průběžné spouštění nezávislých dávek, jež pracují bez synchronizace, provádíme často s mnohem rozsáhlejšími skripty (skriptovacími bloky), než jsou jen základní příkazy či jednořádkové konstrukce. V takovém případě potřebujeme skriptovací bloky nejdříve získat ze zdrojových souborů a následně je předat do jednotlivých úloh k samotnému provedení. Následující ukázka naznačuje jednu z možností, jak takovou sestavu úloh spustit.

1. Před samotným spuštěním si připravíme zdrojové kódy jednotlivých úloh, jež uložíme jako jednotlivé soubory. Pro naši dávku využijeme jednoduché úlohy, jejichž velikost však není v zásadě nijak omezena. Obsah zdrojových souborů vypadá takto:

services.ps1, processes.ps1, jobs2.ps1

```
=>cat .\services.ps1
get-service | Where-Object {$_.status -like "*run*"}

=>cat .\processes.ps1
get-process | Where-Object {$_.WS -gt 10MB}

=>
```

2. Samotná dávka začíná přípravou proměnné obsahující cestu k našim zdrojovým souborům. Tu využijeme následně při jejich načítání – zde používáme stejně jako v minulém příkladu hešovací tabulku, která nám uskladní dvojice jméno=skript.

```
#definujeme jednotlive ulohy
```

```
$base_path = "C:\\Patrik\\Work\\PowerShellBook2\\temp"

$akce = ConvertFrom-StringData @"
sluzby={Invoke-expression (cat "$($base_path).\\processes.ps1")}
procesy={Invoke-expression (cat "$($base_path).\\services.ps1")}
"@
```

Příkaz *Invoke-Expression* jsme zavolali na pomoc proto, aby provedl načtený textový obsah souborů – tento příkaz tedy na náš výslovný povel chápe zdrojový text jako kód a provede jej. Povšimněme si zdvojeného znaku lomítka v adresářové cestě – zde nesmíme zapomínat na skutečnost, že celá konstrukce je zapisována uvnitř konstrukce *Here-String*, a proto jde vlastně o řetězec (citaci v uvozovkách). Proto je třeba lomítko „odrazit", aby nebylo chápáno jako uvozovací znak pro vložení speciálních sekvencí (tzv. sekvence escape). Dvě lomítka tedy zdůrazňují: toto je prostě lomítko, a ne něco jiného.

3. Pokračujeme již známým způsobem. Nejdříve odstraníme existující joby a posléze začneme samotné provádění úloh.

```
#preventivni odstraneni jobu stejneho jmena

$akce.GetEnumerator() | % { get-Job -name "Uloha_$($_.Name)"} -ea `
silentlycontinue | Remove-Job

#spoustime joby

$akce.GetEnumerator() | % { Start-Job -name "Uloha_$($_.Name)" -ScriptBlock `
(Invoke-Expression $_.Value) -OutVariable $_.Name}
```

4. Na závěr opět počkáme na provedení úloh a posléze na výpis dat do výstupních souborů.

```
#cekame na dokonceni jobu

$akce.GetEnumerator() | % { Wait-Job -name "Uloha_$($_.Name)"}

#export dat do souboru CSV

$akce.GetEnumerator() | % { Receive-Job -name "Uloha_$($_.Name)" -keep | Export-
Csv -path "c:\$($_.Name).csv" -NoTypeInformation}
```

Shrnutí

PowerShell 2 zavádí významnou možnost spouštění nezávislých úloh prostřednictvím izolovaných, nesynchronizovaných jobů. Takovéto úlohy běží stejně na pozadí bez blokování interaktivní konzoly jako na vzdálených počítačích bez potřeby interaktivní vzdálené relace.

Důležité k zapamatování

- Nezávislá úloha v PowerShellu – job – vždy vzniká v podobě objektové reprezentace se všemi výhodami, jež toto uložení dat přináší.

- Job může být snadno spuštěn na vzdáleném počítači, aniž bychom museli vstupovat do interaktivní relace se vzdálenou ovládací konzolou.
- Vlastnosti objektu typu job dovolují průběžně kontrolovat stav úlohy a po jejím dokončení prověřit, zda vrací datovou sadu, a případně ji načíst kdykoliv po skončení úlohy.
- Joby je možno provádět stejně snadno na lokálním jako na vzdáleném počítači.

Plánování úloh

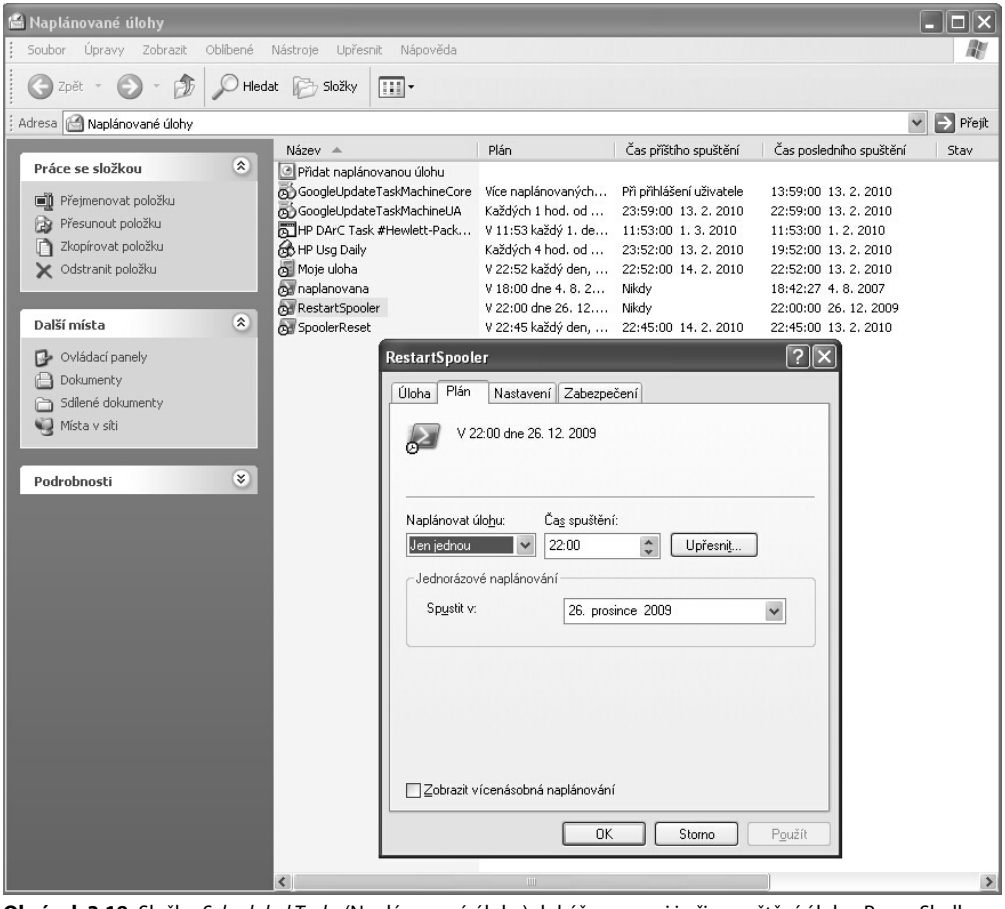
Úvod

Možnost spustit naplánované úlohy či operace je neodmyslitelně spjata s jakoukoliv automatizací práce v operačním systému. Dlouhá řada typických správcovských činností vyžaduje neinteraktivní provedení, navíc v době, kdy správce často typicky nebývá přítomen, a vlastně to ani není potřeba. Naplánované úlohy s sebou přinášejí ještě přinejmenším jednu výhodu – možnost pravidelného opakování za stejných podmínek, což bývá nejslabší stránka správce z masa a kostí.

Operační systém Windows obsahuje službu *Scheduled tasks* (*Naplánované úlohy*, též *Task Scheduler*), jež zodpovídá za spouštění naplánovaných akcí a také mimo jiné za ověření oprávnění uživatele, který úlohy plánuje. Tato služba po ověření totožnosti a oprávnění přebírá za akci zodpovědnost a před i po plánovaném spuštění má k dispozici údaje o stavu úlohy.

Služba *Scheduled Tasks* pro nás bude při povídání o PowerShellu zajímavá hned z několika pohledů, na něž se postupně podíváme. Jako první vyjmenujme situaci, kdy služba spouští skript v PowerShellu jako naplánovanou úlohu – využíváme ji tedy jako mechanizmus v jeho původním určení s tím, že powershellový skript je úlohou samotnou. Druhý případ je vlastně obráceným úhlem pohledu – ukážeme si, jak můžeme pomocí PowerShellu plánovat úlohy, a budeme jej tedy využívat jako ovládací rozhraní pro službu samotnou. Třetí situace pak bude jakási prémie: služba naplánovaných úloh totiž může elegantně vyřešit obvyklý problém, kdy chceme povýšit práva na spuštění jen určitého skriptu a přitom jej chceme aktivovat interaktivně, vlastně „na počkání“. I zde nám naplánovaná úloha dobře poslouží.

Tvůrci Windows trošku zkomplikovali život všem správcům tím, že v průběhu vývoje existovaly různé mechanizmy plánování úloh, k nimž se navíc nepřistupuje pomocí skriptů jednotným způsobem. Příkaz klasického shellu AT je tradičním a „historickým“ způsobem plánování úloh a je možné jej ovládat také pomocí třídy WMI jménem *Win32_ScheduledJob*. Tato třída s velmi slibným jménem naopak nespolupracuje s pozdější, „normální“ službou *Scheduled Tasks*, která se vyskytuje třeba ve Windows 2000, XP či 2003. Tato varianta je dostupná ve starém shellu pomocí příkazu *Schtasks.exe*, avšak pro její ovládání ve skriptech (WSH či PowerShell) je pro změnu nutno doinstalovat knihovnu DLL, která poskytuje potřebné rozhraní COM (jak je dále popsáno). Tato knihovna byla k dispozici buďto z dílny Microsoftu jako součást produktu Site Server (což asi mnoho správců jen tak zavádět nebude), nebo ve volné distribuci jako freeware z dílny jednoho nadšence (jak ukážeme dále). Krom toho programátoři pracující s .NET Framewor-



Obrázek 3.18: Služba *Scheduled Tasks* (Naplánované úlohy) dokáže pomoci i při spouštění úloh v PowerShellu. Stejně tak ji můžeme pomocí PowerShellu ovládat.

kem se také nemohli smířit s tím, že služba ve verzi 1 nemá pořádné rozhraní pro psaní aplikací v jazycích .NETu, a proto napsali analogickou knihovnu s potřebným rozhraním na bázi .NET Frameworku. Tato knihovna nabízí přímé použití v PowerShellu jako u jiných tříd, jež jsou zabaleny v běžné distribuci .NET Frameworku. Služba stejného jména – *Scheduled Tasks* – je pak dostupná i v nejnovějších variantách Windows (Vista, 2008, 7), avšak byla přepracována, takže již sice nabízí vlastní rozhraní COM, ale jiné než verze předchozí. Situaci shrnuje také tabulka. V našich ukázkách se budeme věnovat všem důležitým variantám.

Tabulka 3.2: Nástroje pro plánování úloh ve Windows a možnosti jejich ovládání

Služba	Verze Windows	Klasické ovládání	Rozhraní ke skriptování
AT.exe	Všechna	Cmd.exe – AT.exe	Win32_ScheduledJob (rozhraní WMI)

Služba	Verze Windows	Klasické ovládání	Rozhraní ke skriptování
Scheduled Tasks 1	2000, XP, 2003	Cmd.exe – Schtasks.exe	Task Scheduler DLL 1 (rozhraní COM) Task Scheduler DLL 1 (.NET)
Scheduled Tasks 2	Vista, 2008, 7	Cmd.exe – Schtasks.exe	Task Scheduler DLL 2 (rozhraní COM)

Ovládání služby Scheduled Tasks pomocí COM na Windows XP

Služba Scheduled Tasks je v operačním systému Windows XP a starších vybudována poněkud jiným způsobem než na systémech novějších (Windows Vista a pozdější), někdy se dokonce mluví o Scheduled Tasks verze 1 a 2. Starší varianta ve Windows XP má zásadní nevýhodu – chybí jí ovládací „prvky", díky nimž bychom mohli službu snadno uchopit a ovládat prostřednictvím rozhraní COM, stejně jako řadu jiných aplikací a služeb. Tomuto účelu může posloužit speciální knihovna DLL, vyvinutá mimo společnost Microsoft, jež zavádí objektový model pro ovládání služby i na počítače s Windows XP a starší. Ukážeme si použití tohoto rozhraní, s nímž lze samozřejmě pracovat i v PowerShellu.

scheduler1.ps1

1. Stáhneme potřebnou knihovnu z internetových stránek: *http://www.tlviewer.org/mstask/*

Upozornění: Doporučovaná knihovna DLL není výrobkem společnosti Microsoft a není výchozí součástí Windows. Autorská práva na její použití se tedy neshodují s právy na systém Windows a PowerShell jako takový. Totéž platí o případné technické podpoře.

2. Zaregistrujeme knihovnu v systému Windows, aby byla přístupná přes rozhraní COM.

```
=>regsvr32 .\TaskScheduler.dll

Program RegSvr32.exe potvrdí registraci hlášením.
```

Obrázek 3.19: Potřebnou knihovnu zaregistrujeme obvyklým způsobem pomocí příkazu programu Regsvr32.exe

3. Vytvoříme si objektovou proměnnou, která bude reprezentovat agent služby naplánovaných úloh. Zaregistrovaná knihovna nám poskytne potřebný objektový model.

```
$agent = New-Object -ComObject "Scheduler.SchAgent"
```

4. Zavoláme proměnnou-službu a příslušnou metodu, jež zajistí připojení k „živému systému".

```
=>$agent.Refresh()
```

5. Založíme novou proměnnou, jež bude reprezentovat námi vytvořenou úlohu. Postupně přiřadíme několik vlastností, jež budou naši úlohu určovat.

```
$uloha = $agent.CreateTask('Moje uloha')
$uloha.ApplicationName = 'calc.exe'
$uloha.Creator = 'Patrik'
$uloha.SetAccountInformation('stroj1\patrik','********')
```

Metoda *CreateTask* zakládá novou objektovou proměnnou, z vyjmenovaných vlastností je pak důležitá především ta, jež určuje uživatelské jméno a heslo účtu, pod jehož právy dojde k ověření a autorizaci úlohy.

6. Po zapsání potřebných vlastností zavoláme metodu, jež data skutečně zapíše do „inventáře" služby, a poté můžeme vyzkoušet její bezprostřední spuštění.

```
=>$uloha.Save()
=>$uloha.Run()
```

7. Budeme pokračovat v nastavení úlohy naplánováním jejího spouštění. Pro tento účel si musíme vytvořit další objektovou proměnnou, jež bude zastupovat spouštěcí plán.

```
=>$spoust = $uloha.Triggers.add()
=>$spoust

DaysInterval   : 1
Day            :
Months         :
DaysOfTheWeek  :
Week           :
WeeksInterval  :
Text           : Aktivační procedura nebyla nastavena na platnou hodnotu.
TriggerType    : 1
Duration       : 0
Interval       : 0
Flags          : 0
BeginDay       : 25. 12. 2009 0:00:00
EndDay         : 30. 11. 1999 0:00:00
RandomInterval : 0
StartTime      : 30. 12. 1899 22:12:00
```

Popis nového objektu nám krásně ukazuje, jaké hodnoty lze přesně nastavit – pokud službu zběžně známe z grafického rozhraní, jsou nám parametry přinejmenším povědomé.

8. Přiřadíme potřebné hodnoty pro automatický naplánovaný start služby.

```
$spoust.TriggerType = 1
$spoust.BeginDay = get-date
$spoust.StartTime = ((get-date).addminutes(5))
```

Vlastnost *TriggerType* určuje základní plánovací interval (v tomto případě denně), další dvě vlastnosti jsou svým názvem dosti výmluvné. Výčet možných nastavení tím nekončí – jde o ukázku výchozí kombinace, jež postačí pro základní plán.

9. Založený plán je potřeba zapsat a celé nastavení úlohy uložit, aby je služba vzala na vědomí.

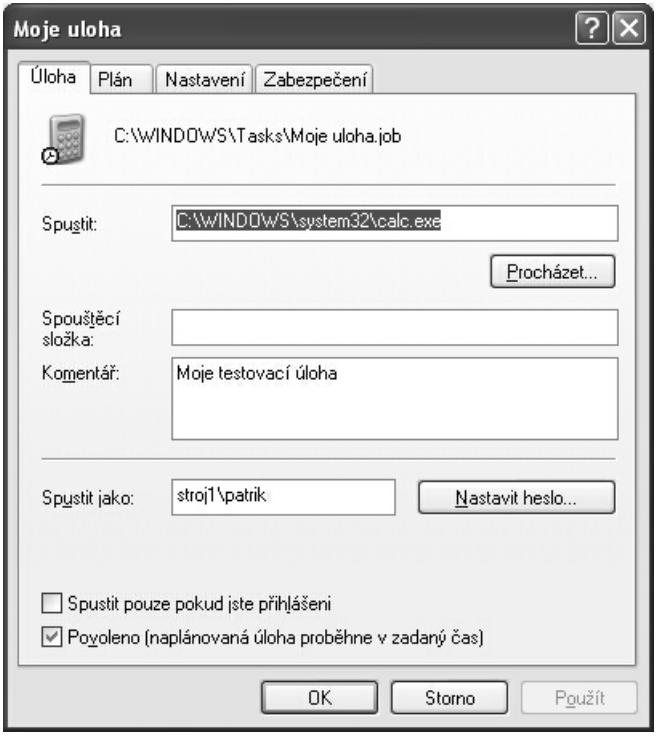
```
=>$spoust.Update()
=>$uloha.Save()
```

10. Nastavení úlohy je možno ověřit jak v grafickém rozhraní, tak přímým dotazem na službu. Naše úloha bude po prvním spuštění vypadat takto:

```
=>$agent.Refresh()
=>$agent | Where-Object {$_.name -like "Moje*"}

ApplicationName    : C:\WINDOWS\system32\calc.exe
CommandLine        :
Comment            :
Creator            : Patrik
ErrorRetryCount    :
ErrorRetryInterval :
ExitCode           : 0
Flags              : 0
IdleWait           :
MaxRunTime         : 259200000
Priority           : 32
Status             : 267008
LastRunTime        : 25. 12. 2009 22:52:00
NextRunTime        : 26. 12. 2009 22:52:00
FileName           : C:\WINDOWS\Tasks\Moje uloha.job
Name               : Moje uloha
WorkingDirectory   :
AccountName        : stroj1\patrik
Triggers           : System.__ComObject
WorkItemData       :
```

11. Na závěr se můžeme přesvědčit, že změna nastavení v grafickém rozhraní je rovnocenná změně pomocí objektového modelu. Přidali jsme komentář v grafickém rozhraní a objektový model poskytl následující informaci:

```
=>$agent.Refresh()
=>$agent | Where-Object {$_.name -like "Moje*"}

ApplicationName    : C:\WINDOWS\system32\calc.exe
CommandLine        :
Comment            : Moje testovací úloha
...
```



Obrázek 3.20: Naplánovaná úloha i s parametry může vzniknout také dílem PowerShellu, pokud sáhneme po vhodné doplňující knihovně pro rozhraní COM

Seznam naplánovaných úloh na Windows XP

Služba Naplánované úlohy může být ovládána pomocí objektového rozhraní, které jsme popsali v předchozím příkladu. Sestavení seznamu naplánovaných úloh pomocí něho je poměrně snadným úkolem, a to i na vzdáleném počítači.

1. Vytvoříme objekt reprezentující službu naplánovaných úloh.

```
$agent = New-Object -ComObject "Scheduler.SchAgent"
```

2. V případě potřeby určíme vzdálený počítač, pro nějž budeme seznam zjišťovat.

```
$agent.TargetComputer = "\\stroj1"
$agent.Refresh()
```

Upozornění: Pokud bude jméno nepřeložitelné a vzdálený stroj nebude dostupný, rozhraní zahlásí již v této fázi chybu a nenechá nás proměnnou vyrobit.

3. Aktivujeme spojení se službou pomocí příslušné metody a posléze již můžeme službu dotazovat na aktuálně naplánované úlohy.

```
=>$agent | fl Name,applicationname,creator
```

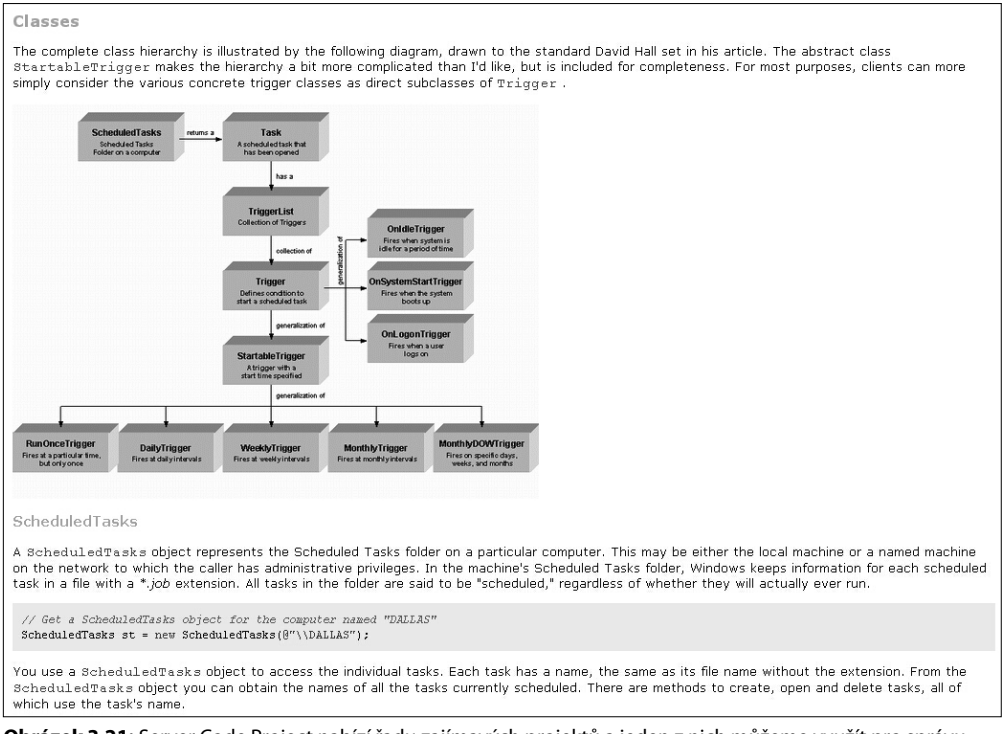
```
Name            : GoogleUpdateTaskMachineCore
ApplicationName : C:\Program Files\Google\Update\GoogleUpdate.exe
Creator         : SYSTEM

Name            : Moje uloha
ApplicationName : C:\WINDOWS\system32\calc.exe
Creator         : Patrik

Name            : naplanovana
ApplicationName : C:\WINDOWS\system32\windowspowershell\v1.0\powershell.exe
Creator         : Patrik
...
```

4. PowerShell nám nabízí prověřené prostředky pro uložení kompletní sady dat do strukturovaných souborů, kterých zde můžeme využít.

```
=>$agent | Export-Clixml c:\tasks.xml
=>$agent | Export-Csv c:\tasks.csv
```

Ovládání služby Scheduled Tasks pomocí tříd .NET

V úvodu jsme uvedli, že zpřístupnění služby *Scheduled Tasks* pro skriptování a programování se nepozdávalo mimo jiné také jednomu z programátorů na platformě .NET, který připravil zapouzdření této služby do tříd tohoto novějšího běhového prostředí. Autor uveřejnil zdarma knihovnu, jež představuje další alternativu přístupu ke službě *Scheduled Tasks* v první verzi (tedy 2000-XP-2003).



Classes

The complete class hierarchy is illustrated by the following diagram, drawn to the standard David Hall set in his article. The abstract class `StartableTrigger` makes the hierarchy a bit more complicated than I'd like, but is included for completeness. For most purposes, clients can more simply consider the various concrete trigger classes as direct subclasses of `Trigger` .

ScheduledTasks

A `ScheduledTasks` object represents the Scheduled Tasks folder on a particular computer. This may be either the local machine or a named machine on the network to which the caller has administrative privileges. In the machine's Scheduled Tasks folder, Windows keeps information for each scheduled task in a file with a *.job* extension. All tasks in the folder are said to be "scheduled," regardless of whether they will actually ever run.

```
// Get a ScheduledTasks object for the computer named "DALLAS"
ScheduledTasks st = new ScheduledTasks(@"\\DALLAS");
```

You use a `ScheduledTasks` object to access the individual tasks. Each task has a name, the same as its file name without the extension. From the `ScheduledTasks` object you can obtain the names of all the tasks currently scheduled. There are methods to create, open and delete tasks, all of which use the task's name.

Obrázek 3.21: Server Code Project nabízí řadu zajímavých projektů a jeden z nich můžeme využít pro správu naplánovaných úloh

Tip: Knihovna s potřebnými třídami je ke stažení na této stránce: *http://www.codeproject.com/KB/cs/tsnewlib.aspx*

Na tomto místě je k dipozici také další dokumentace.

Použití této knihovny je dosti podobné jako u verze s rozhraním COM, proto naznačíme jen hlavní postupy jejího použití, jež bude dále zřejmé.

scheduler2.ps1

1. Načteme knihovnu do relace PowerShellu, aby byla přístupná pro tvorbu objektů.

```
=>[Reflection.Assembly]::LoadFile("C:\psh\taskscheduler.dll")
```

2. Práci s úlohami zahájíme tvorbou objektu, jenž reprezentuje celou službu.

```
=>$sch = New-Object taskscheduler.scheduler
```

3. Nový objekt obsahuje základní kolekci *Tasks*, která reprezentuje existující naplánované úlohy.

```
=>$sch.Tasks | select name, creator,applicationname | fl *

Name            : User_Feed_Synchronization-{85B293B1-25AA-435B-8486-
AA9D2015FF40}
Creator         : Patrik
ApplicationName : C:\WINDOWS\system32\msfeedssync.exe

Name            : naplanovana
Creator         : Patrik
ApplicationName : C:\WINDOWS\system32\windowspowershell\v1.0\powershell.exe

Name            : Moje uloha
Creator         : Patrik
ApplicationName : C:\WINDOWS\system32\calc.exe
...
```

4. Založíme novou úlohu tím, že přidáme nový prvek do kolekce úloh existujících. Tento prvek si uložíme do proměnné, jíž budeme následně nastavovat parametry.

```
=>$uloha = $sch.Tasks.NewTask("DotNetUloha")
=>$uloha.ApplicationName = "DotNetUloha.exe"
=>$uloha.Creator = "Patrik"
=>$uloha.Comment = "Testovaci DotNetUloha"
```

5. Nyní úlohu naplánujeme, a to opět pomocí kolekce *Triggers*, která za jejich „spouštění“ zodpovídá. Nejdříve si připravíme objekt s časovým údajem a poté jej úloze přímo přiřadíme.

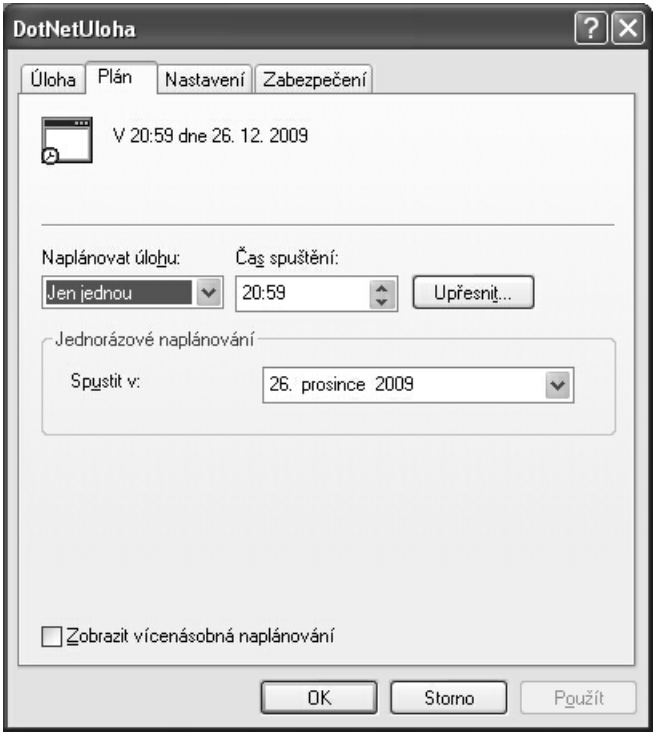
```
=>$casovac = New-Object TaskScheduler.RunOnceTrigger((get-date).addminutes(10))
=>$uloha.Triggers.Add($casovac)
```

Povšimněme si, že tato implementace obsahuje jednotlivé třídy pro určité typy spouštění služby – my jsme použili třídu pro časovač *RunOnce*, která zajistí jediné provedení. Na výše uvedené webové stránce najdeme kompletní dokumentaci.

6. Nezbývá, než přiřadit oprávnění pro spuštění a následně úlohu definitivně uložit.

```
=>$uloha.SetAccountInformation("stroj1\patrik","heslo")
=>$uloha.Save()
```

7. Můžeme se přesvědčit mimo jiné i v grafickém rozhraní, zda byla úloha správně vytvořena, viz následující obrázek.



Obrázek 3.22: Naplánovanou úlohu můžeme doladit do posledních detailů, ani časový plán spouštění nepředstavuje problém

8. Úlohu můžeme v případě potřeby odstranit velmi jednoduchým způsobem:

```
=>$sch.Tasks | Where-Object {$_.name -like "DotNet*"}

Name                    : DotNetUloha
Triggers                : {V 20:59 dne 26. 12. 2009}
ApplicationName         : DotNetUloha.exe
AccountName             :
Comment                 : Testovaci DotNetUloha
Creator                 : Patrik
...

=>$sch.Tasks.Delete("DotNetUloha")

=>$sch.Tasks | Where-Object {$_.name -like "DotNet*"} | Measure-Object
```

```
Count    : 0
...
```

Ovládání programu Schtasks.exe

Program *Schtasks.exe* je klíčovým nástrojem pro automatizovanou konfiguraci a správu služby Naplánované úlohy. Když se poprvé objevil, znamenal rozhodně posun vpřed při správě této služby, na druhou stranu s sebou nese některé neduhy, jako je třeba lokalizace parametrů do různých jazyků dle verze Windows (což je obecně tragická praxe tvůrců tohoto OS). PowerShell můžeme přímo použít na spouštění tohoto konzolového programu a na ovládání služby, pokud nám nevyhovují jiné varianty přes rozhraní COM či .NET.

schtasks.ps1

1. Nastavíme si v našem skriptu výchozí proměnné, jež posléze předáme příkazu *Schtasks.exe*.

```
#nazev ulohy
$TN = "Nocni udrzba"
#ucet pro ulohu -- lokalni system
$RU = "SYSTEM"
#plan spousteni -- jednou
$SC = "Jen jednou"
#uloha
$TR = "calc.exe"
```

2. Proměnné určující čas a datum spouštění můžeme přiřadit dynamicky – v tomto případě nám PowerShell výrazně pomůže při zpracování údajů týkajících se data. Povšimněme si, že časové údaje jsou přeformátovány tak, aby textový řetězec na výstupu přesně odpovídal požadavkům na parametry příkazu *Schtasks.exe* (tento neumí pracovat s objekty, ale jen s textovým vstupem).

```
#cas spusteni - od ted za 10 minut
$ST = $ST = "{0:HH:mm:ss}"  -f ((get-date).addminutes(10))
#datum spusteni - tento den
$SD = get-date -UFormat "%d/%m/%Y"
```

   Obsah proměnných můžeme otestovat:

```
> $ST
23:48:52
> $SD
26/12/2009
```

3. Pustíme program *Schtasks.exe* s příslušnými přepínači a předáme mu vstupní parametry pomocí proměnných.

```
schtasks.exe /Create /TN $TN /RU $RU /SC $SC /ST $ST /SD $SD /TR $TR

Info: Naplánovaná úloha Nocni udrzba bude vytvořena pod uživatelským jménem ("NT
AUTHORITY\SYSTEM").

Úspěch: Naplánovaná úloha Nocni udrzba byla úspěšně vytvořena.
```

4. Stejný program nám poslouží ke zběžnému ověření, jak je úloha nastavena.

```
=>schtasks.exe /Query

Název úlohy       Čas příštího spuštění    Stav
============================================================
...
Nocni udrzba       23:58:00, 26. 12. 2009
...
```

Spuštění úlohy napsané v PowerShellu

Skripty napsané v jazyce PowerShell můžeme plánovat pro automatické spuštění stejně jako jakékoliv jiné automatické akce ve Windows. Nesmíme pouze zapomínat, že zdrojové soubory v jazyce PowerShell s příponou *.ps1* jsou typicky asociovány s jiným programem, než je samotný interpretr *powershell.exe*. Musíme tedy při plánování automatické úlohy výslovně uvést, že bude spuštěn program *PowerShell.exe* a jemu bude předána cesta ke zdrojovému souboru jako jeden z parametrů. Následující příklad ukazuje postup naplánování úlohy napsané v PowerShellu – skript bude restartovat službu Spooler a provede to každý den. Ke konfiguraci použijeme knihovnu pro .NET Framework, jejíž ukázky jsou i v jiných postupech této kapitoly.

scheduler3.ps1

1. Načteme knihovnu a vytvoříme proměnnou reprezentující službu Naplánovaných úloh.

```
=>[Reflection.Assembly]::LoadFile("C:\TaskSchedulerDotNet\taskscheduler.dll")
=>$sch = New-Object taskscheduler.scheduler
```

2. Založíme novou proměnnou zastupující novou úlohu a přiřadíme jí první popisné informace.

```
=>$spool = $sch.Tasks.NewTask("SpoolerReset")
=>$spool.Creator = "Patrik"
=>$spool.Comment = "Restart sluzby Spooler"
```

3. V dalším kroku přiřadíme úloze jméno spustitelného programu, v našem případě tedy spustitelný interpretr PowerShellu.

```
=>$spool.ApplicationName = `
"%systemroot%\system32\WindowsPowerShell\v1.0\powershell.exe"

=>$spool.ApplicationName
C:\WINDOWS\system32\WindowsPowerShell\v1.0\powershell.exe
```

4. Prozatím jsme službě neřekli, jaký zdrojový soubor bude vlastně v PowerShellu spuštěn. Nyní přiřadíme aplikaci přepínače, mezi nimiž bude i odkaz na zdrojový soubor naší úlohy.

```
=>$spool.Parameters = `
"-noprofile -widowstyle hidden -file C:\temp\Restart-spooler.ps1"
```

5. Dále musíme poskytnout uživatelský účet a také načasování spouštění úlohy.

```
=>$spool.SetAccountInformation("stroj1\patrik","patrik")
```

```
=>
=>$casovac = New-Object TaskScheduler.DailyTrigger(22,45,1)
=>$casovac

DaysInterval      : 1
StartHour         : 22
StartMinute       : 45
BeginDate         : 26. 12. 2009 0:00:00
HasEndDate        : False
EndDate           : 1. 1. 0001 0:00:00
DurationMinutes   : 0
IntervalMinutes   : 0
KillAtDurationEnd : False
Disabled          : False

=>$spool.Triggers.Add($casovac)
0
```

Za povšimnutí stojí volání třídy *DailyTrigger*, která vytváří časovač. Její jméno napovídá, že nastavuje spouštění v každodenním režimu, a tři číslice jsou parametry předané tzv. konstruktoru – označují hodinu a minutu spuštění a také počet dní určující periodu (více je pojednáno o konstruktorech v kapitole 4).

6. Uložíme novou úlohu a přesvědčíme se o jejích nastaveních.

```
=>$spool.Save()
=>$spool

Name                    : SpoolerReset
Triggers                : {V 22:45 každý den, poprvé 26. 12. 2009}
ApplicationName         : C:\WINDOWS\system32\WindowsPowerShell\v1.0\powershell.
exe
AccountName             : stroj1\patrik
Comment                 : Restart sluzby Spooler
Creator                 : Patrik
...
NextRunTime             : 26. 12. 2009 22:45:00
Parameters              : -noprofile -noninteractive -widowstyle hidden -file
C:\temp\Restart-spooler.ps1
Priority                : Normal
Status                  : NeverRun
WorkingDirectory        :
Hidden                  : False
Tag                     :
```

7. Jakmile mine alespoň první termín spuštění úlohy, můžeme její stav prověřit.

```
=>$sch.Tasks | Where-Object {$_.name -like "*spoolerreset*"} | `
fl name,MostRecentRunTime,NextRunTime

Name              : SpoolerReset
MostRecentRunTime : 26. 12. 2009 22:45:00
NextRunTime       : 27. 12. 2009 22:45:00
```

Poznámka: Interpretr PowerShell.exe nabízí přepínač *WindowStyle*, pomocí nějž můžeme určit podobu okna, v němž úloha po spuštění poběží. V našem příkladu jsme vybrali hodnotu *hidden*, jež okno ukrývá – skutečné chování je však poněkud odlišné. PowerShell po startu na malou chvíli okno ve skutečnosti ukáže a posléze uposlechne povel a opět je ukryje. Někteří administrátoři to považují za vadu a snaží se ukrytí provést dokonale tak, aby nic ani „neprobleskло". Autor toto nepovažuje za natolik zásadní, a proto namísto postupu nabízí odkaz na stránky jedněch z řešitelů problému. Shrnuto: jde to, ale poněkud krkolomně.

http://blog.sapien.com/index.php/2006/12/20/schedule-hidden-powershell-tasks/

Shrnutí

Podpora skriptování naplánovaných úloh pomocí služby *Scheduled Tasks* je dosti proměnlivá a závisí na verzi Windows, v níž pracujeme.

Důležité k zapamatování

- Třída *Win32_ScheduledJob* má sice velmi slibné jméno, ale její použití se vztahuje výhradně k úlohám naplánovaným pomocí příkazu *AT.exe*. Pokud jej příliš nepoužíváte, tato třída vám toho mnoho neposkytne.
- Rozhraní COM, jež voláme pomocí objektového modelu a třídy *Schedule.Service*, je dostupné pouze od verze Windows Vista výše, neboť Microsoft službu Scheduled Tasks přepracoval na „verzi 2".

Moduly v PowerShellu

Úvod

Koncepce tzv. modulů byla zavedena ve verzi 2 a jednak nahrazuje, jednak podstatně rozšiřuje technologii tzv. snap-inů, zavedenou již v první verzi PowerShellu. Moduly jsou nesmírně důležitým prostředkem pro opakované využití existujících skriptů, a proto se u nich zastavíme.

Modul v PowerShellu si můžeme jednoduše představit jako kus zdrojového kódu, jejž můžeme opakovaně volat (tedy jako funkci), ale který je definován (a fyzicky uložen) v odděleném souboru – jakési knihovně předem připravených „užitečných součástek". Tato koncepce není ničím převratným, neboť podobné principy se uplatňují v řadě jazyků (včetně platformy WSH ve Windows), a důležitým je tak především fakt, že se do PowerShellu konečně dostala v provedení, jež si vysvětlíme.

Moduly mají oproti snap-inům z první verze jednu klíčovou výhodu: mohou být napsány v jazyce PowerShell (tedy jako prosté skripty) a nemusejí být kompilovány z programovacího jazyka C#. Od této chvíle se tedy staly prostředkem nejen pro programátory, ale i pro „skriptaře" a správce. Modulem se zkrátka může stát kus našeho kódu v PowerShellu, jenž se osvědčil a může být volán z jiných skriptů pomocí odkazu. Při jejich přesunu, kopírování a distribuci není třeba žádné registrace, a proto stačí je prostě přenést jako soubor do cílového umístění.

Moduly mají přisouzeno určité výchozí chování, díky němuž je poměrně snadné je začít používat. Základní logiku si ukážeme na následujícím sledu příkladů. Řekněme, že často potřebujeme spojovat různé řetězce a chybí nám na to příkaz v PowerShellu. Vyrobíme si proto modul, jenž nám bude pro takovouto práci vždy dostupný. Řekli jsme si, že modul je samostatný soubor s potřebným kódem (skriptem), a proto začněme jeho umístěním. PowerShell má uloženy cesty k modulům, jež bude hledat, v následující proměnné prostředí:

```
$ENV:PSModulePath

=>$env:PSModulePath
C:\Documents and Settings\Patrik\Dokumenty\WindowsPowerShell\Modules;
C:\WINDOWS\system32\WindowsPowerShell\v1.0\Modules\
```

Výpis naznačuje, že moduly je možno ukládat na dvě místa, jednak jako uživatelské (do profilu uživatele), jednak jako společné (do sdíleného úložiště pro všechny uživatele na daném stroji). V uvedených adresářích je potřeba pro nový modul vyrobit podadresář se jménem, jež bude odpovídat jménu profilu:

```
=>mkdir C:\WINDOWS\system32\WindowsPowerShell\v1.0\Modules\StringOps

    Directory: C:\WINDOWS\system32\WindowsPowerShell\v1.0\Modules

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----        6. 10. 2009     22:49            StringOps
```

V takto připraveném adresáři pak vyrobíme samotný soubor s našimi „součástkami". Začněme nejjednodušší variantou – funkcí pro spojování řetězců, po níž jsme volali. Založíme tedy soubor *StringOps.psm1*, jehož obsah bude vypadat následovně:

StringOps.psm1

```
function SpojRetezce
{
foreach ($arg in $args)
        {

        $finalstring = $finalstring + $arg

        }
return $finalstring
}
```

Náš nový modul tedy zatím obsahuje jedinou funkci, kterou vzápětí využijeme.

Moduly je potřeba do PowerShellu povolat – o jejich stavu se přesvědčíme takto:

```
=>Get-Module
=>
```

PowerShell prozatím zjevně náš modul nezná, je třeba jej inicializovat pomocí příslušného příkazu:

```
=>Import-Module .\StringOps

=>Get-Module

ModuleType Name                     ExportedCommands
---------- ----                     ----------------
Script     StringOps                SpojRetezce
```

Za povšimnutí stojí vlastnost *ExportedCommands*: tím, že jsme zavedli náš modul a PowerShell jej vzal na vědomí, rovněž jsme automaticky „exportovali“ či „zveřejnili“ funkce, které v něm existují. Co to znamená v praxi? Nic menšího než to, že danou funkci můžeme rovnou volat:

```
=>SpojRetezce Prvni Druhy
PrvniDruhy

=>SpojRetezce Skakal pes pres oves
Skakalpespresoves

=>SpojRetezce "$(pwd)" \NovyPodadresarProModul
C:\WINDOWS\system32\windowspowershell\v1.0\Modules\NovyPodadresarProModul
```

Funkce, kterou jsme nadefinovali v úplně samostatném skriptu, je tedy díky importu modulu okamžitě dostupná a funkční. Pojďme vyzkoušet vylepšení našeho modulu přidáním další funkce, jež bude vypadat následovně (tento kód byl přidán k existujícímu v souboru *StringOps.psm1*):

```
function OpakujRetezec
([string]$retezec,[int]$pocet)
{

$finalstring = $retezec * $pocet

return $finalstring
}
```

Pro jistotu starou verzi modulu „zapomeneme“ a znovu natáhneme do PowerShellu:

```
=>Get-Module .\StringOps | Remove-Module
=>Import-Module .\StringOps
```

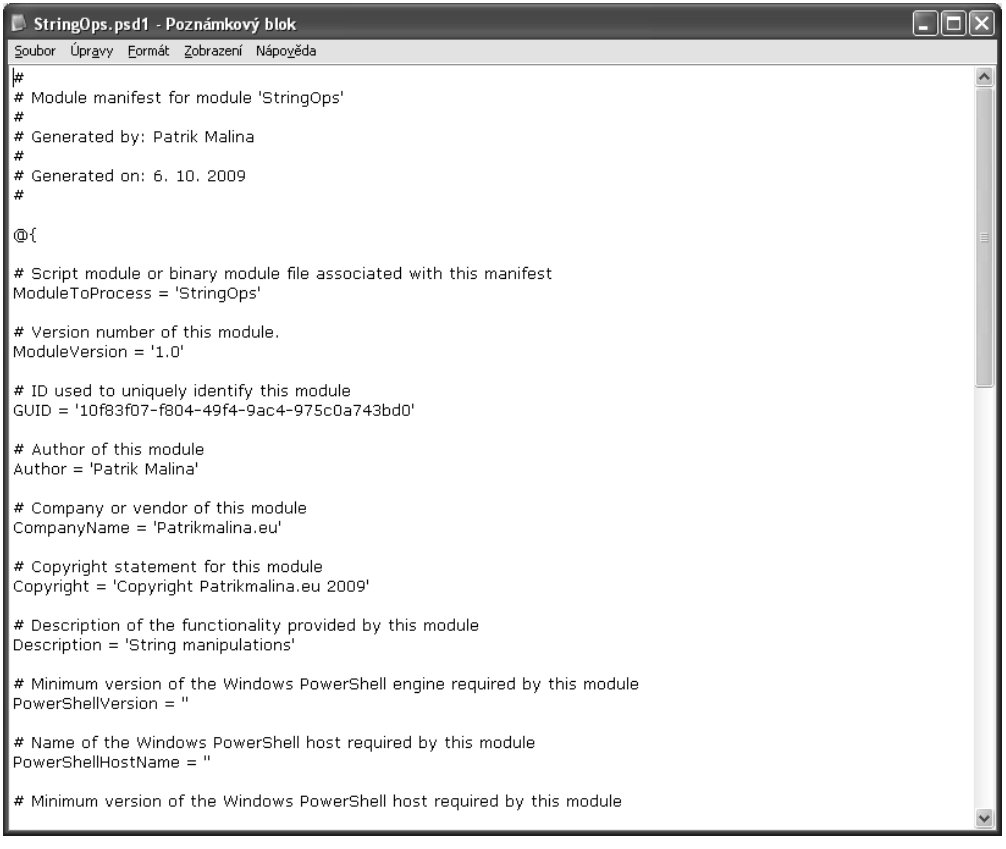
Zkusíme nyní zkontrolovat, jaká je jeho konfigurace:

```
=>Get-Module | fl

Name              : StringOps
Path              : C:\WINDOWS\system32\windowspowershell\v1.0\Modules\StringOps\
                    StringOps.psm1
Description       :
ModuleType        : Script
Version           : 0.0
NestedModules     : {}
ExportedFunctions : {OpakujRetezec, SpojRetezce}
```

```
ExportedCmdlets   : {}
ExportedVariables : {}
ExportedAliases   : {}
```

Je zřejmé, že nyní modul nabízí již obě funkce. Vyzkoušejme tedy náš přírůstek:

```
=>OpakujRetezec -retezec "Ať žije PowerShell! " -pocet 5

Ať žije PowerShell! Ať žije PowerShell! Ať žije PowerShell! Ať žije PowerShell! Ať
žije PowerShell!

=>OpakujRetezec "Nech žije PowerShell! " Necislo

OpakujRetezec : Cannot process argument transformation on parameter 'pocet'. Cannot
convert value "
Necislo" to type "System.Int32". Error: "Vstupní řetězec nemá správný formát."
At line:1 char:14
+ OpakujRetezec <<<<  "Nech žije PowerShell! " Necislo
    + CategoryInfo          : InvalidData: (:) [OpakujRetezec], ParameterBindin...
mationException
    + FullyQualifiedErrorId : ParameterArgumentTransformationError,OpakujRetezec
```



Obrázek 3.23: Modul v PowerShellu může být doprovázen rozsáhlými popisnými daty, která jsou zapsána v takovémto doprovodném souboru

První příklad ukazuje, že nová funkce pracuje zhruba tak, jak jsme chtěli, a to včetně možnosti zadávat parametry jejich jménem. Druhá ukázka pak naznačuje očekávaný výsledek v případě, že namísto tvrdě vyžadované celočíselné hodnoty zkusíme zadat řetězec. PowerShell nám dle očekávání oznamuje, že slovo *Necislo* na celé číslo nepřevedl, ač se snažil sebevíce.

Moduly mohou být velmi rozsáhlými strukturami s řadou užitečných součástí, jež budeme postupně upravovat a vyvíjet. Při jejich rozsáhlejším použití můžeme ocenit možnost připojit k modulům popisnou informaci (metadata), která je pak s nimi spojena. Takovémuto popisu se říká *manifest modulu* a můžeme jej založit pomocí speciálního příkazu přímo v PowerShellu. Následující příklad ukazuje interaktivní tvorbu nového manifestu, jenž bude na závěr automaticky uložen do souboru s příponou *.psd1* v adresáři s modulem:

StringOps.psd1

```
=>New-ModuleManifest

cmdlet New-ModuleManifest at command pipeline position 1
Supply values for the following parameters:

Path: C:\WINDOWS\system32\windowspowershell\v1.0\Modules\StringOps\StringOps.psd1
NestedModules[0]:
Author: Patrik Malina
CompanyName: Patrikmalina.eu
Copyright: Copyright Patrikmalina.eu 2009
ModuleToProcess: StringOps
Description: String manipulations
TypesToProcess[0]:
FormatsToProcess[0]:
RequiredAssemblies[0]:
FileList[0]: C:\WINDOWS\system32\windowspowershell\v1.0\Modules\StringOps\Strin-
gOps.psm1
FileList[1]:

=>
```

PowerShell se doptá na všechna potřebná data a poté založí soubor, který vypadá zhruba jako na obrázku 3.23.

Pokud nyní modul znovu zavedeme, můžeme se přesvědčit, že data z manifestu jsou přenesena do popisného objektu:

```
=>Import-Module .\StringOps

=>Get-Module | fl *

ExportedCommands    : {SpojRetezce, OpakujRetezec}
Name                : StringOps
Path                : C:\WINDOWS\system32\windowspowershell\v1.0\Modules\StringOps\
                      StringOps.psm1
Description         : String manipulations
```