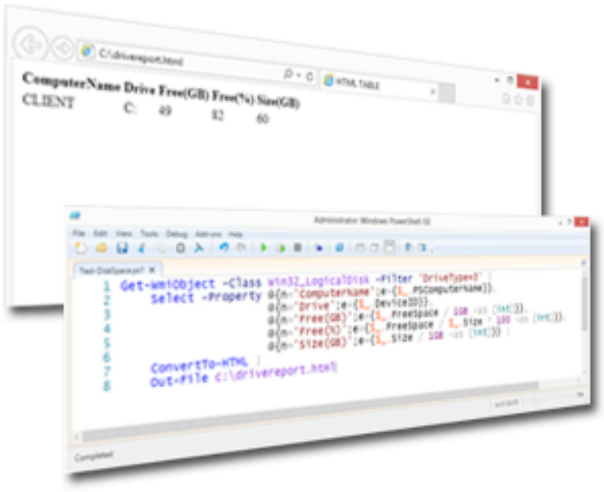
Creating HTML Reports in PowerShell

Don Jones



```
ComputerName Drive Free(GB) Free(%) Size(GB)
CLIENT       C:    49       82      60
```

```
Get-WmiObject -Class Win32_LogicalDisk -Filter 'DriveType=3' |
    Select -Property @{n='ComputerName';e={$_.PSComputerName}},
                     @{n='Drive';e={$_.DeviceID}},
                     @{n='Free(GB)';e={$_.FreeSpace / 1GB -as [int]}},
                     @{n='Free(%)';e={$_.FreeSpace / $_.Size * 100 -as [int]}},
                     @{n='Size(GB)';e={$_.Size / 1GB -as [int]}} |

    ConvertTo-HTML |
    Out-File c:\drivereport.html
```

PowerShell.org
Resources, Answers, and Education

http://PowerShellBooks.com • http://PowerShell.org

Creating HTML Reports in PowerShell

by Don Jones

Foreword

I set out to create this guide because, as more and more people jump into PowerShell, they seem to be jumping into reporting. And, unfortunately, they're making a lot of beginner mistakes that end up making the job tougher, and the results less attractive. It's quite understandable, though: creating HTML reports requires an unholy blend of skills. You have to be an administrator to retrieve the information, a Web designer to make it look good, and quite frankly a hacker to get everything you want into the right place. So with that in mind, I set out to create a book that'll help a bit.

Since I'm not making any money from this book, I do hope you'll take a moment to consider the other books I've written and co-authored, all of which can be found at http://PowerShellBooks.com. Your purchases there help pay the mortgage and the utility bills, and that gives me the time to work on projects like this one.

If you have something you'd like to add to this book, or you find an error, or you just need additional help, please drop a message into the "PowerShell Q&A" forum at http://PowerShell.org. That site also features an impressive array of other free resources related to PowerShell, and I think you'll enjoy the time you spend there.

Don Jones

Note This is the October 2012 edition of this guide. You'll find that this guide is periodically updated with additional material, corrections, and expansions. The one, authoritative source for the latest edition is http://PowerShellBooks.com - please check to make sure you've got the latest!

Contents

HTML Report Basics

First, understand that PowerShell isn't limited to creating reports in HTML. But I like HTML because it's flexible, can be easily e-mailed, and can be more easily made to look pretty than a plain-text report. But before you dive in, you do need to know a bit about how HTML works.

An HTML page is just a plain text file, looking something like this:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"  "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>HTML TABLE</title>
</head><body>
<table>
<colgroup><col/><col/><col/><col/><col/></colgroup>
<tr><th>ComputerName</th><th>Drive</th><th>Free(GB)</th><th>Free(%)</th><th>Size(GB)</th></tr>
<tr><td>CLIENT</td><td>C:</td><td>49</td><td>82</td><td>60</td></tr>
</table>
</body></html>
```

When read by a browser, this file is rendered into the display you see within the browser's window. The same applies to e-mail clients capable of displaying HTML content. While you, as a person, can obviously put anything you want into the file, if you want the output to look right you need to follow the rules that browsers expect.

One of those rules is that each file should contain one, and only one, HTML document. That's all of the content between the <HTML> tag and the </HTML> tag (tag names aren't case-sensitive, and it's common to see them in all-lowercase as in the example above). I mention this because one of the most common things I'll see folks do in PowerShell looks something like this:

```
Get-WmiObject -class Win32_OperatingSystem | ConvertToHTML | Out-File report.html
Get-WmiObject -class Win32_BIOS | ConvertTo-HTML | Out-File report.html -append
Get-WmiObject -class Win32_Service | ConvertTo-HTML | Out-File report.html -append
```

"Aaarrrggh," says my colon everytime I see that. You're basically telling PowerShell to create three complete HTML documents and jam them into a single file. While some browsers (Internet Explorer, notable) will figure that out and display something, it's just wrong. Once you start getting fancy with reports, you'll figure out pretty quickly that this approach is painful. It isn't PowerShell's fault; you're just not following the rules. Hence this guide!

You'll notice that the HTML consists of a lot of other tags, too: <TABLE>, <TD>, <HEAD>, and so on. Most of these are *paired,* meaning they come in an opening tag like <TD> and a closing tag like </TD>. The <TD> tag represents a table cell, and everything between those tags is considered the contents of that cell.

The <HEAD> section is important. What's inside there isn't normally visible in the browser; instead, the browser focuses on what's in the <BODY> section. The <HEAD> section provides additional meta-data, like what the title of the page will be (as displayed in the browser's window title bar, not in the page itself), any style sheets or scripts that are attached to the page, and so on. We're going to do some pretty awesome stuff with the <HEAD> section, trust me.

You'll also notice that this HTML is pretty "clean," as opposed to, say, the HTML output by Microsoft Word. This HTML doesn't have a lot of visual information embedded in it, like colors or fonts. That's *good,* because it follows correct HTML practices of separating formatting information from the document structure. It's disappointing at first, because your HTML pages look really, really boring. But we're going to fix that, also.

In order to help the narrative in this book stay focused, I'm going to start with a single example. In that example, we're going to retrieve multiple bits of information about a remote computer, and format it all into a pretty, dynamic HTML report. Hopefully, you'll be able to focus on the *techniques* I'm showing you, and adapt those to your own specific needs.

In my example, I want the report to have five sections, each with the following information:

- Computer Information
    - The computer's operating system version, build number, and service pack version.
    - Hardware info: the amount of installed RAM and number of processes, along with the manufacturer and model.
    - An list of all processes running on the machine.
    - A list of all services which are set to start automatically, but which aren't running.
    - Information about all physical network adapters in the computer. Not IP addresses, necessarily - hardware information like MAC address.

I realize this isn't a universally-interesting set of information, but these sections will allow be to demonstrate some specific techniques. Again, I'm hoping that you can adapt these to your precise needs.

Gathering the Information

I'm a big fan of modular programming. Big, big fan. With that in mind, I tend to write functions that gather the information I want to be in my report - and I'll usually do one function per major section of my report. You'll see in a bit how that's beneficial. By writing each function individually, I make it easier to use that same information in other tasks, and I make it easier to debug each one. The trick is to have each function output a single type of object that combines all of the information for that report section. I've created five functions, which I've pasted into a single script file. I'll give you each of those functions one at a time, with a brief commentary for each. Here's the first:

```
 function Get-InfoOS {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)][string]$ComputerName
    )
    $os = Get-WmiObject -class Win32_OperatingSystem -ComputerName $ComputerName
    $props = @{'OSVersion'=$os.version;
               'SPVersion'=$os.servicepackmajorversion;
               'OSBuild'=$os.buildnumber}
    New-Object -TypeName PSObject -Property $props
}
```

This is a straightforward function, and the main reason I bothered to even make it a function - as opposed to just using Get-WmiObject directly - is that I want different property names, like "OSVersion" instead of just "Version." That said, I tend to follow this exact same programming pattern for all info-retrieval functions, just to keep them consistent.

```
function Get-InfoCompSystem {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)][string]$ComputerName
    )
    $cs = Get-WmiObject -class Win32_ComputerSystem -ComputerName $ComputerName
    $props = @{'Model'=$cs.model;
               'Manufacturer'=$cs.manufacturer;
               'RAM (GB)'="{0:N2}" -f ($cs.totalphysicalmemory / 1GB);
               'Sockets'=$cs.numberofprocessors;
               'Cores'=$cs.numberoflogicalprocessors}
    New-Object -TypeName PSObject -Property $props
}
```

Very similar to the last one. You'll notice here that I'm using the -f formatting operator with the RAM property, so that I get a value in gigabytes with 2 decimal places. The native value is in bytes, which isn't useful for me.

```
function Get-InfoBadService {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)][string]$ComputerName
    )
    $svcs = Get-WmiObject -class Win32_Service -ComputerName $ComputerName `
           -Filter "StartMode='Auto' AND State<>'Running'"
    foreach ($svc in $svcs) {
        $props = @{'ServiceName'=$svc.name;
                   'LogonAccount'=$svc.startname;
                   'DisplayName'=$svc.displayname}
        New-Object -TypeName PSObject -Property $props
    }
}
```

Here, I've had to recognize that I'll be getting back more than one object from WMI, so I have to enumerate through them using a ForEach construct. Again, I'm primarily just renaming properties. I absolutely could have done that with a Select-Object command, but I like to keep the overall function structure similar to my other functions. Just a personal preference that helps me include fewer bugs, since I'm used to doing things this way.

```
function Get-InfoProc {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)][string]$ComputerName
    )
    $procs = Get-WmiObject -class Win32_Process -ComputerName $ComputerName
    foreach ($proc in $procs) {
        $props = @{'ProcName'=$proc.name;
                   'Executable'=$proc.ExecutablePath}
        New-Object -TypeName PSObject -Property $props
    }
}
```

Very similar to the function for services. You can probably start to see how using this same structure makes a certain amount of copy-and-paste pretty effective when I create a new function.

```
function Get-InfoNIC {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)][string]$ComputerName
    )
    $nics = Get-WmiObject -class Win32_NetworkAdapter -ComputerName $ComputerName `
           -Filter "PhysicalAdapter=True"
    foreach ($nic in $nics) {
        $props = @{'NICName'=$nic.servicename;
                   'Speed'=$nic.speed / 1MB -as [int];
                   'Manufacturer'=$nic.manufacturer;
                   'MACAddress'=$nic.macaddress}
        New-Object -TypeName PSObject -Property $props
    }
}
```

The main thing of note here is how I've converted the speed property, which is natively in bytes, to megabytes. Because I don't care about decimal places here (I want a whole number), casting the value as an integer, by using the -as operator, is easier for me than the -f formatting operator. Also, it gives me a chance to show you this technique!

Note that, for the purposes of this book, I'm going to be putting these functions into the same script file as the rest of my code, which actually generates the HTML. I don't normally do that. Normally, info-retrieval functions go into a script module, and I then write my HTML-generation script to load that module. Having the functions in a module makes them easier to use elsewhere, if I want to. I'm skipping the module this time just to keep things simpler for this demonstration. If you want to learn more about script modules, pick up *Learn PowerShell Toolmaking in a Month of Lunches* or *PowerShell in Depth,* both of which are linked at http://PowerShellBooks.com.

Building the HTML

Once the information is retrieved, I can start creating the HTML. For each section of my report, I'm going to generate an *HTML fragment.* This is not a complete HTML page; it's just the HTML needed to display that particular report section. But first, I'm going to add a little bit of code to my script. To make sure you're keeping up, here's the entire thing:

```
 <#
.SYNOPSIS
Generates an HTML-based system report for one or more computers.
Each computer specified will result in a separate HTML file;
specify the -Path as a folder where you want the files written.
Note that existing files will be overwritten.
.PARAMETER ComputerName
One or more computer names or IP addresses to query.
.PARAMETER Path
The path of the folder where the files should be written.
.EXAMPLE
.\New-HTMLSystemReport -ComputerName ONE,TWO -Path C:\Reports\
#>
[CmdletBinding()]
param(
    [Parameter(Mandatory=$True,
               ValueFromPipeline=$True,
               ValueFromPipelineByPropertyName=$True)]
    [string[]]$ComputerName,

    [Parameter(Mandatory=$True)]
    [string]$Path
)
PROCESS {

function Get-InfoOS {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)][string]$ComputerName
    )
    $os = Get-WmiObject -class Win32_OperatingSystem -ComputerName $ComputerName
    $props = @{'OSVersion'=$os.version;
               'SPVersion'=$os.servicepackmajorversion;
               'OSBuild'=$os.buildnumber}
    New-Object -TypeName PSObject -Property $props
}

function Get-InfoCompSystem {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)][string]$ComputerName
    )
    $cs = Get-WmiObject -class Win32_ComputerSystem -ComputerName $ComputerName
    $props = @{'Model'=$cs.model;
               'Manufacturer'=$cs.manufacturer;
               'RAM (GB)'="{0:N2}" -f ($cs.totalphysicalmemory / 1GB);
               'Sockets'=$cs.numberofprocessors;
               'Cores'=$cs.numberoflogicalprocessors}
    New-Object -TypeName PSObject -Property $props
}

function Get-InfoBadService {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)][string]$ComputerName
    )
    $svcs = Get-WmiObject -class Win32_Service -ComputerName $ComputerName `
           -Filter "StartMode='Auto' AND State<>'Running'"
    foreach ($svc in $svcs) {
        $props = @{'ServiceName'=$svc.name;
                   'LogonAccount'=$svc.startname;
                   'DisplayName'=$svc.displayname}
        New-Object -TypeName PSObject -Property $props
    }
}

function Get-InfoProc {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)][string]$ComputerName
    )
    $procs = Get-WmiObject -class Win32_Process -ComputerName $ComputerName
    foreach ($proc in $procs) {
        $props = @{'ProcName'=$proc.name;
                   'Executable'=$proc.ExecutablePath}
        New-Object -TypeName PSObject -Property $props
    }
}

function Get-InfoNIC {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)][string]$ComputerName
    )
    $nics = Get-WmiObject -class Win32_NetworkAdapter -ComputerName $ComputerName `
           -Filter "PhysicalAdapter=True"
    foreach ($nic in $nics) {
        $props = @{'NICName'=$nic.servicename;
                   'Speed'=$nic.speed / 1MB -as [int];
                   'Manufacturer'=$nic.manufacturer;
                   'MACAddress'=$nic.macaddress}
        New-Object -TypeName PSObject -Property $props
    }
}

foreach ($computer in $computername) {
    try {
        $everything_ok = $true
        Write-Verbose "Checking connectivity to $computer"
        Get-WmiObject -class Win32_BIOS -ComputerName $Computer -EA Stop | Out-Null
    } catch {
        Write-Warning "$computer failed"
        $everything_ok = $false
    }

    if ($everything_ok) {
    }
}

}
```

Basically, I've turned this script - which I'm calling New-HTMLSystemReport - into an Advanced Script, accepting its own parameters and even including comment-based help. I want to acknowledge that the contents of the PROCESS script block are not properly indented; that's a function of the page width of this book and not because I'm sloppy!

You'll notice that, near the end, I'm grabbing Win32_BIOS and piping it to Out-Null. That's basically a "ping" for me, although it's specifically testing my ability to get to WMI on the remote machine. I don't care about the result I get back; I just want to see if it works. If it doesn't, I'll display a warning and skip that machine.

The rest of my code is going to go inside the If construct:

```
if ($everything_ok) {
}
```

When we get to the end of the book, I'll give you the entire script again.

So let's start working on the HTML. At a very basic level, the following commands will do what I want:

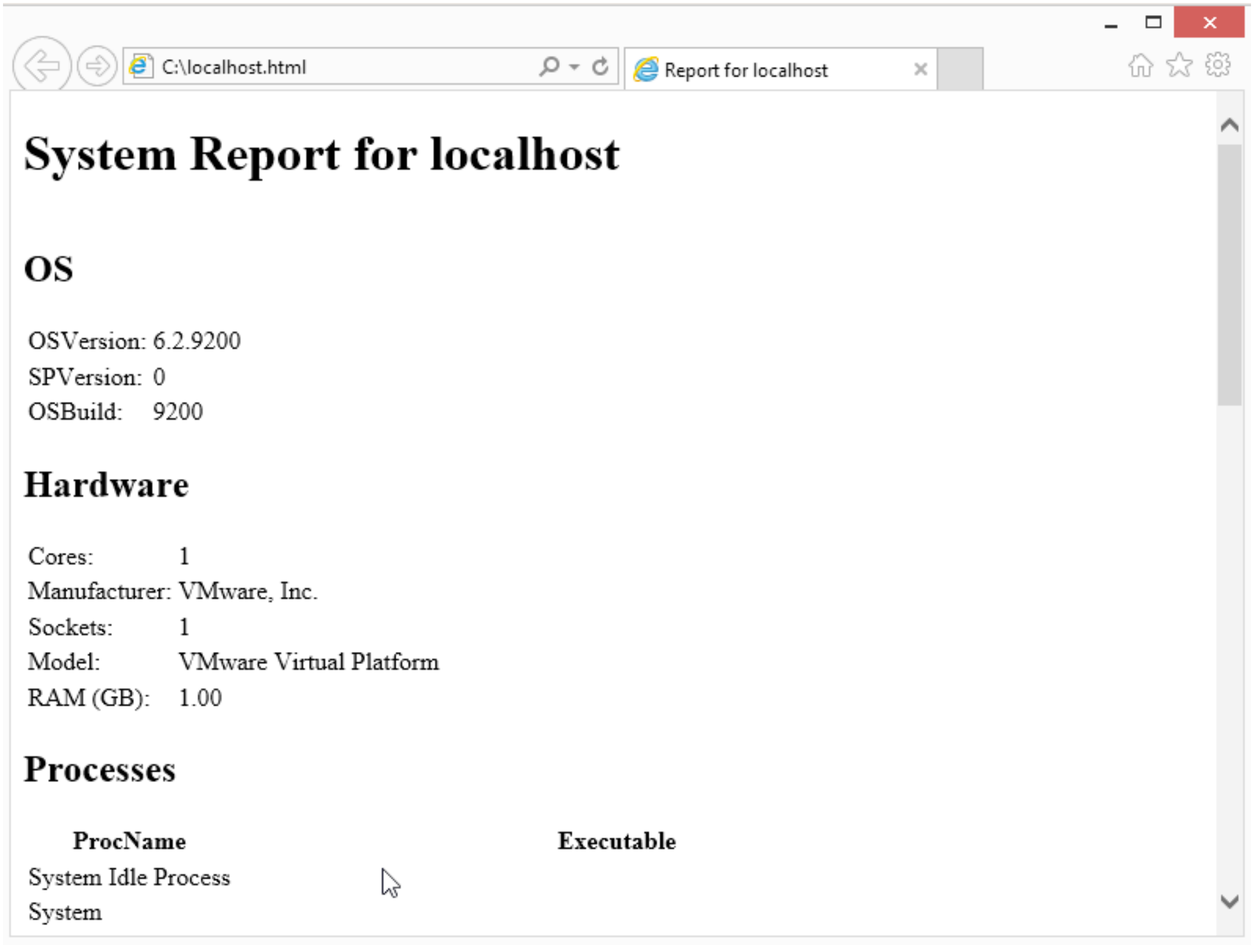
```
 if ($everything_ok) {
    $filepath = Join-Path -Path $Path -ChildPath "$computer.html"
    $html_os = Get-InfoOS -ComputerName $computer |
               ConvertTo-HTML -As List -Fragment -PreContent "<h2>OS</h2>" |
               Out-String
    $html_cs = Get-InfoCompSystem -ComputerName $computer |
               ConvertTo-HTML -As List -Fragment -PreContent "<h2>Hardware</h2>" |
               Out-String
    $html_pr = Get-InfoProc -ComputerName $computer |
               ConvertTo-HTML -Fragment -PreContent "<h2>Processes</h2>" |
               Out-String
    $html_sv = Get-InfoBadService -ComputerName $computer |
               ConvertTo-HTML -Fragment -PreContent "<h2>Check Services</h2>" |
               Out-String
    $html_na = Get-InfoNIC -ComputerName $Computer |
               ConvertTo-HTML -Fragment -PreContent "<h2>NICs</h2>" |
               Out-String

    $params = @{'Title'="Report for $computer";
                'PreContent'="<h1>System Report for $computer</h1>";
                'PostContent'=$html_os,$html_cs,$html_pr,$html_sv,$html_na}
    ConvertTo-HTML @params | Out-File -FilePath $filepath
}
```

Notice that I'm basically just running all five of my functions, and piping them to ConvertTo-HTML. Each time, I'm giving ConvertTo-HTML a -PreContent parameter, which will become the section heading for that section of the report. I'm using an <H2> HTML tag in those headings, to make them stand apart. You can see that each of the five is producing not a full HTML page, but an HTML fragment. The first two are producing a list, which I tend to prefer if I have a single object to display; the last three are using a table, which is the default (you could also put **-As Table** if you wanted to explicitly list that).

Also pay close attention to where I'm piping that HTML: I'm sending it to Out-String. That's because ConvertTo-HTML natively produces an array of strings, which will be problematic for me to use later. Out-String collapses all those down into an object that will play nicely with what I want to do last.

My last trick is to run ConvertTo-HTML one last time, but without the -Fragment switch. In other words, I'm producing the final, completed HTML page. I'm giving it a title for the browser's window bar, and I'm using -PreContent to insert an overall report header. My HTML fragments get fed to -PostContent, and then piped out to my file. You can see at the top of this code snippet where I appended the "*computername*.html" filename to the folder path that was specified.

Here's a portion of the resulting HTML, shown in Internet Explorer:



System Report for localhost

OS

OSVersion:	6.2.9200
SPVersion:	0
OSBuild:	9200

Hardware

Cores:	1
Manufacturer:	VMware, Inc.
Sockets:	1
Model:	VMware Virtual Platform
RAM (GB):	1.00

Processes

ProcName	Executable
System Idle Process	
System	

Not beautiful, but all of the information is there. Beautification is our next step. But before we do that, let's pause for a moment and consider what we've produced: a complete, multi-section report, in HTML, which is suitable for placing onto a Web server, attaching to an e-mail (using Send-MailMessage, you could even make the HTML contents the -Body of the message if you also add the -BodyAsHTML switch). You could even frame the results. Well, maybe not that - let's make it prettier first.

An Aside: The ConvertTo-HTML Cmdlet

Obviously, one of the keys to all of this magic is the ConvertTo-HTML cmdlet. It has several parameters which are worth examining:

- -InputObject accepts input from the pipeline, and this input is used to construct either a list or a table. A table is used by default, specify **-As List** if you want a list instead. I prefer lists whenever there’s only a single object being displayed, but that’s just a personal preference.

- -Body specifies the contents of the HTML <BODY> tag. This content will appear *before* anything else, and the parameter accepts multiple values.

- -PreContent displays next, followed by whatever table or list was generated from -InputObject.

- -PostContent wraps up the display.

- -Title specifies the contents of the <TITLE> tag, which appears in the <HEAD> section and specifies what to display in the browser’s window title bar.

- -Head lets you specify contents for the <HEAD> section; this overrides -Title. Whatever you pass to the -Head parameter needs to be properly formatted HTML, and must be legal for inclusion in the <HEAD> section.

- -CssUri lets you specify the name (or URL) of a Cascading Style Sheet.

You’d think that last one would be the key to making our report prettier, and you’d be almost right.

Beautification of HTML

The HTML we've produced so far is exactly as it should be: plain. HTML is meant to describe the *structure* of a document, such as where the headings are, what tables look like, and so on. A *style sheet,* or more properly a Cascading Style Sheet (CSS) is designed to apply visual formatting to that structure.

CSS can be attached to HTML as a separate file, but I don't like that approach for management reports, mainly because it means you have to mail *two* reports all over the place, instead of just one. I tend to prefer the other approach, which is to *embed* the style information right into the main HTML page. Web developers are cringing right now, but relax. Reports like this are *meant* to be more standalone.

I'm going to start by creating the following CSS file, and saving it as C:\style.css. Now, in practice, you'd probably put this on a file share someplace. We're not going to link HTML pages directly to it, but will rather copy the contents into each HTML page we create.

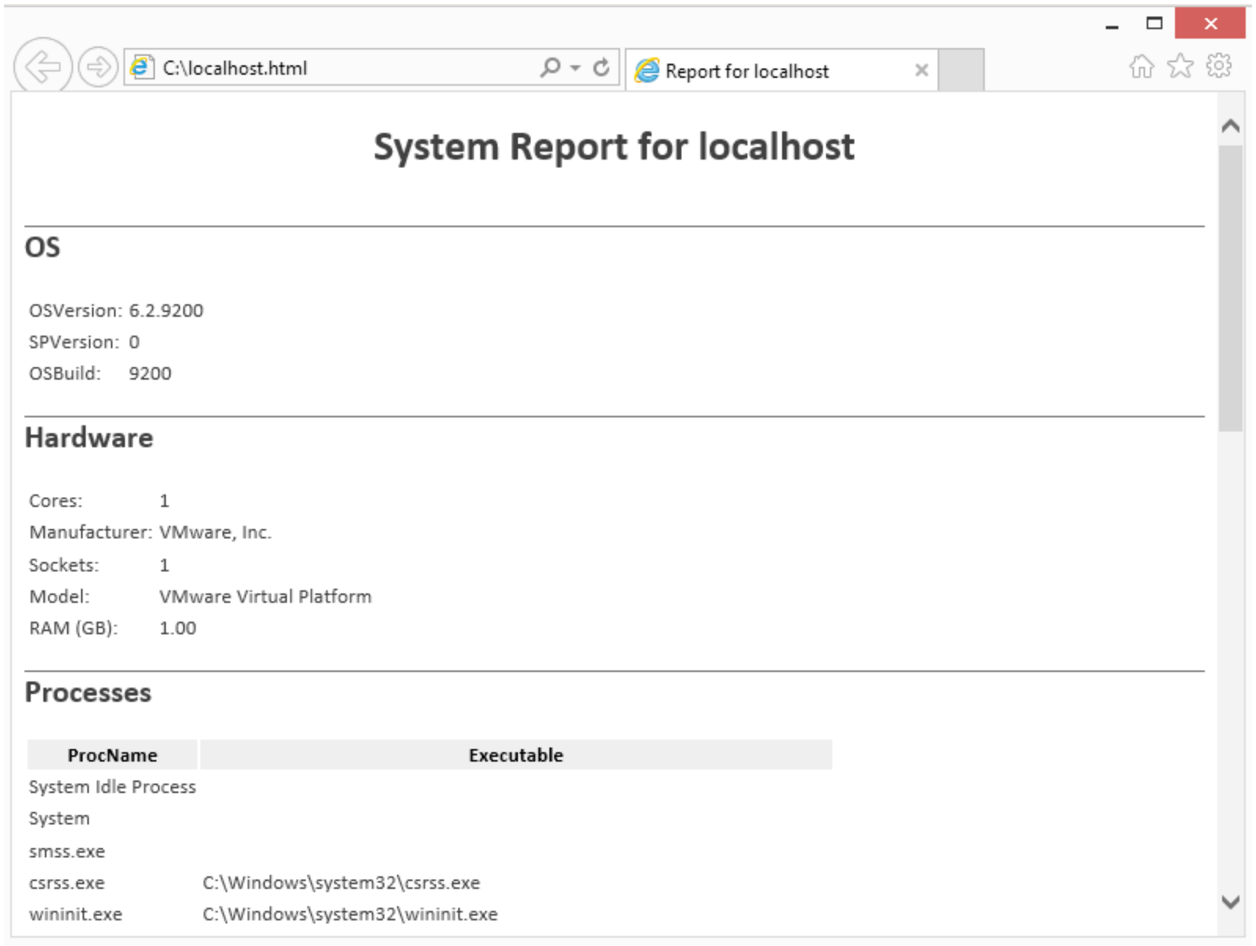
```
 body {
    color:#333333;
    font-family:Calibri,Tahoma;
    font-size: 10pt;
}
h1 {
    text-align:center;
}
h2 {
    border-top:1px solid #666666;
}
th {
    font-weight:bold;
    color:#000000;
    background-color:#eeeeee;
}
```

Now I'm going to modify a portion of my script, as follows, to utilize that:

```
foreach ($computer in $computername) {

   $style = "<style>$(get-content C:\style.css)</style>"

   try {
       $everything_ok = $true
       Write-Verbose "Checking connectivity to $computer"
       Get-WmiObject -class Win32_BIOS -ComputerName $Computer -EA Stop | Out-Null
   } catch {
       Write-Warning "$computer failed"
       $everything_ok = $false
   }

   if ($everything_ok) {
       $filepath = Join-Path -Path $Path -ChildPath "$computer.html"
       $html_os = Get-InfoOS -ComputerName $computer |
                  ConvertTo-HTML -As List -Fragment -PreContent "<h2>OS</h2>" |
                  Out-String
       $html_cs = Get-InfoCompSystem -ComputerName $computer |
                  ConvertTo-HTML -As List -Fragment -PreContent "<h2>Hardware</h2>" |
                  Out-String
       $html_pr = Get-InfoProc -ComputerName $computer |
                  ConvertTo-HTML -Fragment -PreContent "<h2>Processes</h2>" |
                  Out-String
```

```
            $html_sv = Get-InfoBadService -ComputerName $computer |
                       ConvertTo-HTML -Fragment -PreContent "<h2>Check Services</h2>" |
                       Out-String
            $html_na = Get-InfoNIC -ComputerName $Computer |
                       ConvertTo-HTML -Fragment -PreContent "<h2>NICs</h2>" |
                       Out-String

            $params = @{'Head'="<title>Report for $computer</title>$style";
                        'PreContent'="<h1>System Report for $computer</h1>";
                        'PostContent'=$html_os,$html_cs,$html_pr,$html_sv,$html_na}
            ConvertTo-HTML @params | Out-File -FilePath $filepath
        }
    }
```

I've highlighted the two changes I made. The first one reads in the contents of that CSS file, inserting it between two HTML <STYLE> tags. The second one changes from using the -Title parameter of ConvertTo-HTML, and instead uses the more-flexible -Head parameter, specifying an entire <HEAD> section. I'm inserting both a <TITLE> tag (so that I still get a title in the browser window title bar), and my inline, or embedded, style sheet. Here's the new result in Internet Explorer:



System Report for localhost

OS

```
OSVersion: 6.2.9200
SPVersion: 0
OSBuild:   9200
```

Hardware

```
Cores:        1
Manufacturer: VMware, Inc.
Sockets:      1
Model:        VMware Virtual Platform
RAM (GB):     1.00
```

Processes

ProcName	Executable
System Idle Process	
System	
smss.exe	
csrss.exe	C:\Windows\system32\csrss.exe
wininit.exe	C:\Windows\system32\wininit.exe

Awesome, no? All I did is create some style for four HTML tags: <BODY>, which sets the default for the page, <H1> and <H2>, and the <TH> tag used for table header rows. Of course, you need to dig into CSS to learn more about what styles you can set and it can get pretty complex. But the results can be pretty astounding.

Note If you'd like a complete reference and tutorial on CSS, you can get one online for free. Just head to http://w3schools.com/css3/default.asp, where you'll find links to both examples, tutorials, and references.

Let's quickly review: Suppose you followed my naming convention and named your script New-HTMLSystemReport. Also suppose you have an organizational unit in Active Directory named Servers, and that you want to generate a report for each server in that OU. Using Microsoft's ActiveDirectory PowerShell module, you could run this (I'm using COMPANY.COM as my AD domain name; you'd obviously adjust that):

```
Get-ADComputer -filter * -searchBase "ou=servers,dc=company,dc=com" |
Select-Object -ExpandProperty Name |
C:\MyScripts\New-HTMLSystemReport -Path C:\HTMLReports
```

After a bit of waiting, you'd have your HTML reports. Although, to be honest, the way I've hardcoded the location of the CSS template file bugs me. So here's a final script, with that parameterized:

```
 <#
.SYNOPSIS
Generates an HTML-based system report for one or more computers.
Each computer specified will result in a separate HTML file;
specify the -Path as a folder where you want the files written.
Note that existing files will be overwritten.
.PARAMETER ComputerName
One or more computer names or IP addresses to query.
.PARAMETER Path
The path of the folder where the files should be written.
.PARAMETER CssPath
The path and filename of the CSS template to use.
.EXAMPLE
.\New-HTMLSystemReport -ComputerName ONE,TWO `
                       -Path C:\Reports\ `
                       -CssPath c:\style.css
#>
[CmdletBinding()]
param(
    [Parameter(Mandatory=$True,
               ValueFromPipeline=$True,
               ValueFromPipelineByPropertyName=$True)]
    [string[]]$ComputerName,

    [Parameter(Mandatory=$True)]
    [string]$Path,

    [Parameter(Mandatory=$True)]
    [string]$CssPath
)
PROCESS {

function Get-InfoOS {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)][string]$ComputerName
    )
    $os = Get-WmiObject -class Win32_OperatingSystem -ComputerName $ComputerName
    $props = @{'OSVersion'=$os.version;
               'SPVersion'=$os.servicepackmajorversion;
               'OSBuild'=$os.buildnumber}
    New-Object -TypeName PSObject -Property $props
}

function Get-InfoCompSystem {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)][string]$ComputerName
    )
    $cs = Get-WmiObject -class Win32_ComputerSystem -ComputerName $ComputerName
    $props = @{'Model'=$cs.model;
```

```
                'Manufacturer'=$cs.manufacturer;
                'RAM (GB)'="{0:N2}" -f ($cs.totalphysicalmemory / 1GB);
                'Sockets'=$cs.numberofprocessors;
                'Cores'=$cs.numberoflogicalprocessors}
    New-Object -TypeName PSObject -Property $props
}

function Get-InfoBadService {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)][string]$ComputerName
    )
    $svcs = Get-WmiObject -class Win32_Service -ComputerName $ComputerName `
           -Filter "StartMode='Auto' AND State<>'Running'"
    foreach ($svc in $svcs) {
        $props = @{'ServiceName'=$svc.name;
                   'LogonAccount'=$svc.startname;
                   'DisplayName'=$svc.displayname}
        New-Object -TypeName PSObject -Property $props
    }
}

function Get-InfoProc {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)][string]$ComputerName
    )
    $procs = Get-WmiObject -class Win32_Process -ComputerName $ComputerName
    foreach ($proc in $procs) {
        $props = @{'ProcName'=$proc.name;
                   'Executable'=$proc.ExecutablePath}
        New-Object -TypeName PSObject -Property $props
    }
}

function Get-InfoNIC {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)][string]$ComputerName
    )
    $nics = Get-WmiObject -class Win32_NetworkAdapter -ComputerName $ComputerName `
           -Filter "PhysicalAdapter=True"
    foreach ($nic in $nics) {
        $props = @{'NICName'=$nic.servicename;
                   'Speed'=$nic.speed / 1MB -as [int];
                   'Manufacturer'=$nic.manufacturer;
                   'MACAddress'=$nic.macaddress}
        New-Object -TypeName PSObject -Property $props
    }
}

foreach ($computer in $computername) {

    $style = "<style>$(get-content $csspath)</style>"

    try {
        $everything_ok = $true
        Write-Verbose "Checking connectivity to $computer"
        Get-WmiObject -class Win32_BIOS -ComputerName $Computer -EA Stop | Out-Null
    } catch {
        Write-Warning "$computer failed"
        $everything_ok = $false
    }
```

```
        if ($everything_ok) {
            $filepath = Join-Path -Path $Path -ChildPath "$computer.html"
            $html_os = Get-InfoOS -ComputerName $computer |
                       ConvertTo-HTML -As List -Fragment -PreContent "<h2>OS</h2>" |
                       Out-String
            $html_cs = Get-InfoCompSystem -ComputerName $computer |
                       ConvertTo-HTML -As List -Fragment -PreContent "<h2>Hardware</h2>" |
                       Out-String
            $html_pr = Get-InfoProc -ComputerName $computer |
                       ConvertTo-HTML -Fragment -PreContent "<h2>Processes</h2>" |
                       Out-String
            $html_sv = Get-InfoBadService -ComputerName $computer |
                       ConvertTo-HTML -Fragment -PreContent "<h2>Check Services</h2>" |
                       Out-String
            $html_na = Get-InfoNIC -ComputerName $Computer |
                       ConvertTo-HTML -Fragment -PreContent "<h2>NICs</h2>" |
                       Out-String

            $params = @{'Head'="<title>Report for $computer</title>$style";
                        'PreContent'="<h1>System Report for $computer</h1>";
                        'PostContent'=$html_os,$html_cs,$html_pr,$html_sv,$html_na}
            ConvertTo-HTML @params | Out-File -FilePath $filepath
        }
    }

    }
```

Now, you'd run something like this for the same example:

```
    Get-ADComputer -filter * -searchBase "ou=servers,dc=company,dc=com" |
    Select-Object -ExpandProperty Name |
    C:\MyScripts\New-HTMLSystemReport -Path C:\HTMLReports -CssPath C:\style.css
```

Obviously, you'd adjust the file paths to suit your system.

We could stop right here and you'd have a marvelous HTML report. But let's keep going.

Making Tables Prettier

One area where I think CSS is a bit weak is in formatting tables. For example, in order to make alternating rows have a different background color, you have to have a table cell tag that includes a class name, like <TD class="typeA">. PowerShell doesn't emit class names into its HTML, though, which would seem to present an insurmountable hurdle.

It isn't surmountable, but it's pretty darn difficult. For this approach, I'm going to take advantage of the fact that PowerShell produces such well-formed HTML, meaning it is essentially **X**HTML, which is XML, which PowerShell can manipulate quite handily. I'm going to do this with my individual HTML fragments, before I assemble the final page, because it's a lot easier that way.

This does require an understanding of some HTML/XML basics. For example, consider this HTML table:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"  "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>HTML TABLE</title>
</head><body>
<table>
<colgroup><col/><col/><col/><col/><col/></colgroup>
<tr><th>ComputerName</th><th>Drive</th><th>Free(GB)</th><th>Free(%)</th><th>Size(GB)</th></tr>
<tr><td>CLIENT</td><td>C:</td><td>49</td><td>82</td><td>60</td></tr>
</table>
</body></html>
```

First, it says right there that this is XHTML, which is awesome. Second, notice the <TABLE> tag. Everything within it - all of the <COLGROUP> and <TR> tags - are *children* of the <TABLE>. Within each <TR>, you have the children of *that* tag, which are either <TH> or <TD>. We're going to rely on this hierarchical relationship. Check out this function, which I'm adding to my script:

```
function Set-AlternatingCSSClasses {
   [CmdletBinding()]
   param(
       [Parameter(Mandatory=$True,ValueFromPipeline=$True)]
       [string]$HTMLFragment,

       [Parameter(Mandatory=$True)]
       [string]$CSSEvenClass,

       [Parameter(Mandatory=$True)]
       [string]$CssOddClass
   )
   [xml]$xml = $HTMLFragment
   $table = $xml.SelectSingleNode('table')
   $classname = $CSSOddClass
   foreach ($tr in $table.tr) {
       if ($classname -eq $CSSEvenClass) {
           $classname = $CssOddClass
       } else {
           $classname = $CSSEvenClass
       }
       $class = $xml.CreateAttribute('class')
       $class.value = $classname
       $tr.attributes.append($class) | Out-null
   }
   $xml.innerxml | out-string
}
```

This will accept an HTML fragment, provided it is valid XML and that it contains a <TABLE> tag. It'll go through that table, adding the specified CSS class names to the <TR> tags. Now, there's a trick to this: The old HTML fragments I was generating contain a top-level <H2> tag from my -PreContent switch, as well as a top-level <TABLE> tag. Having two top-level elements isn't valid XML, so I'm going to have to modify the rest of my script to work with this:

```
        if ($everything_ok) {
            $filepath = Join-Path -Path $Path -ChildPath "$computer.html"
            $html_os = Get-InfoOS -ComputerName $computer |
                       ConvertTo-HTML -As List -Fragment -PreContent "<h2>OS</h2>" |
                       Out-String

            $html_cs = Get-InfoCompSystem -ComputerName $computer |
                       ConvertTo-HTML -As List -Fragment -PreContent "<h2>Hardware</h2>" |
                       Out-String

            $html_pr = Get-InfoProc -ComputerName $computer |
                       ConvertTo-HTML -Fragment |
                       Out-String |
                       Set-AlternatingCSSClasses -CSSEvenClass 'even' -CssOddClass 'odd'
            $html_pr = "<h2>Processes</h2>$html_pr"

            $html_sv = Get-InfoBadService -ComputerName $computer |
                       ConvertTo-HTML -Fragment |
                       Out-String |
                       Set-AlternatingCSSClasses -CSSEvenClass 'even' -CssOddClass 'odd'
            $html_sv = "<h2>Check Services</h2>$html_sv"

            $html_na = Get-InfoNIC -ComputerName $Computer |
                       ConvertTo-HTML -Fragment |
                       Out-String |
                       Set-AlternatingCSSClasses -CSSEvenClass 'even' -CssOddClass 'odd'
            $html_na = "<h2>NICs</h2>$html_na"

            $params = @{'Head'="<title>Report for $computer</title>$style";
                        'PreContent'="<h1>System Report for $computer</h1>";
                        'PostContent'=$html_os,$html_cs,$html_pr,$html_sv,$html_na}
            ConvertTo-HTML @params | Out-File -FilePath $filepath
        }
```

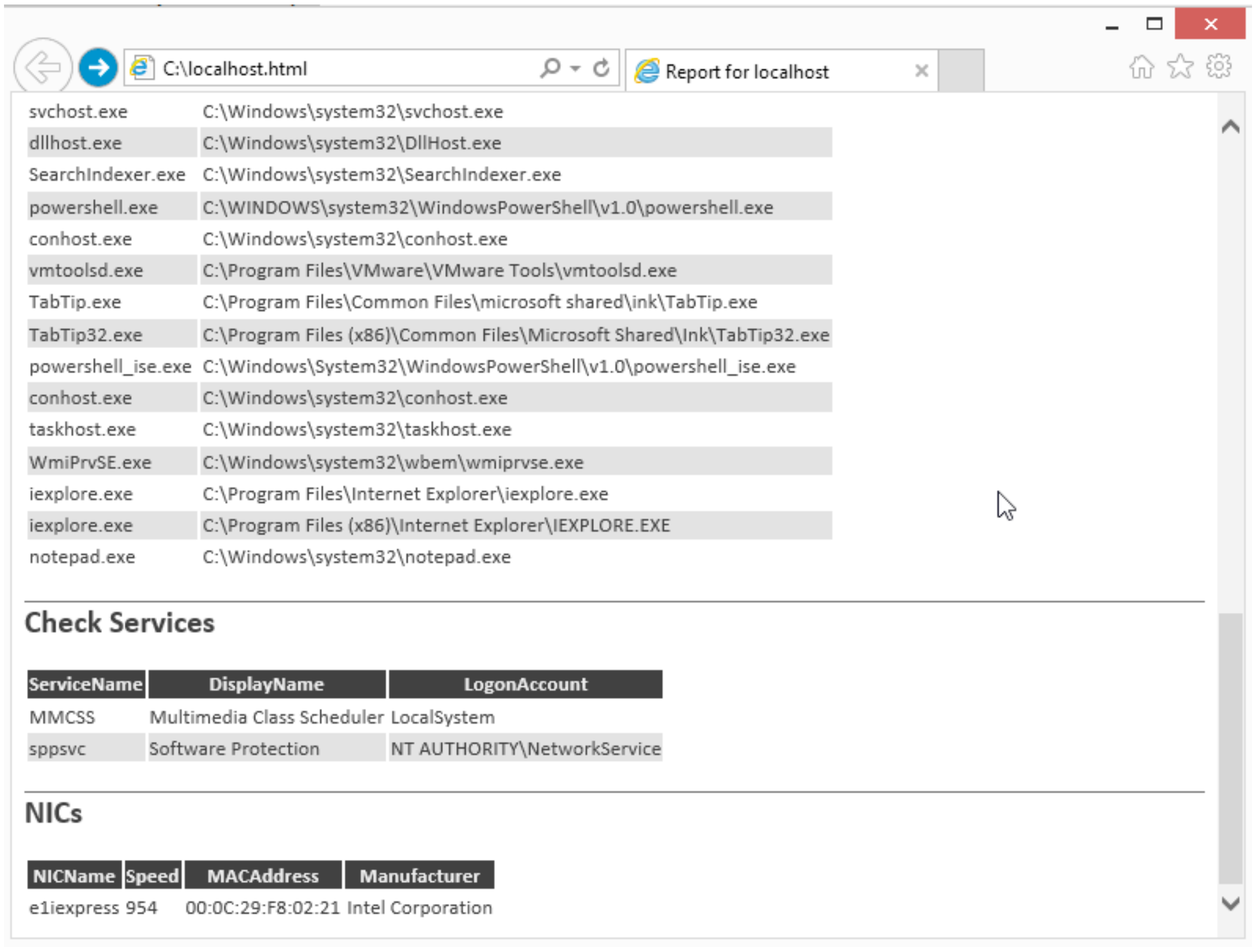
So I'm taking the HTML fragment *without* the -PreContent, piping it to Out-String, and then piping it to my new Set-AlternatingCSSClasses function. I'm telling it to use the CSS class names "even" and "odd." When I get the result back, I'm manually prepending the <H2> heading to each fragment.

I've also added a bit to my CSS file. Here's the whole thing:

```
    body {
        color:#333333;
        font-family:Calibri,Tahoma;
        font-size: 10pt;
    }
    h1 {
        text-align:center;
    }
    h2 {
        border-top:1px solid #666666;
    }
    th {
        font-weight:bold;
        color:#eeeeee;
        background-color:#333333;
```

```
}
.odd  { background-color:#ffffff; }
.even { background-color:#dddddd; }
```

Notice that, while HTML tags like <BODY> and <H1> are referred to by just the tag name in the CSS, my class names are preceded by a period. The final result looks like this:



svchost.exe	C:\Windows\system32\svchost.exe
dllhost.exe	C:\Windows\system32\DllHost.exe
SearchIndexer.exe	C:\Windows\system32\SearchIndexer.exe
powershell.exe	C:\WINDOWS\system32\WindowsPowerShell\v1.0\powershell.exe
conhost.exe	C:\Windows\system32\conhost.exe
vmtoolsd.exe	C:\Program Files\VMware\VMware Tools\vmtoolsd.exe
TabTip.exe	C:\Program Files\Common Files\microsoft shared\ink\TabTip.exe
TabTip32.exe	C:\Program Files (x86)\Common Files\Microsoft Shared\Ink\TabTip32.exe
powershell_ise.exe	C:\Windows\System32\WindowsPowerShell\v1.0\powershell_ise.exe
conhost.exe	C:\Windows\system32\conhost.exe
taskhost.exe	C:\Windows\system32\taskhost.exe
WmiPrvSE.exe	C:\Windows\system32\wbem\wmiprvse.exe
iexplore.exe	C:\Program Files\Internet Explorer\iexplore.exe
iexplore.exe	C:\Program Files (x86)\Internet Explorer\IEXPLORE.EXE
notepad.exe	C:\Windows\system32\notepad.exe

Check Services

ServiceName	DisplayName	LogonAccount
MMCSS	Multimedia Class Scheduler	LocalSystem
sppsvc	Software Protection	NT AUTHORITY\NetworkService

NICs

NICName	Speed	MACAddress	Manufacturer
e1iexpress	954	00:0C:29:F8:02:21	Intel Corporation

So now I have awesome alternating colors on my table rows. This is *cool.* I made the table header rows a bit darker to make them stand out even more.

The Final Product

Here's my final script, which I saved as New-HTMLSystemReport.ps1:

```
 <#
.SYNOPSIS
Generates an HTML-based system report for one or more computers.
Each computer specified will result in a separate HTML file;
specify the -Path as a folder where you want the files written.
Note that existing files will be overwritten.
.PARAMETER ComputerName
One or more computer names or IP addresses to query.
.PARAMETER Path
The path of the folder where the files should be written.
.PARAMETER CssPath
The path and filename of the CSS template to use.
.EXAMPLE
.\New-HTMLSystemReport -ComputerName ONE,TWO `
                       -Path C:\Reports\ `
                       -CssPath c:\style.css
#>
[CmdletBinding()]
param(
    [Parameter(Mandatory=$True,
               ValueFromPipeline=$True,
               ValueFromPipelineByPropertyName=$True)]
    [string[]]$ComputerName,

    [Parameter(Mandatory=$True)]
    [string]$Path,

    [Parameter(Mandatory=$True)]
    [string]$CssPath

)
PROCESS {

function Get-InfoOS {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)][string]$ComputerName
    )
    $os = Get-WmiObject -class Win32_OperatingSystem -ComputerName $ComputerName
    $props = @{'OSVersion'=$os.version;
               'SPVersion'=$os.servicepackmajorversion;
               'OSBuild'=$os.buildnumber}
    New-Object -TypeName PSObject -Property $props
}

function Get-InfoCompSystem {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)][string]$ComputerName
    )
    $cs = Get-WmiObject -class Win32_ComputerSystem -ComputerName $ComputerName
    $props = @{'Model'=$cs.model;
               'Manufacturer'=$cs.manufacturer;
               'RAM (GB)'="{0:N2}" -f ($cs.totalphysicalmemory / 1GB);
               'Sockets'=$cs.numberofprocessors;
               'Cores'=$cs.numberoflogicalprocessors}
    New-Object -TypeName PSObject -Property $props
}

function Get-InfoBadService {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)][string]$ComputerName
    )
    $svcs = Get-WmiObject -class Win32_Service -ComputerName $ComputerName `
           -Filter "StartMode='Auto' AND State<>'Running'"
    foreach ($svc in $svcs) {
        $props = @{'ServiceName'=$svc.name;
                   'LogonAccount'=$svc.startname;
                   'DisplayName'=$svc.displayname}
        New-Object -TypeName PSObject -Property $props
    }
}

function Get-InfoProc {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)][string]$ComputerName
    )
    $procs = Get-WmiObject -class Win32_Process -ComputerName $ComputerName
    foreach ($proc in $procs) {
        $props = @{'ProcName'=$proc.name;
                   'Executable'=$proc.ExecutablePath}
        New-Object -TypeName PSObject -Property $props
    }
}

function Get-InfoNIC {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)][string]$ComputerName
    )
    $nics = Get-WmiObject -class Win32_NetworkAdapter -ComputerName $ComputerName `
           -Filter "PhysicalAdapter=True"
    foreach ($nic in $nics) {
        $props = @{'NICName'=$nic.servicename;
                   'Speed'=$nic.speed / 1MB -as [int];
                   'Manufacturer'=$nic.manufacturer;
                   'MACAddress'=$nic.macaddress}
        New-Object -TypeName PSObject -Property $props
    }
}

function Set-AlternatingCSSClasses {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True,ValueFromPipeline=$True)]
        [string]$HTMLFragment,

        [Parameter(Mandatory=$True)]
        [string]$CSSEvenClass,

        [Parameter(Mandatory=$True)]
        [string]$CssOddClass
    )
    [xml]$xml = $HTMLFragment
    $table = $xml.SelectSingleNode('table')
    $classname = $CSSOddClass
    foreach ($tr in $table.tr) {
        if ($classname -eq $CSSEvenClass) {
            $classname = $CssOddClass
        } else {
            $classname = $CSSEvenClass
        }
        $class = $xml.CreateAttribute('class')
        $class.value = $classname
        $tr.attributes.append($class) | Out-null
    }
    $xml.innerxml | out-string
}

foreach ($computer in $computername) {

    $style = "<style>$(get-content $csspath)</style>"

    try {
        $everything_ok = $true
        Write-Verbose "Checking connectivity to $computer"
        Get-WmiObject -class Win32_BIOS -ComputerName $Computer -EA Stop | Out-Null
    } catch {
        Write-Warning "$computer failed"
        $everything_ok = $false
    }

    if ($everything_ok) {
        $filepath = Join-Path -Path $Path -ChildPath "$computer.html"
        $html_os = Get-InfoOS -ComputerName $computer |
                   ConvertTo-HTML -As List -Fragment `
                                  -PreContent "<h2>OS</h2>" |
                   Out-String

        $html_cs = Get-InfoCompSystem -ComputerName $computer |
                   ConvertTo-HTML -As List -Fragment `
                                  -PreContent "<h2>Hardware</h2>" |
                   Out-String

        $html_pr = Get-InfoProc -ComputerName $computer |
                   ConvertTo-HTML -Fragment |
                   Out-String |
                   Set-AlternatingCSSClasses -CSSEvenClass 'even' -CssOddClass 'odd'
        $html_pr = "<h2>Processes</h2>$html_pr"

        $html_sv = Get-InfoBadService -ComputerName $computer |
                   ConvertTo-HTML -Fragment |
                   Out-String |
                   Set-AlternatingCSSClasses -CSSEvenClass 'even' -CssOddClass 'odd'
        $html_sv = "<h2>Check Services</h2>$html_sv"

        $html_na a= Get-InfoNIC -ComputerName $Computer |
                   ConvertTo-HTML -Fragment |
                   Out-String |
                   Set-AlternatingCSSClasses -CSSEvenClass 'even' -CssOddClass 'odd'
        $html_na = "<h2>NICs</h2>$html_na"

        $params = @{'Head'="<title>Report for $computer</title>$style";
                    'PreContent'="<h1>System Report for $computer</h1>";
                    'PostContent'=$html_os,$html_cs,$html_pr,$html_sv,$html_na}
        ConvertTo-HTML @params | Out-File -FilePath $filepath
    }
}

}
```

And now here's my CSS template, which I save as Style.css:

```
 body {
    color:#333333;
    font-family:Calibri,Tahoma;
    font-size: 10pt;
}
h1 {
    text-align:center;
}
h2 {
    border-top:1px solid #666666;
}

th {
    font-weight:bold;
    color:#eeeeee;
    background-color:#333333;
}
.odd  { background-color:#ffffff; }
.even { background-color:#dddddd; }
```

And, just for fun, I'm going to give you one final version. This one eliminates the need to have Style.css as a separate file, and instead embeds it right into the script. For this version, I've removed the -CssFile parameter, since it's no longer needed.

```
 <#
.SYNOPSIS
Generates an HTML-based system report for one or more computers.
Each computer specified will result in a separate HTML file;
specify the -Path as a folder where you want the files written.
Note that existing files will be overwritten.
.PARAMETER ComputerName
One or more computer names or IP addresses to query.
.PARAMETER Path
The path of the folder where the files should be written.
.PARAMETER CssPath
The path and filename of the CSS template to use.
.EXAMPLE
.\New-HTMLSystemReport -ComputerName ONE,TWO `
                       -Path C:\Reports\
#>
[CmdletBinding()]
param(
    [Parameter(Mandatory=$True,
               ValueFromPipeline=$True,
               ValueFromPipelineByPropertyName=$True)]
    [string[]]$ComputerName,

    [Parameter(Mandatory=$True)]
    [string]$Path
)
PROCESS {

$style = @"
<style>
body {
    color:#333333;
    font-family:Calibri,Tahoma;
    font-size: 10pt;
}
h1 {
    text-align:center;
}
h2 {
    border-top:1px solid #666666;
}

th {
    font-weight:bold;
    color:#eeeeee;
    background-color:#333333;
}
.odd  { background-color:#ffffff; }
.even { background-color:#dddddd; }
</style>
"@

function Get-InfoOS {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)][string]$ComputerName
    )
    $os = Get-WmiObject -class Win32_OperatingSystem -ComputerName $ComputerName
    $props = @{'OSVersion'=$os.version;
               'SPVersion'=$os.servicepackmajorversion;
               'OSBuild'=$os.buildnumber}
    New-Object -TypeName PSObject -Property $props
}

function Get-InfoCompSystem {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)][string]$ComputerName
    )
    $cs = Get-WmiObject -class Win32_ComputerSystem -ComputerName $ComputerName
    $props = @{'Model'=$cs.model;
               'Manufacturer'=$cs.manufacturer;
               'RAM (GB)'="{0:N2}" -f ($cs.totalphysicalmemory / 1GB);
               'Sockets'=$cs.numberofprocessors;
               'Cores'=$cs.numberoflogicalprocessors}
    New-Object -TypeName PSObject -Property $props
}

function Get-InfoBadService {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)][string]$ComputerName
    )
    $svcs = Get-WmiObject -class Win32_Service -ComputerName $ComputerName `
           -Filter "StartMode='Auto' AND State<>'Running'"
    foreach ($svc in $svcs) {
        $props = @{'ServiceName'=$svc.name;
                   'LogonAccount'=$svc.startname;
                   'DisplayName'=$svc.displayname}
        New-Object -TypeName PSObject -Property $props
    }
}

function Get-InfoProc {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)][string]$ComputerName
    )
    $procs = Get-WmiObject -class Win32_Process -ComputerName $ComputerName
    foreach ($proc in $procs) {
        $props = @{'ProcName'=$proc.name;
                   'Executable'=$proc.ExecutablePath}
        New-Object -TypeName PSObject -Property $props
    }
}

function Get-InfoNIC {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)][string]$ComputerName
    )
    $nics = Get-WmiObject -class Win32_NetworkAdapter -ComputerName $ComputerName `
           -Filter "PhysicalAdapter=True"
    foreach ($nic in $nics) {
        $props = @{'NICName'=$nic.servicename;
                   'Speed'=$nic.speed / 1MB -as [int];
                   'Manufacturer'=$nic.manufacturer;
                   'MACAddress'=$nic.macaddress}
        New-Object -TypeName PSObject -Property $props
    }
}

function Set-AlternatingCSSClasses {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True,ValueFromPipeline=$True)]
        [string]$HTMLFragment,

        [Parameter(Mandatory=$True)]
        [string]$CSSEvenClass,

        [Parameter(Mandatory=$True)]
        [string]$CssOddClass
    )
    [xml]$xml = $HTMLFragment
    $table = $xml.SelectSingleNode('table')
    $classname = $CSSOddClass
    foreach ($tr in $table.tr) {
        if ($classname -eq $CSSEvenClass) {
            $classname = $CssOddClass
        } else {
            $classname = $CSSEvenClass
        }
        $class = $xml.CreateAttribute('class')
        $class.value = $classname
        $tr.attributes.append($class) | Out-null
    }
    $xml.innerxml | out-string
}

foreach ($computer in $computername) {
    try {
        $everything_ok = $true
        Write-Verbose "Checking connectivity to $computer"
        Get-WmiObject -class Win32_BIOS -ComputerName $Computer -EA Stop | Out-Null
    } catch {
        Write-Warning "$computer failed"
        $everything_ok = $false
    }

    if ($everything_ok) {
        $filepath = Join-Path -Path $Path -ChildPath "$computer.html"
        $html_os = Get-InfoOS -ComputerName $computer |
                   ConvertTo-HTML -As List -Fragment `
```

```
                               -PreContent "<h2>OS</h2>" |
                   Out-String

        $html_cs = Get-InfoCompSystem -ComputerName $computer |
                   ConvertTo-HTML -As List -Fragment `
                               -PreContent "<h2>Hardware</h2>" |
                   Out-String

        $html_pr = Get-InfoProc -ComputerName $computer |
                   ConvertTo-HTML -Fragment |
                   Out-String |
                   Set-AlternatingCSSClasses -CSSEvenClass 'even' -CssOddClass 'odd'
        $html_pr = "<h2>Processes</h2>$html_pr"

        $html_sv = Get-InfoBadService -ComputerName $computer |
                   ConvertTo-HTML -Fragment |
                   Out-String |
                   Set-AlternatingCSSClasses -CSSEvenClass 'even' -CssOddClass 'odd'
        $html_sv = "<h2>Check Services</h2>$html_sv"

        $html_na = Get-InfoNIC -ComputerName $Computer |
                   ConvertTo-HTML -Fragment |
                   Out-String |
                   Set-AlternatingCSSClasses -CSSEvenClass 'even' -CssOddClass 'odd'
        $html_na = "<h2>NICs</h2>$html_na"

        $params = @{'Head'="<title>Report for $computer</title>$style";
                    'PreContent'="<h1>System Report for $computer</h1>";
                    'PostContent'=$html_os,$html_cs,$html_pr,$html_sv,$html_na}
        ConvertTo-HTML @params | Out-File -FilePath $filepath
    }
}

}
```

So there you have it. I hope you've enjoyed, and I hope you find this useful.