WSA - HW - 2:

5. Find and document as much as possible for Windows PowerShell. Try to compare PowerShell with some classic Unix/Linux shells.

**Windows PowerShell** is Microsoft's task automation and configuration management framework, consisting of a command-line shell and associated scripting language built on .NET Framework. PowerShell provides full access to COM and WMI, enabling administrators to perform administrative tasks on both local and remote Windows systems as well as WS-Management and CIM enabling management of remote Linux systems and network devices.
In PowerShell, administrative tasks are generally performed by *cmdlets* (pronounced *command-lets*), which are specialized .NET classes implementing a particular operation. Sets of cmdlets may be combined together in *scripts*, *executables* (which are standalone applications), or by instantiating regular .NET classes (or WMI/COM Objects).[2][3] These work by accessing data in different data stores, like the filesystem or registry, which are made available to the PowerShell runtime via Windows PowerShell *providers*.
Windows PowerShell also provides a hosting API with which the Windows PowerShell runtime can be embedded inside other applications. These applications can then use Windows PowerShell functionality to implement certain operations, including those exposed via the graphical interface. This capability has been used by Microsoft Exchange Server 2007[2][4] to expose its management functionality as PowerShell cmdlets and providers and implement the graphical management tools as PowerShell hosts which invoke the necessary cmdlets. Other Microsoft applications including Microsoft SQL Server 2008[5] also expose their management interface via PowerShell cmdlets. With PowerShell, graphical interface-based management applications on Windows are layered on top of Windows PowerShell. A PowerShell scripting interface for Windows products is mandated by Microsoft's Common Engineering Criteria.[6]
Windows PowerShell includes its own extensive, console-based help, similar to man pages in Unix shells, via the Get-Help cmdlet and updatable with fresh content using the Update-Help cmdlet and web based content via the -online switch to Get-Help.

**Windows PowerShell Background**

Every released version of Microsoft DOS and Microsoft Windows for personal computers has included a command-line interface tool (shell). These are COMMAND.COM (in installations relying on MS-DOS, including Windows 9x) and cmd.exe (in Windows NT family operating systems). The shell is a command line interpreter that supports a few basic commands. For other purposes, a separate console application must be invoked from the shell. The shell also includes a scripting language (batch files), which can be used to automate various tasks. However, the shell cannot be used to automate all facets of GUI functionality, in part because command-line equivalents of operations exposed via the graphical interface are limited, and the scripting language is elementary and does not allow the creation of complex scripts. In Windows Server 2003, the situation was improved,[7] but scripting support was still considered unsatisfactory.
Microsoft attempted to address some of these shortcomings by introducing the Windows Script Host in 1998 with Windows 98, and its command-line based host: cscript.exe. It integrates with the Active Script engine and allows scripts to be written in compatible languages, such as JScript and VBScript, leveraging the APIs exposed by applications via COM. However, it too has its own deficiencies: it is not integrated with the shell, its documentation is not very accessible, and it quickly gained a reputation as a system vulnerability vector after several high-profile computer viruses exploited weaknesses in its

security provisions. Different versions of Windows provided various special-purpose command line interpreters (such as netsh and WMIC) with their own command sets. None of them were integrated with the command shell; nor were they interoperable.
By 2002 Microsoft had started to develop a new approach to command line management, including a shell called Monad (also known as Microsoft Shell or MSH). The shell and the ideas behind it were published in August 2002 in a white paper entitled Monad Manifesto.[8] Monad was to be a new extensible command shell with a fresh design that would be capable of automating a full range of core administrative tasks. Microsoft first showed off Monad at the Professional Development Conference in Los Angeles in September 2003. A private beta program began a few months later which eventually led to a public beta program.
Microsoft published the first Monad public beta release on June 17, 2005, Beta 2 on September 11, 2005, and Beta 3 on January 10, 2006. Not much later, on April 25, 2006 Microsoft formally announced that Monad had been renamed *Windows PowerShell*, positioning it as a significant part of their management technology offerings.[9] Release Candidate 1 of PowerShell was released at the same time. A significant aspect of both the name change and the RC was that this was now a component of Windows, and not an add-on product.
Release Candidate 2 of PowerShell version 1 was released on September 26, 2006 with final Release to the web (RTW) on November 14, 2006 and announced at TechEd Barcelona. PowerShell for earlier versions of Windows was released on January 30, 2007.[10]
PowerShell v2.0 development began, as is usual at Microsoft[*citation needed*], before PowerShell v1.0 even shipped. During the development, Microsoft shipped three community technology preview (CTP). Microsoft made these releases available to the public. The last CTP release of Windows PowerShell v2.0 was made available in December 2008.
PowerShell v2.0 was completed and released to manufacturing in August 2009, as an integral part of Windows 7 and Windows Server 2008 R2. Versions of PowerShell for Windows XP, Windows Server 2003, Windows Vista and Windows Server 2008 were released in October 2009 and are available for download for both 32-bit and 64-bit platforms.[11]

**Windows PowerShell Overview**

Windows PowerShell can execute four kinds of named commands:[12]

- *cmdlets*, which are .NET programs designed to interact with PowerShell
- PowerShell scripts (files suffixed by .ps1)
- PowerShell functions
- standalone executable programs

If a command is a standalone executable program, PowerShell.exe launches it in a separate process; if it is a cmdlet, it is executed in the PowerShell process. PowerShell provides an interactive command line interface, wherein the commands can be entered and their output displayed. The user interface, based on the Win32 console, offers customizable tab completion but lacks syntax highlighting. PowerShell enables the creation of *aliases* for cmdlets, which are textually translated by PowerShell into invocations of the original commands. PowerShell supports both named and positional parameters for commands. In executing a cmdlet, the job of binding the argument value to the parameter is done by PowerShell itself, but for external executables, arguments are parsed by the external executable independently of PowerShell interpretation.[*citation needed*]
PowerShell makes use of the *pipeline* concept. As with Unix pipelines, PowerShell pipelines are used to compose complex commands, allowing the output of one command to be passed as the input to another, using the | operator. But unlike its Unix counterpart, the PowerShell pipeline is an object pipeline. The data passed between cmdlets are fully typed objects, rather than character streams. When data is piped as objects, the elements they encapsulate retain

their structure and types across cmdlets, without the need for any serialization or explicit parsing of the stream, as would be the need if only character streams were shared. An object can also encapsulate certain functions that work on the contained data, which become available to the recipient command for use.[13][14][*dead link*] For the last cmdlet in a pipeline, PowerShell automatically pipes its output object to the Out-Default cmdlet, which transforms the objects into a stream of format objects and then renders those to the screen.[15][16]
Because all PowerShell objects are .NET objects, they share a .ToString() method, which retrieves the text representation of the data in an object. Windows PowerShell uses this method to convert an object to text. In addition, it also allows formatting definitions to be specified, so the text representation of objects can be customized by choosing which data elements to display, and how. However, in order to maintain backwards compatibility, if an external executable is used in a pipeline, it receives a text stream representing the object, instead of directly integrating with the PowerShell type system.[17][18][19]
The PowerShell *Extended Type System* (*ETS*) is based on the .NET type system, but with extended semantics (for example, propertySets and third-party extensibility). For example, it enables the creation of different views of objects by exposing only a subset of the data fields, properties, and methods, as well as specifying custom formatting and sorting behavior. These views are mapped to the original object using XML-based configuration files.[20]

**Windows PowerShell Cmdlets**

Cmdlets are specialized commands in the PowerShell environment that implement specific functions. These are the native commands in the PowerShell stack. Cmdlets follow a *<verb>-<noun>* naming pattern, such as *Get-ChildItem*, helping to make them self-descriptive.[21] Cmdlets output their results as objects, or collections thereof (including arrays), and can optionally receive input in that form, making them suitable for use as recipients in a pipeline. But, whereas PowerShell allows arrays and other collections of objects to be written to the pipeline, cmdlets always process objects individually. For collections of objects, PowerShell invokes the cmdlet on each object in the collection, in sequence.[21]
Cmdlets are specialized .NET classes, which the PowerShell runtime instantiates and invokes when they are run. Cmdlets derive either from Cmdlet or from PSCmdlet, the latter being used when the cmdlet needs to interact with the PowerShell runtime.[21] These base classes specify certain methods - BeginProcessing(), ProcessRecord() and EndProcessing() - which the cmdlet's implementation overrides to provide the functionality. Whenever a cmdlet is run, these methods are invoked by PowerShell in sequence, with ProcessRecord() being called if it receives pipeline input.[22] If a collection of objects is piped, the method is invoked for each object in the collection. The class implementing the Cmdlet must have one .NET attribute - CmdletAttribute - which specifies the verb and the noun that make up the name of the cmdlet. Common verbs are provided as an enum.[23][24]
If a cmdlet receives either pipeline input or command-line parameter input, there must be a corresponding property in the class, with a mutator implementation. PowerShell invokes the mutator with the parameter value or pipeline input, which is saved by the mutator implementation in class variables. These values are then referred to by the methods which implement the functionality. Properties that map to command-line parameters are marked by ParameterAttribute[25] and are set before the call to BeginProcessing(). Those which map to pipeline input are also flanked by ParameterAttribute, but with the ValueFromPipeline attribute parameter set.[26]
The implementation of these cmdlet classes can refer to any .NET API and may be in any .NET language. In addition, PowerShell makes certain APIs available, such as WriteObject(), which is used to access PowerShell-specific functionality, such as writing resultant objects to

the pipeline. Cmdlets can use .NET data access APIs directly or use the PowerShell infrastructure of PowerShell *Providers*, which make data stores addressable using unique paths. Data stores are exposed using drive letters, and hierarchies within them, addressed as directories. Windows PowerShell ships with providers for the file system, registry, the certificate store, as well as the namespaces for command aliases, variables, and functions.[27] Windows PowerShell also includes various cmdlets for managing various Windows systems, including the file system, or using Windows Management Instrumentation to control Windows components. Other applications can register cmdlets with PowerShell, thus allowing it to manage them, and, if they enclose any datastore (such as databases), they can add specific providers, as well.[*citation needed*]
In PowerShell V2, a more portable version of Cmdlets called Modules have been added. The PowerShell V2 release notes state:
"Modules allow script developers and administrators to partition and organize their Windows PowerShell code in self-contained, reusable units. Code from a module executes in its own self-contained context and does not affect the state outside of the module. Modules also enable you to define a restricted runspace environment by using a script."[*citation needed*]

**Windows PowerShell Pipeline**

PowerShell implements a *pipeline* concept, which enables the output of one cmdlet to be piped as input to another cmdlet. For example, the output of the Get-Process cmdlet could be piped to the Sort-Object cmdlet (e.g., to sort the objects by handle count), and then to the Where-Object to filter any process that has less than 1 MB of paged memory, and then finally to the Select-Object cmdlet to select just the first 10 (i.e., the 10 processes based on handle count).[*citation needed*]
PowerShell pipeline differs from the Unix analog in that structured .NET objects are passed between stages in the pipeline instead of typically unstructured text. Using objects eliminates the need to explicitly parse text output to extract data.[28]

**Windows PowerShell Scripting**

Windows PowerShell includes a dynamically typed scripting language which can implement complex operations using cmdlets imperatively. The scripting language supports variables, functions, branching (if-then-else), loops (while, do, for, and foreach), structured error/exception handling and closures/lambda expressions,[29] as well as integration with .NET. Variables in PowerShell scripts have names that start with $; they can be assigned any value, including the output of cmdlets. Strings can be enclosed either in single quotes or in double quotes: when using double quotes, variables will be expanded even if they are inside the quotation marks. According to the variable syntax, if the path to a file is enclosed in braces preceded by a dollar sign (as in ${C:\foo.txt}), it refers to the contents of the file. If it is used as an L-value, anything assigned to it will be written to the file. When used as an R-value, it will be read from the file. If an object is assigned, it is serialized before storing it.[*citation needed*]
Object members can be accessed using . notation, as in C# syntax. PowerShell provides special variables, such as $args, which is an array of all the command line arguments passed to a function from the command line, and $_, which refers to the current object in the pipeline.[30] PowerShell also provides arrays and associative arrays. The PowerShell scripting language also evaluates arithmetic expressions entered on the command line immediately, and it parses common abbreviations, such as GB, MB, and KB.[31][32]

Using the function keyword, PowerShell provides for the creation of functions, the following general form:[33]

```
function name (Param1, Param2)
{
  Instructions
}
```

The defined function invoke in either of the following forms:[33]

```
name value1 value2
name -Param1 value1 -Param2 value2
```

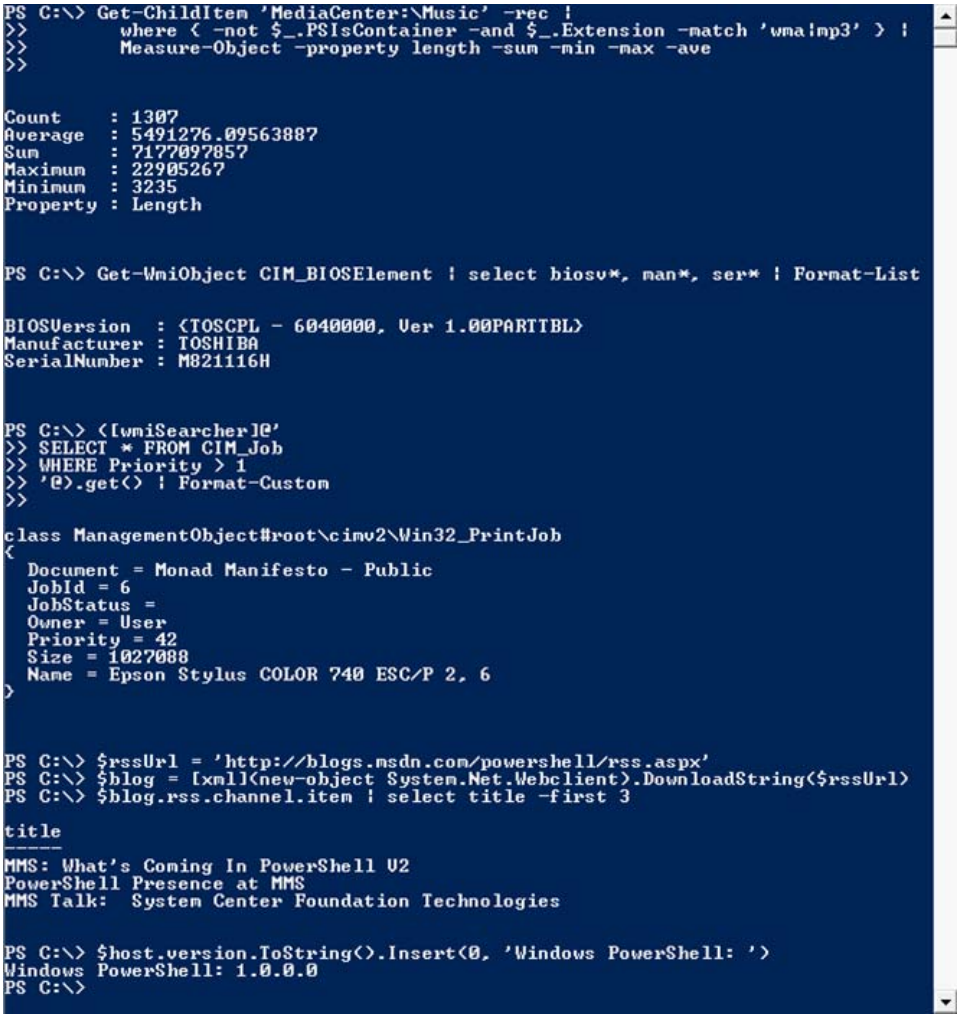
PowerShell supports named parameters, positional parameters, switch parameters and dynamic parameters.[33]
PowerShell allows any .NET methods to be called by providing their namespaces enclosed in brackets ([]), and then using a pair of colons (::) to indicate the static method.[34] For example, [System.Console]::WriteLine("PowerShell"). Objects are created using the New-Object cmdlet. Calling methods of .NET objects is accomplished by using the regular . notation.[34]
PowerShell scripting language accepts strings, both raw and escaped. A string enclosed between single quotation marks is a raw string while a string enclosed between double quotation marks is an escaped string. PowerShell treats straight and curly quotes as equivalent.[35]
For error handling, PowerShell provides a .NET-based exception handling mechanism. In case of errors, objects containing information about the error (Exception object) are thrown, which are caught using the trap keyword. However, the action-or-error is configurable; in case of an error, PowerShell can be configured to silently resume execution, without trapping the exception.[36]
Scripts written using PowerShell can be made to persist across sessions in a .ps1 file. Later, either the entire script or individual functions in the script can be used. Scripts and functions are used analogously with cmdlets, in that they can be used as commands in pipelines, and parameters can be bound to them. Pipeline objects can be passed between functions, scripts, and cmdlets seamlessly. However, script execution is disabled by default and must be enabled explicitly.[37] PowerShell scripts can be signed to verify their integrity, and are subject to Code Access Security.[38]
The PowerShell scripting language supports binary prefix notation similar to the scientific notation supported by many programming languages in the C-family.[*citation needed*]

**Windows PowerShell Hosting**

Another use of PowerShell is being embedded in a management application, which uses the PowerShell runtime to implement the management functionality. For this, PowerShell provides a managed hosting API. Via the APIs, the application can instantiate a *runspace* (one instantiation of the PowerShell runtime), which runs in the application's process and is exposed as a Runspace object.[2] The state of the runspace is encased in a SessionState object. When the runspace is created, the Windows PowerShell runtime initializes the instantiation, including initializing the providers and enumerating the cmdlets, and updates the SessionState object accordingly. The Runspace then must be opened for either synchronous processing or asynchronous processing. After that it can be used to execute commands.[*citation needed*]

To execute a command, a pipeline (represented by a Pipeline object) must be created and associated with the runspace. The pipeline object is then populated with the cmdlets that make up the pipeline. For sequential operations (as in a PowerShell script), a Pipeline object is created for each statement and nested inside another Pipeline object.[2] When a pipeline is created, Windows PowerShell invokes the pipeline processor, which resolves the cmdlets into their respective assemblies (the *command processor*) and adds a reference to them to the pipeline, and associates them with InputPipe, Outputpipe and ErrorOutputPipe objects, to represent the connection with the pipeline. The types are verified and parameters bound using reflection.[2] Once the pipeline is set up, the host calls the Invoke() method to run the commands, or its asynchronous equivalent - InvokeAsync(). If the pipeline has the Write-Host cmdlet at the end of the pipeline, it writes the result onto the console screen. If not, the results are handed over to the host, which might either apply further processing or display it itself.[*citation needed*]
The hosting APIs are used by Microsoft Exchange Server 2007 to provide its management GUI. Each operation exposed in the GUI is mapped to a sequence of PowerShell commands (or pipelines). The host creates the pipeline and executes them. In fact, the interactive PowerShell console itself is a PowerShell host, which interprets the scripts entered at command line and creates the necessary Pipeline objects and invokes them.[*citation needed*]

Some capture of PowerShell v.1.0



**Comparison of cmdlets with similar commands:**

The following table contains a selection of the cmdlets that ship with PowerShell, noting similar commands in other well-known command-line interpreters. Many of these similar commands come out-of-the-box defined as aliases within PowerShell, making it easy for people familiar with other common shells to start working.

| PowerShell (Cmdlet) | PowerShell (Alias) | CMD.EXE / COMMAND.COM | Unix shell | Description |
|---|---|---|---|---|
| Get-ChildItem | gci, dir, ls | dir | ls | List all files / directories in the (current) directory |
| Get-Content | gc, type, cat | type | cat | Get the content of a file |
| Get-Command | gcm | help | help, which | List available commands |
| Get-Help | help, man | help | man | Help on commands |
| Clear-Host | cls, clear | cls | clear | Clear the screen[Note 1] |
| Copy-Item | cpi, copy, cp | copy | cp | Copy one or several files / a whole directory tree |
| Move-Item | mi, move, mv | move | mv | Move a file / a directory to a new location |
| Remove-Item | ri, del, erase, rmdir, rd, rm | del, erase, rmdir, rd | rm, rmdir | Delete a file / a directory |
| Rename-Item | rni, ren, mv | ren, rename | mv | Rename a file / a directory |
| Get-Location | gl, pwd | cd | pwd | Display the current directory/present working directory. |
| Pop-Location | popd | popd | popd | Change the current directory to the directory most recently pushed onto the stack |
| Push-Location | pushd | pushd | pushd | Push the current directory onto the stack |
| Set-Location | sl, cd, chdir | cd, chdir | cd | Change the current directory |
| Tee-Object | tee | n/a | tee | Pipe input to a file or variable, then pass the input along the pipeline |
| Write-Output | echo, write | echo | echo | Print strings, variables etc. to standard output |
| Get-Process | gps, ps | tlist,[Note 2] tasklist[Note 3] | ps | List all currently running processes |
| Stop-Process | spps, kill | kill,[Note 2] taskkill[Note 3] | kill[Note 4] | Stop a running process |
| Select-String | sls | find, findstr | grep | Print lines matching a pattern |
| Set-Variable | sv, set | set | env, export, set, setenv | Set the value of a variable / create a variable |
| Invoke-WebRequest | iwr | n/a | wget, curl | Gets content from a web page on the Internet |

**Alternative implementation**

A project named **Pash** (the name is a pun on the well-known "bash" Unix shell[81]) has been an open source and cross-platform reimplementation of PowerShell via the Mono framework. Pash was created by Igor Moochnick, written in C# and was released under the GNU General Public License. Pash development stalled in 2008, but development was restarted in 2012.[81][82]

**SOME Examples**

Examples are provided first using the long-form canonical syntax and then using more terse UNIX-like and DOS-like aliases that are set up in the default configuration. For a list of all aliases, use the Get-Alias Cmdlet.

**Print "Hello, World!" to console:**

```
PS> Write-Host "Hello, World!"
```

**Stop all processes that begin with the letter p:**

```
PS> Get-Process p* | Stop-Process
PS> ps p* | kill
```

**Find the processes that use more than 1000 MB of memory and kill them:**

```
PS> Get-Process | Where-Object { $_.WS -gt 1000MB } | Stop-Process   # V1/V2 Syntax
PS> Get-Process | Where-Object WS -gt 1000MB | Stop-Process          # V3 Syntax
PS> ps | ? WS -gt 1000MB | kill                                     # V3 Syntax
```

**Calculate the number of bytes in the files in a directory:**

```
PS> Get-Childitem | Measure-Object -Property Length -Sum
PS> ls  | measure length -s
PS> dir | measure length -s
```

**Determine whether a specific process is no longer running:**

```
PS> $processToWatch = Get-Process Notepad
PS> $processToWatch.WaitForExit()
PS> (ps notepad).WaitForExit()
```

**Change the case of a string from lower to upper:**

```
PS> 'hello, world!'.ToUpper()
```

**Insert the string 'ABC' after the first character in the word 'string' to have the result 'sABCtring':**

```
PS> 'string'.Insert(1, 'ABC')
```

**Download a specific RSS feed and show the titles of the 8 most recent entries:**

```
PS> $rssUrl = 'http://blogs.msdn.com/powershell/rss.aspx'
PS> $blog = [xml](new-object System.Net.WebClient).DownloadString($rssUrl)
PS> $blog.rss.channel.item | select title -first 8

PS> $x = new-object xml
PS> $x.load('http://blogs.msdn.com/powershell/rss.aspx')
PS> $x.rss.channel.item | select title -f 8
```

**Sets $UserProfile to the value of the UserProfile environment variable**

```
PS> $UserProfile = $env:UserProfile
```

**Call a static method of a .Net object**

```
PS> [System.Math]::Sqrt(16)
4
```

**Run a command line executable with arguments:**

```
PS> [Array]$arguments = '-h', '15', 'www.Wikipedia.com'
PS> tracert $arguments
```

**Get the serial number of a remote computer from WMI:**

```
PS> Get-WmiObject -ComputerName MyServer -Class Win32_BIOS | Select-Object
```

**SerialNumber**

```
PS> gwmi -co MyServer Win32_BIOS | select SerialNumber
```

**Windows PowerShell File extensions:**

**PS1** – Windows PowerShell shell script[59]
**PSD1** – Windows PowerShell data file (for Version 2)[60]
**PSM1** – Windows PowerShell module file (for Version 2)[61]
**PS1XML** – Windows PowerShell format and type definitions [19][62]
**CLIXML** - Windows PowerShell serialized data[63]
**PSC1** – Windows PowerShell console file [64]
**PSSC** - Windows PowerShell Session Configuration file[65]

**повече информация за Windows PowerShell:**

http://en.wikipedia.org/wiki/Windows_PowerShell
http://technet.microsoft.com/en-us/library/bb978526.aspx
http://technet.microsoft.com/en-us/library/ee221100.aspx
http://www.powershellpro.com/powershell-tutorial-introduction/tutorial-powershell-cmdlet/
http://blogs.technet.com/b/musings_of_a_technical_tam/archive/2012/06/04/windows-powershell-self-training-guide.aspx