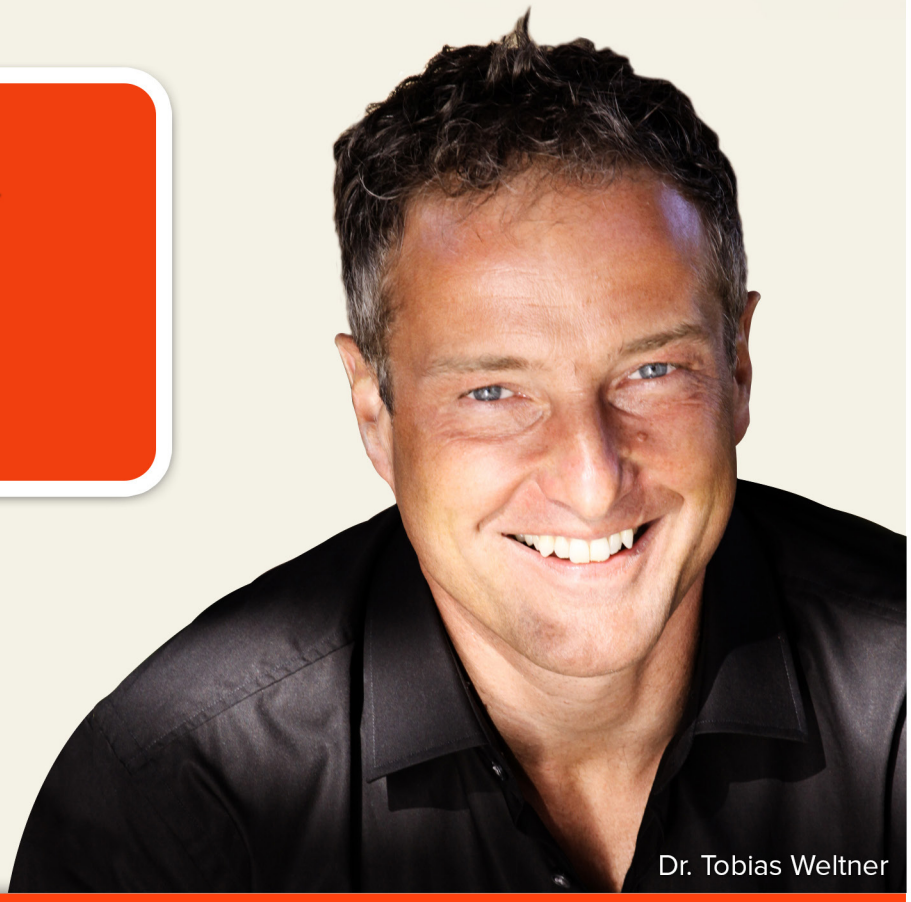
PowerTips MONTHLY

Part of the PowerShell.com reference library, brought to you by **Dr. Tobias Weltner**

Volume 11 April 2014

This Month's Topic:

XML-Related Tasks



Dr. Tobias Weltner

Table of Contents

1. XML-Related Cmdlets

Handling XML data is greatly simplified by these built-in PowerShell cmdlets:

Cmdlet	Purpose
`New-Object -TypeName XML`	Create new XML object that can load XML data from file, text, or Internet
`Select-Xml`	Select information from an XML object using the XML query language XPath
`New-WebServiceProxy`	Retrieve XML-based information from a web service
`Export-Clixml`	Export objects to XML
`Import-Clixml`	Imports XML object descriptions and recreates objects
`ConvertTo-Xml`	Converts objects to XML representations

In this document, you will find plenty of examples on how to use these cmdlets.

2. XPath

XPath is a built-in XML query language that you can use to find data inside XML documents. You will find plenty of example on how to use XPath in this document as well.

Here is a quick XPath refresher, providing you with the most important XPath details you should know. To understand XPath, you need to know what a “node” is, and what an “attribute” is.

A “node” is a “tag” in XML. Remember that XML is case-sensitive. This would define a node named “machine”:

```
<machine>
</machine>
```

An “attribute” is a piece of information embedded in a node. This would add an attribute “description” to the node “machine”:

```
<machine description=”some description”>
</machine>
```

XPath can query for information inside an XML document. These are the “selectors” you can use to find nodes and attributes:

Selector	Purpose
Nodename	Selects all nodes with the name submitted
/	Selects from the root node
//	Selects from the current node on, anywhere in the document
.	Selects the current node
..	Selects parent of current node
@	Selects an attribute

XPath supports these operators:

Operator	Purpose
|	Combines two node sets: //servers | //workstations
+	Addition
-	Subtraction
*	Multiplication
div	Division
=	Equal
!=	Not equal
<	Less than
<=	Less or equal
>	Greater than
>=	Greater or equal
or	Logical OR
and	Logical AND
mod	Modulus (division remainder)

You can further refine XPath queries by using functions. XPath 1.0 supports these functions (this is not a complete list. It is a list of the most commonly used functions):

Function	Purpose
`last()`	Position of last node in set
`position()`	Position of current node in set
`count(nodeset)`	Number of nodes in node set
`starts-with(string1, string2)`	Checks whether string1 starts with string2
`contains(string1, string2)`	Checks whether string1 contains string2
`substring(string, number1, number2)`	Returns the part of the string that starts at position “number1” and is “number2” characters long
`translate(string1, string2, string3)`	Replaces characters in string1. All characters in string2 are replaced by the characters in string3.

Again, you will find examples for all of these functions in this document.

3. Creating New XML Files Manually

XML data is plain text, so you could use a simple text editor or cmdlets such as Set-Content to construct XML data manually.

This would define XML data content and load it into an XML object:

```
$data = @'
<xml>
  <machines>

    <machine>
      <name>Server 1</name>
      <ip>10.10.12.100</ip>
    </machine>

    <machine>
      <name>Server 2</name>
      <ip>10.10.12.105</ip>
    </machine>

    <machine>
      <name>Server 3</name>
      <ip>10.10.12.210</ip>
    </machine>

  </machines>
</xml>
'@

$xml = New-Object -TypeName XML
$xml.LoadXml($data)
```

Author Bio

Tobias Weltner is a long-term Microsoft PowerShell MVP, located in Germany. Weltner offers entry-level and advanced PowerShell classes throughout Europe, targeting mid- to large-sized enterprises. He just organized the first German PowerShell Community conference which was a great success and will be repeated next year (more on www.pscommunity.de).. His latest 950-page “PowerShell 3.0 Workshop” was recently released by Microsoft Press.

To find out more about public and in-house training, get in touch with him at tobias.weltner@email.de.

And this is how you would access the XML data:

```
PS> $xml.xml.machines.machine

name                                                                 ip
----                                                                 --
Server 1                                                             10.10.12.100
Server 2                                                             10.10.12.105
Server 3                                                             10.10.12.210
```

Basically, ".xml.machines.machine" represents the nodes inside your XML data that need to be traversed to get to the data sets.

PowerShell cmdlets and parameters are generally not case-sensitive. XML however is case-sensitive. It makes a difference whether you capitalize an XML node name or not. If you manually construct XML data and casing does not match, you will get errors. So if you had capitalized a starting tag but not capitalized the corresponding ending tag, then XML would treat both as different:

```
$data = @'
<Xml>
  (...)
</xml>
'@
```

The result would look similar to this:

```
Exception calling "LoadXml" with "1" argument(s): "The 'XML' start tag on line 1 position 2
does not match the end tag of 'xml'. Line
20, position 3."
At line:27 char:1
+ $xml.LoadXml($data)
+ ~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [], MethodInvocationException
    + FullyQualifiedErrorId : DotNetMethodException
```

While it is perfectly fine to construct XML data manually, it requires you to obey all XML syntax rules manually, too.

4. Creating New XML Files Programmatically

XML is picky (it is case-sensitive, for example, and requires strict starting and ending tags), so chances are that manually created XML data will contain syntactical errors.

That's why a more solid approach uses a special object that is designed to create well-formed XML.

This piece of code creates a hypothetical inventory report (the file created here will be used by other examples in this document):

```
# this is where the document will be saved
$Path = "$env:temp\inventory.xml"

# get an XMLTextWriter to create the XML
$XmlWriter = New-Object System.XML.XmlTextWriter($Path,$Null)

# choose a pretty formatting
$xmlWriter.Formatting = 'Indented'
$xmlWriter.Indentation = 1
$XmlWriter.IndentChar = "`t"

# write the header
$xmlWriter.WriteStartDocument()

# set XSL statements
$xmlWriter.WriteProcessingInstruction("xml-stylesheet", "type='text/xsl' href='style.xsl'")

# create root element "machines" and add some attributes to it
$XmlWriter.WriteComment('List of machines')
$xmlWriter.WriteStartElement('Machines')
$XmlWriter.WriteAttributeString('current', $true)
$XmlWriter.WriteAttributeString('manager', 'Tobias')

# add a couple of random entries
for($x=1; $x -le 10; $x++)
{
  $server = 'Server{0:0000}' -f $x
  $ip = '{0}.{1}.{2}.{3}' -f  (0..256 | Get-Random -Count 4)

  $guid = [System.GUID]::NewGuid().ToString()

  # each data set is called "machine", add a random attribute to it
  $XmlWriter.WriteComment("$x. machine details")
  $xmlWriter.WriteStartElement('Machine')
  $XmlWriter.WriteAttributeString('test', (Get-Random))

  # add three pieces of information
  $xmlWriter.WriteElementString('Name',$server)
  $xmlWriter.WriteElementString('IP',$ip)
  $xmlWriter.WriteElementString('GUID',$guid)

  # add a node with attributes and content
  $XmlWriter.WriteStartElement('Information')
  $XmlWriter.WriteAttributeString('info1', 'some info')
  $XmlWriter.WriteAttributeString('info2', 'more info')
  $XmlWriter.WriteRaw('RawContent')
  $xmlWriter.WriteEndElement()

  # add a node with CDATA section
  $XmlWriter.WriteStartElement('CodeSegment')
  $XmlWriter.WriteAttributeString('info3', 'another attribute')
  $XmlWriter.WriteCData('this is untouched code and can contain special characters /\@<>')
  $xmlWriter.WriteEndElement()
```

Technical Editor Bio

Aleksandar Nikolic, Microsoft MVP for Windows PowerShell, a frequent speaker at the conferences (Microsoft Sinergija, PowerShell Deep Dive, NYC Techstravaganza, KulenDayz, PowerShell Summit) and the co-founder and editor of the PowerShell Magazine (http://powershellmagazine.com). He is also available for one-on-one online PowerShell trainings. You can find him on Twitter: https://twitter.com/alexandair

```
  # close the "machine" node
  $xmlWriter.WriteEndElement()
}

# close the "machines" node
$xmlWriter.WriteEndElement()

# finalize the document
$xmlWriter.WriteEndDocument()
$xmlWriter.Flush()
$xmlWriter.Close()

notepad.exe $path
```

Attention: The XMLTextWriter object takes care of correct matching case and XML structure, but it is your responsibility to construct valid XML content. The XMLTextWriter writes anything you want, including invalid XML element names.

So when you use WriteStartElement(), make sure you are submitting only one word (no spaces or special characters), or else you could end up with invalid XML content after all.

```
# this is where the XML sample file was saved
$Path = "$env:temp\inventory.xml"

# load it into an XML object
$xml = New-Object -TypeName XML
$xml.Load($Path)
# note: if your XML is malformed, you will get an exception here
# always make sure your node names do not contain spaces

# simply traverse the nodes and select the information you want
$Xml.Machines.Machine | Select-Object -Property Name, IP
```

You may see approaches like the next one–don't do that. It is much slower than the previous approach:

```
# this is where the XML sample file was saved
$Path = "$env:temp\inventory.xml"

# load it into an XML object
[XML]$xml = Get-Content $Path
$Xml.Machines.Machine | Select-Object -Property Name, IP
```

5. Loading RSS Feeds and other Internet Data

An XML object can download XML data from URLs, too. RSS feeds typically deliver pure XML data, so you can tap any RSS feed and then select the data you are interested in.

```
$url = 'http://sxp.microsoft.com/feeds/3.0/ServerTools/ServerToolsTopPosts'

$XML = New-Object XML
$XML.Load($url)

$XML.rss.channel.item |
  Select-Object -Property MobileTitle, Description, LastUpdated, Link |
  Out-GridView
```

Note that “rss.channel.item” are simply the nested nodes that need to be traversed to get to the data you want to display. Have a look at your XML raw data to find out what the names of nodes are. They can be different for RSS feeds and XML files.

6. Accessing Internet XML Data through Proxy

Loading Internet-based XML data via an XML object may fail if you do not have direct Internet access.

XML objects do not support anything fancy like a proxy servers or authentication. If you need to go through a proxy, or authenticate, you cannot directly download the data with an XML object.

Instead, use Invoke-WebRequest and its rich options for specifying a proxy and/or credentials. Attention: While Invoke-WebRequest can load almost anything from the Internet, this code example expects XML data. So make sure the URL you are using is in fact returning XML data. Do not use plain HTML pages. HTML cannot be loaded into an XML object.

Once this cmdlet has downloaded the data for you, feed it into an XML object.

```
$url = 'http://sxp.microsoft.com/feeds/3.0/ServerTools/ServerToolsTopPosts'
$request = Invoke-WebRequest -Uri $url -UseBasicParsing

$xmlText = $request.Content

$XML = New-Object XML
$XML.LoadXml($xmlText)

$XML.rss.channel.item | Select-Object -Property MobileTitle, Description, LastUpdated, Link |
  Out-GridView
```

Note that this example is not using a proxy nor is it using credentials. Simply add the appropriate parameters to Invoke-WebRequest.

Note also how the raw content retrieved from the URL is stored in $xmlText and then loaded into the XML object using LoadXML() (not Load()).

7. Selecting XML Data

When you have loaded XML data into an XML object, you may want to pick data. If you use the hypothetical server inventory created earlier, then you may want to find the IP address for a given server.

You can simply use PowerShell pipeline cmdlets to extract the information:

```
# this is where the XML sample file was saved
$Path = "$env:temp\inventory.xml"

# load it into an XML object
$xml = New-Object -TypeName XML
$xml.Load($Path)

$Xml.Machines.Machine |
  Where-Object { $_.Name -eq 'Server0009' } |
  Select-Object -Property IP, {$_.Information.info1}
```

The result may look similar to this:

```
PS> $Xml.Machines.Machine |
  Where-Object { $_.Name -eq 'Server0009' } |
  Select-Object -Property IP, {$_.Information.info1}


IP                                                  $_.Information.info1
--                                                  --------------------
33.50.113.105                                       some info
```

While this approach is perfectly OK, it is not very efficient, and search options are limited. A much more efficient approach is using the XML query language XPath to find the information you are after.

9. Using XPath to Search for XML Nodes

To find all nodes named “Machine” that have a “Name” subnode of “Server0009”, this would be the appropriate XPath query:

```
# this is where the XML sample file was saved
$Path = "$env:temp\inventory.xml"

# load it into an XML object
$xml = New-Object -TypeName XML
$xml.Load($Path)

$item = Select-Xml -Xml $xml -XPath '//Machine[Name="Server0009"]'
$item.Node
```

```
PS> $item.Node


test        : 2074887351
Name        : Server0009
IP          : 33.50.113.105
GUID        : 3eeaed7e-7397-4bfb-9e09-0d957e4b57b0
Information : Information
CodeSegment : CodeSegment
```

You could now use Select-Object to select the pieces of information you need:

```
PS> $item.Node | Select-Object -Property Name, IP

Name       IP
----       --
Server0009 33.50.113.105
```

When you look at the original XML, then PowerShell has translated this piece of XML information into an object:

```
PS> $item.Node.InnerXML
<Name>Server0009</Name>
<IP>33.50.113.105</IP>
<GUID>3eeaed7e-7397-4bfb-9e09-0d957e4b57b0</GUID>
<Information info1="some info" info2="more
 info">RawContent
</Information>
<CodeSegment info3="another attribute"><![CDATA[this is untouched code and can contain
special characters
 /\@<>]]>
</CodeSegment>
```

As you can see, the node “Information” has additional attribute information such as “info1” and “info2”. The node “CodeSegment” has text information.

To get to the attribute information or information in subnodes, and include them in your output, use script blocks instead of property names.

```
PS> $item.Node | Select-Object -Property IP, {$_.Information.Info1} , {
$_.CodeSegment.'#cdata-section' }

IP              $_.Information.Info1    $_.CodeSegment.'#cdata-section'
--              --------------------    ---------------------------
33.50.113.105   some info               this is untouched code a...
```

Or, use hash tables to use natural column header text:

```
$infoAttribute = @{
  Name = 'Info1'
  Expression = {$_.Information.Info1}
}

$cData = @{
  Name = 'Code'
  Expression = { $_.CodeSegment.'#cdata-section' }
}


$item.Node | Select-Object -Property IP, $infoAttribute, $cData
```

Now, the columns use the text that you specified in the hash table:

```
$item.Node | Select-Object -Property IP, $infoAttribute , $cData

IP                      Info1                   Code
--                      -----                   ----
33.50.113.105           some info               this is untouched code a...
```

10. Using XPath to Search for Attributes

Attributes are pieces of information that are embedded inside an XML node. Here is an example:

```
<Information info1="some info" info2="moreinfo">
RawContent
</Information>
```

The node “Information” contains two attributes named “info1” and “info2”.

To search for all XML nodes of type “Information” that have an attribute named “info1” with the text “some info”, use this example:

```
# this is where the XML sample file was saved
$Path = "$env:temp\inventory.xml"

# load it into an XML object
$xml = New-Object -TypeName XML
$xml.Load($Path)

$item = Select-Xml -Xml $xml -XPath '//Information[@info1="some info"]'
$item | ForEach-Object {
  $_.Node
}
```

And this would be the result:

```
PS> $item.Count
10


PS> $item | ForEach-Object {
  $_.Node
}


info1                    info2                    #text
-----                    -----                    -----
some info                more info                RawContent
some info                more info                RawContent
some info                more info                RawContent
some info                more info                RawContent
some info                more info                RawContent
some info                more info                RawContent
some info                more info                RawContent
some info                more info                RawContent
some info                more info                RawContent
some info                more info                RawContent
```

Likewise, this would search for all attributes named “info1” anywhere in your XML document, in any node:

```
# this is where the XML sample file was saved
$Path = "$env:temp\inventory.xml"

# load it into an XML object
$xml = New-Object -TypeName XML
$xml.Load($Path)

$item = Select-Xml -Xml $xml -XPath '//@info1'
$item | ForEach-Object {
  $_.Node
}
```

11. Using Advanced Searches

Sometimes, you may want to search for nodes that do not exactly match an expression. You may want to get nodes that start with a given text, for example.

To do advanced searches, you can use built-in XPath functions. XPath functions can search text for matches, convert text to lowercase, analyze node position, and more. Just take a look at the following examples to get a feeling for what you can do with this.

This example identifies all nodes of type “Machine” where the subnode “Name” starts with “Serv” (note again that XML is case-sensitive. This would not match instances where the name starts with “serv”):

```
# this is where the XML sample file was saved
$Path = "$env:temp\inventory.xml"

# load it into an XML object
$xml = New-Object -TypeName XML
$xml.Load($Path)

$item = Select-Xml -Xml $xml -XPath '//Machine[starts-with(Name,"Serv")]'
$item.Node
```

And this would always get the last node of type “Machine”

```
# this is where the XML sample file was saved
$Path = "$env:temp\inventory.xml"

# load it into an XML object
$xml = New-Object -TypeName XML
$xml.Load($Path)

$item = Select-Xml -Xml $xml -XPath '//Machine[last()]'
$item.Node
```

This example would retrieve the first 3 nodes of type “Machine”:

```
# this is where the XML sample file was saved
$Path = "$env:temp\inventory.xml"

# load it into an XML object
$xml = New-Object -TypeName XML
$xml.Load($Path)

$item = Select-Xml -Xml $xml -XPath '//Machine[position()<4]'
$item.Node
```

And this example would find all nodes of type “Machine” where the subnote “Name” contains the text “0009”:

```
# this is where the XML sample file was saved
$Path = "$env:temp\inventory.xml"

# load it into an XML object
$xml = New-Object -TypeName XML
$xml.Load($Path)

$item = Select-Xml -Xml $xml -XPath '//Machine[contains(Name, "0009")]'
$item.Node
```

12. Mastering Case-Sensitivity

XML data is case-sensitive, so you need to be very careful when you create XPath filters. If the casing does not match, then you may not get back the results you were looking for.

This example tries to find a machine with the name “Server0009”, however the casing is incorrect. In the XML file, the name is spelled “Server0009”, but the query looks for “server0009”:

```
# this is where the XML sample file was saved
$Path = "$env:temp\inventory.xml"

# load it into an XML object
$xml = New-Object -TypeName XML
$xml.Load($Path)

$item = Select-Xml -Xml $xml -XPath '//Machine[Name="server0009"]'
$item.Node
```

One easy workaround is to convert the node you are checking to lowercase text, and always make sure you, too, submit a lowercase search term. Unfortunately, XPath by default does not contain a lower-case() function. The only readily available work-around to turn text into lowercase would be to use the translate() function. It replaces characters.

So this would turn regular letters to lowercase (you would have to extend the range to cover special characters as well):

```
translate(Name, 'ABCDEFGHIJKLMNOPQRSTUVWXYZ', 'abcdefghijklmnopqrstuvwxyz')
```

And this example now accepts any lowercase search term and matches it with any node name, case-insensitive (since the node name is “translated” to a lowercase text):

```
# this is where the XML sample file was saved
$Path = "$env:temp\inventory.xml"
```

```
# load it into an XML object
$xml = New-Object -TypeName XML
$xml.Load($Path)

$item = Select-Xml -Xml $xml -XPath "//Machine[translate(Name, 'ABCDEFGHIJKLMNOPQRSTUVWXYZ',
'abcdefghijklmnopqrstuvwxyz')='server0009']"
$item.Node
```

13. Changing XML Data

The information stored in an XML object can be changed and updated. You do not need to care about the underlying complex XML structure. Simply use XPath to find the element you want to change, do the change, and optionally save the changed XML object back to file.

This will load the sample inventory XML data, then look for the server “Server0006”, and add a number of attributes.

```
# this is where the XML sample file was saved
$Path = "$env:temp\inventory.xml"

# load it into an XML object
$xml = New-Object -TypeName XML
$xml.Load($Path)

$item = Select-Xml -Xml $xml -XPath '//Machine[Name="Server0006"]'

$item.node.Name               = 'NewServer0006'
$item.node.IP                 = '10.10.10.12'
$item.node.Information.Info1  = 'new attribute info'

$NewPath = "$env:temp\inventory2.xml"
$xml.Save($NewPath)

notepad.exe $NewPath
```

As you can see, the XML will be saved back to file, and you do not need to worry about well-formed XML. Your change is incorporated into the newly generated XML file.

14. Changing Multiple Instances of XML Data

Batch changes to multiple pieces of information in an XML file is straightforward, too.

You can use a PowerShell pipeline. This would load the XML sample data, then look for any “Machine” node, and change the node name.

After all updates are made, the XML file is saved back to a new file, and now incorporates all changes:

```
# this is where the XML sample file was saved
$Path = "$env:temp\inventory.xml"

# load it into an XML object
$xml = New-Object -TypeName XML
$xml.Load($Path)

foreach ($item in (Select-Xml -Xml $xml -XPath '//Machine'))
{
    $item.node.Name = 'Prod_' + $item.node.Name
}

$NewPath = "$env:temp\inventory2.xml"
$xml.Save($NewPath)

notepad.exe $NewPath
```

By using regular expressions, you can do even much more sophisticated changes. This example would rename all machines to "PCxxx", taking the old server numbers and carrying them over to the new names, turning them into 2-digit numbers:

```
# this is where the XML sample file was saved
$Path = "$env:temp\inventory.xml"

# load it into an XML object
$xml = New-Object -TypeName XML
$xml.Load($Path)

Foreach ($item in (Select-Xml -Xml $xml -XPath '//Machine'))
{
    if ($item.node.Name -match 'Server(\d{4})')
    {
      $item.node.Name = 'PC{0:00}' -f [Int]$matches[1]
    }
}

$NewPath = "$env:temp\inventory2.xml"

$xml.Save($NewPath)

notepad.exe $NewPath
```

15. Appending New Data to Existing XML

Adding new nodes to an existing XML requires you to add nodes that are identical to the existing nodes.

One easy strategy would be to take an existing XML node and clone it.

This would take the sample inventory XML data, pick the first machine node, and clone it.

The cloned node can then be changed, and once done, is inserted into the XML document:

```
# this is where the XML sample file was saved
$Path = "$env:temp\inventory.xml"

# load it into an XML object
$xml = New-Object -TypeName XML
$xml.Load($Path)

# clone an existing node structure
$item = Select-Xml -Xml $xml -XPath '//Machine[1]'
$newnode = $item.Node.CloneNode($true)

# update the information as needed
# all other information is defaulted to the values from the original node
$newnode.Name = 'NewServer'
$newnode.IP = '1.2.3.4'

# get the node you want the new node to be appended to
$machines = Select-Xml -Xml $xml -XPath '//Machines'
$machines.Node.AppendChild($newnode)

$NewPath = "$env:temp\inventory2.xml"

$xml.Save($NewPath)

notepad.exe $NewPath
```

This approach would add the new node always at the end of the XML file. You can, however, add and insert it anywhere.

Here are some example lines that you can use alternatively:

```
# add it to the top of the list
$machines.Node.InsertBefore($newnode, $item.node)


# add it after "Server0007"
$parent = Select-Xml -Xml $xml -XPath '//Machine[Name="Server0007"]'
$machines.Node.InsertAfter($newnode, $parent.node)
```

16. Removing XML Data

To completely remove a dataset from an XML object, select the node you want to remove, and then remove the node from the XML document.

This would take the example inventory data and remove the dataset referring to machine “Server0007”:

```
# this is where the XML sample file was saved
$Path = "$env:temp\inventory.xml"

# load it into an XML object
$xml = New-Object -TypeName XML
$xml.Load($Path)

# remove "Server0007"
$item = Select-Xml -Xml $xml -XPath '//Machine[Name="Server0007"]'
$null = $item.Node.ParentNode.RemoveChild($item.node)
```

17. Persisting Objects with XML

You can save objects to disk by turning their inner structure into XML. This is known as “dehydrating” or “serializing” an object. The object will be disconnected from its original source. You can then transfer the file, and “rehydrate” or “deserialize” anytime you want.

This code will take a list of services, and serialize them to file:

```
$path = "$env:temp\services.xml"

Get-Service | Export-Clixml -Path $path
```

To view the autogenerated XML, open the file in notepad:

```
PS> notepad $path

PS>
```

To “deserialize” the file, read it in again:

```
PS> Import-Clixml -Path $path

Status   Name               DisplayName
------   ----               -----------
Running  AdobeARMservice    Adobe Acrobat Update Service
Stopped  AeLookupSvc        Application Experience
Stopped  ALG                Application Layer Gateway Service
(...)
```

This type of serialization occurs automatically behind the scene when you run commands via PowerShell Remoting or transfer information from a background job.

You could use the “deserialized” data to compare system snapshots. For example, to check whether service status has changed between the time when you serialized the service list, and now, try this:

```
$snapshot1 = Import-Clixml -Path $path
$snapshot2 = Get-Service

Compare-Object -ReferenceObject $snapshot1 -DifferenceObject $snapshot2 -Property Name,
Status
```

When you get back nothing, then there are no changes. Else, you get back each change with a “side indicator”, indicating in which snapshot the change occurred.

18. Finding Type Definitions Using XPath

PowerShell enhances many .NET types and adds additional information. These changes are defined in XML files.

To list all .NET types that get enhanced by default, you can use Select-Xml and an XPath query:

```
PS> Select-Xml $env:windir\System32\WindowsPowerShell\v1.0\types.ps1xml -Xpath /Types/Type/
Name | Select-Object -expand Node | Sort-Object '#text'

#text
-----
Deserialized.Microsoft.PowerShell.Commands.Internal.Format.FormatInfoData
Deserialized.System.Diagnostics.Process
Deserialized.System.Enum
Deserialized.System.Globalization.CultureInfo
Deserialized.System.Management.Automation.Breakpoint
Deserialized.System.Management.Automation.BreakpointUpdatedEventArgs
Deserialized.System.Management.Automation.DebuggerCommand
(...)
```

19. Formatting XML Files

XML content does not necessarily have to be pretty. Pretty XML however makes it easier for humans to read the data.

By loading XML content into an XML object, you can ask PowerShell to prettify it and add indentation. Here is a function called Format-XML that takes the path to an existing XML file and asks for an output file.

It then reads the input file and writes a prettified version to the destination file. Optionally, the function will open the generated file.

```
function Format-Xml  {
  param
  (
    [Parameter(Mandatory=$true)]
    $PathXML,

    $Indent=2,

    $Destination ="$env:temp\out.xml",

    [switch]
    $Open
  )
```

```
  $xml = New-Object XML
  $xml.Load($PathXML)

  $StringWriter           = New-Object System.IO.StringWriter
  $XmlWriter              = New-Object System.XML.XmlTextWriter $StringWriter
  $xmlWriter.Formatting   = 'indented'
  $xmlWriter.Indentation  = $Indent

  $xml.WriteContentTo($XmlWriter)

  $XmlWriter.Flush()
  $StringWriter.Flush()

  Set-Content -Value ($StringWriter.ToString()) -Path $Destination

  if ($Open) { notepad.exe $Destination }
}
```

And this is how you'd use it:

```
PS> Format-Xml -PathXML $env:windir\Starter.xml -Open -Indent 1

PS> Format-Xml -PathXML $env:windir\Starter.xml -Open -Indent 4

PS>
```

Because of the –Open parameter, the result is opened automatically in Notepad, and you can see the results of the different indentation settings.

20. Output Scheduled Tasks to XML

Scheduled tasks are internally described by XML. You can use schtasks.exe to dump the XML definition of any scheduled task. This can be useful to look at all scheduled task settings, or to change the XML and then re-import the scheduled task.

```
function Export-ScheduledTask {
  param(

    $TaskName = $(schtasks.exe /QUERY /FO CSV |
      ConvertFrom-CSV |
      Out-GridView -Title 'Select Task' -PassThru |
      Select-Object -ExpandProperty TaskName
    ),

    $XMLOutputPath = "$env:temp\scheduledtask.xml",

    [Switch]
    $Open
  )

  schtasks.exe /QUERY /TN $TaskName /XML | Out-File $XMLOutputPath
  if ($Open)
  {
    notepad.exe $XMLOutputPath
  }
}
```

To export a scheduled task, run the function like this for example:

```
PS> Export-ScheduledTask -Open

PS>
```

Since you did not submit a task name, a grid view window will open and show all scheduled tasks available. Select one. The task is exported to file, and the file is opened in Notepad. Now you can see all task settings. And you can also save the file and use it to create this particular scheduled task on other machines.

21. Creating Scheduled Tasks from XML

If you have an XML definition for a scheduled task, then here is a simple function that reads this information in and creates the scheduled task from the XML definition.

```
function Import-ScheduledTask {
  param(
    [Parameter(Mandatory=$true)]
    $Jobname,

    [Parameter(Mandatory=$true)]
    $Path,

    $ComputerName=$null
  )

  if ($ComputerName -ne $null) {
    $option = "/S $ComputerName"
  } else {
    $option = ''
  }
  schtasks.exe /CREATE /TN $jobname /XML $path $option
}
```

You can use this technique to clone a scheduled task to multiple machines, or you can first export a scheduled task to XML, then adjust all the advanced settings inside the XML file, and finally re-import the scheduled task from the adjusted XML file.

23. Getting Weather Forecast from an Airfield near You

PowerShell can access web services and automatically retrieve information such as weather forecasts. Information emitted by web services is typically delivered as XML data.

New-WebServiceProxy does all the work for you, and you only need to submit the web service URL. Note however that New-WebServiceProxy requires direct Internet access and cannot work over proxy servers.

This piece of code returns the names of all airfield weather stations in the United States:

```
$weather = New-WebServiceProxy -Uri http://www.webservicex.com/globalweather.asmx?WSDL
$data = $weather.GetCitiesByCountry('United States')

$cities = [xml]$data

$cities.NewDataSet.Table |
  Select-Object -ExpandProperty City
```

This is what the list may look like:

```
PS> $cities.NewDataSet.Table |
  Select-Object -ExpandProperty City
Claiborne Range, Airways Facilit
Payson
Custer, Custer County Airport
Andover, Aeroflex-Andover Airport
Nogales Automatic Meteorological Observing System
Elkhart / Elkhart-Morton County
Saint Johnsbury
Salmon
White Sands
(...)
```

The data returned by the web service is casted to [XML]. $cities now contains an XML object, and you can traverse its nodes until you get to the list of nodes you are after.

Next, you can specify one of the airfields to get current weather information:

```
$weather = New-WebServiceProxy -Uri http://www.webservicex.com/globalweather.asmx?WSDL
$data = [xml]$weather.GetWeather('Seattle, Seattle Boeing Field','United States')
$data.CurrentWeather
```

The result will look something like this:

```
PS> $data = [xml]$weather.GetWeather('Seattle, Seattle Boeing Field','United States')
$data.CurrentWeather


Location         : SEATTLE BOEING FIELD, WA, United States (KBFI) 47-33N 122-19W 4M
Time             : Mar 11, 2014 - 12:53 PM EDT / 2014.03.11 1653 UTC
Wind             :  Calm:0
Visibility       :  10 mile(s):0
SkyConditions    :  mostly clear
Temperature      :  46.0 F (7.8 C)
DewPoint         :  39.9 F (4.4 C)
RelativeHumidity :  79%
Pressure         :  30.6 in. Hg (1036 hPa)
Status           : Success
```

To get the current temperature, for example, you could access the appropriate object property:

```
PS> $data.CurrentWeather.Temperature
 46.0 F (7.8 C)

PS> $data.CurrentWeather.RelativeHumidity
 79%

PS>
```

24. Searching Object Properties

Objects can be extremely complex and consist of many nested properties. So it may not be obvious or easy to find a particular property.

By converting an object to XML, you could use XPath to find all nodes containing the property you are looking for. So XML can be used to analyze objects.

Simply use ConvertTo-Xml to turn an object into an XML object:

```
PS> $host | ConvertTo-Xml

xml                                                Objects
---                                                -------
version="1.0"                                      Objects


PS> $object = $host | ConvertTo-XML

PS> $object.InnerXml
```

```
<?xml version="1.0"?><Objects><Object Type="System.Management.Automation.Internal.Host.
InternalHost"><Property Name="Name" Type="System.
String">Windows PowerShell ISE Host</Property><Property Name="Version" Type="System.Version">4.0</
Property><Property Name="InstanceId" T
ype="System.Guid">e3ed5df7-ae36-4c1f-9378-f2333159f690</Property><Property Name="UI" Type="System.
Management.Automation.Internal.Host.In
ternalHostUserInterface"><Property Type="System.String">System.Management.Automation.Internal.Host.
InternalHostUserInterface</Property><
/Property><Property Name="CurrentCulture" Type="Microsoft.Windows.PowerShell.Gui.Internal.
ISECultureInfo">de-DE</Property><Property Name
="CurrentUICulture" Type="Microsoft.Windows.PowerShell.Gui.Internal.ISECultureInfo">en-US</
Property><Property Name="PrivateData" Type="M
icrosoft.PowerShell.Host.ISE.ISEOptions"><Property Type="System.String">Microsoft.PowerShell.Host.
ISE.ISEOptions</Property><Property Nam
e="Window" Type="System.Management.Automation.PSNoteProperty">Microsoft.Windows.PowerShell.Gui.
Internal.MainWindow</Property></Property>
<Property Name="IsRunspacePushed" Type="System.Boolean">False</Property><Property Name="Runspace"
Type="System.Management.Automation.Run
spaces.LocalRunspace"><Property Type="System.String">System.Management.Automation.Runspaces.
LocalRunspace</Property></Property></Object>
</Objects>

PS>
```

As you can see, ConvertTo-Xml automatically traverses all nested object properties and turns them into XML. Next, you can use XPath to find any property you want inside the object. Use the –Depth parameter to tell ConvertTo-Xml how deeply you want to traverse the inner object structure.

Here is a ready-to-use function named “Get-ObjectProperty” that encapsulates all of this and makes it easy for you to search for properties inside of objects.

```
Function Get-ObjectProperty
{
  param
  (
    $Name = '*',
    $Value = '*',
    $Type = '*',
    [Switch]$IsNumeric,

    [Parameter(Mandatory=$true,ValueFromPipeline=$true)]
    [Object[]]$InputObject,

    $Depth = 4,
    $Prefix = '$obj'
  )

  Begin
  {
    $x = 0
    Function Get-Property
    {
      param
      (
        $Node,
        [String[]]$Prefix
      )

      $Value = @{Name='Value'; Expression={$_.'#text' }}
      Select-Xml -Xml $Node -XPath 'Property' | ForEach-Object {$i=0} {
        $rv = $_.Node | Select-Object -Property Name, $Value, Path, Type
        $isCollection = $rv.Name -eq 'Property'

        if ($isCollection)
        {
          $CollectionItem = "[$i]"
          $i++
          $rv.Path = (($Prefix) -join '.') + $CollectionItem
```

```
        }
        else
        {
          $rv.Path = ($Prefix + $rv.Name) -join '.'
        }

        $rv

        if (Select-Xml -Xml $_.Node -XPath 'Property'  )
        {
          if ($isCollection)
          {
            $PrefixNew = $Prefix.Clone()
            $PrefixNew[-1] += $CollectionItem
            Get-Property -Node $_.Node -Prefix ($PrefixNew )
          }
          else
          {
            Get-Property -Node $_.Node -Prefix ($Prefix + $_.Node.Name )
          }
        }
      }
    }
  }

  Process
  {
    $x++
    $InputObject |
    ConvertTo-Xml -Depth $Depth |
    ForEach-Object { $_.Objects } |
    ForEach-Object { Get-Property $_.Object -Prefix $Prefix$x  } |
    Where-Object { $_.Name -like "$Name" } |
    Where-Object { $_.Value -like $Value } |
    Where-Object { $_.Type -like $Type } |
    Where-Object { $IsNumeric.IsPresent -eq $false -or $_.Value -as [Double] }
  }
}
```

Here are some examples. For example, to find all properties in the $host PowerShell object that refer to colors, try this:

```
$host | Get-ObjectProperty -Depth 2 -Name *color* | Out-GridView
```

Or, to find all numeric properties contained in process objects, try this:

```
Get-Process -Id $pid | Get-ObjectProperty -Depth 5 –IsNumeric
```

Note how –Depth controls how many nested levels you want to traverse. Use –Depth 1 or –Depth 2 unless you suspect that the property you are looking for is deeply nested.

You can even search for properties containing a specific data type. This would find all properties in service objects that are string:

```
Get-Service -Name spooler | Get-ObjectProperty -Type System.String
```

25. Adding Custom Methods to Types

You can teach PowerShell to add members to object types by default. This is done by the Extended Type System (ETS). There are many ways to do this. One is to create an XML file with the extension ps1xml, and read the file into the extended file system.

To add a new method called Words() to every object of type string, launch Notepad and enter this:

```
<?xml version="1.0" encoding="utf-8"?>
<Types>
  <Type>
    <Name>System.String</Name>
```

```
    <Members>
      <ScriptMethod>
        <Name>Words</Name>
        <Script>
$this.Split()
</Script>
      </ScriptMethod>
    </Members>
  </Type>
</Types>
```

Save it as “myExtension.ps1xml”, and then load it into the PowerShell ETS:

```
PS> Update-TypeData myExtension.ps1xml

PS>
```

Immediately, every new string has a new method called Words():

```
PS> $text = "This is a text"

PS> $text.Words()
This
is
a
text

PS> $text.Words().Count
4

PS>
```

Note that beginning in PowerShell 3.0, Update-TypeData can add type extensions on the fly, too, without the need for an external XML file:

```
PS> Update-TypeData -MemberType ScriptMethod -MemberName GetWords -Value { $this.Split() }
-TypeName String

PS> "Hello World".GetWords()
Hello
World

PS>
```