# Chapter 14. XML

In today’s world, data is no longer presented in plain-text files. Instead, XML (Extensible Markup Language) has evolved to become a de facto standard because it allows data to be stored in a flexible yet standard way. PowerShell takes this into account and makes working with XML data much easier than before.

**Topics Covered:**

* [Taking a Look At XML Structure](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/06/chapter-14-xml.aspx#taking-a-look-at-xml-structure)
* [Loading and Processing XML Files](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/06/chapter-14-xml.aspx#loading-and-processing-xml-files)
  + [Accessing Single Nodes and Modifying Data](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/06/chapter-14-xml.aspx#accessing-single-nodes-and-modifying-data)
  + [Using SelectNodes() to Choose Nodes](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/06/chapter-14-xml.aspx#using-selectnodes-to-choose-nodes)
  + [Accessing Attributes](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/06/chapter-14-xml.aspx#accessing-attributes)
  + [Adding New Nodes](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/06/chapter-14-xml.aspx#adding-new-nodes)
* [Exloring the Extended Type System](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/06/chapter-14-xml.aspx#exploring-the-extended-type-system)
  + [The XML Data of the Extended Type System](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/06/chapter-14-xml.aspx#the-xml-data-of-the-extended-type-system)
  + [Finding Pre-Defined Views](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/06/chapter-14-xml.aspx#finding-pre-defined-views)

**Taking a Look At XML Structure**

XML uses tags to uniquely identify pieces of information. A tag is a pair of angle brackets like the ones used for HTML documents. Typically, a piece of information is delimited by a start and end tag. The end tag is preceded by "/"; the result is called a "node", and in the next example, the node is called "Name":

<Name>Tobias Weltner</Name>

Nodes can be decorated with attributes. Attributes are stored in the start tag of the node like this:

<staff branch="Hanover" Type="sales">...</staff>

If a node has no particular content, its start and end tags can be combined, and the ending symbol "/" drifts toward the end of the tag. If the branch office in Hanover doesn't have any staff currently working in the field, the tag could look like this:

<staff branch="Hanover" Type="sales"/>

The following XML structure describes two staff members of the Hanover branch office who are working in the sales department.

<staff branch="Hanover" Type="sales">

<employee>

<Name>Tobias Weltner</Name>

<function>management</function>

<age>39</age>

</employee>

<employee>

<Name>Cofi Heidecke</Name>

<function>security</function>

<age>4</age>

</employee>

</staff>

The XML data is wrapped in an XML node which is the top node of the document:

<?xml version="1.0" ?>

This particular header contains a version attribute which declares that the XML structure conforms to the specifications of XML version 1.0. There can be additional attributes in the XML header. Often you find a reference to a "schema", which is a formal description of the structure of that XML file. The schema could, for example, specify that there must always be a node called "staff" as part of staff information, which in turn could include as many sub-nodes named "staff" as required. The schema would also specify that information relating to name and function must also be defined for each staff member.

Because XML files consist of plain text, you can easily create them using any editor or directly from within PowerShell. Let's save the previous staff list as an xml file:

$xml = @'

<?xml version="1.0" standalone="yes"?>

<staff branch="Hanover" Type="sales">

<employee>

<Name>Tobias Weltner</Name>

<function>management</function>

<age>39</age>

</employee>

<employee>

<Name>Cofi Heidecke</Name>

<function>security</function>

<age>4</age>

</employee>

</staff>

'@ | Out-File $env:temp\employee.xml

XML is case-sensitive!

**Loading and Processing XML Files**

To read and evaluate XML, you can either convert the text to the XML data type, or you can instantiate a blank XML object and load the XML from a file or a URL in the Internet. This line would read the content from a file $env:temp\employee.xml and convert it to XML:

$xmldata = [xml](Get-Content $env:temp\employee.xml)

A faster approach uses a blank XML object and its Load() method:

$xmldata = New-Object XML

$xmldata.Load("$env:temp\employee.xml")

Conversion or loading XML from a file of course only works when the XML is valid and contains no syntactic errors. Else, the conversion will throw an exception.

Once the XML data is stored in an XML object, it is easy to read its content because PowerShell automatically turns XML nodes and attributes into object properties. So, to read the staff from the sample XML data, try this:

$xmldata.staff.employee

Name function Age

---- ----- -----

Tobias Weltner management 39

Cofi Heidecke security 4

**Accessing Single Nodes and Modifying Data**

To pick out a specific node from a set of nodes, you can use the PowerShell pipeline and Where-Object. This would pick out a particular employee from the list of staff. As you will see, you can not only read data but also change it.

$xmldata.staff.employee | Where-Object { $\_.Name -match "Tobias Weltner" }

Name function Age

---- ----- -----

Tobias Weltner management 39

$employee = $xmldata.staff.employee | Where-Object { $\_.Name -match "Tobias Weltner" }

$employee.function = "vacation"

$xmldata.staff.employee

Name function Age

---- ----- -----

Tobias Weltner vacation 39

Cofi Heidecke security 4

If you want to save changes you applied to XML data, call the Save() method:

$xmldata.Save("$env:temp\updateddata.xml")

**Using SelectNodes() to Choose Nodes**

Another way of picking nodes is to use the method SelectNode() and its so-called XPath query language. So, to get to the employee data below the staff node, use this approach:

$xmldata.SelectNodes('staff/employee')

Name function Age

---- ----- -----

Tobias Weltner management 39

Cofi Heidecke security 4

The result is pretty much the same as before, but XPath is very flexible and supports wildcards and additional control. The next statement retrieves just the first employee node:

$xmldata.SelectNodes('staff/employee[1]')

Name function Age

---- ----- -----

Tobias Weltner management 39

If you'd like, you can get a list of all employees who are under the age of 18:

$xmldata.SelectNodes('staff/employee[age<18]')

Name function Age

---- ----- -----

Cofi Heidecke security 4

To the last employee on the list, use this approach:

$xmldata.SelectNodes('staff/employee[last()]')

$xmldata.SelectNodes('staff/employee[position()>1]')

Alternatively, you can also use an *XpathNavigator*:

# Create navigator for XML:

$xpath = [System.XML.XPath.XPathDocument][System.IO.TextReader][System.IO.StringReader]`  
(Get-Content $env:temp\employee.xml | Out-String)

$navigator = $xpath.CreateNavigator()

# Output the last employee name of the Hanover branch office:

$query = "/staff[@branch='Hanover']/employee[last()]/Name"

$navigator.Select($query) | Format-Table Value

Value

-----

Cofi Heidecke

# Output all employees of the Hanover branch office except for Tobias Weltner:

$query = "/staff[@branch='Hanover']/employee[Name!='Tobias Weltner']"

$navigator.Select($query) | Format-Table Value

Value

-----

Cofi Heidecke

**Accessing Attributes**

Attributes are pieces of information that describe an XML node. If you'd like to read the attributes of a node, use *Attributes*:

$xmldata.staff.Attributes

#text

-----

Hanover

sales

Use *GetAttribute()* if you'd like to query a particular attribute:

$xmldata.staff.GetAttribute("branch")

Hanover

Use *SetAttribute()* to specify new attributes or modify (overwrite) existing ones:

$xmldata.staff.SetAttribute("branch", "New York")

$xmldata.staff.GetAttribute("branch")

New York

**Adding New Nodes**

If you'd like to add new employees to your XML, use *CreateElement()* to create an employee element and then fill in the data. Finally, add the element to the XML:

# Create new node:

$newemployee = $xmldata.CreateElement("employee")

$newemployee.InnerXML = '<Name>Bernd Seiler</Name><function>expert</function>'

# Write nodes in XML:

$xmldata.staff.AppendChild($newemployee)

# Check result:

$xmldata.staff.employee

Name function Age

---- ----- -----

Tobias Weltner management 39

Cofi Heidecke security 4

Bernd Seiler expert

# Output plain text:

$xmldata.get\_InnerXml()

<?xml version="1.0"?><Branch office staff="Hanover" Type="sales"><employee>  
<Name>Tobias Weltner</Name><function>management</function><age>39</age>  
</employee><employee><Name>Cofi Heidecke</Name><function>security</function>  
<age>4</age></employee><employee><Name>Bernd Seiler</Name><function>  
expert</function></employee></staff>

**Exploring the Extended Type System**

The PowerShell Extended Type System (ETS) is XML-based, too. The ETS is responsible for turning objects into readable text. PowerShell comes with a set of xml files that all carry the extension ".ps1xml". There are format-files and type-files. Format-files control which object properties are shown and how the object structure is represented. Type-format files control which additional properties and methods should be added to objects.

With the basic knowledge about XML that you gained so far, you can start exploring the ETS XML files and learn more about the inner workings of PowerShell.

**The XML Data of the Extended Type System**

Whenever PowerShell needs to convert an object into text, it searches through its internal "database" to find information about how to best format and display the object. This database really is a collection of XML files in the PowerShell root folder*$pshome*:

Dir $pshome\\*.format.ps1xml

All these files define a multitude of Views, which you can examine using PowerShell XML support.

[xml]$file = Get-Content "$pshome\dotnettypes.format.ps1xml"

$file.Configuration.ViewDefinitions.View

Name ViewSelectedBy TableControl

---- -------------- ------------

System.Reflection.Assembly ViewSelectedBy TableControl

System.Reflection.AssemblyName ViewSelectedBy TableControl

System.Globalization.CultureInfo ViewSelectedBy TableControl

System.Diagnostics.FileVersionInfo ViewSelectedBy TableControl

System.Diagnostics.EventLogEntry ViewSelectedBy TableControl

System.Diagnostics.EventLog ViewSelectedBy TableControl

System.Version ViewSelectedBy TableControl

System.Drawing.Printing.PrintDo... ViewSelectedBy TableControl

Dictionary ViewSelectedBy TableControl

ProcessModule ViewSelectedBy TableControl

process ViewSelectedBy TableControl

PSSnapInInfo ViewSelectedBy

PSSnapInInfo ViewSelectedBy TableControl

Priority ViewSelectedBy TableControl

StartTime ViewSelectedBy TableControl

service ViewSelectedBy TableControl

(...)

**Finding Pre-Defined Views**

Pre-defined views are interesting because you can use the *-View* parameter to change the way PowerShell presents results with the cmdlets *Format-Table* or *Format-List*.

Get-Process | Format-Table -View Priority

Get-Process | Format-Table -View StartTime

To find out which views exist, take a look into the format.ps1xml files that describe the object type.

[xml]$file = Get-Content "$pshome\dotnettypes.format.ps1xml"

$view = @{ Name='ObjectType'; Expression= {$\_.ViewSelectedBy.TypeName}}

$file.Configuration.ViewDefinitions.View | Select-Object Name, $view |  
 Where-Object { $\_.Name -ne $\_. ObjectType } | Sort-Object ObjectType

Name ObjectType

---- ----------

Dictionary System.Collections.DictionaryEntry

DateTime System.DateTime

Priority System.Diagnostics.Process

StartTime System.Diagnostics.Process

process System.Diagnostics.Process

process System.Diagnostics.Process

ProcessModule System.Diagnostics.ProcessModule

DirectoryEntry System.DirectoryServices.DirectoryEntry

PSSnapInInfo System.Management.Automation.PSSnapI...

PSSnapInInfo System.Management.Automation.PSSnapI...

service System.ServiceProcess.ServiceController

Here you see all views defined in this XML file. The object types for which the views are defined are listed in the second column. The *Priority* and *StartTime* views, which we just used, are on that list. However, the list just shows views that use Table format. To get a complete list of all views, here is a more sophisticated example:

[xml]$file = Get-Content "$pshome\dotnettypes.format.ps1xml"

$view = @{ Name='ObjectType'; Expression= {$\_.ViewSelectedBy.TypeName}}

$type = @{ Name='Type'; expression={if ($\_.TableControl) { "Table" } elseif ($\_.ListControl) {  
 "List" } elseif ($\_.WideControl) { "Wide" } elseif ($\_.CustomControl) { "Custom" }}}

$file.Configuration.ViewDefinitions.View | Select-Object Name, $view, $type |  
 Sort-Object ObjectType | Group-Object ObjectType | Where-Object { $\_.Count -gt 1} |  
 ForEach-Object { $\_.Group}

Name ObjectType Type

---- ---------- ----

Dictionary System.Collections.Dict... Table

System.Collections.Dict... System.Collections.Dict... List

System.Diagnostics.Even... System.Diagnostics.Even... List

System.Diagnostics.Even... System.Diagnostics.Even... Table

System.Diagnostics.Even... System.Diagnostics.Even... Table

System.Diagnostics.Even... System.Diagnostics.Even... List

System.Diagnostics.File... System.Diagnostics.File... List

System.Diagnostics.File... System.Diagnostics.File... Table

Priority System.Diagnostics.Process Table

process System.Diagnostics.Process Wide

StartTime System.Diagnostics.Process Table

process System.Diagnostics.Process Table

PSSnapInInfo System.Management.Autom... Table

PSSnapInInfo System.Management.Autom... List

System.Reflection.Assembly System.Reflection.Assembly Table

System.Reflection.Assembly System.Reflection.Assembly List

System.Security.AccessC... System.Security.AccessC... List

System.Security.AccessC... System.Security.AccessC... Table

service System.ServiceProcess.S... Table

System.ServiceProcess.S... System.ServiceProcess.S... List

System.TimeSpan System.TimeSpan Wide

System.TimeSpan System.TimeSpan Table

System.TimeSpan System.TimeSpan List

Remember there are many format.ps1xml-files containing formatting information. You'll only get a complete list of all view definitions when you generate a list for all of these files.