*Enhance Your Productivity and Enable
Rapid Application Development*
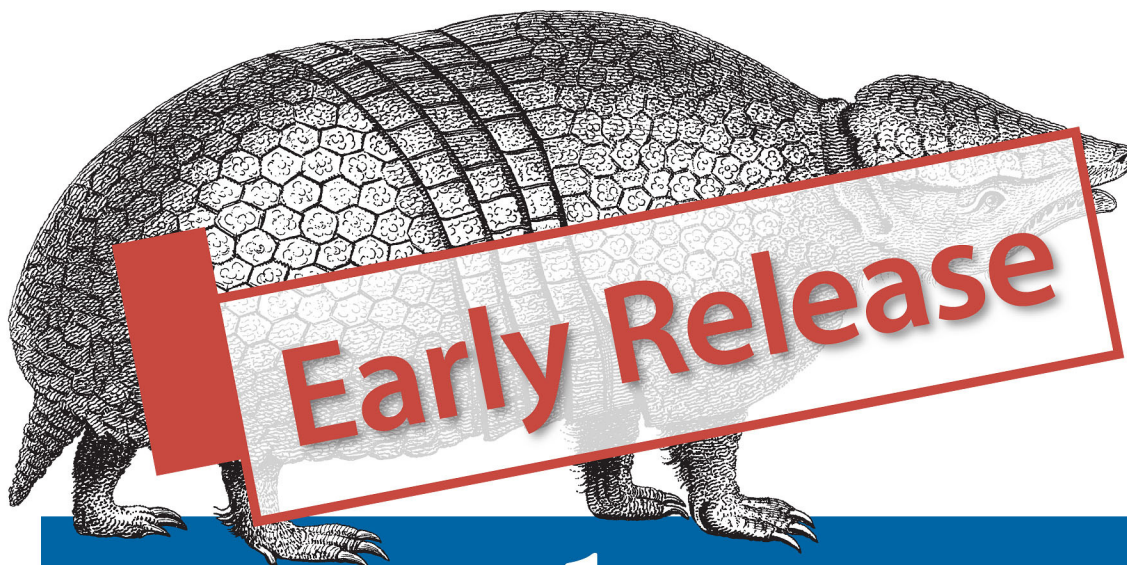
Early Release

Windows PowerShell

for Developers

Douglas Finke

2

The Dime Tour

"Scripting and system programming are symbiotic. Used together, they produce programming environments of exceptional power." - John Ousterhout, creator of Tcl

PowerShell provides rapid turnaround during development for a number of reasons. It eliminates compile time, it's an interpreter and makes development more flexible by allowing programming during application runtime, and it sits on top of powerful components, the .NET framework, connecting them together.

If you want to write PowerShell scripts you need to learn the PowerShell syntax and its building blocks, like Cmdlets, Functions and how to tap into PowerShell's ecosystem, including the .Net framework, third party DLLs and DLLs you create.

There's a lot to cover, even in the dime tour, so here goes.

The Object Pipeline

These 63 characters are what hooked me when I saw my first PowerShell demo.

The Game Changer

```
Get-Process | Where {$_.Handles -gt 750} | Sort PM -Descending

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    965      43   173992     107044   602   157.34   2460 MetroTwit
    784      21    88196      83588   290    19.84   5776 chrome
    952      44    39456      20100   287    29.27   2612 explorer
    784      34    34268       2836   109     4.56   3712 SearchIndexer
   1158      28    18868      14048   150     6.21    956 svchost
    779      14     3784       3900    36     4.46    580 lsass
```

They convey key concepts in PowerShell's value proposition, maximizing effort and reducing time. Here are the highlights.

- Using cmdlets to compose solutions, Get-Process, Where, Sort
- Piping .NET objects, not just text
- Eliminating parsing and praying. No need to count spaces, tabs and other whitespace to pull out the Handles value and then converting it to numeric for the comparison
- Working with .NET properties directly, `$_.Handles` in the `Where` and `PM` in the `Sort`
- Less brittle. If someone adds properties to the output of Get-Process, my code is not affected. I am working with an object-oriented pipeline.

Automation References

When you create a Console Application Project in Visual Studio, the wizard adds these using statements for you:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

In a PowerShell session, created by launching the console or ISE (Integrated Scripting Environment). PowerShell does more work for you. By default, there is a lot available to you in a single PowerShell session. I'll cover how to import DLLs that are not included later using the `Add-Type` Cmdlet or the .Net framework directly using `[Reflection.Assembly]::Load*`.

When you load up ISE, you'll have access to more DLLs because ISE is a WPF application and namespaces like the PresenationCore, PresentationFramework and WndowsBase. This is a PowerShell snippet I used to print out what DLLs are reference.

```
[System.AppDomain]::CurrentDomain.GetAssemblies() |
    Where {$_.location} |
    ForEach { Split-Path -Leaf $_.location } |
    Sort
```

Results

```
Microsoft.CSharp.dll
Microsoft.Management.Infrastructure.dll
Microsoft.PowerShell.Commands.Management.dll
Microsoft.PowerShell.Commands.Utility.dll
Microsoft.PowerShell.ConsoleHost.dll
Microsoft.PowerShell.Security.dll
mscorlib.dll
System.Configuration.Install.dll
System.Core.dll
System.Data.dll
System.DirectoryServices.dll
System.dll
System.Dynamic.dll
System.Management.Automation.dll
System.Management.dll
System.Numerics.dll
System.Transactions.dll
System.Xml.dll
```

PowerShell does this automatically for you so you are ready to go when you launch the console or editor.

Semicolons

They are optional. I don't use them in my scripts, too much noise and typing. They are perfectly legal though and coming from C#, it is hard to lose that muscle memory of adding semicolons.

```
PS C:\> $s = "Hello";
PS C:\> $s += " World"; $s += "!";
PS C:\> $s;
Hello World!
```

I do use them on the command line when I have multiple statements.

```
PS C:\> clear; dir *.cs
```

The good news is if you copy paste C# code, tweak it and forget the semicolon, the PowerShell code will still run.

Use them if you like, I prefer less typing and I go without.

Return Statements

They are optional too. I briefly ran a PowerShell Script Club in New York City. James Brundage, of Start-Automating, created the idea of the script club while he was on the PowerShell team and ramping up other groups in Microsoft. One of the Scripting Club rules is, write only the script you need, no more.

While this is correct.

```
function SayHello ($p) {
    return "Hello $p"
}

SayHello World
```

This is preferred.

```
function SayHello ($p) {
    "Hello $p"
}

SayHello World
```

There will be plenty of times when you do return in a function because it short circuits the execution of the script at that point.

Remember, when using a dynamic language like PowerShell it is *ceremony vs. essence*. Prefer essence.

Datatypes

Optional too. In the following example, `$a = "Hello"` is the same as `var a = "Hello";` in C#. Each environment recognizes the variable as a String.

```
$a = "Hello"
```

3

```
$a # Prints Hello
$a = 1
$a += 1
$a # Prints 2
$a = 1,2,3,"a" # Create an array of different types

[int]$a = "Hello" # Error: Cannot convert value "Hello" to type "System.Int32".
```

PowerShell reduces your typing by not requiring you to explicitly define the type of variables, essence v. ceremony. This is a significant time saver and great when you are trying to plow through some quick prototypes on your own. When you need to take it to a more formal level, for example, sharing you script with someone else or putting your script into production. Typing of your variables is at your fingertips. Passing a string to either parameter causes throws an error, which can be caught.

```
function Do-PrecisionCalculation {
    param (
        [Double]$Latitude,
        [Double]$Longitude
    )

    [Math]::Sin($Latitude) * [Math]::Sin($Longitude)
}
```

Exception handling

PowerShell supports try/catch/finally, that should feel familiar to .Net developers. PowerShell Version 1 introduced the trap statement that still works; I prefer try/catch/finally.

Trap

Break

```
trap {"trapped: $($error[0])"; break}
1/0
"done"
```

Results

```
trapped: Attempted to divide by zero.
Attempted to divide by zero.
At line:3 char:1
+ 1/0
+ ~~~
    + CategoryInfo          : NotSpecified: (:) [],
ParentContainsErrorRecordException
    + FullyQualifiedErrorId : RuntimeException
```

Continue

```
trap {"trapped: $($error[0])"; continue}
1/0
"done"
```

Results

```
trapped: Attempted to divide by zero.
done
```

Try/Catch/Finally

```
try {
    1/0
    "Hello World"
} catch {
    "Error caught: $($error[0])"
} finally {
    "Finally, Hello World"
}
```

Results

```
Error caught: Attempted to divide by zero.
Finally, Hello World
```

Quoting Rules

One key item I want to dial in on here is that the back tick ` is the escape not the backslash \. Note: The backslash is still the escape character for Regular Expressions and PowerShell does support .NET RegEx's.

```
"A string"
A string

"A string with 'Quotes'"
A string with 'Quotes'

"A string with `"Escaped Quotes`""
A string with "Escaped Quotes"

$s = "PowerShell"
"A string with a variable: $s"
A string with a variable: PowerShell

"A string with a quoted variable: '$s'"
A string with a quoted variable: 'PowerShell'

'Variables are not replaced inside single quotes: $s'
Variables are not replaced inside single quotes: $s
```

PowerShell Subexpressions in String

> Expandable strings can also include arbitrary expressions by using the subexpression notation. A subexpression is a fragment of PowerShell script code that's replaced by the value resulting from the evaluation of that code. Here are examples of subexpression expansion in strings.
>
> Bruce Payette - Designer of the PowerShell Language

```
$process = (Get-Process)[0]
```

5

```
$process.PM      # Prints 31793152
"$process.PM"     # System.Diagnostics.Process (AddInProcess32).PM
"$($process.PM)" # Prints 31793152
```

Your mileage will vary; the PM property will have a different value on your system. The key here is, if you do not wrap $process.PM in a subexpression $(…)   you'll get a result you'd never expect.

Here-Strings

Here-Strings are a way to specify blocks of string literals. It preserves the line breaks and other whitespace, including indentation, in the text. It also allows variable substitution and command substitution inside the string. Here-Strings also follow the quoting rules already outlined

Great Code Generation Techniques

This is a block of string literals, in it; I show how single and double quotes can be printed. Plus, I embed a variable $name that gets expanded. Note: I set $name outside of the Here-String to World.

```
$name = "World"

$HereString = @"
This is a here-string
It can contain multiple lines
"Quotes don't need to be escaped"
Plus, you can include variables 'Hello $name'
"@
```

Here-String Output

```
This is a here-string
It can contain multiple lines
"Quotes don't need to be escaped"
Plus, you can include variables 'Hello World'
```

C# Code

Here-String make code generation easy and more readable. I can copy a snippet of C#, drop it into the Here-String, drop in some variables for substitution and I'm off to the races.

```
$methodName = "Test"
$code = @"
public void $methodName()
{
    System.Console.WriteLine("This is from the $methodName method.");
}
"@

$code
```

Results

```
public void Test()
{
    System.Console.WriteLine("This is from the Test method.");
```

6

```
}
```

Script Blocks, Closures and Lambdas

A closure (also lexical closure, function closure, function value or functional value) is a function together with a referencing environment for the non-local variables of that function. A closure allows a function to access variables outside its typical scope. The `&` is the function call operator.

```
$n = "PowerShell"
$closure = {"Hello $n"}
& $closure
Hello PowerShell
```

A scriptblock can have a name, this is called a function.

```
function Add5 ($num) {
    $num + 5
}

Add5 5
10
```

Or it can be a function without a name.

```
$add5 = {param($num) $num + 5}
& $add5 5 # The call operator works with parameters too!
10
```

Scriptblocks, Dynamic Languages and Design Patterns

This example demonstrates one way to apply the strategy design pattern. Because PowerShell is a dynamic language, far less structure is needed to get the job done. I want to employ two strategies, both are doing multiplication. One uses the multiplication operator while the other does multiple additions. I could have named each scriptblock, thereby creating a function, `function  CalcByMult($n,$m) {}` and function `CalcByManyAdds($n,$m) {}`

```
# $sampleData is a multi-dimensional array
$sampleData = @(
    ,(3,4,12)
    ,(5,-5,-25)
)

# $strategies is an array of scriptblocks
$strategies =
{param($n,$m) $n*$m},
{
    param($n,$m)
    1..$n | ForEach {$result = 0} { $result += $m } {$result}
}

ForEach($dataset in $sampleData) {
    ForEach($strategy in $strategies) {
        & $strategy $dataset[0] $dataset[1]
    }
}
```

The nested ForEach loops first loops through the sample data and then through each of the strategies. On the first pass, `& $strategy $Dataset[0] $Dataset[1]` expands to and runs `& {param($n,$m) $n*$m} 3 4.` This produces the result `12`. Next time though the inner loop, I'll have the same parameters but the strategy will change to calculating the result doing multiple adds.

Arrays

A PowerShell array is your .NET System.Array. Plus, PowerShell makes interacting with them simpler. You can still work with arrays in the traditional way through subscripts, but there is more.

It is dead simple to create arrays in PowerShell, separate the items with commas and if they text, wrap them in quotes.

```
$animals = "cat", "dog", "bat"
$animals.GetType()

IsPublic IsSerial Name     BaseType
-------- -------- ----     --------
True     True     Object[] System.Array

$animals

cat
dog
bat
```

Creating an Empty Array

As simple as it is to create an array with items it is equally simple to create an empty array using `@()`. This is a special form of subexpression.

```
$animals = @()
$animals.Count
0
```

Adding an Array Item

I can easily add elements to an array using `+=` operator.

```
$animals = "cat", "dog", "bat"
$animals += "bird"
$animals

cat
dog
bat
bird
```

Retrieving an Element

Accessing a specific array element can be done in a familiar way using subscripts.

```
$animals = "cat", "dog", "bat"
$animals[1]
dog
```

Array Slicing

Array slicing is an operation that extracts certain elements form an array and returns them as an array. I can print out the first two elements using the PowerShell Range notation, 0..1 or I can print out the last element of the array using -1.

```
$animals = "cat", "dog", "bat"

$animals[0..1]
cat
dog

$animals[-1] # Get the last element
bat
```

Finding Array Elements

You can use PowerShell comparison operators with arrays too. Here I am searching the array for elements –ne (not equal) to cat.

```
$animals = "cat", "dog", "bat"
$animals –ne 'cat'
dog
bat
```

I use the –like operator and get wild cards.

```
$animals = "cat", "dog", "bat"
$animals –like '*a*'
cat
bat
```

Reversing an Array

Using the static method Reverse from the Array class, I can invert the elements and then print them. Another example of the seamlessness of PowerShell and the .NET framework.

```
$animals = "cat", "dog", "bat"
[array]::Reverse($animals)
$animals

# Prints
bat
dog
cat
```

Parenthesis and Commas

Coming from C# to PowerShell, parenthesis requires a little extra cognitive effort. They show up where you expect them, around and between parameters to a function.

```
function Test ($p, $x) {
    "This is the value of p $p and the value of x $x"
}
```

If you use them when you call the function Test, you get unexpected results.

```
Test (1, 2)
This is the value of p 1 2 and the value of x
```

The previous example gave results we didn't want. Here is how you do it to get the results you'd expect.

```
Test 1 2
This is the value of p 1 and the value of x 2
```

Calling Test with (1, 2) actually passes the number 1 and 2 as an array to the parameter $p and then PowerShell unrolls that it the string is printed.

This takes practice but don't worry, it is absolutely worth the investment.

Hash tables

A hash table or hash map is a data structure that lets you map keys (e.g., a person's name), to their associated values (e.g., their telephone number). A hash table implements an associative array.

Creating an Empty Hash Table

The @{} creates an empty hash table, similar to the @() used to create the empty array.

```
$h = @{}
$h.Count
0
$h.GetType()
IsPublic IsSerial Name       BaseType
-------- -------- ----       --------
True     True     Hashtable  System.Object
```

Adding a Hash Table Item

Once I have an empty hash table I can map keys and values to them. With PowerShell, I can use either the traditional approach or dot notation.

```
$h = @{}
$h["Item0"] = 0 # More ceremony
$h.Item1 = 1 # Notice, dot notation
$h.Item2 = 2
$h # Prints the Hash table
Name  Value
----  -----
Item1 1
Item0 0
Item2 2
```

Initializing a Hash Table with Items

Here I create a hash table and two keys with values. Then, using dot notation I print out the value of the key named Item1.

```
$h = @{Item1=1;Item2=2}
$h.Item1 # dot notation and no casting
1
```

Concatenating Hash Tables

The addition operator also works on hash tables. Strings and Arrays will work with addition operator also.

```
$h1 = @{a=1;b=2}
$h2 = @{c=3;d=4}

$h1+$h2

# Prints
Name Value
---- -----
d    4
c    3
b    2
a    1
```

Get-Member CmdLet

Gets the members (properties and methods) of objects, there I ran `Get-Help Get-Member` for you. This is reflection at the command line. As a developer it is one of the key cmdlets I use regularly. I get all of the right information about an object right there; it's type, plus all the methods, properties, events and more. When working with a script, this is very handy I can just add an `$obj | Get-Member` in the script and inspect all these details about an object I am working with.

```
1.0 | Get-Member


   TypeName: System.Double

Name          MemberType Definition
----          ---------- ----------
CompareTo     Method     int CompareTo(System.Object value)
Equals        Method     bool Equals(System.Object obj), bo
GetHashCode   Method     int GetHashCode()
GetType       Method     type GetType()
GetTypeCode   Method     System.TypeCode GetTypeCode()
ToBoolean     Method     bool ToBoolean(System.IFormatProvi
ToByte        Method     byte ToByte(System.IFormatProvider
ToChar        Method     char ToChar(System.IFormatProvider
ToDateTime    Method     System.DateTime ToDateTime(System.
ToDecimal     Method     decimal ToDecimal(System.IFormatPr
ToDouble      Method     double ToDouble(System.IFormatProv
ToInt16       Method     System.Int16 ToInt16(System.IForma
ToInt32       Method     int ToInt32(System.IFormatProvider
ToInt64       Method     long ToInt64(System.IFormatProvide
ToSByte       Method     System.SByte ToSByte(System.IForma
ToSingle      Method     float ToSingle(System.IFormatProvi
ToString      Method     string ToString(), string ToString
ToType        Method     System.Object ToType(type conversi
ToUInt16      Method     System.UInt16 ToUInt16(System.IFor
ToUInt32      Method     System.UInt32 ToUInt32(System.IFor
ToUInt64      Method     System.UInt64 ToUInt64(System.IFor
```

Filtering with Get-Member

Notice it tells you the type there at the top. I used a double as an example, if I used a reference type, I would see properties, events and more. With `Get-Member` you can filter on `MemberType` too.

```
New-Object Net.Webclient | Get-Member -MemberType Property


    TypeName: System.Net.WebClient


Name                   MemberType Definition
----                   ---------- ----------
BaseAddress            Property   System.String BaseAddress {get;set;}
CachePolicy            Property   System.Net.Cache.RequestCachePolicy
Container              Property   System.ComponentModel.IContainer
Credentials            Property   System.Net.ICredentials Credentials
Encoding               Property   System.Text.Encoding Encoding {get;set;}
Headers                Property   System.Net.WebHeaderCollection Headers
IsBusy                 Property   System.Boolean IsBusy {get;}
Proxy                  Property   System.Net.IWebProxy Proxy {get;set;}
QueryString            Property   System.Collections.Specialized.NameVal
ResponseHeaders        Property   System.Net.WebHeaderCollection ResponseHea
Site                   Property   System.ComponentModel.ISite Site {get;set;}
UseDefaultCredentials  Property   System.Boolean UseDefaultCredentials {get;se
```

Using Get-Member with Collections

Here is some PowerShell magic that is useful and sometimes not what you want.

```
$range = 1..10
$range | Get-Member
```

Piping the `$range` to Get-Member, PowerShell prints out the details about the different elements in the array, not the collection itself. In this case, since I used the range operator 1..10 all the elements are int32, so I get the details about the type Int32.

```
    TypeName: System.Int32


Name        MemberType Definition
----        ---------- ----------
ToBoolean   Method     bool ToBoolean(System.IFormatProvider provid
ToByte      Method     byte ToByte(System.IFormatProvider provider)
ToChar      Method     char ToChar(System.IFormatProvider provider)
ToDateTime  Method     System.DateTime ToDateTime(System.IFormatPro
ToDecimal   Method     decimal ToDecimal(System.IFormatProvider pro
ToDouble    Method     double ToDouble(System.IFormatProvider provi
```

If the `$range` were heterogeneous, `Get-Member` would return the details for each type. To be more accurate, the PowerShell Object Flow Engine would do that. I won't be discussing the flow engine though.

What if I wanted to get the details on `$range` though? Simple, use the `-InputObject` on the `Get-Member` cmdlet.

```
$range = 1..10
Get-Member -InputObject $range
```

Here is an edited version of what is returned about the collection `$range`.

```
   TypeName: System.Object[]

Name              MemberType            Definition
----              ----------            ----------
Count             AliasProperty         Count = Length
Add               Method                int Add(System.Object value)
Clear             Method                System.Void Clear()
GetEnumerator     Method                System.Collections.IEnumerato
GetLowerBound     Method                int GetLowerBound(int dimensi
IndexOf           Method                int IndexOf(System.Object val
Initialize        Method                System.Void Initialize()
Insert            Method                System.Void Insert(int index,
IsReadOnly        Property              bool IsReadOnly {get;}
IsSynchronized    Property              bool IsSynchronized {get;}
Length            Property              int Length {get;}
```

Looking into PowerShell cmdlets deeper you'll often find options where piping or passing parameters, while a different mindset, yield the results that you want. This speaks to the cognitive shift of PowerShell and is worth the time you invest.

Inject a GUI into the PowerShell Command Line

Let's say I get too much output at the command line from `Get-Member`. No problem, let's pipe to a GUI using `Out-GridView`. `Out-GridView` comes with PowerShell, ready to go out of the box see Figure 2-1.

```
New-Object Net.Webclient | Get-Member | Out-GridView
```
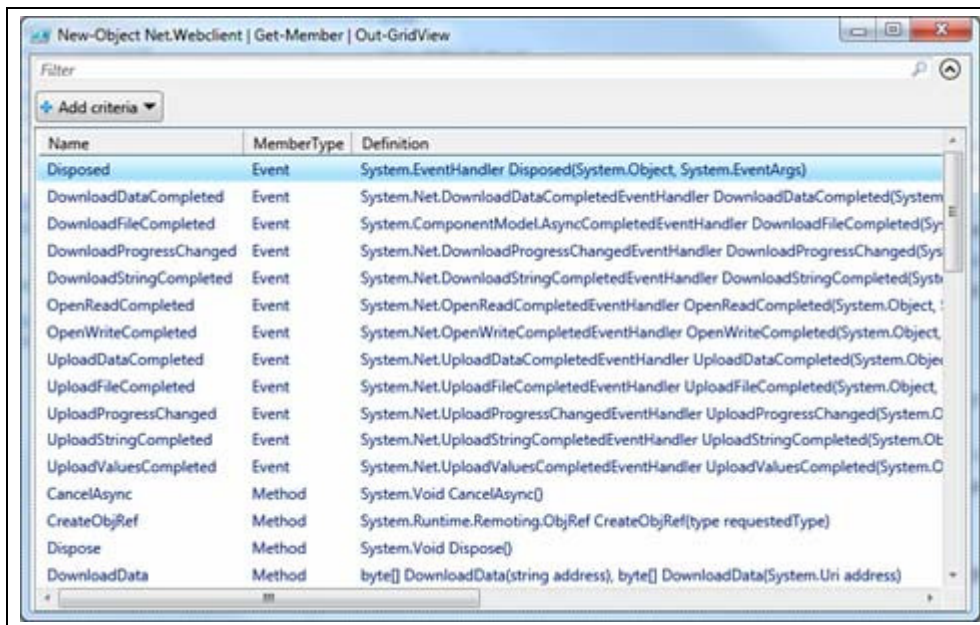

Figure 2-1. Injecting a GUI

I recommend playing with `Out-GridView`. It has a Filter, which subsets the list as you type. In version 3 it has a `-PassThru` parameter that lets you select items and they get passed down the pipeline when you click Ok.

`Out-GridView` saves time and effort when debugging. In a multi-line script, I can add a line where I pipe a variable containing an array of objects to it, run the script and this window pops up. Great way to inspect what happened.

New-Object CmdLet

`New-Object` creates an instance of a Microsoft .NET Framework object. I'll "new" up a COM object, Internet Explorer, and then I'll new up a native PowerShell object, PSObject, and add properties to it. I'll show the streamlined PowerShell V3 syntax. Finally I'll work with a .NET framework object

Launching Internet Explorer

Here in 3 lines of PowerShell I can create a COM object, call a method on it and set a property. I don't know how many lines are needed to get this done in C#. Remember the ProgID? This is how we used to interact with COM objects; Here I am using the ProgID InternetExplorer.Application, then I'm navigating to the Google page and the making IE visible. If you've got a ProgID, PowerShell can make short work of it.

```
$ie = New-Object -ComObject InternetExplorer.Application
$ie.Navigate2("http://www.google.com")
$ie.Visible = $true
```

Creating a New PowerShell Object

PSObject is the PowerShell Object. It is the root of the synthetic type system in PowerShell. So here I am creating a new one and adding two properties to it Name and Age. Plus, I am setting values to them.

```
$obj = New-Object PSObject -Property @{
    Name = "John"
    Age = 10
}

$obj.GetType()
IsPublic IsSerial Name            BaseType
-------- -------- ----            --------
True     False    PSCustomObject System.Object

$obj
Age Name
--- ----
 10 John
```

PowerShell V3 is More Pithy

Version 3 of PowerShell comes with a ton of new things. Here, I am getting the same results as the previous example, but in less typing. Less typing, more results is what PowerShell is all about.

```
[PSCustomObject] @{
    Name = "John"
    Age = 10
}

Name Age
---- ---
```

```
John   10
```

Using the .Net Framework

I can also instantiate .NET framework components. This is a primary use case as a .NET developer. I use this to instantiate the components I write and deliver as DLLs.

```
$wc = New-Object Net.WebClient
[xml]$resp = $wc.DownloadString("http://feeds.feedburner.com/DevelopmentInABlink")
$resp.rss.channel.item| ForEach {$_.Title}

NumPy 1.5 Beginnerâ€™s Guide
Design Patterns in Dynamic Languagesâ€"PowerShell
Analyze your C# Source files using PowerShell and the Get-RoslynInfo Cmdlet
Using PowerShell in Roslynâ€™s C# Interactive Window
PowerShell â€" Handling CSV and JSON
My Philly .Net PowerShell Code Camp Presentation
PowerShell for .Net Developersâ€"A Survey
My New York City PowerShell Code Camp Presentation
PowerShell vNext â€" Web Service Entities
Reading RSS Feedsâ€"Even easier in PowerShell V3
```

Add-Member

Here I used `Add-Member` to extend the .Net string object and added `Reverse`, which reverses the letters in the string. I created a new `ScriptProperty` (`Add-Member` can add other types like `NoteProperty`) and in the scriptblock, I referenced object and its properties using the `$this` variable.

```
$s = "Hello World" |
    Add-Member -PassThru ScriptProperty Reverse {$this[$this.Length..0] -join ""}

$s
Hello World

$s.Reverse
dlroW olleH
```

Add-Type

Adds a Microsoft .NET Framework type (a class) to a Windows PowerShell session. `Add-Type` has a few parameters; `TypeDefinition` lets me compile C# code on the fly. It also supports VB.NET. I'll also show the `Path` parameter too, that lets me load a DLL into a PowerShell session.

Compiling C# On The Fly

You should recognize the text Inside the `Here-String`, the stuff between the `@""@`. That is a C# `MyMathClass` class with a single method `Add`. I am passing the Here-String to the –TypeDefinition parameter and the Add-Type cmdlet with compile it on the fly, in memory, into the current PowerShell session. If I am running a script, it is compiles the code just for the life of that script.

```
Add-Type -TypeDefinition @"
public class MyMathClass {
    public int Add(int n1, int n2) {
```

```
        return n1 + n2;
    }
}
"@
```

Newing Up The Class

After compiling it, I want to use it so I use the New-Object. This is equivalent to `var obj = new MyMathClass();`. From there I print out the objects type and then use Get-Member to get the details of the object I am working with.

```
$obj = New-Object MyMathClass
$obj.GetType()

IsPublic IsSerial Name        BaseType
-------- -------- ----        --------
True     False    MyMathClass System.Object


$obj | Get-Member


   TypeName: MyMathClass

Name        MemberType Definition
----        ---------- ----------
Add         Method     int Add(int n1, int n2)
Equals      Method     bool Equals(System.Object obj)
GetHashCode Method     int GetHashCode()
GetType     Method     type GetType()
ToString    Method     string ToString()
```

Calling the Add Method On MyMathClass

Let's exercise the newly minted code by adding the numbers 1 to 5 to themselves and printing them out. It's important to note here, I am not telling PowerShell how to print or loop. I don't check for the end of the stream, or end of file. It just works.

```
1..5 | ForEach {$obj.Add($_,$_)}
2
4
6
8
10
```

Wait, I Don't Have the Source

What if I didn't give you the C# source code? No problem. Use the –Path parameter and let Add-Type do the rest.

```
Add-Type –Path C:\Temp\MyMathClass.dll
```

This is similar to adding a reference to a project and then a using statement. You can apply the previous PowerShell statements for the same results.

I could also have used the .Net framework to get the job done.

```
[Reflection.Assembly]::LoadFile("C:\Temp\MyMathClass.dll")
```

Check out my blog post for more detail How To Load .NET Assemblies In A PowerShell Session

What the %? And other aliases

PowerShell has an aliasing system allowing you to create or change an alias for a cmdlet or for a command element, such as a function, a script, a file, or other executable.

So, `%` is aliased to `ForEach` and `?` aliased to `Where`. These two PowerShell lines are equivalent, finding the even numbers, multiplying them by 2 and printing them.

```
1..10 | ? {$_ % 2 -eq 0} | % {$_*2}
1..10 | Where {$_ % 2 -eq 0} | ForEach {$_*2}
4
8
12
16
20
```

In my PowerShell profile, `$PROFILE`, I alias `vs` to the Visual Studio executable. So whenever I need to launch it, I type vs and press enter.

```
Set-Alias vs
 "C:\Program Files\Microsoft Visual Studio 10.0\Common7\ide\devenv.exe"
```

Modules

PowerShell modules are fundamental to organizing your scripts. You can place your scripts in sub folders and from the module you can recursively find them all and dot source them into a PowerShell session. It's a fantastic way to speed development. You can just drop a script into a directory below your module (has a psm1 extension) do a Import-Module –Force <module name> and you're ready to rock.

Here is a list of Modules on my box. They are probably different than yours because I have PowerShell V3 CTP2 installed on Windows 7.

```
Get-Module -ListAvailable | Select Name
AppLocker
BitsTransfer
CimCmdlets
Microsoft.PowerShell.Core
Microsoft.PowerShell.Diagnostics
Microsoft.PowerShell.Host
Microsoft.PowerShell.Management
Microsoft.PowerShell.Security
Microsoft.PowerShell.Utility
Microsoft.WSMan.Management
PSDiagnostics
PSScheduledJob
PSWorkflow
TroubleshootingPack
```

Modules are your friend. Learn them, love them, and use them. It is how Microsoft teams deliver their PowerShell functionality. Once you grow beyond a few scripts that interact, it is the preferred packaging mechanism.

Let's say I have this script stored in a PowerShell file in my scripts directory, `C:\Scripts\MyCountScript.ps1`.

```
$count = 0
function Add-OneTocount { $count += 1 }
function Get-Count        { $count }
```

I can source this script by putting a dot '.' and the fully qualified script file name. ".
`C:\Scripts\MyCountScript.ps1`". Dot sourcing with load and run the script, variables become global as do functions. This is good news and bad news. The good news is, it lets me rapidly iterate and problem solve. The bad news is, if I deliver this as is to a colleague or client and they have a script they dot source and it uses `$count`, we'll have a collision.

Modules helps with scoping and that is just the beginning of what modules do, remember, this is the dime tour. I will illustrate quickly how to ramp up easily on modules. I can rename my script to `C:\Scripts\MyCountScript.psm1`, note, I only changed `ps1` to `psm1`. Now I need to "load" it and I cannot dot source it so I'll use `Import-Module`.

```
Import-Module C:\Scripts\MyCountScript.psm1
```

That's it! Now `$count` is not visible outside of the module and we are safe.

There's a lot more to modules, again. Learn them, love them, and use them.

Summary

Ok, that's the end of the tour. We did a nice swim across the surface, dipped under for a couple of feet and a bit of a deep dive. PowerShell V2 had a couple hundred cmdlets, PowerShell V3 over four hundred, Windows Server 8 delivers over 2300 cmdlets. That's a lot of good stuff. Plus, it doesn't include PowerShell remoting, background jobs, Windows Workflow, idiomatic aspects, best practices, tips and tricks and much, much more.

I started the chapter with a quote from John Ousterhout, creator of Tcl, "Scripting and system programming are symbiotic. Used together, they produce programming environments of exceptional power."

PowerShell is a distributed automation platform, it is as interactive and composable as BASH/KSH (UNIX Shells), it is as programmatic as Perl/Python/Ruby, it is production ready, and it ships with Windows 7 and soon Windows 8.

It requires your investment. The good news is you can become very productive very quickly, just learning some basics. When you're ready, you develop your PowerShell skills more and you'll benefit by using it to support your development process, deliver more powerful products, make your product more manageable, deliver faster and better functionality and enable System Integrators and End-Users generate custom solutions based on your product.

Want to know how? I invite you to read on.

3

Getting Started

Installing PowerShell

Installing PowerShell is as simple as installing any other application. Even better, it comes installed with Windows 7, Windows Server 2008 R2, Windows 8 and Windows Server 8. PowerShell is also available for previous versions Windows XP, 2003 and Vista.

As I mention, PowerShell v3 comes installed with Windows 8, (as I am writing this there is a CTP2 release for Windows 7. You download PowerShell v3 CTP2 from HERE). New cmdlets and language features are abundant in this more robust version; all designed to make you more productive and lower the barrier of entry to using PowerShell.

If you are running an older Microsoft Windows OS, I encourage you to update that, too, however, no worries though, PowerShell v2 can run on these boxes. You can get that version HERE. Make sure to download the right PowerShell for your OS and architecture.

> While there is no PowerShell version for UNIX, Linux or Mac, Microsoft did license the PowerShell Language under the Community Promise. We'll see if any developers pick up from here and implement PowerShell on non-Windows boxes.

Checking the PowerShell Version

Depending on your Windows OS, you can navigate to PowerShell in many ways First, get to the command prompt and type in:

```
PS C:\> $PSVersionTable

Name                      Value
----                      -----
WSManStackVersion         3.0
PSCompatibleVersions      {1.0, 2.0, 3.0}
```

```
SerializationVersion        1.1.0.1
BuildVersion                6.2.8158.0
PSVersion                   3.0
CLRVersion                  4.0.30319.239
PSRemotingProtocolVersion   2.103
```

This gives you lots of good information about the PowerShell version running on your box. Including what version of .NET you are going against, noted as CLRVersion in PowerShell. I'm running PowerShell v3 CTP3. I can run PowerShell in version 2 mode, if possible you should too.
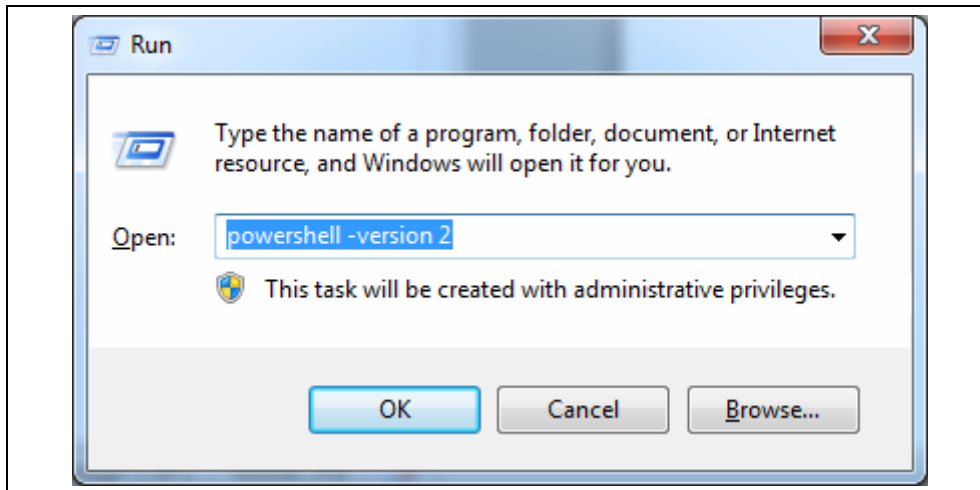
Figure 3-1. Using the –version parameter

Here is what I get when I look at the $PSVersionTable variable. Notice I only have two compatible versions and am using .NET 2.0, CLRVersion. When PowerShell v2 was delivered, only .NET 2.0 was released. PowerShell v3 works with .NET Framework 4.0.

```
PS C:\> $PSVersionTable

Name                        Value
----                        -----
CLRVersion                  2.0.50727.5448
BuildVersion                6.1.7601.17514
PSVersion                   2.0
WSManStackVersion           2.0
PSCompatibleVersions        {1.0, 2.0}
SerializationVersion        1.1.0.1
PSRemotingProtocolVersion   2.1
```

Interactivity, the key to PowerShell

The prompt is up so let's work the PowerShell REPL. A *REPL* is a Read, Eval, Print, Loop. This means that when you type some PowerShell command and press enter, those commands are read, evaluated, results are printed (or errors) and the console loops back and waits to do it again. Let's try it.

```
PS C:\> 2 + 2 * 3
8
```

```
PS C:\>
```

So, PowerShell is just a big calculator? Not exactly. If you try that in a DOS prompt, what happens? You get an error. Notice, the result is printed and we get the prompt back, ready to complete your next request.

Now type in the "Hello World" quoted string. Press Enter and you get back the same thing you typed, without the quotes. PowerShell evaluated that for you, it shows the E in REPL. Also, we didn't have to explicitly specify that we wanted it to be printed. PowerShell just "knew" to do that. These are great time-saving aspects of PowerShell; not to mention, they cut down on keystrokes too.

```
PS C:\> "Hello World"
Hello World
```

Let's tap into the .NET Framework now. Type in:

```
PS C:\> [System.Math]::Pow(2, 3)
8
```

What you've done is input the `System.Math` namespace and called the static method `Pow()`. Get used to the syntax; you'll be using it again and again. Square brackets '[]' around the fully qualified type name and two colons '::' indicate I'm calling the static method. This is the syntax the PowerShell team has decided on.

Let's create a variable, set it to a string and then inspect its type. You may be familiar with the `GetType()` method from C#.

```
PS C:\> $a = "Hello"
PS C:\> $a.GetType()

IsPublic IsSerial Name   BaseType
-------- -------- ----   --------
True     True     String System.Object
```

Set the variable $a to a string, and you'll see that in fact, it is by using the `GetType()` method. This is very handy when running/debugging PowerShell scripts. You can slap a `GetType()` on a variable and find out exactly what type it is. Now, how to run a PowerShell script?

Running a PowerShell Script

Setting the Execution Policy

The execution policy is part of the security strategy of Windows PowerShell. It determines whether you can load configuration files (including your Windows PowerShell profile) and run scripts, and it determines which scripts, if any, must be digitally signed before they will run.

When PowerShell is installed, the execution policy is set to `Restricted`. This means, PowerShell will not load configuration files or run scripts. Even though you are restricted from using scripts, interactive commands can still be run. If you're new to PowerShell, better safe than sorry.

Once you are more comfortable with using PowerShell and scripts written by others, you can ease the restriction.

> Finding PowerShell Information Online from the Command Line
>
> You change the policy by using the `Set-ExecutionPolicy` cmdlet. You can find more information about the `Set-ExecutionPolicy` cmdlet by typing the following.
>
> Get-Help Set-ExecutionPolicy -Online
>
> The cool part, the `-Online` parameter will launch the browser and navigate to the cmdlet web page.

RemoteSigned Is Good for You

There are a few options you can use with the `-ExecutionPolicy` parameter found on the `Set-ExecutionPolicy` cmdlet. Many users set the execution policy to *RemoteSigned* which means that all scripts and configuration files downloaded from the Internet must be signed by a trusted publisher. This "protects" you so if you download a script or get one in an email and you try to run it, PowerShell will prompt and warn you before letting you continue. As you gain experience, you could choose the `UnRestricted` setting. Setting the execution policy to `UnRestricted` comes with risks which means you can launch scripts that could disable or destroy information on your system.

I run in `UnRestricted` mode but have been working with PowerShell for a few years. I'm comfortable with what scripts I am about to run based on knowing the authors and where I have downloaded the scripts from.

```
PS C:\> Set-ExecutionPolicy Unrestricted
```

Here's an example of why `RemoteSigned` is a good idea. Ed Wilson, Microsoft employee and author of PowerShell books produces the content for "Hey, Scripting Guy!" blog and is the driving force behind the Windows PowerShell Scripting Games. Ed invited me to be a judge at the games. I downloaded one of the entries for review and then ran it. The creator of the script had unintentionally added some WMI code that disabled the Ethernet card. I ran the script and then spent the next hour trying to figure out why I couldn't connect to the Internet and how to re-enable the card.

If had the `RemoteSigned` execution policy set, it would have prompted me that I was running a script I had downloaded and I may have chosen not to run it. This is especially handy if you end up with a folder with scripts from mixed sources.

Running Scripts with Execution Policy Set to Restricted

Let's run the test script again with the policy set to Restricted.

```
PS C:\> .\test.ps1
File C:\test.ps1 cannot be loaded because the execution
of scripts is disabled on this system. For more information,
see about_Execution_Policies at
http://go.microsoft.com/fwlink/?LinkID=135170.
At line:1 char:1
```

```
+ .\test.ps1
+ ~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [], PSSecurityException
    + FullyQualifiedErrorId : UnauthorizedAccess
```

You can set the execution policy to one of several settings so you don't get this message and can run the script. You'll need to do a little research to figure out what setting is most appropriate for you. You need to *run the console as administrator* in order to effect the `Set-ExecutionPolicy` changes because it is a setting in the Registry.

Here are all the possible options for the `-ExecutionPolicy` parameter on the `Set-ExcutionPolicy` cmdlet.

Restricted	Does not load configuration files or run scripts. "Restricted" is the default execution policy
AllSigned	Requires that all scripts and configuration files be signed by a trusted publisher, including scripts that you write on the local computer.
RemoteSigned	Requires that all scripts and configuration files downloaded from the Internet be signed by a trusted publisher.
Unrestricted	Loads all configuration files and runs all scripts. If you run an unsigned script that was downloaded from the Internet, you are prompted for permission before it runs.
Bypass	Nothing is blocked and there are no warnings or prompts.
Undefined	Removes the currently assigned execution policy from the current scope. This parameter will not remove an execution policy that is set in a Group Policy scope.

Now we're set to run a script

Let's try a simple script. The script test.ps1 contains the quoted string "Hello World".

```
PS C:\> Get-Content .\test.ps1
"Hello World"


PS C:\> .\test.ps1
Hello World
```

In order to run a PowerShell script, you'll need to place ".\" before the name of the script thereby noting it is in the current directory. You can provide the full path to the script. Scripts can be specified by full path or relative path.

Again, notice that there is no compilation step, you just execute and go. Even though there is no compilation step and PowerShell is a dynamic language, it is based on .NET which proves to be beneficial in many ways.

PowerShell works within the .NET Framework and as such we can perform reflection at the command line using `Get-Member`, (see more on this topic in Chapter 4). As well as use the `GetType()` method to see the underlying .NET type of object you are manipulating. Reflection is the process by which you can observe (do type introspection) and modify its own structure and behavior at runtime. Here we just did some observing.

PowerShell ISE

ISE (pronounced *ice*) is free and is available as soon as you install PowerShell, or are using Microsoft operating systems like Windows 7 or Windows 8 that has PowerShell already installed.

Windows PowerShell Integrated Scripting Environment (ISE) is a graphical host application for Windows PowerShell. Windows PowerShell ISE lets you run commands, and write, edit, run, test, and debug scripts in an environment that displays syntax in colors and that supports Unicode.

Windows PowerShell ISE is designed for users at all levels of proficiency. Beginners will appreciate the syntax colors and the context-sensitive Help. Multiline editing makes it easy to try the examples that you copy from the Help topics and from other sources. Advanced users will appreciate the availability of multiple execution environments, the built-in debugger, and the extensibility of the Windows PowerShell ISE object model.

Other PowerShell Editors

PowerShell does have a few free editors specifically tailored for use with it. There are a number of other editors which support the editing of many different programming languages and typically the PowerShell community has stepped up in delivering extensions for syntax highlighting, build tools and more.

PowerGUI is an extensible graphical administrative console for managing systems based on Windows PowerShell. These include Windows OS (XP, 2003, Vista), Exchange 2007, Operations Manager 2007 and other new systems from Microsoft. The tool allows you to use the rich capabilities of Windows PowerShell in a familiar and intuitive GUI console. PowerGUI can be downloaded here: http://powergui.org/downloads.jspa

PowerShell Analyzer is an integrated development environment that focuses on the leveraging PowerShell as a dynamic language. It's goal is simply to allow users to be as productive as possible in sculpting, running, interpreting results and refactoring everything from the "one-liners" PowerShell is famous for, to fully fledged production quality scripts. PowerShell Analyzer can be downloaded here: http://www.powershellanalyzer.com/

Professional PowerShell Script Editor (PowerSE). PowerSE is an advanced IDE Console, plus it has all the features you come to expect from a professional editor. It supports color syntax highlighting, IntelliSense (PowerShell, WMI, and .NET), tab completion, context sensitive help system and much more. PowerSE can be downloaded here: http://powerwf.com/products/powerse.aspx

PrimalScript: The Integrated Scripting Environment for PowerShell. It doesn't matter what your niche is - system, database or network administrator, web developer or end-user developer; you probably need to work with multiple technologies, languages and file formats at the same time. Take charge of your script development regardless of what language you use and combine PrimalScript's powerful editing and packaging abilities with your scripting skills. PrimalScript can be downloaded here: http://www.sapien.com/software/primalscript

PowerShell Plus: Learn PowerShell fast using the Interactive Learning Center. Run PowerShell commands with the powerful interactive console. Debug PowerShell 10X

faster with the advanced script editor. Execute scripts remotely using customized configurations. Access hundreds of pre-loaded scripts in the QuickClick library. Search and download thousands of community scripts. Enable team collaboration using Source Control integration. PowerShell Plus can be downloaded here: http://www.idera.com/PowerShell/powershell-plus/

There are other editors out there that have powerful capabilities and are highly customizable to your needs.

Vim: stands for 'Vi Improved' (vi is a "Visual Editor"). Vim is an advanced text editor that seeks to provide the power of the de-facto Unix editor 'Vi', with a more complete feature set. Vim can be downloaded here: http://www.vim.org/index.php. Plus, default syntax coloring for Windows PowerShell can be downloaded here: http://www.vim.org/scripts/script.php?script_id=1327

Notepad++: is a free (as in "free speech" and also as in "free beer") source code editor and Notepad replacement that supports several languages. Notepad++ can be downloaded here: http://notepad-plus-plus.org/

This is a sampling of what is available to use to editing of PowerShell scripts, running and debugging. Each has options out of the box and different levels of customizability.

Experiment and enjoy!

PowerShell and Visual Studio

Visual Studio is used to develop console and graphical user interface applications along with Windows Forms applications, web sites, web applications, and web services in both native code together with managed code for all platforms supported by Microsoft Windows, Windows Mobile, Windows CE and .NET Framework.

Since you can embed PowerShell in a C# application, see Chapter 5 "Add a PowerShell Command Line to Your GUI", both Microsoft and PowerShell MVPs have written PowerShell consoles that work directly in and with Visual Studio.

NuGet is a free, open source developer-focused package management system for the .NET platform intent on simplifying the process of incorporating third-party libraries into a .NET application during development. NuGet also comes with a PowerShell console that runs inside Visual Studio. NuGet can be downloaded here: http://nuget.codeplex.com/

StudioShell is written and maintained by Jim Christopher, PowerShell MVP, here on CodePlex. If you've ever implemented a Visual Studio extension, such as an add-in or a package, you know how convoluted this space has become. You have to become an expert in your tooling if you want to change it. StudioShell changes this landscape by exposing many of Visual Studio's extensibility points in a simple and consistent way. It makes the Visual Studio IDE interactive and discoverable.

The PowerShell Community

PowerShell has a thriving community; there are open source projects and script repositories, forums and even a PowerShell Magazine. If you are thinking of where to get started, have a question or wondering if someone else has already created it. These are

the places to check. Plus, you can get a look at some advanced usages of PowerShell and contribute solutions based on these that are already there.

Many of us are on the forums, answering questions on StackOverflow (Search [powershell])and involved on Twitter (#powershell).

CodePlex- Search CodePlex for PowerShell, and you will find that there are over 450 open source projects and it is growing. Everything from tools that bring features from the UNIX world to Azure management cmdlets, testing frameworks, SQL Server integration scripts, Facebook and Twitter integration and so much more.

PoShCode.org - The PowerShell Code Repository is maintained by another PowerShell MVP, Joel "Jaykul" Bennett.

PowershellCommunity.org is a community-run and vendor-sponsored organization that provides evangelism for all things PowerShell through news, forums, user group outreach, script repository, and other resources.

PowerShell Magazine - I am a co-founder and editor of the PowerShell Magazine, along with four other great guys and PowerShell MVPs, Ravikanth Chaganti, Aleksandar Nikolić, Shay Levy, and Steven Murawski.

Check out the site, submit an article and just enjoy the targeted PowerShell content from some of the best scripters in the community.

The Future of PowerShell on Window 8

PowerShell was released as a separate download more than six years ago. Jeffrey Snover, a creator of PowerShell, wrote the Monad Manifesto in 2002 (Monad was the code name for PowerShell). Then, PowerShell debuted as part of the Windows 7 operating system in 2009. A few hundred million copies of Windows 7 have been licensed which means, there are a few hundred million copies of PowerShell out there installed and ready to go.

This year, 2012, Windows 8 will be delivered and with it PowerShell v3. In addition, Windows Server 8 will also be released. PowerShell v3 has numerous enhancements across the entire product, shipping with hundreds more PowerShell cmdlets for the client, and in the Windows Server 8 case over 2000 cmdlets.

Plus, Microsoft is not the only company delivering PowerShell-enabled software. VMWare, Cisco, Intel, Citrix and SPLUNK are now doing this-just to name a few.

Summary

We've barely covered the basics here. There is an entire ocean of PowerShell waiting and that doesn't include third-party PowerShell systems, community delivered scripts or the internal Microsoft teams outfitting their products.

You can say PowerShell is about 10 years old, maybe older, since its inception stemming from the Monad Manifesto. The team that developed PowerShell drew inspiration from systems developed over 30+ years ago in DEC and IBM. And PowerShell is as programmable as Perl, Python and Ruby and takes it cues from UNIX shells.

The community is thriving, which is a fundamental component to any new language and approach. Microsoft has over 50 PowerShell MVPs worldwide, providing feedback to

the Microsoft PowerShell team as well as the other teams who are developing cmdlets and surfacing their APIs for easy consumption in PowerShell.

PowerShell is a distributed automation platform and is surfaced as a command line, scripting language and API. Think, embedding PowerShell in your C# app and check out the chapter on "Add a PowerShell Command Line to Your GUI".

Jeffrey Snover says, "If you're planning on working with Microsoft systems for the next year, invest some time with PowerShell, it'll make your life simpler."

4

Accelerating Delivery

In this chapter I'm going to work through different types of text extraction and manipulation. This can play into creating code generators which take over the task of writing repetitive infrastructure code, eliminating grunt work. PowerShell's ability to work in this way, reading text, XML, reading DLL metadata, enables productivity and consistency while driving up the quality of the deliverable.

Being able to rip through numerous source code files looking for text in a specific context and extracting key information is super useful, primarily I can locate key information quickly. Plus, because I can generate a .NET object with properties, I can easily pipe the results and do more work easily.

- Export the results to a CSV and do analysis on them with Excel
- Catalog strings for use by QA/QC
- Create lists of classes, methods and functions

Scanning for `const` definitions

These examples read C# files looking for strings containing the word const, extracting the variable name and value. Scanning for strings across files can be applied in many ways like, searching SQL files, PowerShell scripts, JavaScript files, HTML files and more. Once the information is extracted you can again use it in many ways, for example, catalog string for internationalization, code analysis, create indexes of classes methods and functions, locate global variables. The list goes on and on.

```
public const int Name = "Dog";
const double Freezing = 32;
```

This first reader will look for `const` definitions in C# files, like the one above and produce the following output.

```
FileName Name     Value
-------- ----     -----
test1.cs Name     "Dog"
test1.cs Freezing 32
```

I will show two versions of the code. The first will read a single file and the second will search a directory for all C# files and process them. Both examples are nearly identical, differing only in how I work with the `Select-String` cmdlet.

Reading a Single C# File

This is an example of a single C# file, test.cs. It has three `const` variables defined. Two scoped at the class level and one at the method level.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        public const string Test = "test";
        public const int TestX = 1;

        static void Main(string[] args)
        {
            const double PI = 3.14;
        }
    }
}
```

Next up, I show the PowerShell script to scan and extract the const pattern.

Using Select-String

It's import to note, I am doing pattern matching here, not parsing. If one of these lines of code is in a comment, this reader will find it because it cannot tell if it is a comment or "real" line of code.

The reader will find these `const` definitions and then output them in this format. This is an array of PowerShell objects each having three properties, `FileName`, `Name`, and `Value`.

```
$regex = "\s+const\s+\w+\s+(?<name>.*)\s+=\s+(?<value>.*);"

Select-String $regex .\test.cs |
    ForEach {
        $fileName = $_.Path
        ForEach($match in $_.Matches) {
            New-Object PSObject -Property @{
                FileName = $fileName
                Name     = $match.Groups["name"].Value
                Value    = $match.Groups["value"].Value
            }
        }
    }
```

The Result

```
FileName Name  Value
-------- ----  -----
test.cs  Test  "test"
```

```
test.cs  TestX 1
test.cs  PI    3.14
```

`Select-String` finds text in files or strings. For UNIX folks, this is equivalent to `grep`. In this example, I am using a Regular Expression with the named groups, "name" and "value". `Select-String` can also find text using the `-SimpleMatch` keyword, meaning Select-String will not interpret the pattern as a regular expression statement.

So, the parameters I am passing are the pattern and file name. If there are matches found, they are piped to a `ForEach`. I capture the `$fileName` from the property `$_.Path` (`$_` is the current item in the pipeline) and then pipe the matches (`$_.Matches`) to another `ForEach`. In it I create a new `PSObject` on the fly with three properties, `FileName`, `Name` and `Value`. Where did Name and Value come from? They came from the named groups in the regular expression

I extracted data and created a custom output type using `Select-String` and `New-Object` PSObject. I can rip through any text based file searching for information and then present it as a .NET object with properties. I could have even piped this data to `Export-Csv .\MyFile.CSV`, which converts it to comma separated values, save it to a file and then I could do an Invoke-Item `.\MyFile.CSV` opening it in Excel, parsed and ready to go.

Reading C# Files in a Directory

In this example, I use `Select-String` again. The difference is I am doing a `dir` for files ending in .cs and then piping them to `Select-String`. From there, the process is the same as before.

```
$regex = "\s+const\s+\w+\s+(?<name>.*)\s+=\s+(?<value>.*);"

dir *.cs | Select-String $regex |
    ForEach {
        $fileName = $_.Path
        ForEach($match in $_.Matches) {
            New-Object PSObject -Property @{
                FileName = $fileName
                Name    = $match.Groups["name"].Value
                Value   = $match.Groups["value"].Value
            }
        }
    }
```

Result of Reading and Extracting Info from Multiple C# Files

```
FileName Name      Value
-------- ----      -----
test.cs  Test      "test"
test.cs  TestX     1
test.cs  PI        3.14
test1.cs Color     "Red"
test1.cs Name      "Dog"
test1.cs Freezing 32
```

PowerShell simplifies the process of traversing directories while searching for patterns in the text. Then taking those results and transforming them into objects with properties. I

could further pipe these results to other PowerShell built-in cmdlets or my own functions to do all kinds of work for me.

Consider refactoring this script so you can vary either the RegEx or files you want to search for but keep the same type of output.

This is a two foot dive into what you can do using PowerShell's Select-String, regular expressions and creating objects with properties. There is an entire ocean of possibility this technique can applied to with text files. Once the strings have been extracted and are in the form of a list of PowerShell objects, you can generate a wide variety of output, including, HTML documentation, other programmatic elements and much more.

A Template Engine

A template engine is software that is designed to process templates and content information to produce output documents. Writing a simple template engine in PowerShell is straightforward. This approach lets me write many different types of templates in text and then leverage PowerShell to dynamically generate the file's content based on variables or more complex logic.

The Engine

```
function Invoke-Template {
    param(
        [string]$Path,
        [Scriptblock]$ScriptBlock
    )

    function Get-Template {
        param($TemplateFileName)

        $content = [IO.File]::ReadAllText(
            (Join-Path $Path $TemplateFileName) )
        Invoke-Expression "@`"`r`n$content`r`n"@"
    }

    & $ScriptBlock
}
```

Template engines typically include features common to most high-level programming languages, with an emphasis on features for processing plain text.

Such features include:

- Variables and Functions

- Text replacement

- File inclusion

- Conditional evaluation and loops

Additionally, because we are using PowerShell to accomplish this, we get all of these benefits plus we can use all of PowerShell's features, cmdlets and more.

The parameter $ScriptBlock is the script block I'll pass in a later example. In order to execute it I use the & (call operator). Invoke-Template supports a "keyword", Get-Template. I define this keyword simply by creating a function names Get-

`Template`. Here I nest that function inside the `Invoke-Template` function. `Get-Template` take one parameter, `$TemplateFileName`.

In essence, this DSL has three moving parts. The execution of the script block, which calls `Get-Template`, the reading of the contents of that file, using the .NET Framework's `System.IO.File.ReadAllText` static method and finally using PowerShell's `Invoke-Expression` to evaluate the content just read as though it were a here-string.

I want to draw your attention to how `Invoke-Template` takes a `-ScriptBlock` as a second parameter. Practically speaking, `Invoke-Template` is an internal DSL (domain specific language). So I have the entire PowerShell ecosystem available to me and I can get really creative inside this script block calling cmdlets, getting templates and doing code generation. This opens the door for lots of automation possibilities, saving me time, effort and reducing defects in my deliverables.

A Single Variable

Let's use the template engine in a simple example. I set up this template in a file called `TestHtml.htm` in the subdirectory `etc`.

```
<h1>Hello $name</h1>
```

I use an HTML tag plus PowerShell syntax to define the variable for replacement, `$name`. Here are contents of the `TestHtml.htm`. Note, this is the verbose version. I explicitly specifying the parameter names `-Path`, `-ScriptBlock`, `-TemplateName`.

```
# dot-source it
. .\Invoke-Template.ps1

Invoke-Template -Path "$pwd\etc" -ScriptBlock {
    $name = "World"
    Get-Template -TemplateFileName TestHtml.htm
}
```

Here's the terse approach, letting PowerShell bind the parameters.

```
# dot-source it
. .\Invoke-Template.ps1

Invoke-Template "$pwd\etc" {
    $name = "World"
    Get-Template TestHtml.htm
}
```

Results

```
<h1>Hello World</h1>
```

While the intent of code is clearer using named parameters. I prefer less typing and typically write my code as terse as possible. Both versions are possible because of the magic behind PowerShell's parameter binding.

Multiple Variables

Expanding on the theme of variable replacement I'll replace two variables. The template is a blend of C# and PowerShell variables, after the variable replacement, it'll be a C# property.

```
public $type $name {get; set;}
```

And now, the script.

```
Invoke-Template "$pwd\etc" {
    $type = "string"
    $name = "FirstName"
    Get-Template properties.txt
}
```

Results

```
public string FirstName {get; set;}
```

`Invoke-Template` stitches the variables and template together and I think it is important to extrapolate here. You can have any number of `Invoke-Template` calls in a single script, each pointing to a different file path for its set of templates. Plus, the code inside the script block can be far more involved in setting up numerous variables and calling `Get-Template` multiple times, pulling in any number of templates.

Calling Multiple Templates

Here I want to create both public and private C# variable. I do this by calling different templates. I am demoing multiple templates. I want to create two properties, a `string` `FirstName` and a `DateTime Date`. For the `Date` property though I want a `get` and a `private set`. I create a file in the `etc` directory called `privateSet.txt` and stub what I want to generate.

This is the contents of Test-MultipleVariableTemplate.ps1.

```
# dot-source it
. .\Invoke-Template.ps1

Invoke-Template "$pwd\etc" {

    $type = "string"
    $name = "FirstName"
    Get-Template properties.txt

    $type = "DateTime"
    $name = "Date"
    Get-Template privateSet.txt
}
```

Results

This is incredibly useful; for example, I can write PowerShell code that reads the schema of a SQL table grabs the column names, datatypes and generates an entire C# class that maps my table to an object. Yes there are tools that do this but just a few lines of PowerShell enable these key processes and give you control of the entire workflow.

Plus, most off the shelf products are not always able to let us have fine grain control over the acquisition, process and output of the results. There are always exceptions.

```
public string FirstName {get; set;}
public DateTime Date {get; private set;}
```

This is just a small sampling of what is possible to do with Invoke-Template. It's a very powerful way to organize simple text replacement and get a lot done. Let's move on to some more involved scripts.

Complex Logic

In this example, I'm using the built-in `Import-Csv` cmdlet to read a CSV file (comma-separated value file).

```
Type, Name
string, LastName
int, Age
```

Here, piping the contents of the CSV to the ForEach, setting the appropriate variables and finally calling the template properties.txt.

```
Invoke-Template "$pwd\etc" {
    Import-Csv $pwd\properties.csv | ForEach {
        $type = $_.Type
        $name = $_.Name
        Get-Template properties.txt
    }
}
```

Results

```
public string LastName {get; set;}
public int Age {get; set;}
```

The template is the same as the previous example and the PowerShell script to make this happen is nearly identical. The main difference being the input is from a CSV file.

I can continue to add properties to the CSV file; rerun the script and code generate as many C# properties as is needed. With a little creativity, I can see this as a first step in code generating an entire C# class, ready for compilation.

UML Style Syntax

I want to show how flexible PowerShell is. I created a file containing properties in UML syntax and then use the built-in PowerShell cmdlet `Import-Csv` to read the file and convert it to an array of PowerShell objects each having the properties `Name` and `Type`. By default, `Import-Csv` reads the first line and uses it to name the properties. I override that by specifying `Name` and `Type` in `-Header` property. Plus I override the default delimiter "," using `-Delimiter` property to ":".

```
LastName  : string
FirstName : string
Age       : int
City      : string
State     : string
Zip       : int
```

```powershell
. .\Invoke-Template.ps1

Invoke-Template "$pwd\etc" {
    Import-Csv -Path .\Uml.txt -Header "Name","Type" -Delimiter ":" |
        ForEach {
            $name = $_.Name
            $type = $_.Type
            Get-Template properties.txt
        }
}
```

Results

```csharp
public string LastName  {get; set;}
public string FirstName  {get; set;}
public int Age  {get; set;}
public string City  {get; set;}
public string State  {get; set;}
public int Zip  {get; set;}
```

With a little imagination you can work up a number interesting useful formats that make it simple to represent information and then transform it into many other types of output.

Reading XML

PowerShell is not limited to reading CSV files, so I have options. As a developer, XML is typical part of my daily diet. I'll play off the previous example of generating C# properties; this time using XML drives the input to the process.

```xml
<properties>
    <property>
        <type>string</type>
        <name>City</name>
    </property>
    <property>
        <type>string</type>
        <name>State</name>
    </property>
    <property>
        <type>string</type>
        <name>Zip</name>
    </property>
</properties>
```

Let's read the XML, convert it.

```powershell
Invoke-Template "$pwd\etc" {
    ([xml](Get-Content .\Properties.xml)).properties.property |
        ForEach {
            $type = $_.type
            $name = $_.name
            Get-Template properties.txt
        }
}
```

This is the same script as the Complex Logic version, instead of reading from a comma separated value file with `Import-Csv`, I read the file using `Get-Content`, applying the PowerShell `[xml]` accelerator and dot notate over the nodes.

Results

```
public string City {get; set;}
public string State {get; set;}
public string Zip {get; set;}
```

There it is, the transformation of XML data into C# properties. The separation of the text being replaced from the PowerShell that processes the input really highlights the essence part of using PowerShell. A handful of script to process and transform information into C#, very readable and maintainable.

Bonus Round

I now will invoke all three scripts one after the other. The PowerShell engine takes care of handling the output from all of them. I am bringing together information from three disparate sources.

```
.\Test-MultipleVariableTemplate.ps1
.\Test-ComplexLogicTemplate.ps1
.\Test-ReadXMLTemplate.ps1
```

Results

I can easily pipe this to `Set-Content Person.cs` and I am well on my way to generating code that compiles.

```
public string FirstName {get; set;}
public string LastName {get; set;}
public int Age {get; set;}
public string City {get; set;}
public string State {get; set;}
public string Zip {get; set;}
```

Using this and PowerShell I have tremendous reach. I can pull information from numerous sources; a database, Excel, a web service, a web page, just to name a few. Plus, you can call `Get-Template` multiple times in the same script each pointing to different templates and produce a number of different outputs.

Generating PowerShell Functions from C# Methods

I'm going to compile a C# class, `MyMath`, on the fly, using the built-in `Add-Type` cmdlet. Note: `Add-Type` also lets me load either a DLL or C# source file. Now I have a new type, `MyMath`, loaded in my PowerShell session. I can use the methods on the .NET Framework's System.Type class like, `GetMethods()` on this type to get info.

```
$code = @"
    public class MyMath
    {
        public int  MyAdd(int n1, int n2)      { return n1 + n2; }
        public int  MySubtract(int n1, int n2) { return n1 - n2; }
        public int  MyMultiply(int n1, int n2) { return n1 * n2; }
        public int  MyDivide(int n1, int n2)   { return n1 / n2; }
        public void MyTest() {System.Console.WriteLine("Test");}
```

```
        }
"@

Add-Type -TypeDefinition $code
```
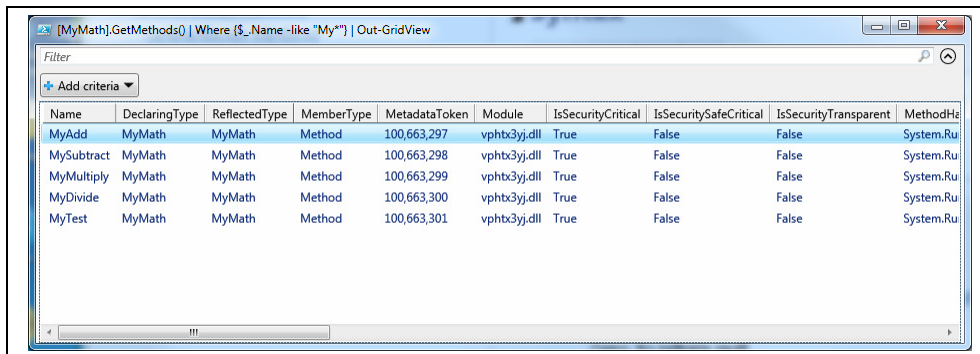
Here I take the output of `GetMethods()` and display it in a GUI using `Out-GridView`.

```
[MyMath].GetMethods() | Where {$_.Name -like "My*"} | Out-GridView
```


Figure 4-1. Inject a GUI in your pipeline – Showing Methods on a C# object

PowerShell is based on .NET so here I tap into the framework and use `GetMethods()` on the type `MyMath`. First, I'll create the variable `$code` to hold my C# class and its methods. Then, `Add-Type` will compile the code in the current PowerShell session. Lastly, I use brackets `[]` around the name of my class `MyMath`, indicating to PowerShell it is a type and then I can call `GetMethods()`. I use this approach a lot when working will C# code/DLLs at the command line. I have used the "long form" of the code in the script example for clarity. When I do this at the command line I like the pithy version better, saving time, effort and keystrokes.

In PowerShell version 3, it gets simpler. Cleaner, less noise, fewer keystrokes and more essence. Here the `Where` syntax loses the curly braces, and the `$_`.

```
[MyMath].GetMethods() | Where Name -like "My*" | Out-GridView
```

Get Parameters

I'll take that last line of PowerShell, from the previous example, pipe it to the `ForEach`, calling the .NET `GetParameters()` method. Then I'll pipe it to `Out-GridView` and get a nice summary of parameter information on `MyMath` code implementation.

```
[MyMath].GetMethods() | Where {$_.Name -like "My*"} |
    ForEach {
        $_.GetParameters()
    } | Out-GridView
```

Figure 4-2. Showing C# Parameters from Method Signatures

Pulling It All Together

If I wanted, I could type this by hand. This gives me full access to MyMath in PowerShell. PowerShell is an *automation platform*; I'm a lazy coder so I'll write a script to make it happen.

```
$MyMath = New-Object MyMath

function Invoke-MyAdd ($n1, $n2) {$MyMath.MyAdd($n1, $n2)}
function Invoke-MySubtract ($n1, $n2) {$MyMath.MySubtract($n1, $n2)}
function Invoke-MyMultiply ($n1, $n2) {$MyMath.MyMultiply($n1, $n2)}
function Invoke-MyDivide ($n1, $n2) {$MyMath.MyDivide($n1, $n2)}
function Invoke-MyTest () {$MyMath.MyTest()}
```

Wrapping MyMath in PowerShell functions is a gateway to many capabilities. For example, I can interact with MyMath at the command line, in scripts, write tests and pipe results to the rest of the PowerShell ecosystem. PowerShell enables me to compose code in ways I cannot in a system language like C#. In this simple example I let PowerShell handle parameters through parameter binding so I can focus less on mechanics and more on problem solving.

```
Invoke-MyAdd 1 3
1..10 |
    ForEach {Invoke-MyAdd $_ $_} |
    ForEach {Invoke-MyMultiply $_ $_}
```

I've shown PowerShell code that can get the methods and parameters for an object which is loaded into a PowerShell session. The next script will combine these and using a Here-String, will create the PowerShell functions that fully wrap MyMath signatures in a PowerShell way.

One line gets a bit funky. In the Get-Parameter function I have "`` `$$($_.Name)``". This is needed in order to generate the $n1. I use the PowerShell escape character ` before the first $, otherwise PowerShell will interpret that as $$. That is a PowerShell automatic variable, which contains the last token in the last line received. The $($_.Name) is a subexpression, and is a simple rule to memorize when you want to expand variables in strings.

```
function Get-Parameter ($target) {
    ($target.GetParameters() |
        ForEach {
            "`$$($_.Name)"
        }
```

```
    ) -join ", "
}

@"
`$MyMath = New-Object MyMath
$([MyMath].GetMethods() | Where {$_.Name -like "My*"} | ForEach {

    $params = Get-Parameter $_

@"

function Invoke-$($_.Name) ($params) {`$MyMath.$($_.Name)($($params))}
"@
})

"@
```

Result

```
function Invoke-MyAdd ($n1, $n2) {$MyMath.MyAdd($n1, $n2)}
function Invoke-MySubtract ($n1, $n2) {$MyMath.MySubtract($n1, $n2)}
function Invoke-MyMultiply ($n1, $n2) {$MyMath.MyMultiply($n1, $n2)}
function Invoke-MyDivide ($n1, $n2) {$MyMath.MyDivide($n1, $n2)}
function Invoke-MyTest () {$MyMath.MyTest()}
```

Generating PowerShell wrappers is a scalable approach. Compare this to manually transforming the C# method signatures to PowerShell functions. Plus, if my C# code is still changing, I have a single script solution to wrapping my C# functions and make them PowerShell ready. Again, this saves time, effort and I'll have fewer finger errors.

This example is for illustration. With some additional thought and work, I can make it generic by parameterizing the final snippet.

I can:

- Add a $Type parameter, which lets me pass in any type for inspection
- Add a Where filter parameter, to be used in when the methods are piped from GetMethods
- Add a variable name parameter, so I don't have to hard code $MyMath

A final thought, the text manipulation tools that PowerShell brings table are invaluable in doing many types of transforms. In the next sections you'll see a few more ways. These ideas are not new. PowerShell's deep integration to Windows and the .NET Framework are what make it possible for developers to optimize their efforts.

Calling PowerShell Functions from C#

Next, I'm going to compile more C# and then create a custom object rather than a PSModuleInfo object using New-Module and the -AsCustomObject property. I'll create a single PowerShell function called test and store it the variable $module so I can pass it to the constructor in the C# class. Finally, I'll call the C# InvokeTestMethod. InvokeTestMethod looks up the PowerShell test function in the module that was passed in the constructor. If it is found, Invoke is called, all the ducks line up, and it prints "Hello World".

> This next example using Add-Type will work if you're using PowerShell v3.
>
> If you are using PowerShell v2 and have not added *powershell.exe.config* to point to .NET 4.0, see Appendix A "How to run PowerShell with .NET 4.0".
>
> If you're not sure what version of the .NET runtime you're session is using, type `$PSVersionTable` and look for the `CLRVersion` entry.

```
Add-Type @"
using System.Management.Automation;

public class InvokePSModuleMethod
{
    PSObject module;
    public InvokePSModuleMethod(PSObject module)
    {
        this.module = module;
    }

    public void InvokeTestMethod()
    {
        var method = module.Methods["test"];

        if(method != null) method.Invoke();
    }
}
"@

$module = New-Module -AsCustomObject {
    function test { "Hello World" | Out-Host }
}

(New-Object InvokePSModuleMethod $module).InvokeTestMethod()
```

That's a long trek to get Hello World printed. I could have just type "Hello World" at the command line. There's a method to the madness.

In the next section, I will use these pieces to create a visitor that uses PowerShell v3's new access to the AST (Abstract Syntax Tree). I will read PowerShell source code and extract information by parsing it, not just scanning for text patterns.

A hat tip to Jason Shirk, one of the of the PowerShell teams language experts, who shared the technique.

Override C# Methods with PowerShell Functions

Ok, I've shown you how to pull out the metadata from compiled C# code and generate PowerShell functions to wrap them. This is extremely useful when exploring a new .NET DLL. I can quickly extract key information about the component and start kicking the tires right from the command line. Plus, because the .NET component is wrapped in PowerShell functions, I seamlessly plug into the PowerShell ecosystem, further optimizing my time and effort. For example, if the component returns arrays of objects, I

can use the `Where,` `Group` and `Measure` cmdlets to filter and summarize information rapidly.

Moving on to overriding C# base class methods with PowerShell functions.

The next example extracts metadata from a .NET DLL, generates C# methods overriding the base class methods and creates a constructor that takes a PowerShell module.

Each of the C# methods doing the override uses the technique in the previous to look up the method in PowerShell module and call it with the correct parameters.

I'm using the AST capabilities of PowerShell v3 to demonstrate this technique of extracting method signatures from C# and then injecting a PowerShell Module to provide a way to override the implementation, and this is valid for PowerShell v2 and can be applied to .NET solutions employing inheritance.

The Breakdown

I'm going to break this script down into a few sections; the metadata extraction of the PowerShell v3 `AstVisitor` methods, subsequent C# code generation putting the PowerShell "hooks" in place and the creation of the PowerShell custom object using `New-Module`. This will have a PowerShell function called `VisitFunction` and mirrors the method I override in the base class `AstVisitor`. This PowerShell function will be called each time a function is found in our source script. `VisitFunction` takes `$ast` as a parameter and it contains all the information about the function that has been matched in our source script. I'll be pulling out only the name and line number where it was matched.

Looking for PowerShell Functions

In this source script I want to find where all the functions are defined.

```
function test1 {"Say Hello"}
1..10 | % {$_}
function test2 {"Say Goodbye"}
1..10 | % {$_}
function test3 {"Another function"}
#function test4 {"This is a comment"}
```

I can see three functions named `test1`, `test2`, `test3` and they are on lines 1, 3 and 5. The last function, `test4`, is a comment. I included it for two reasons. First, if I was scanning the file using `Select-String` and pattern matching on function, this would show up in the results and it would be misleading. Second, using the AST approach, `test4` will be recognized as a comment and not include in the results when searching for functions.

While it is easy to scan a file visually, if I'm looking at a large script with many functions, I'd like an automated way to know what and where my functions are. Plus, if I can extract this information programmatically, the potential is there to automate many other activities.

Extracting Metadata and Generating C#

```
public override AstVisitAction $FunctionName($ParameterName ast)
{
    var method = module.Methods["$FunctionName"];
    if (method != null)
```

```
    {
        method.Invoke(ast);
    }
    return AstVisitAction.Continue;
}
```

I'm going to generate something a little more complex, leveraging the `Invoke-Template` I built before. The goal is to create a C# class that has all of the override methods found in `System.Management.Automation.Language.AstVisitor`. This is equivalent to being in Visual Studio, inheriting from `AstVisitor`, overriding each method and then providing an implementation.

The implementation I want to provide, for each overridden method, is a lookup for that function name in the module/custom object passed from PowerShell. If one is found, I'll invoke it and pass it the AST for the declaration being visited.

```
[System.Management.Automation.Language.AstVisitor].GetMethods() |
    Where { $_.Name -like 'Visit*' } |
    ForEach {
        $functionName = $_.Name
        $parameterName = $_.GetParameters()[0].ParameterType.Name

        Get-Template AstVisitAction.txt
    }
```

This is the template that gets it done, the file is named `AstVisitAction.txt`.

Now for the PowerShell code snippet that'll figure out the `FunctionName`, `ParameterName` and invoke the template that does the code generation.

The `GetMethods()` method returns a list of methods on the Type `System.Management.Automation.Language.AstVisitor`. I'm filtering the list of methods to only the ones whose names begin with Visit* `Where { $_.Name -like 'Visit*' }`. In the `ForEach` I grab the name of the function `$_.Name` and the name of the parameter type being passed to it, `$_.GetParameters()[0].ParameterType.Name`.

```
using System;
using System.Management.Automation;
using System.Management.Automation.Language;

public class CommandMatcher : AstVisitor
{
    PSObject module;
    public CommandMatcher(PSObject module)
    {
        this.module = module;
    }

    $methodOverrides
}
```

The template sets up references, a constructor and backing store for the module being passed in. The key piece is the `$methodOverrides` variable. This will contain all the text generated from the previous template, `AstVisitAction.txt`.

```
. .\Invoke-Template.ps1
```

```
Invoke-Template $pwd\etc {

    $methodOverrides = Invoke-Template $pwd\etc {
      [System.Management.Automation.Language.AstVisitor].GetMethods() |
          Where { $_.Name -like 'Visit*' } |
          ForEach {
              $functionName = $_.Name
              $parameterName = $_.GetParameters()[0].ParameterType.Name

              Get-Template AstVisitAction.txt
          }
    }

    Get-Template CommandMatcher.txt
}
```

This is the completed script that generates a C# class ready for compilation. This class handles visiting any PowerShell source, calling out to a PowerShell function to handle the node that is visited. I'll show that next.

Fortunately, it's not necessary to understand the recursive descent parser mechanism. Fundamental is the metadata extraction and code generation which is the glide path to using the `Add-Type` and compiling useful code on the fly in the current context.

The PowerShell Module

Now that I have code generated all of the overrides for the base class `AstVisitor`. I want to create a PowerShell module to pass to it that will be called back on every time a PowerShell function definition is detected.

```
$m = New-Module -AsCustomObject {

    $script:FunctionList = @()

    function VisitFunctionDefinition ($ast) {
        $script:FunctionList += New-Object PSObject -Property @{
            Kind = "Function"
            Name = $ast.Name
            StartLineNumber = $ast.Extent.StartLineNumber
        }
    }

    function GetFunctionList {$script:FunctionList}
}
```

I store this in the variable `$m`, I'll pass it to the constructor later.

I added a helper function `GetFunctionList` which returns the script scoped variable. `FunctionList` is initialized to an empty array to start and is populated in `VisitFunctionDefinition`.

Each time a function declaration is matched, the PowerShell function `VisitFunctionDefinition` is invoked. I then emit a PowerShell object with three parameters, `Kind`, `Name`, and `StartLineNumber`. I hard code `Kind`, for simplicity, and get the other two values from the data passed in the `$ast` variable.

Testing it all

I'll create a reusable helper function that takes a PowerShell script and returns the AST (Abstract Syntax Tree) that can be "visited"; I'll call it Get-Ast. Next, I'll "new" up the CommandMatcher I built in C# during code generation phase and pass in $m which contains my PowerShell module with the function I want to get invoked. The variable $ast contains the AST of the script passed in the Here-String. The variable $ast is a System.Management.Automation.Language.ScriptBlockAst and the method I want to invoke is Visit(). I will pass $matcher, which is my custom visitor, to it. Finally, I will call $m.GetFunctionList() displaying the details about the functions that were found.

```
function Get-Ast
{
    param([string]$script)

    [System.Management.Automation.Language.Parser]::ParseInput(
        $script,
        [ref]$null,
        [ref]$null
    )
}

$matcher = New-Object CommandMatcher $m

$ast = Get-Ast @'
function test {"Say Hello"}
1..10 | % {$_}
function test1 {"Say Goodbye"}
1..10 | % {$_}
function test2 {"Another function"}
'@

$ast.Visit($matcher)
$m.GetFunctionList()
```

Results

This correctly finds the three functions in my test script. Displaying the name of the function and the line it is on.

```
Name   StartLineNumber Kind
----   --------------- ----
test                 1 Function
test1                3 Function
test2                5 Function
```

You can easily rework this to process a single script or an entire directory of scripts. In addition, a filename can be added as a property, thus enabling filtering of function names and filenames. This way I can semantically scan any number of PowerShell scripts for a particular function name and quickly locate the file and line number where it lives.

Plus, I can add more functions to the PowerShell module to match on parameters, variable expressions and more. From there, I'd create a new PSObject with the properties I wanted and I have a list of key information about my scripts that I could programmatically act on.

Using PowerShell's `System.Management.Automation.Language` library like this is one application of what it can do. There is a lot to explore here that is beyond the scope of this book. If you're familiar with the tool ReSharper from JetBrains and its ability to refactor C# code, that is the potential of `System.Management.Automation.Language`. For example, being able to rename part of a PowerShell function name and ripple that change through an entire script accurately. Another example, is extracting a section of PowerShell code as a function, naming it, adding it to the script and replacing where it came from with the new function name. Doing static analysis along the lines of a lint tool, PSLint.

This doesn't come for free. You need to learn the ins and outs of this library. These would be great open source tools for PowerShell as well as opportunities to learn deeper parts of what this platform offers.

Summary

In this chapter I showed several ways to use PowerShell to work with information, transform it and position it for consumption elsewhere. The information was stored in C# files, text files and it was even extracted directly from compiled DLLs. These ideas can also be extended to SQL Server schemas, XML, JSON and even Microsoft Excel. PowerShell easily integrates with all of this because it is based on .NET.

As a developer, I reuse and expand on these approaches for every project I work on. I actively seek out patterns in the workflow and automate them. This has numerous benefits. Code generation has been around as long as software languages have been. PowerShell's deep integration to the .NET platform and its object pipeline optimizes the development effort. Being able to crack open a DLL, inspect methods, and parameters all from within a subexpression in a `Here-String` and then compile it on the fly all in a page of code enables me, and other developers, to iterate through ideas at an (even more) rapid pace.

Finally, being able to extend C# solutions by invoking PowerShell, and here is the key, without having to touch the original C# code, is huge. Scripting languages are sometimes referred to as glue languages or system integration languages. PowerShell, being based on .NET, takes this to a new level.

5

Add PowerShell to Your GUI

PowerShell Empowers Others to Customize

Roy Osherove points out that "adding scripting support to your application is one of the most valuable things you can do for your client, letting them add value to your software, and keep it current over time with little or no overhead from the developers"

After adding PowerShell to an application, developers, end-users, testers and system integrators are able to customize the application's logic to better match their specific needs. This approach is an efficient use of resources, with developers focusing their efforts on core functionality while allowing others most familiar with their domain knowledge to easily and independently customize it to meet their needs. Using PowerShell in this way, there is no need to distribute the application source code for other developers to extend the application. As a result, you do not need to support multiple versions of the application.

Adding PowerShell to an application not only speeds the development of software, there are common areas of customization for applications which may include modification to match particular businesses processes, automation of repetitive tasks, the addition of unique features and access to internal and remote data.

Embedding PowerShell in your C# Application

PowerShell is surfaced as a command line application (the console), a scripting language and an API. Here I'll show you the API and how simple it is to create the PowerShell engine, call some cmdlets and print out the results.

> You'll need to add a reference to PowerShell. To do so, open the project file as a text file and add the following line into the <ItemGroup> section:
>
> <Reference Include="System.Management.Automation" />

I've setup two C# extension methods `ExecutePS()` and `WritePS()`. `ExecutePS()` extends `strings` and `WritePS()` extends `List<PSObject>`. The strings are the PowerShell commands and the `List<PSObject>` are the results of invoking those commands.

By default, the `ExecutePS()` method prints the results to the console. If you pass false to `ExecutePS()` it returns a list of PSObjects. PowerShell version 3 takes a dependency on the Dynamic Language Runtime (DLR) and `PSObject` implement `IDynamicObject`. This lets me do a `foreach` over the results and take advantage of late binding to get at the `ProcessName`.

> The foreach block requires PowerShell v3 installed. If you only have PowerShell v2, comment out the foreach block and the example will run cleanly.

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Management.Automation;

class Program
{
    private static void Main()
    {
        var script = "Get-Process | Where {$_.Handles -gt 400}";

        // Let the extension method
        // print out the results
        script.ExecutePS();

        // In PowerShell v3, PSObject
        // implements IDynamicObject
        foreach (dynamic item in
            script.ExecutePS(writeToConsole: false))
        {
            Console.WriteLine(item.ProcessName);
        }
    }
}

public static class PSExtensions
{
    public static List<PSObject> ExecutePS(
        this string script, bool writeToConsole = true)
    {
        var powerShell = PowerShell
                .Create()
                .AddScript(script);

        if (writeToConsole)
        {
            powerShell.AddCommand("Out-String");

            powerShell
                .Invoke<PSObject>()
```

```
                    .ToList()
                    .WritePS();
        }

        // Lets the caller act on the returned collection
        // of PowerShell objects
        return powerShell
            .Invoke<PSObject>()
            .ToList();
    }

    public static void WritePS(this List<PSObject> psResults)
    {
        psResults.ForEach(i => Console.WriteLine(i));
    }
}
```

The script variable contains the PowerShell commands. The `ExecutePS()` method creates the PowerShell engine and then adds the commands with the `AddScript()` method. I use the `AddCommand()` method to append the `Out-String` cmdlet to whatever is specified in the script variable. This tells PowerShell to covert the objects returned to their string representations. PowerShell will execute all of this after I call the `Invoke()` method.

The `Invoke()` method returns an array of PowerShell PSObjects. Just as `System.Object` is the root of the type system in .NET, `PSObject` is the root of the synthetic type system in PowerShell.

The `WritePS()` method extends `List<PSObject>` by looping through the results and printing it to the console.

This example shows how easy it is to include the PowerShell engine your application. You can use all of the PowerShell cmdlets with this approach including external scripts and modules developed by you, others in the community, third parties or Microsoft.

I do not include any error management, profile loading, or REPL console here. If you want to see one in action plus how to load your applications objects into the PowerShell run space, read on and see what you can do with the Beaver Music application.

The Beaver Music application is a WPF GUI application and I layer a simple WPF PowerShell console into it. It is a command line in my WPF GUI, capable of working with all the objects in my app including the MEF container and more.

The good news, it's included with the book and can be easily hooked into any of your GUI apps too.

Beaver Music Application

The reference application for this chapter is a very simple music album management system. It supports create, read, update and delete (CRUD) of albums. Beaver Music has the functionality you'd expect, a couple of dialogs for adding and changing album information and you can delete albums. What I want to focus on is the PowerShell Console button. It is a WPF application that has the PowerShell engine embedded in it. PowerShell, a distributed automation platform, is surfaced as console, scripting language and an API. The custom PowerShell Console uses this surfaced API in conjunction with

the Beaver Music Application so it can be scripted and automated. Similar to the way Microsoft Excel can be automated with the embedded Visual Basic for Application scripting language
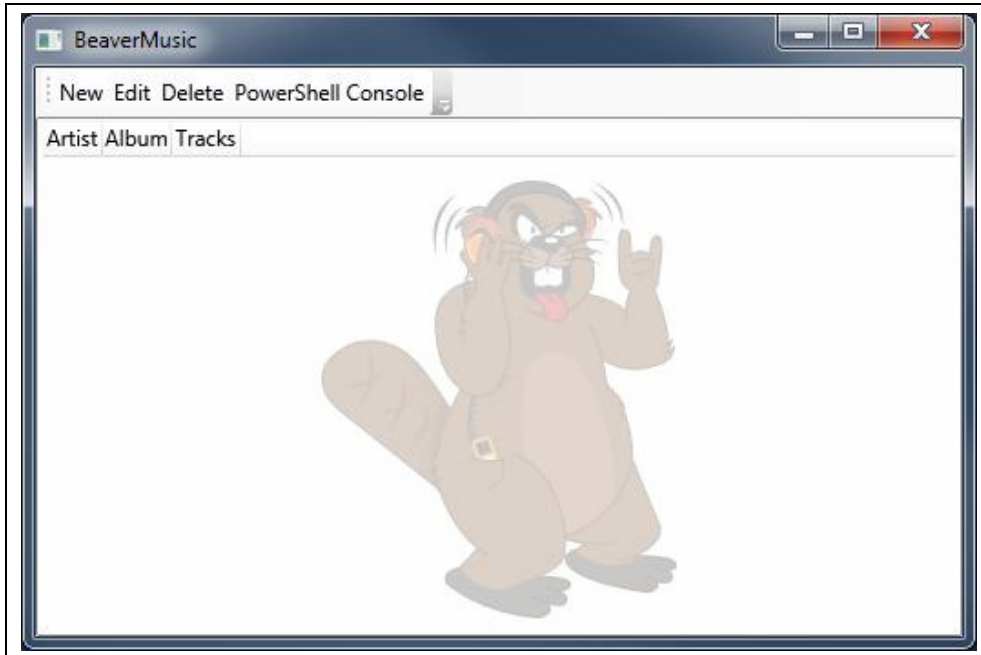
Figure 5-1. The Beaver Music App

PowerShell Console

After clicking on the "PowerShell Console" button you'll see Figure 5-2.
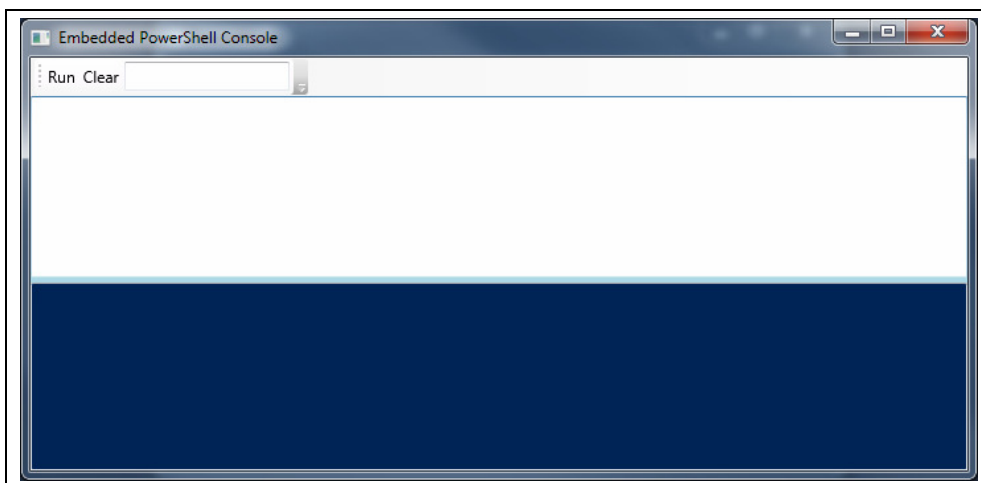
Figure 5-2. PowerShell Console

In addition, each time the console is launched, I inject variables which are instances of the running components. For example, the top pane is a WPF textbox (Windows

Presentation Foundation (or WPF) is a computer-software graphical subsystem for rendering user interfaces in Windows-based applications.). I added textbox reference into the PowerShell runspace, setting it to the variable name `$ScriptPane`. If I want to change the background color of the `$ScriptPane` I can type `$ScriptPane.Background = "Cyan"`, press F5 and it will change background color of the textbox, at runtime.

The top pane is where you type in PowerShell commands or scripts, then either click Run or press F5 to execute it. The bottom pane will show the results. It's a full fledge PowerShell engine. So you can type any valid PowerShell, you can even type invalid PowerShell and you'll see the errors in the bottom pane. The BeaverMusic PowerShell Console is custom and does not have the all the niceties found in the Microsoft PowerShell console or in the Integrated Scripting Environment (ISE).

The source to this application is included so feel free to enhance it or build your own. If you do, it'd be great if you posted it to a social coding platform like Github so others could use it, change it and benefit from it too.

What makes it custom? You should be able to embed this console in any WPF application. The console is a WPF component layered on top of a PowerShell engine. Plus, it supports a profile, type `notepad $profile` and press F5. You can store PowerShell functions here and they will be available each time you run the application. Also, custom variables will be added through both the profile and the C# code which are not available in other PowerShell consoles like the command line and PowerShell ISE (Integrated Script Environment).

I've also injected live instances of the main BeaverMusic application. For example, the WPF application has an Album Repository. The repository is the in-memory data store for holding all the albums. I've done this using the `AddVariable()` method.

```
PSConfig.AddVariable("AlbumRepository", _albumRepository);
```

This means I can get to the live instance of `_albumRepository` from the PowerShell variable `$AlbumRepository`. Since this is a PowerShell console, I can inspect the methods on that variable using `Get-Member`, see Figure 5-3.
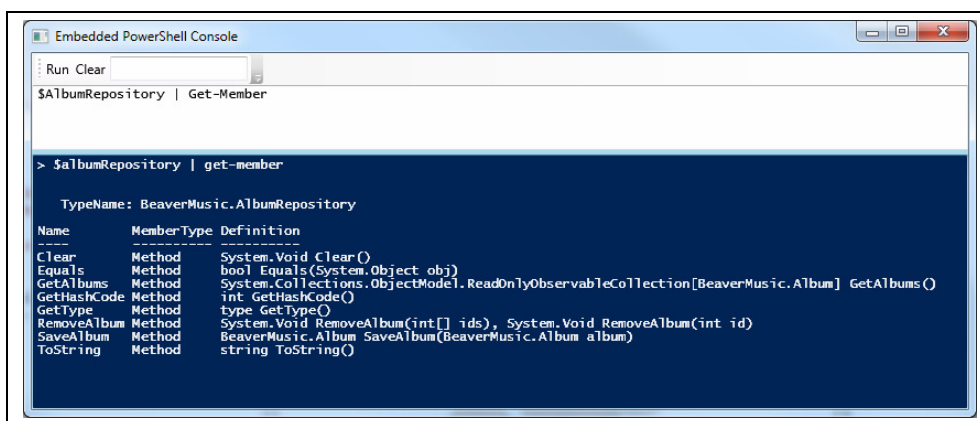
Figure 5-3. Inspecting the methods on AlbumRepository

Foundational Functions

The live objects of the Beaver Music application are added the PowerShell Engine. Now I can write PowerShell functions that take advantage of them. I've created five of them, and they follow the PowerShell naming standards of Verb-Noun. You can find the PowerShell approved verbs by typing Get-Verb at the command prompt. The ones I create are Add-Album, Clear-Album, Get-Album, New-Album and Remove-Album. They support the CRD of the CRUD model nicely; I did not implement an Update function PowerShell.

I want to highlight that on three of the five functions I have decorated the parameters with either ValueFromPipelineByPropertyName or ValueFromPipeline. These two attributes really make PowerShell sing when piping objects between functions.

New-Album

```
function New-Album  {

    param(
        [Parameter(ValueFromPipelineByPropertyName=$true)]
        [string]$Name,
        [Parameter(ValueFromPipelineByPropertyName=$true)]
        [string]$Artist
    )

    Process {
        $album = New-Object BeaverMusic.Album

        $album.Name   = $Name
        $album.Artist = $Artist

        $album
    }
}
```

This is the New-Album function. For each its parameters I attribute them with ValueFromPipelineByPropertyName which indicates that the parameter can take values from a property of the incoming pipeline object that has the same name as this parameter.

New-Album takes two parameters, a Name and Artist, and returns a BeaverMusic.Album object with those properties set, see Figure 5-4. The next example leverages the pipeline and the ValueFromPipelineByPropertyName.
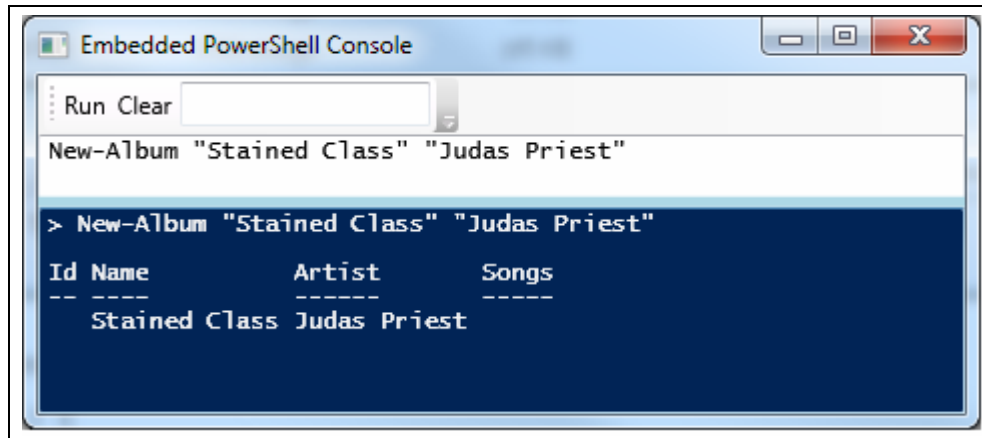
Figure 5-4. New-Album in Action

Add-Album

```
function Add-Album {
    param(
        [Parameter(ValueFromPipeline=$true)]
        $album
    )

    Process {
        $AlbumRepository.SaveAlbum($album) | Out-Null
    }
}
```

ValueFromPipeline indicates whether the parameter can take values from incoming pipeline objects. I need to specify a Process block which indicates it will execute once for each $album that is passed from the pipeline. An added benefit is I can also use traditional parameter passing.

```
ForEach($Album in $AlbumList) {
    Add-Album $Album
}
```

This assumes $AlbumList contains an array of PowerShell objects that have been set up using the New-Album function. These objects will have two properties, Name and Artist.

Next up, I'll show a different and far simpler syntax that fully leverages PowerShell's parameter binding mechanism that is enabled with the ValueFromPipeline and Process block approach.

Import-Csv

I have *all* of PowerShell available to me and I don't want to build up my list of albums by hand. Instead, I'll store a list of them (or download one) in a CSV file**Error! Reference source not found.**.
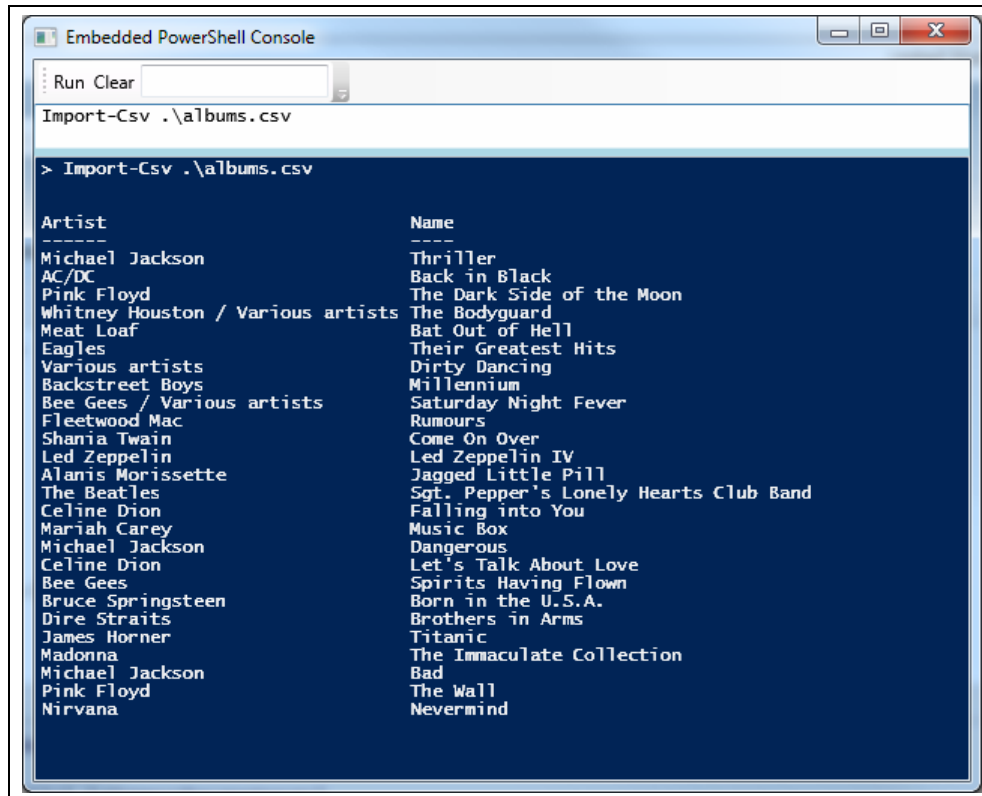
```
Artist,Name
"Michael Jackson","Thriller"
```

```
"AC/DC","Back in Black"
"Pink Floyd","The Dark Side of the Moon"
"Whitney Houston / Various artists","The Bodyguard"
"Meat Loaf","Bat Out of Hell"
"Eagles","Their Greatest Hits"
"Various artists","Dirty Dancing"
"Backstreet Boys","Millennium"
"Bee Gees / Various artists","Saturday Night Fever"
"Fleetwood Mac","Rumours"
"Shania Twain","Come On Over"
"Led Zeppelin","Led Zeppelin IV"
"Alanis Morissette","Jagged Little Pill"
"The Beatles","Sgt. Pepper's Lonely Hearts Club Band"
"Celine Dion","Falling into You"
"Mariah Carey","Music Box"
"Michael Jackson","Dangerous"
"Celine Dion","Let's Talk About Love"
"Bee Gees","Spirits Having Flown"
"Bruce Springsteen","Born in the U.S.A."
"Dire Straits","Brothers in Arms"
"James Horner","Titanic"
"Madonna","The Immaculate Collection"
"Michael Jackson","Bad"
"Pink Floyd","The Wall"
"Nirvana","Nevermind"
```

I use the PowerShell cmdlet `Import-Csv` to read the file. This cmdlet creates an array of objects each having an artist and name property. These property names are decided from the first line of the file, see Figure 5-5.

Figure 5-5. Import-Csv from albums.csv

Next, I'll pipe this to New-Album. Here is the parameter binding at work, remember we set that up using `ValueFromPipeline` and `Process block`. The Import-Csv is transformed into album objects with the correct properties set. Finally, I pipe the results to Add-Album so they are stored in the Album Repository and ultimately displayed in the Beaver Music main window, see Figure 5-6.

```
Import-Csv .\albums.csv | New-Album | Add-Album
```
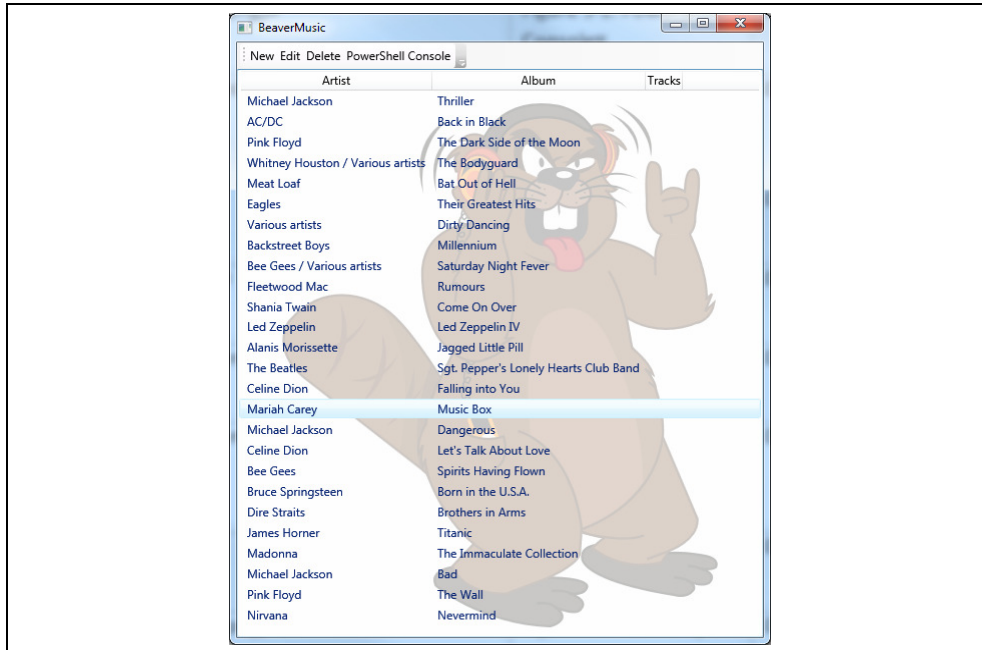
Figure 5-6. Importing and adding albums to the repository

Notice, I do not have to handle looping, end of files or parameter passing. This is very different approach to programming coming from C# and I can't stress enough how much time and effort this saves. In less than 50 characters, I'm exercising (testing) several code paths in my application. With a few more characters, I'll be clearing and filtering the list of albums and even pulling data from the Internet to create lists albums.

Get-Album and Clear-Album

I wrapped the previous 50 characters in function and called it `Import-Default`. I'm now exercising a chunk of my app with 15 characters. I type that in and add `Get-Album`. `Get-Album` reads all the albums currently in the repository, Figure 5-8.
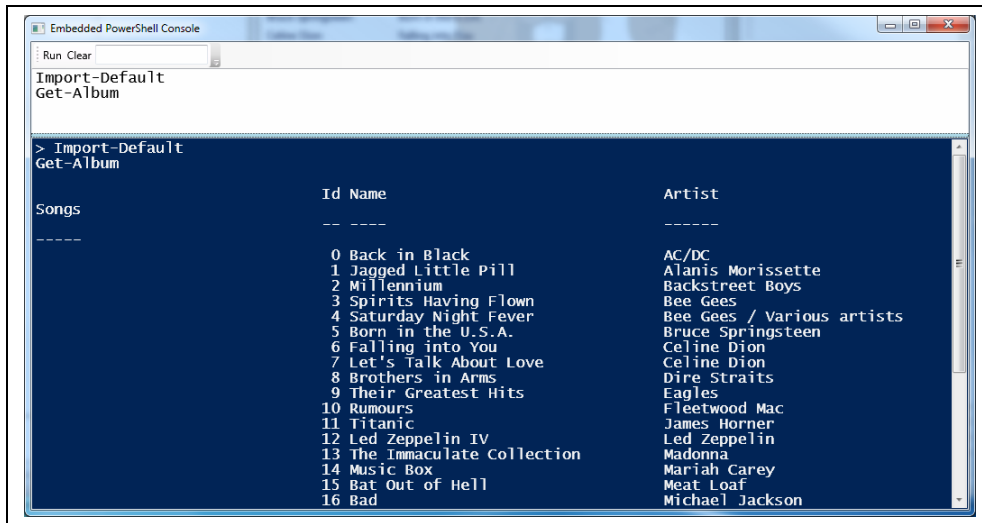
Figure 5-7. These commands update the GUI and the Results pane
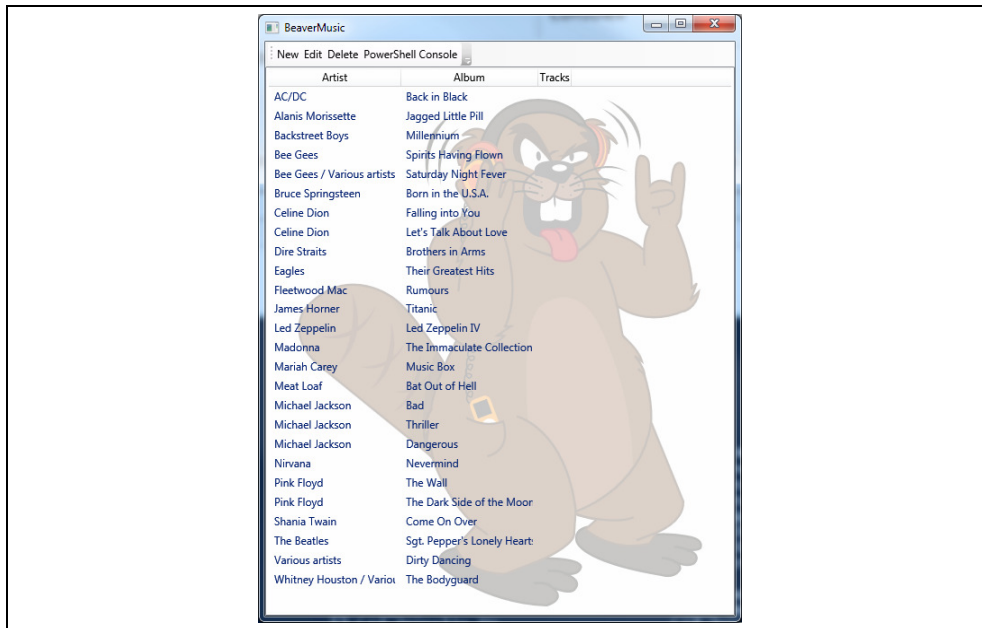
Figure 5-8. Importing and retrieving albums

Manage Applications better with PowerShell

Now I'm interacting with live data in a live environment in my application. If I run this script again, pressing F5, I will now have duplicate records, so I'll add the `Clear-Album` at the top so I can work with an empty repository each time.

Let's use some more built in PowerShell. I know there are 26 songs in my CSV file, let's make sure that after pushing all that data through the multiple code paths, I in fact end up with that number of albums in the repository, Figure 5-9.

So I clear the repository, import the defaults, retrieve all the albums and count them with the PowerShell cmdlet `Measure (which is an alias to Measure)`, and sure enough, it is the correct count.
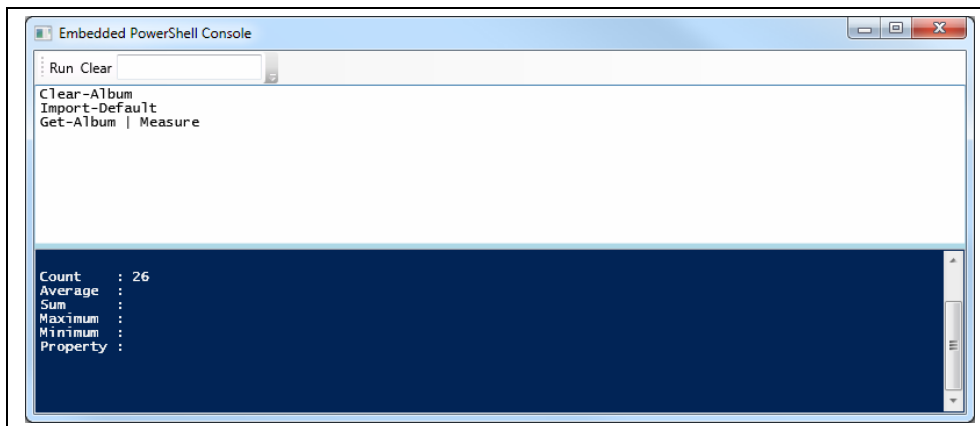
Figure 5-9. Counting the albums

One more tweak and I it reads like a unit test Figure 5-10.
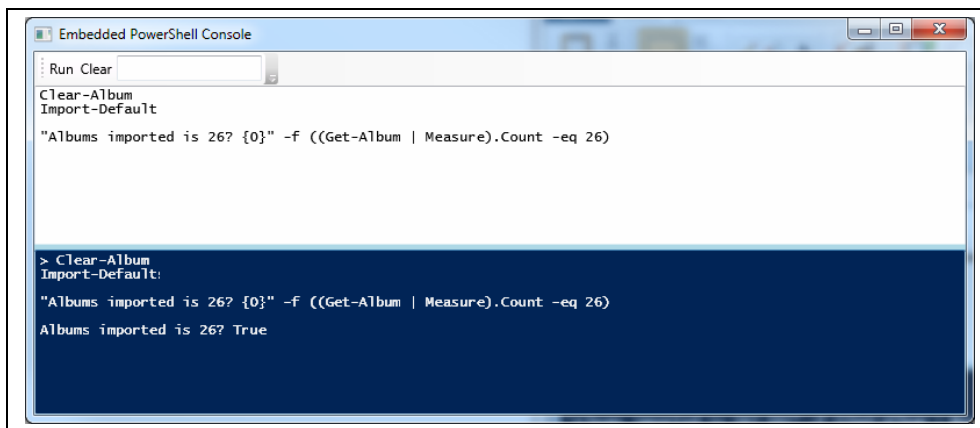
Figure 5-10. Asserting the number of albums added

Import Albums from the Web

Information is stored in many formats as well as many locations. When I am testing the Beaver Music application I like to flow lots of different data through it. This exercises different aspects and lets me handle data they I may not expect. Often is the case that the Web has ready-made data sources I can tap into. Perhaps I'd need to scrub the data a bit, deleting unwanted details, combing others and then emitting PowerShell objects with properties so I can let it play into the pipeline. So I took the albums.csv and made it available from my website. I'll create a new function `Get-AlbumFromWeb`, and then pipe it just the way I did before, first to `New-Album` then `Add-Album`, and bingo I get

the same number of albums. This time, I reached out over the Internet, got my data, and displayed it all from within the same PowerShell console, Figure 5-11.
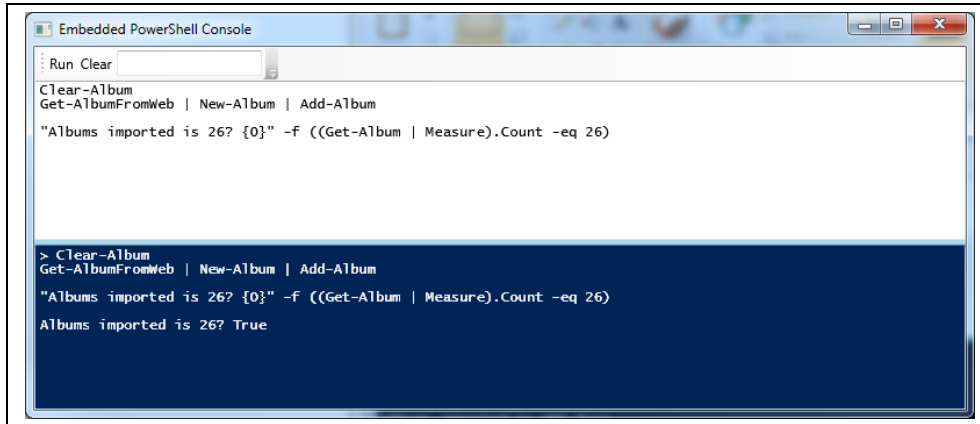

Figure 5-11. Importing the albums from a Web Site

Function Get-AlbumFromWeb

```
function Get-AlbumFromWeb {
    $wc=New-Object Net.WebClient
    $url="http://dougfinke.com/PowerShellForDevelopers/albums.csv"

    $wc.DownloadString($url) |
        ConvertFrom-Csv
}
```

Interacting with the web is not native in PowerShell v2 (it is in v3); so I reach into the .NET Framework and create a new `Net.WebClient` and use the `DownloadString()` method. I'm pulling down the contents of a CSV file so I can pipe it to `ConvertFrom-Csv` (another built-in PowerShell cmdlet) and now the data is ready to be piped to my functions that load it into the music repository.

PowerShell v3

I've got PowerShell v3 CTP2 installed, so I can replace my function `Get-AlbumFromWeb` with this new one in v3, `Invoke-RestMethod,` to get the same result with fewer lines of code, Figure 5-12.

Figure 5-12. PowerShell v3 Invoke-RestMethod

Out-GridView

```
Import-Default
Get-Album | Out-GridView
```

Out-GridView is a great tool that debuted in PowerShell v2. Bottom line, it is a separate interactive window that supports filtering, sorting and in PowerShell v3 you can use the –PassThru parameter so you can select items and have the passed through to the pipeline. I use it in the custom PowerShell Console to great effect.
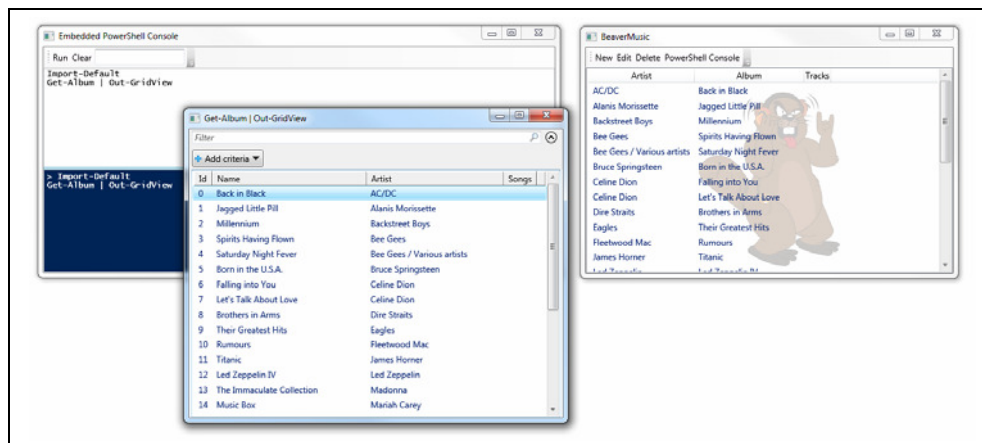

Figure 5-13. Dumping live data to Out-GridView

Export-ToExcel

Getting data into Excel is extremely helpful for analysis. Plus, then I have access to PivotTables, charting and more. PowerShell doesn't have anything out the box for working with Excel, not to worry. Transforming data is another sweet spot for PowerShell.

```
function Export-ToExcel {
    param(
        $fileName = "$pwd\BeaverMusic.csv"
```

```
    )

    Get-Album |
        Export-Csv –NoTypeInformation $fileName

    Invoke-Item $fileName
}
```

The `Export-ToExcel` function uses one of the foundation functions, `Get-Album`, it gets all the albums in the repository and then pipes it to `Export-Csv`, another built-in PowerShell cmdlet. `Export-Csv` takes an array of objects and saves it to a file, in a comma separated file format. It gets the names for the data columns from the names of the properties on the object. In the last line of the script I call `Invoke-Item` (yet another built-in PowerShell cmdlet), passing it the file name used in the export. `Invoke-Item` performs the default action on the specified item; in this case the default action associated with CSV files opens it in Excel, Figure 5-14.

Just a note, I regularly use the Export-Csv/Invoke-Item in both the console and ISE. I find it invaluable way to work with data.
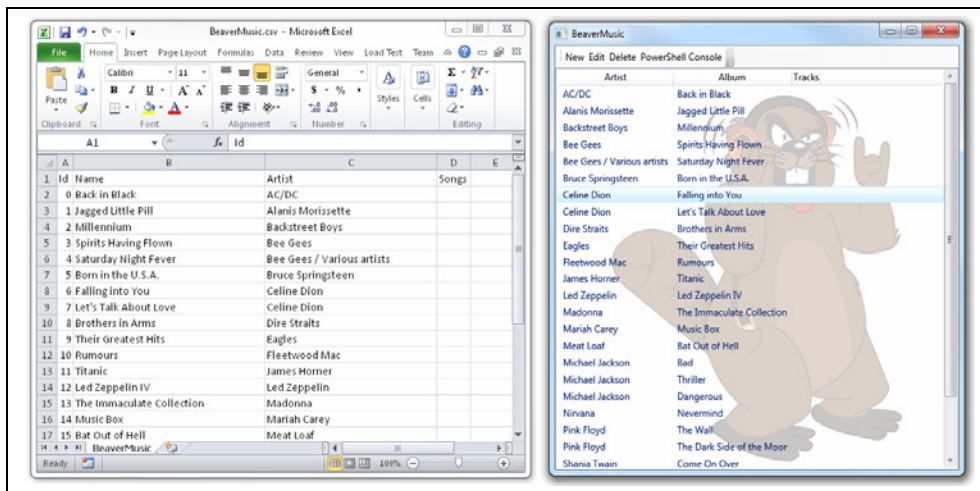
```
Import-Default
Export-Excel
```


Figure 5-14. Dumping live data to Excel

Interacting with MEF

MEF is Microsoft's Managed Extensibility Framework; it's a composition layer for .NET that improves the flexibility, maintainability and testability of applications. The principle purpose of MEF is extensibility; to serve as a 'plug-in' framework for when the developer of the application and the developer of the plug-in are different and have no particular knowledge of each other beyond a published interface library.

Another problem space MEF addresses that's different from the usual IoC suspects, and one of MEFs strengths, is [extension] discovery. It has a lot of, well, extensible discovery mechanisms that operate on metadata you can associate with extensions.

```
$MEFHelper.GetMEFCatalog.Parts | Select DisplayName
$Contract = "BeaverMusic.UI.Shell.AlbumListViewModel"
$MEFHelper.GetExport($Contract).NewAlbumCommand.Execute($null)
```

I've injected a C# instance of MEFHelper and tied it to the PowerShell variable MEFHelper. MEFHelper is a C# instance which has a few methods,for example, the GetExport() takes a contractName and has this implementation.

```
return ExportProvider.GetExport<object>(contractName).Value;
```

Using this and the other methods, I can discover what MEF Parts are in the catalog, then retrieve a live instance from the MEF catalog and act on it. Here I'm looking for the Album List View Model; from there I know I can get at the command that launches the new album dialog window, Figure 5-15.

This opens doors to providing an extensive automatable infrastructure for an application. Plus, being mindful of the development of the static components will enable them to seamlessly work in PowerShell.

I find developing my .NET components with an eye towards PowerShell integration actually helps me create a better designed infrastructure.
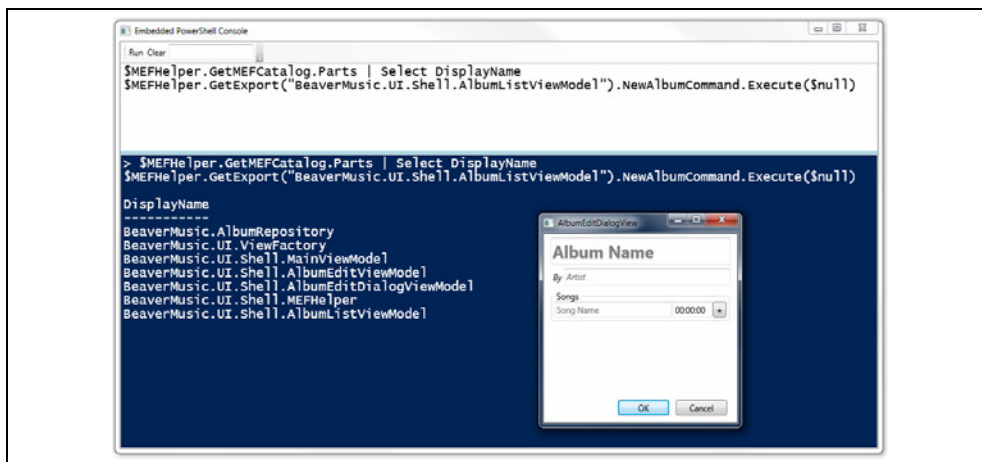

Figure 5-15. Returning MEF Catalog Parts

Discover the Executable Commands

So I want to find all of the commands I can execute on the AlbumListViewModel. Using a combination of the $MEFHelper and PowerShell's Get-Member, I can find all the properties whose name ends in Command using the wildcard *Command. This is how I found the NewAlbumCommand in the previous example in the live running application. Calling the Execute() method then brought up the dialog window where I could enter album details.

```
$MEFHelper.GetExport('BeaverMusic.UI.Shell.AlbumListViewModel') |
    Get-Member -MemberType Property -Name *Command


   TypeName: BeaverMusic.UI.Shell.AlbumListViewModel


Name                     MemberType
----                     ----------
```

```
DeleteAlbumCommand        Property
EditAlbumCommand          Property
NewAlbumCommand           Property
PowerShellConsoleCommand  Property
```

The discovery and application doesn't end here. I could go further and call methods and properties on the dialog to set default information, wrapping that in a PowerShell function is in essence creating a macro for that part of the system. Providing this for end users can really open up productivity.

Show-NewAlbumDialog

I want to wrap all that in a function which handles the contract name and the MEF interaction. Plus, it is the start of a little vocabulary for my application, Figure 5-16.

```
function Show-NewAlbumDialog {
    $contractName = "BeaverMusic.UI.Shell.AlbumListViewModel"
    $MEFHelper.GetExport($contractName).NewAlbumCommand.Execute($null)
}
```
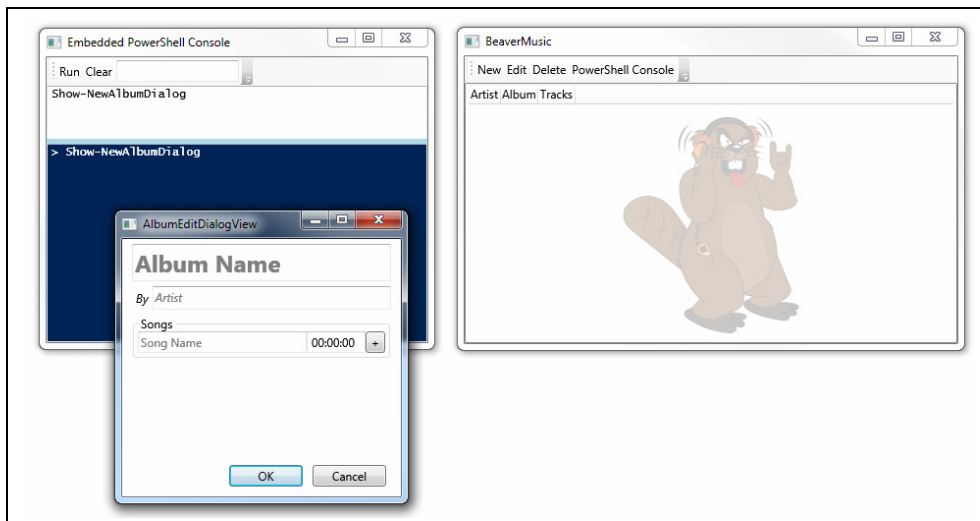

Figure 5-16. Using Show-NewAlbumDialog

Performance Counters

I've only scratched the surface of PowerShell's reach. In this example, I'll tap into the Windows performance counters using `Get-PrivateBytes`, which wraps `Get-Counter` (a built-in PowerShell cmdlet), showing the amount of memory the application is using before and after I retrieve music information from Yahoo using YQL in the query.

```
"Private Bytes before loading albums from Yahoo $(Get-PrivateBytes)"

Clear-Album
Get-YahooMusic | New-Album | Add-Album

"Private Bytes after loading albums from Yahoo $(Get-PrivateBytes)"
```

Results

```
Get-YahooMusic | New-Album | Add-Album

"Private Bytes after loading albums from Yahoo $(Get-PrivateBytes)"
Private Bytes before loading albums from Yahoo 93548544
Private Bytes after loading albums from Yahoo 95088640
```

It's not a far leap from here to exercising code paths in your application and gathering metrics about memory, CPU, disk activity and more.

I've added the code for `Get-PrivateBytes` and `Get-YahooMusic` for reading convenience.

Get-PrivateBytes

```
function Get-PrivateBytes {
    $counterName="\Process($(Get-CurrentProcessName))\Private Bytes"
    (Get-Counter $counterName).CounterSamples |
        Select -Expand CookedValue
}
```

`Get-Counter` comes with PowerShell and it is able to retrieve performance counter data, the same that you'd see in PerfMon. I grab the `CounterSamples` property and then expand the `CookedValue` property. This is an extremely useful way to execute different code paths of your application and then measuring it.

Get-YahooMusic

`Get-YahooMusic` uses the `WebClient` in the .NET Framework so I can download a string via the Yahoo API using their YQL to query music data. It is returned as XML, no problem for PowerShell, and I transform that result into an array of objects with the correct properties, `Artist` and `Name` and then pipe it directly into the application.

```
function Get-YahooMusic {
    $wc = New-Object Net.WebClient
    $url = "http://query.yahooapis.com/v1/public/yql?q=select * from
music.release.popular"
    [xml]$xml = $wc.DownloadString($url)

    $xml.query.results.Release |
        ForEach {
            New-Object PSObject -Property @{
                Artist=$_.artist.name
                Name=$_.Title
            }
        }
}
```

Once you get going with this approach, it is really powerful the reach you have for data acquisition. No C# needed!

```
Get-YahooMusic | New-Album | Add-Album
```
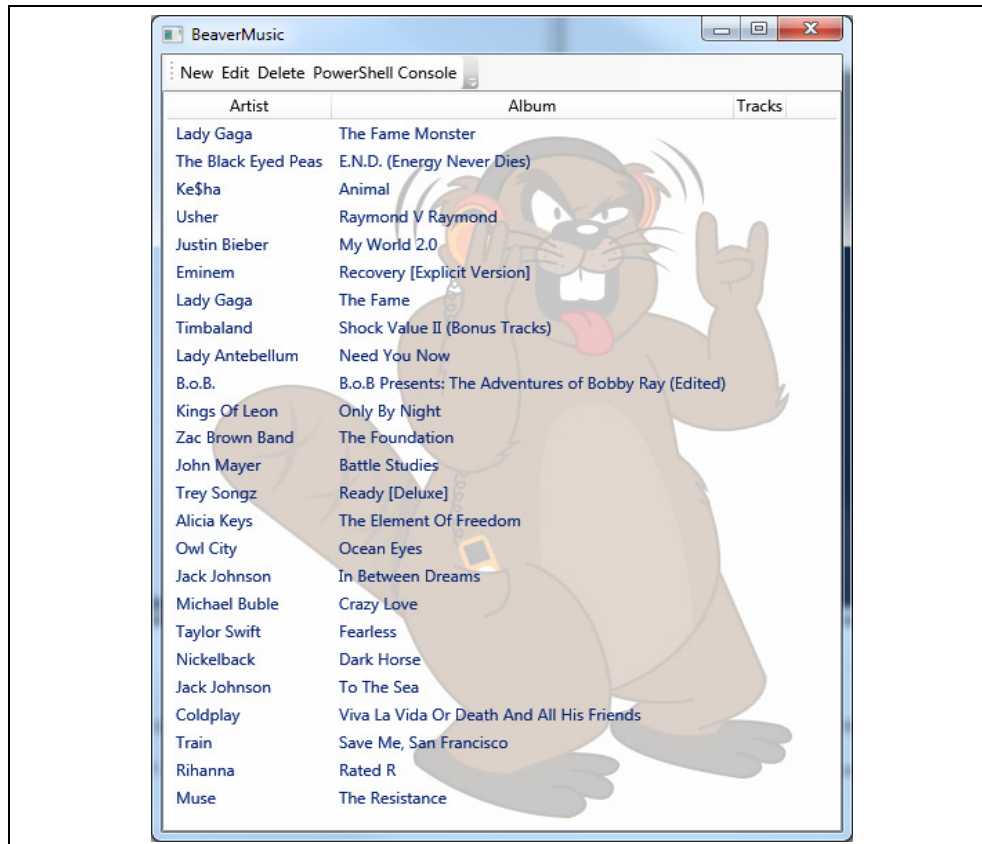
Figure 5-17. Get-YahooMusic Results

From here I can go wild with the Yahoo interface. I can easily add parameters to the function to be pass to the YQL to subset the data. Plus, I can quickly set up other functions that use different YQL to retrieve other types of music details.

Wiring a TextBox to Execute PowerShell Code

I've included the Chinook sample XML music file along with two PowerShell function `Get-ChinookData` and `Import-BearverMusic`. You can find the Chinook Database on CodePlex. Run `Get-ChinookData` and it will pull the information from the XML file and display it as albums in the console. You can also filter the data by artist name. The filtering uses a match, so you don't need to be exact. Piping this to the `Import-BeaverMusic` function will clear the albums in the main window list view and add the new albums, Figure 5-18.

Here is the function `Get-ChinookData`. It uses the PowerShell XML accelerator to create an `XmlDocument` from the contents of the file ChinookData.xml. Rather than using the built-in Get-Content cmdlet I use the `ReadAllLines()` method on the `IO.File` namespace. This is a faster way to read a file.

Also, I cache the data by checking for the global variable `$global:ChinookData`. In the last line of the script I filter the data by artist using the `Where` cmdlet. Note, if `$artist` is not specified, all of data is returned.

```
function Get-ChinookData ($artist) {
    if(!$global:ChinookData) {
        [xml]$global:ChinookData =
            [IO.File]::ReadAllLines("$pwd\ChinookData.xml")
    }

    if(!$global:artists) {

        $global:artists = @{}
        $global:ChinookData.ChinookDataSet.Artist |
            Foreach {
                $artists.($_.ArtistId)= $_.Name
            }
    }

    $(ForEach($item in $global:ChinookData.ChinookDataSet.Album) {
        New-Album $item.Title $artists.($item.ArtistId)
    }) | Where {$_.Artist -match $artist}
}
```

Here I'm looking for any artist with the word Kiss in it.
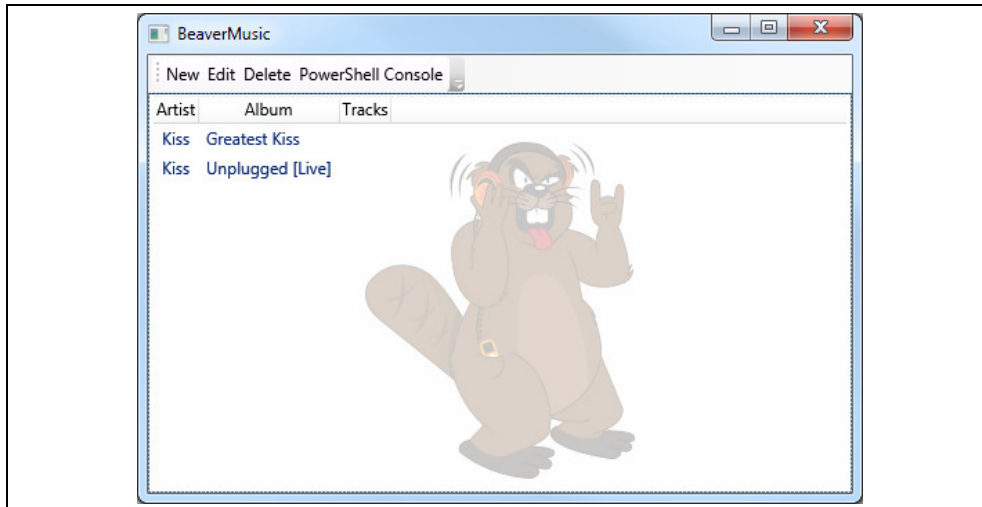
```
Get-ChinookData Kiss | Import-BeaverMusic
```


Figure 5-18. Using Get-ChinookData to import albums

Do it in the PreviewKeyDown

Once I've written PowerShell scripts and tested them like this, it'd be a shame if I could only use them at the console. I could really reduce the test matrix by not having to re-implement reading the XML and adding the albums to the repository in C# code. Plus I could save a lot of time reusing those PowerShell functions directly in the application.

```
private void Artist_PreviewKeyDown(
    object sender, System.Windows.Input.KeyEventArgs e)
{
    if (e.Key == Key.Enter)
    {
        var script = "Get-ChinookData "
```

```
                + Artist.Text + " | Import-BeaverMusic";

        script.ExecutePS();
        e.Handled = true;
    }
}
```

This is a bit of C# code that reacts to keystrokes in the textbox that sits in the PowerShell Console. When the enter key is pressed, I construct PowerShell command as a string.

```
var script = "Get-ChinookData "
             + Artist.Text + " | Import-BeaverMusic";
```
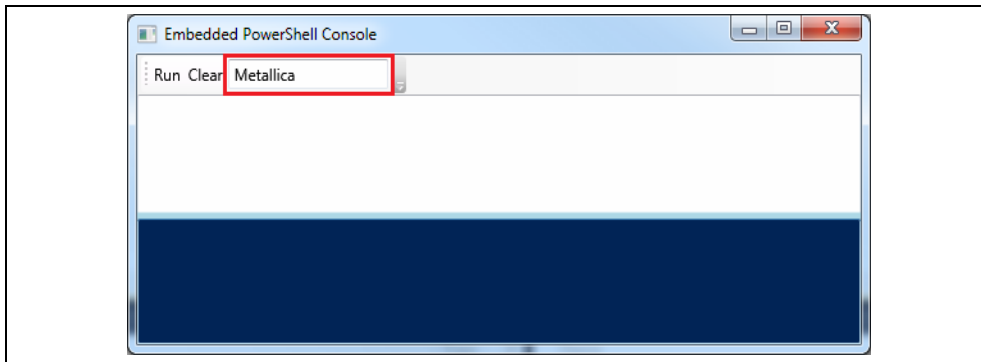

Figure 5-19. Executing PowerShell from a WPF Textbox

After entering Metallica, and pressing enter, the `script` variable look like this.

```
"Get-ChinookData Metallica | Import-BeaverMusic"
```

This line of C# code calls the extension method `ExecutePS()` and the results are as if you typed it all in the PowerShell Console and the main window is updated with the results.
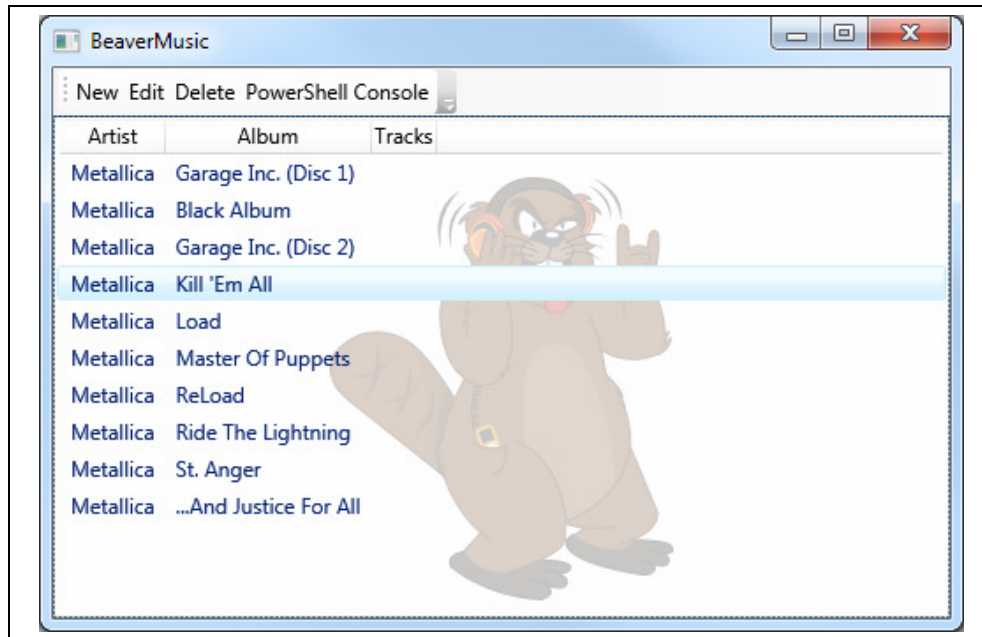
```
script.ExecutePS();
```

Figure 5-20. Results - Executing PowerShell from a WPF Textbox

This is a fantastic and simple way to expose PowerShell functionality in your .NET application. There is so much more that can be done here. I'll leave it to the reader for now.

Running Script and Debugging the C#

Here's a neat trick. Open the `BeaverMusic.sln`, navigate to the BeaverMusic Project and edit the `AlbumRepository.cs`. Set a breakpoint in the `GetAlbums()` method. Run the application, launch the PowerShell console type `Get-Album` in the script pane and press F5.

You'll hit the break after running the script and be in the live running application. You'll be able to step through the C# code, inspect variables and view the call stack just as you would expect. That is too cool!

This is very powerful. You can create scripts that quickly put the application into a reproducible state and then debug it. When bugs are reported, you can potentially email PowerShell scripts around that repro the bugs; this is much more reliable than reading bug reports.
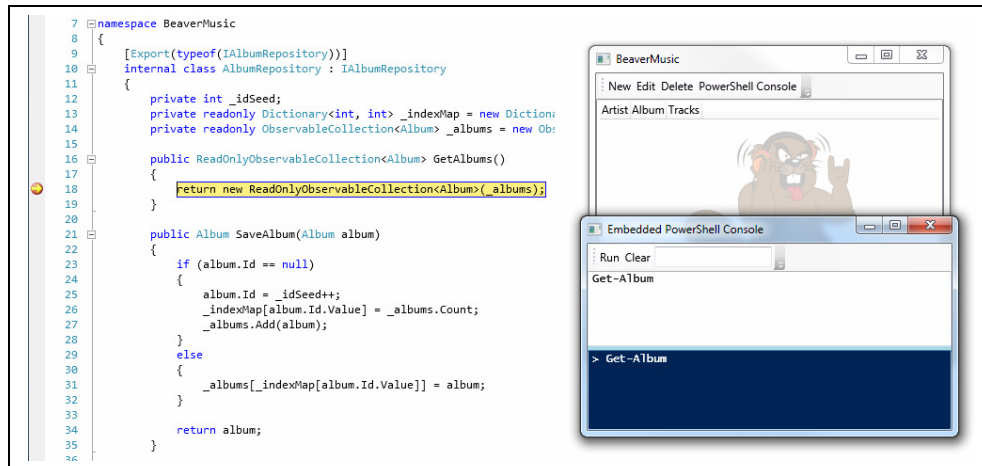
Figure 5-21. Debugging in the Beaver Music App

How do I get the PowerShell Console in My App?

```
PSConsole _console;
public void LaunchPowerShellConsole()
{
    PSConfig.AddVariable("AlbumRepository", _albumRepository);
    PSConfig.Profile = "BeaverMusicProfile.ps1";

    _console = new PSConsole();
    _console.Closing +=
    new System.ComponentModel.CancelEventHandler(
      (w, e) => _console = null
    );

    _console.Show();
}
```

It's pretty easy. First, download the code. You can add this method to your app. Then either remove or customize the `PSConfig.AddVariable` and `PSConfig.Profile` statements.

PSConfig.Profile

Use this to define the name of the PowerShell script that will be stored `$profile`. The console looks to see if this file exists and evaluates the script content in the context of the session.

PSConfig.AddVariable

```
PSConfig.AddVariable("AlbumRepository", _albumRepository);
```

That's how I inject the object model of my application into the PowerShell session. I use

`_albumRepository` which is the instantiated object and I am tying it to the PowerShell variable name `$AlbumRepository`. In the launched console, I can access it with `$AlbumRepository`. Earlier in the chapter, Figure 5-3, I showed how you can inspect the methods of this object at run time. You can inject any kind and any number of

variables into the PowerShell session using the `AddVariable()` method. Plus, you can name them whatever you'd like and once inside the PowerShell console you can access them by prefixing the name with a `$`.

The PowerShell Console Code

I'm not going to do a narrative on the code that supports the PowerShell console and its configuration. It's about 125 lines, a couple pages of code. I'm including here as a reference.

PS.cs

```csharp
namespace EmbeddedPSConsole
{
    public static class PS
    {
        public static string ExecutePS(this string script)
        {
            var sb = new
                StringBuilder(string.Format("> {0}\r", script));

            powerShell.AddScript(script);
            powerShell.AddCommand("Out-String");
            powerShell.AddParameter("Width", 133);

            try
            {
                var results = powerShell.Invoke();
                if (powerShell.Streams.Error.Count > 0)
                {
                    foreach (var err in powerShell.Streams.Error)
                    {
                        AddErrorInfo(sb, err);
                    }
                    powerShell.Streams.Error.Clear();
                }
                else
                {
                    foreach (var item in results)
                    {
                        sb.Append(item);
                    }
                }
            }
            catch (System.Exception ex)
            {
                sb.Append(ex.Message);
            }

            powerShell.Commands.Clear();
            return sb.ToString();
        }

        static PowerShell _powerShell;

        static PowerShell powerShell
        {
```

```csharp
            get
            {
                if (_powerShell == null)
                {
                    _powerShell = PowerShell.Create();
                    powerShell.Runspace = PSConfig.GetPSConfig;
                    if (!string.IsNullOrEmpty(PSConfig.Profile) &&
File.Exists(PSConfig.Profile))
                    {
                        var script =
                          File.ReadAllText(PSConfig.Profile);
                        _powerShell.AddScript(script);
                        _powerShell.Invoke();
                        powerShell.Commands.Clear();
                    }
                }

                return _powerShell;
            }
        }

        private static void AddErrorInfo(StringBuilder sb,
                                         ErrorRecord err)
        {
            sb.Append(err.ToString());
            sb.AppendFormat("\r\n   +{0}",
                err.InvocationInfo.PositionMessage);
            sb.AppendFormat("\r\n   + CategoryInfo          :{0}",
                err.CategoryInfo);
            sb.AppendFormat("\r\n   + FullyQualifiedErrorId :{0}",
                err.FullyQualifiedErrorId.ToString());
            sb.AppendLine();
        }
    }
}
```

PSConfig.cs

```csharp
namespace EmbeddedPSConsole
{
    public class PSConfig
    {
        private static string _profile;
        private static Runspace _rs;

        public static Runspace GetPSConfig { get { return rs; } }

        public static string Profile
        {
            get
            {
                return _profile;
            }
            set
            {
                _profile = value;
```

```
                AddVariable("profile",
System.IO.Path.Combine(Environment.CurrentDirectory, _profile));
                PS.ExecutePS("$a='Executes so the profile is loaded.'");
            }
        }

        private static Runspace rs
        {
            get
            {
                if (_rs == null)
                {
                    _rs = RunspaceFactory.CreateRunspace();
                    _rs.ThreadOptions =
                        PSThreadOptions.UseCurrentThread;
                    _rs.Open();

                    return _rs;
                }
                return _rs;
            }
        }

        public static void AddVariable(string name, object value)
        {
            rs.SessionStateProxy.SetVariable(name, value);
        }
    }
}
```

Parting Thoughts

- How would you implement coloring each row of music by a specific artist, via script?

- Find

Summary

So that's the walkthrough of the Beaver Music application. Surfacing the internals of you application provides numerous benefits to you, your team and your client. And because PowerShell is based on .NET, this approach is virtually seamless.

Providing a scripting language for an application is not a new idea. Perhaps you've hear of Visual Basic for Applications? This is built into most Microsoft Applications. In Microsoft Excel for example, you can record activities and then save the VBA scripts for later.

Companies like Autodesk, world leader in 3D design software, also offered VBA as an embedded scripting language for their product.

In the gaming industry, Lua is the scripting language used by World of Warcraft for Interface Customization.

Why is PowerShell preferred over say VBA, IronPython or IronRuby? These are excellent choices for their dynamic capabilities for example and PowerShell is as programmable.

PowerShell is tuned differently. A simple example being, Get-ChildItem | Sort LastWriteTime –Descending | Select FullName. Mirroring this in the other languages is a a challenge, extending it as simply as it can be done in PowerShell, difficult at best.

Plus PowerShell provides common functions like sorting/filtering/grouping/formatting/outputting and more. In addition to PowerShell's growing integration with the rest of the Windows Platform, as PowerShell grows, so does your application.

<div style="text-align:right;color:red;font-size:3em;">6</div>

PowerShell and the Internet

PowerShell interacts really well with the Web, being able to access files, XML, JSON , web services and more, directly from the Internet. PowerShell does not have cUrl or GNU Wget type support out of the box, but because PowerShell is an amazing glue language that is deeply integrated with the .NET Framework, this is one place where PowerShell capabilities really shines, in connecting a set of powerful underlying components. Plus, PowerShell v3 makes this easier using the cmdlets Invoke-WebRequest, Invoke-RestMethod, ConvertTo-Json and ConvertFrom-Json.

It's interesting to note, even though PowerShell was envisioned over a decade ago and v2 was delivered in 2009, PowerShell is able to keep pace with daily development needs.

Taking advantage of something like JSON (JavaScript Object Notation), a lightweight data-interchange format, over the web using is easy using .NET libraries designed to parse this format and presenting it in a way consumable by PowerShell.

In this chapter I'll show code that will let you pull down different formatted information from websites. The amount of public information available is enormous. Contributed by individuals, companies and governments creating huge datasets and giving us potential insight into a myriad of things and is easily accessible via PowerShell.

Net.WebClient

One cool PowerShell demo I like to give is to show how to pull down the details of a blog's RSS feed, in three lines of code.

```
$url  = "http://feeds.feedburner.com/DevelopmentInABlink"
$feed = (New-Object Net.WebClient).DownloadString($url)
([xml]$feed).rss.channel.item | Select title, pubDate
```

This simple code gives us the following:

```
title                                              pubDate
-----                                              -------
Using PowerShell to solve Project Euler: Problem 1 Sun, 08 Jan
PowerShell and IEnumerable<T>                      Sat, 24 Dec
```

```
PowerShell, Windows Azure and Node.js                  Sat, 17 Dec
How to find the second to last Friday in December – Usin... Sat, 17 Dec
PowerShell – Using the New York Times Semantic Web APIs  Sun, 04 Dec
My First PowerShell V3 ISE Add-on                       Sun, 04 Dec
Use PowerShell V3 to Find Out About Your Twitter Followers  Thu, 24 Nov
```

Using the `Net.WebClient` class from the .NET Framework, the `DownloadString()` method retrieves the RSS as a string. Next, using the PowerShell XML accelerator, `[xml]`, the data in the `$feed` variable is transformed into an XmlDocument and I dot notate over it to get to the item details. Piping this to `Select`, I pull out just the Title and PubDate.

Wrap that code in a PowerShell Function

Good PowerShell script discipline is to wrap snippets like these in functions. It helps organize your code and makes them composable.

```
function Get-WebData {

    param([string]$Url, [Switch]$Raw)

    $wc   = New-Object Net.WebClient
    $feed = $wc.DownloadString($Url)

    if($Raw) { return $feed }

    [xml]$feed
}
```

`Get-WebData` takes two parameters the `$Url`, which is the resource on the site you're hitting and `$Raw`. If you don't specify `$Raw`, `Get-WebData` tries to accelerate the string returned from the as an `XmlDocument`.

```
$url = "http://feeds.feedburner.com/DevelopmentInABlink"
(Get-WebData $url).rss.channel.item |
    select title, pubDate
```

This code is a simplified version from unlike the one from the beginning of this chapter. PowerShell v3 adds a number of new functions. Later in this chapter, under the heading "Invoke-RestMethod" we'll see one of the new functions the PowerShell team added that obsoletes a function like `Get-WebData` and has more capability.

Reading CSV formatted data from the Web

Retrieving the contents of a file containing data in the CSV format requires the `-Raw` parameter on the `Get-WebData` function.

```
$url = "http://dougfinke.com/PowerShellForDevelopers/albums.csv"
(Get-WebData $url -Raw) | ConvertFrom-Csv
```

Results

```
Artist                          Name
------                          ----
Michael Jackson                 Thriller
AC/DC                           Back in Black
Pink Floyd                      The Dark Side of the Moon
Whitney Houston / Various artists The Bodyguard
```

```
Meat Loaf                    Bat Out of Hell
Eagles                       Their Greatest Hits
Various artists              Dirty Dancing
Backstreet Boys              Millennium
```

The results can then be piped to the PowerShell Cmdlet `ConvertFrom-Csv` which transforms that data into an array of PowerShell objects with properties.

Reading XML formatted data from the Web

Retrieving the contents of a file containing XML is easy using the `Get-WebData` function.

```
$url = "http://dougfinke.com/PowerShellForDevelopers/albums.xml"
(Get-WebData $url).albums.album
```

I dot notate through nodes in the results transforms that data into an array of PowerShell objects with properties.

The Structure of XML data

Here is a snippet of the Xml I'll read. Notice the structure. Now map it to the dot notation I used in the PowerShell script. I can loop through all the data simply using `albums.album`.

```
<albums>
  <album>
    <artist>Michael Jackson</artist>
    <name>Thriller</name>
  </album>
  <album>
    <artist>AC/DC</artist>
    <name>Back in Black</name>
  </album>
<albums>
```

Results

Here is the transformed Xml into PowerShell objects.

```
Artist                       Name
------                       ----
Michael Jackson              Thriller
AC/DC                        Back in Black
```

US Government Data Sources

Here I'm hitting the US Consumer Product Safety Commission site and pulling down product recall information stored in an Xml format.

```
$url = "http://www.cpsc.gov/cpscpub/prerel/prerel.xml"
(Get-WebData $url).rss.channel.item
```

Data acquisition couldn't be easier.

```
title       : Uni-O Industries Recalls O-Grill Portable Gas Grills
description : The regulator on the grill can leak gas which can ignite
pubDate     : Tue, 03 Jan 2012 16:00:00 GMT
link        : http://www.cpsc.gov/cpscpub/prerel/prhtml12/12077.html
```

The US Government has an entire index of publicly available data accessible through both web services and XML. Google around for other free public resources, if you can think of it, someone has put it on the internet.

Invoke-RestMethod

```
$url = "http://dougfinke.com/PowerShellForDevelopers/albums.csv"
Invoke-RestMethod $url | ConvertFrom-Csv
```

`Invoke-RestMethod` is a new PowerShell v3 cmdlet. It simplifies how you can work with the web. `Invoke-RestMethod` is available to you without having to dot source other scripts or importing a module. That means, you can deliver a script to another user who has PowerShell v3 installed and you're good to go. Using `Get-WebData`, you need to either deliver the extra script file or copy or paste that code into scripts you distribute. Plus, you then own the `Get-WebData` function, testing, enhancing and upgrading.

But wait, there's more. In the next two sections, I take advantage of `Invoke-RestMethod`'s `-ReturnType` parameter which defaults to `Detect`.

Detecting XML

`Invoke-RestMethod` does a lot for us. Like the `(New-Object Net.WebClient).DownloadString()`, it retrieves the content file. Then it goes a step further, auto detecting that it is Xml and returns an `XmlDocument`. The `-ReturnType` takes three values, `Detect`, `Xml` and `Json`.

```
$url = "http://dougfinke.com/PowerShellForDevelopers/albums.xml"
(Invoke-RestMethod $url).albums.album
```

If you know the type of data you're going after, you can short-circuit the detection process.

Detecting JSON

Let's retrieve the same album data, except it's stored in JSON format. This is the same approach as retrieving the Xml data. This time through, Invoke-RestMethod detects the JSON format and automatically converts the JSON into an array of objects of type PSCustomObject with the properties Artist and Name.

```
$url = "http://dougfinke.com/PowerShellForDevelopers/albums.js"
Invoke-RestMethod $url
```

Here a sample of the returned data. This output will be identical whether the XML or JSON format is returned. Thinking it through, this means I can meld the same data across multiple web sites with different formats and produced a uniform output. Pretty powerful!

```
Artist                            Name
------                            ----
Michael Jackson                   Thriller
AC/DC                             Back in Black
Pink Floyd                        The Dark Side of the Moon
Whitney Houston / Various artists The Bodyguard
Meat Loaf                         Bat Out of Hell
Eagles                            Their Greatest Hits
Various artists                   Dirty Dancing
```

```
Backstreet Boys                      Millennium
Bee Gees / Various artists           Saturday Night Fever
Fleetwood Mac                        Rumours
Shania Twain                         Come On Over
```

I've walked through some of the key building blocks for interacting with the data on the web. These data interchange formats are universal. What follows are more interesting applications of the same approach.

PowerShell and the NYT Semantic Web API

With the New York Times Semantic API, you get access to the long list of people, places, organizations and other locations, entities and descriptors that make up the controlled vocabulary used as metadata by The New York Times (sometimes referred to as Times Tags and used for Times Topics pages).

```
Get-SemanticNYT "Obama" |
    Get-SemanticNYTArticles |
    Where links |
        Select -ExpandProperty article_list |
        Select -ExpandProperty results |
        Select date, title, url |
        Out-GridView
```


This script only works in PowerShell v3. I am retrieving the information in JSON format from using the semantic APIs and `Invoke-RestMethod`.

Reading the New York Times – Part 1

```
function Get-SemanticNYT {

    param($query = "obama")

    $uri = "http://api.nytimes.com/svc/semantic/v2/"+
        "concept/search.json?query=$query&api-key=$apiKey"

    (Invoke-RestMethod $uri).results
}
```

Reading the New York Times – Part 2

```
function Get-SemanticNYTArticles {

    param(
        [Parameter(ValueFromPipelineByPropertyName=$true)]
        $concept_name,
        [Parameter(ValueFromPipelineByPropertyName=$true)]
        $concept_type
    )

    Process {
      $uri = "http://api.nytimes.com/svc/semantic/v2/" +
      "concept/name/$concept_type/$concept_name.json?&" +
      "fields=all&api-key=$apiKey"

      (Invoke-RestMethod $uri).results
    }
}
```

The two PowerShell v3 functions used, `Get-SemanticNYT` and `Get-SemanticNYTArticles`, are simple wrappers used to construct NYT Urls correctly. These are passed to the `Invoke-RestMethod` Cmdlet that does the heavy lifting of connecting to the site, pulling down the JSON and transforming it to PowerShell arrays.

`Get-SemanticNYTArticles` makes use of `ValueFromPipelineByPropertyName` and the `Process Block`.

`ValueFromPipelineByPropertyName` indicates that the parameter can take values from a property of the incoming pipeline object that has the same name as this parameter. This means there is a property called `concept_name` and `concept_type` emitted from the `Get-SemanticNYT` function. When I pipe `Get-SemanticNYT` to `Get-SemanticNYTArticles` I leverage PowerShell's parameter binding mechanism. This is one of the essence enabling features of PowerShell. Each item from `Get-SemanticNYT` is automatically passed through the pipeline, and the properties `concept_name` and `concept_type` are bound to the same-named parameters in `Get-SemanticNYTArticles`.

The `Process Block` handles iterating over the data piped, doing the move next and checking for end of stream. This frees me up to create solutions and worry less about the mechanics of passing parameters properly.

Summary

In less than three quarters of a page of PowerShell v3 code I'm querying the New York Times web based articles via their Semantic API, handling a Web REST interaction, transforming JSON to PowerShell (.NET) objects and finally displaying it in a WPF GUI.

Powerful components a developer can easily add to their toolbox.

Stock WebService

There are many sites available that provide stock quotes. This means I need to navigate to the page, type in the symbol press enter and the read the information. What if I want to check several symbols? What if I check stocks every few minutes? Maybe I want to save the stock information details? Even better, I want to do some quick calculations on the

fly. I'm going to use a Web Service to get this done. Web services are typically application programming interfaces (API) or Web APIs that are accessed via Hypertext Transfer Protocol (HTTP) and executed on a remote system hosting the requested services. Web services tend to fall into one of two camps: big Web services and RESTful Web services.

```powershell
function Get-Quote {
    param(
        [Parameter(ValueFromPipeline=$true)]
        [string[]]$symbol,
        [Switch]$Raw
    )

    Begin {
        $url = "http://www.webservicex.net/stockquote.asmx?wsdl"
        $proxy = New-WebServiceProxy $url
    }

    Process {
        $result = $ proxy.GetQuote($symbol)

        if($Raw) { return $result }

        [xml]$result
    }
}

"IBM", "AAPL", "GM", "GE", "MSFT", "GOOG" |
    Get-Quote |
    ForEach {$_.StockQuotes.Stock} |
    Format-Table
```

In this example I am easily retrieveing data for several stock symbols in a single call.

The `New-WebServiceProxy`, inside the `Begin Block` executes only the first time through the function, creates a Web service proxy object that lets you use and manage the Web service in Windows PowerShell.

Then in the `Process Block`, executed for each item in the pipeline, the `GetQuote()` method is called, passing in the `$symbol`. `GetQuote` returns and Xml data source, so using the `[xml]` accelerator, an `XmlDocument` is returned for each symbol that is located.

Dig a little deeper

`New-WebServiceProxy` creates a Web service proxy object that lets you use and manage the Web service in Windows PowerShell. It goes out, reads the WSDL (Web Service Definition Language) and on the fly generates/compiles an object that represents all the methods and parameters that you can access for that service.

I used the `GetQuote()` method and it takes a *symbol*, a string. For example *IBM*, and returns an Xml string containing lots of good information about that stock symbol.

Here is the shape of the Xml returned by `GetQuote()` method is a `Stock` node inside a `StockQuotes` node.

```
<StockQuotes>
    <Stock>
        <Symbol>IBM</Symbol>
        <Last>193.35</Last>
        <Date>2/7/2012</Date>
        <Time>4:01pm</Time>
        <Change>+0.53</Change>
        <Open>192.45</Open>
        <High>194.14</High>
        <Low>191.97</Low>
        <Volume>3432953</Volume>
        <MktCap>224.3B</MktCap>
        <PreviousClose>192.82</PreviousClose>
        <PercentageChange>+0.27%</PercentageChange>
        <AnnRange>151.71 - 194.90</AnnRange>
        <Earns>13.06</Earns>
        <P-E>14.76</P-E>
        <Name>International Bus</Name>
    </Stock>
</StockQuotes>
```

Then pipe the Xml to ForEach to pull out the actual data from `$_.StockQuotes.Stock`.

Symbol	Last	Date	Time	Change	Open
IBM	180.52	1/19/2012	4:02pm	-0.55	181.79
AAPL	427.75	1/19/2012	4:00pm	-1.36	430.03
GM	24.82	1/19/2012	4:00pm	+0.31	24.65
GE	19.15	1/19/2012	4:00pm	+0.13	19.03
MSFT	28.12	1/19/2012	4:00pm	-0.11	28.15
GOOG	639.57	1/19/2012	4:00pm	+6.66	640.97

Being able to get a proxy to a web service in a single line of PowerShell enables many scenarios. For example, quick integration testing, here you could easily query stock symbols with known values and test for to see if they are correct. Don't forget, once the data is pulled from the web service and in the pipeline you can pipe it or transform it to another data format and save it to disk for use in other ways.

Invoke-WebRequest – Another PowerShell Cmdlet

This Cmdlet is another workhorse for integrating the web into PowerShell. It lets you grab web pages and, for example, through the AllElements property you can search for HTML Elements with a certain class name.

Again `Invoke-WebRequest` is available out of the box with PowerShell v3. That means you can write scripts that mash up, scrape and do significant text manipulation of any of your favorite web sites. This means capturing and scrubbing data is a simple operation.

Next up I present a couple scripts using this technique to query Google and Bing about the status of a flight. What is really cool is how few lines of code is needed to get this done. Thinking it through, the composability of PowerShell can really light the way for useful interesting applications.

One key addition to PowerShell v3 is the workflow keyword. Underneath it is using Microsoft Workflow 4.0. In addition, the `ForEach` sprouts a new parameter in this

context, `-Parallel`. Gluing together Parallel workflow and easy web integration makes for a powerful mix of data acquisition.

PowerShell & Google

Flight Status for **Delta Air Lines 269**

On-time			arrives in 25 minutes
Departure	**JFK** New York	**8:04am** (was 8:05am) Jan 20	**Terminal 3** Gate 3
Arrival	**ATL** Atlanta	**10:41am** (was 10:45am) Jan 20	**Terminal N** Gate C51
On-time			departs in 7 hours 24 minutes
Departure	**TLV** Tel Aviv-Yafo	**12:40am** Jan 21	**Terminal 3** Gate B7
Arrival	**JFK** New York	**5:50am** Jan 21	**Terminal 4** Gate B22

Updated 3 minutes ago by flightstats.com - Details

I want to find out the flight status for Delta Air Lines flight 269, I surf to Google and type "flight status for dl 269".

```
function Get-FlightStatus {
    param($query="dl269")

    $url = "https://www.google.com/search?q=flight status for $query"
    $result = Invoke-WebRequest $url
    $result.AllElements |
        Where Class -eq "obcontainer" |
        Select -ExpandProperty innerText
}

Get-FlightStatus
```

Here I type `Get-FlightStatus` at a command line and scrape the Google Page using `Invoke-WebRequest`. I truncated these results for readability.

```
Flight Status for Delta Air Lines 269

On-timearrives in 25 minutes
DepartureJFK8:04am(was 8:05am)Terminal 3
New YorkJan 20Gate 3

Updated 3 minutes ago by flightstats.com - Details
```

The key to scraping pages this way is to find an element that can be as close to uniquely identified as possible. By navigating to the page you want to scrape and doing a "view source" you can look at the resulting HTML and figure out if that is possible. Looking at the results from Google, I saw that the "Flight Results" were in a `div` with a `class` name `obcontainer`. That translates `Where Class -eq "obcontainer"`.

The Target HTML

Using Invoke-WebRequest with the Where cmdlet makes quick work of scraping web sites. Here is the HTML I searched to find a *class* name equal to *obcontainer*.

```
<div class="obcontainer" style="padding-bottom:5px;">
  <div>
    <div>
      <table style="width:34.24em;border-top:0"
        <tr>
          <td >Flight Status for <b>Delta Air Lines 269</b></td>
        </tr>
      </table>
    </div>
    <div>
      <table >
        <tr>
          <td>Updated 3 minutes ago by flightstats.com – <a href=
          class=" fl">Details</a></td>
        </tr>
      </table>
    </div>
  </div>
</div>
```

So, you retrieve the web page with Invoke-WebRequest, filter AllElements by the "key" you are looking for, select the innerText and you're done.

Not all web pages will be this simple but it is worth a few minutes of investment to potentially unlock a data mining opportunity.

PowerShell & Bing

```
function Get-FlightStatus {
    param($query="dl269")

    $url = "http://bing.com?q=flight status for $query"

    $result = Invoke-WebRequest $url

    $result.AllElements |
        Where Class -eq "ans" |
        Select -First 1 -ExpandProperty innerText
}
```

This book is about a Microsoft technology so here is the same query in Bing. The two differences are the "key" to filter on in the Where Cmdlet and you need to do use –First 1 parameter in the Select because Bing returns several answers and the first one is what we want.

```
Flight status for Delta 269
Landed early · Jan 20, 2012
From: New York (JFK) 08:04 AM (was 08:05 AM) · gate 3, terminal 3 · map
To: Atlanta (ATL) 10:33 AM (was 10:45 AM) · gate C51, terminal N · map
Other flight segments · TLV–JFK

Data provided by Bing Travel · Source: www.flightstats.com, 2 minutes ago
```

Overall a very clean and simple approach for querying search engines and pulling out just the details you need.

The good news is, it is not limited to just query engines. It is any public data on web.

PowerShell & the Twitter API

```
. .\Get-WebData.ps1
$result = Get-WebData "http://search.twitter.com/search.rss?q=PowerShell"
$result.rss.channel.item |
    Select title, author
```

Twitter is an information network and communication mechanism that produces more than 200 million tweets a day. The Twitter platform offers access to that data, through their APIs. Each API represents a facet of Twitter, and allows developers to build upon and extend their applications in new and creative ways.

Tapping into the Twitter search API and searching for one of my favorite topics, PowerShell and leveraging the Get-Webdata function I presented earlier, I easily extract the title and author of the tweets containing the word PowerShell.

```
title                           author
-----                           ------
I heart #Powershell. What else ... awanderingmind@twitter.com (Jo...
I hate you people. No, not you.... billinkc@twitter.com (Bill Fel...
#PowerShell Mailbox name not al... ihunger@twitter.com (Jim Hofer)
nothing like writing #PowerShel... Josh_Atwell@twitter.com (Josh ...
#PowerShell Granting permission... ihunger@twitter.com (Jim Hofer)
NewPost:: PowerShell, Active Se... jbmurphy@twitter.com (Jeffrey ...
Configure Git in PowerShell So ... JohnBubriski@twitter.com (John...
Article #5 of 7 for Hey Scripti... proxb@twitter.com (Boe Prox)
Get Powershell to wait for an S... stackfeed@twitter.com (StackOv...
RT @PowerShellGroup: UK PowerSh... OliverZofic@twitter.com (Olive...
GPP Registry Item Level Targeti... AGoodies@twitter.com (A Goodies)
wadehel is windows powershell m... pimapimapima@twitter.com (Adri...
RT @PowerShellGroup: UK PowerSh... ScriptingGuys@twitter.com (MSF...
RT @toenuff: Revert the  #power... ScriptingGuys@twitter.com (MSF...
The future of Exchange administ... alexandair@twitter.com (Aleksa...
```

The resulting Xml returned by the Twitter search API is far richer than these two fields. It contains a link to the image the author uses, the date it was tweeted, a link to the original tweet and more. Plus, this is only the search API. Twitter supports much more, check out my blog post Use PowerShell V3 to Find Out About Your Twitter Followers.

Many web sites support similar APIs and I strongly encourage you investigating PowerShell as a way to rapidly tap into them, opening opportunities to quickly mine data from a single source or across many others.

PowerShell v3 ups the game further by natively supporting Cmdlets like Invoke-WebRequest and Invoke-RestMethod that let me concentrate on the essence of data interaction across heterogeneous data stores on the web.

Unlike ceremonial versions of web interaction, where I need to handle requests, responses, data conversions and more. Using Invoke-RestMethod, I pass a Url and if it is Xml or JSON on the other end, I don't even know it. I'm simply working with an array of objects with properties, piping them to other PowerShell Cmdlets for sorting, grouping, slicing, dicing or using the intermediate results to do lookups through other APIs or on completely different sites.

Summary

We've covered a lot in this chapter. We saw how easy it is to user PowerShell and the Internet, pulling down the contents of files, in three different formats CSV, XML and JSON. Then, we converted them on the fly to .NET (PowerShell) Objects and did some analysis on files. Finally, we pulled down entire web pages and filtering out key details based on HTML Tag names.

Now, you have to check out the next chapter. I'm going to expand on the Twitter code and introduce you to WPF programming using only PowerShell, that's right no XAML, no C#. See you there.