

# Actuar en aplicaciones con COM

## 1. Microsoft Office 2007

En esta sección, nos centraremos en la manipulación de los objetos COM en la forma que interactúan con las aplicaciones informáticas contenidas en el pack Office 2007 (Word, PowerPoint, Excel, etc.). Quizás no lo sepa, pero durante la instalación del pack Office en un sistema operativo Windows, a este último se le acreditan un buen número de objetos COM Microsoft Office, que permiten a los usuarios utilizarlos en sus scripts o programas.

La simple utilización de la función `Get-ProgID` (véase el apartado buscar un objeto) asociada a los nombres Word, PowerPoint, Excel etc., le dará una lista impresionante de objetos COM que podrá utilizar a su conveniencia. Si necesita más información, puede acceder al sitio web MSDN para obtener una descripción detallada de las numerosas propiedades y métodos de cada objeto.

### a. Microsoft PowerPoint 2007

Para darnos cuenta de hasta qué punto la manipulación de la aplicación PowerPoint 2007 es realmente sencilla gracias a los objetos COM, veamos cómo en unas pocas líneas, podemos automatizar la apertura de un archivo. En primer lugar, creamos una instancia del objeto `PowerPoint.Application`.

```
PS > $ObjetoPowerPoint = New-object -ComObject PowerPoint.Application
```

Observemos de paso que después de instanciar el objeto COM `PowerPoint.Application`, el proces `POWERPNT`, correspondiente al proceso de ejecución Microsoft PowerPoint, está lógicamente activo.

```
PS > Get-Process
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
-----	-----	-----	-----	-----	-----	--	-----
25	2	604	1004	25	0,05	2456	acrotray
103	5	1100	1568	32		1104	alg
44	1	364	1168	16		960	ati2evxx
234	8	3324	3256	51	0,33	2944	ccApp
216	6	2472	1148	40		1820	ccEvtMgr
733	15	15000	12952	73		568	CcmExec
217	5	5820	136	66		1876	cisvc
33	2	516	1404	19		1924	DefWatch
447	12	14760	11636	81	41,41	4012	explorer
0	0	0	16	0		0	Idle
302	13	25100	712	154	11,81	3824	POWERPNT

Veamos ahora, una vez más gracias al comando `Get-Member`, qué propiedades o métodos podemos utilizar para modificar el comportamiento de nuestro objeto.

```
PS > $ObjetoPowerPoint | Get-Member
```

TypeName: Microsoft.Office.Interop.PowerPoint.ApplicationClass

Name	MemberType	Definition
----	-----	-----
Activate	Method	void Activate ()
GetOptionFlag	Method	bool GetOptionFlag (int, bool)

Help	Method	void Help (string, int)
LaunchSpelling	Method	void LaunchSpelling (Document...
PPFileDialog	Method	IUnknown PPFileDialog (PpFile...
Quit	Method	void Quit ()
...		

Después de un rápido vistazo a los miembros disponibles, empezamos por hacer visible la aplicación gracias a la propiedad `Visible`.

```
PS > $ObjetoPowerPoint.Visible = 'MsoTrue'
```



Aplicamos a la propiedad `Visible` un valor de tipo `MsoTriState` específico de Microsoft Office. Aunque, según la versión que utilice, podrá estar obligado a utilizar la forma siguiente:

```
PS > $ObjetoPowerPoint.Visible = $true
```

Una vez se inicia la aplicación y es visible, usamos el método `Add` con el fin de crear una presentación vacía. Esto equivale a hacer en PowerPoint «Archivo/Nueva presentación». Una vez se efectúa esta operación, podremos añadir diapositivas manualmente (ya que la ventana aparece en la pantalla) o en línea de comandos.

```
PS > $ObjetoPowerPoint.Presentations.Add()
```



Por supuesto, también es posible abrir una presentación ya existente con el método `Open`:

```
PS > $ObjetoPowerPoint.Presentations.Open(<Ruta del archivo>)
```

En resumen, la utilización de objetos COM Microsoft Office hace accesible mediante script todas las operaciones realizadas gráficamente, abrir/cerrar un archivo, guardar un archivo, añadir una presentación, etc.

Para ofrecerle una percepción más completa, he aquí un ejemplo de script PowerShell. Éste crea una presentación PowerPoint en la que cada diapositiva es una fotografía en formato JPG situada en un directorio determinado. La ruta se pasa por parámetro.

```
#diapo.ps1
#Creación de una presentación PowerPoint a partir de imágenes jpg

Param([string]$path)
$lista = Get-ChildItem $path | Where {$_.extension -eq '.jpg'} |
Foreach{$_.Fullname}

if($lista -ne $null)
{
    $Transition = $true
    $time_diapo = 2
    $objPPT = New-object -ComObject PowerPoint.Application
    $objPPT.Visible = 'MsoTrue'
    $projet = $objPPT.Presentations.Add()
    foreach($image in $lista)
    {
        $slide = $Proyecto.Slides.Add(1, 1)
        $slide.Shapes.AddPicture($image,1,0,0,720,540)
        $slide.SlideShowTransition.AdvanceOnTime = $Transition
        $slide.SlideShowTransition.AdvanceTime = $time_diapo
    }
    $projet.SlideShowSettings.Run()
```

```

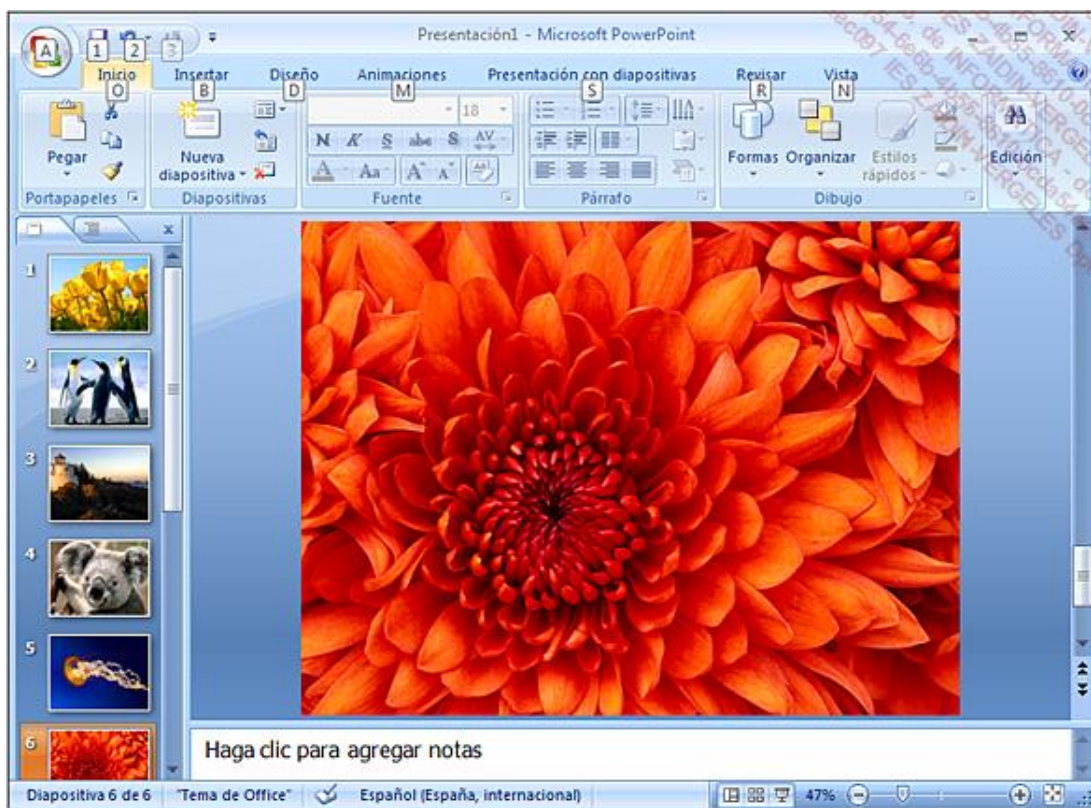
}
Else
{
    Write-Host '¡No existen fotografías en formato JPEG en este directorio!'
}

```

*Ejemplo de utilización con la muestra de imágenes de Windows 7.*

```
PS> ./diapo.ps1 'C:\Users\Public\Pictures\Sample Pictures'
```

El resultado es el siguiente:



## **b. Microsoft Word 2007**

Siempre dentro del dominio de objetos COM del pack Office, vamos a ver los que Word 2007 nos propone.


Como usted sabe, el pack Office 2007 es el origen de las nuevas extensiones que añaden una «x» a las extensiones que ya conocemos (\*.docx, \*.pptx, etc.). Más que un simple cambio, estas nuevas extensiones son la señal de un nuevo tipo de documento Office en formato XML. Por consiguiente, estos archivos no se pueden utilizar con las versiones anteriores del Pack Office. Sin embargo, para mantener una compatibilidad ascendente de documentos Office, el pack 2007 permite guardar los archivos en un modo llamado de «compatibilidad».

Así, seleccionando el formato de registro correcto, podrá seguir compartiendo sus documentos con otras personas que no utilicen Office 2007.

Pero donde esto llega a ser realmente molesto, es en el caso de conversión masiva de documentos. La única solución sigue siendo abrir los documentos uno por uno, guardarlos en el modo de compatibilidad 97-2003, y cerrarlos nuevamente. Es decir, esto es una tarea candidata ideal para convertirse en script PowerShell.

Tomemos el ejemplo de documentos Word. Para abrir y posteriormente grabar un documento, tenemos en primer que instanciar el objeto COM Word.Application y después aplicar el método Open sobre el miembro Document.

```
PS > $objWord = New-object -ComObject Word.Application
PS > $objWord.Visible = 'MsoTrue'
PS > [void]$objWord.Documents.Open(<Nombre del archivo.docx>)
```

 Habrá observado la utilización de «[void]» delante de la llamada al método Open. Esto permite que no se muestre en la consola el resultado del comando. Cuando se utiliza el método Open, éste genera decenas de líneas correspondientes a todas las propiedades del documento abierto. Ahora bien, como no tenemos que hacer nada con la información (en este ejemplo) y ésta nos «contamina» la consola, [void] nos permite no mostrarla. En lugar de utilizar [void] hubiéramos podido asignar este comando a una variable, lo cual sería lo mismo.

Seguidamente, llega el momento delicado de guardar vía el método `SaveAs` y de sus dieciséis parámetros.

Parámetro	Descripción
FileName	Permite definir el nombre del documento. Si el documento ya ha sido guardado, el valor por defecto es su nombre actual.
FileFormat	Permite definir el formato del documento guardado. En el caso de un documento Word, puede tratarse de cualquier valor de <code>WdSaveFormat</code> (véase la tabla sobre los formatos de registro Word).
LockComments	Permite definir un bloqueo de los comentarios si el valor <code>True</code> está asociado. El valor por defecto es <code>False</code> .
Password	Permite definir una contraseña para abrir un documento.
AddToRecentFiles	Permite definir la adición de un documento a la lista de los últimos archivos utilizados si se le asociad el valor <code>True</code> . El valor por defecto es <code>True</code> .
WritePassword	Permite definir una contraseña para la modificación del documento.
ReadOnlyRecommended	Permite definir la apertura del documento en modo sólo lectura si el valor <code>True</code> está asociado. El valor por defecto es <code>False</code> .
EmbedTrueTypeFonts	Permite definir la grabación de políticas TrueType si el valor <code>True</code> está asociado.
SaveNativePictureFormat	Permite grabar en versión Windows gráficos importados de otra plataforma si la variable <code>True</code> está asociada.
SaveFormsData	Define el registro de los datos introducidos por un usuario en un formulario.
SaveAsAOCELetter	Permite guardar un documento en formato AOCE ( <i>Apple Open Collaboration Environment</i> ) si el valor <code>True</code> está asociado.
Encoding	Permite definir la codificación de un documento (Arabic, Cyrillic, etc.).
InsertLineBreaks	Permite insertar los retornos de carro de cada fin de línea si el valor <code>True</code> está asociado. Esta propiedad puede ser interesante en el caso de un registro en formato de texto.
AllowSubstitutions	Permite, en el caso de un registro en formato de texto, sustituir los símbolos del documento por una apariencia similar.
LineEnding	Permite definir la manera en que Word va a señalar los saltos de líneas para los documentos registrados en formato de texto.

AddBiDiMarks	Permite la adición de caracteres de control al archivo para conservar la dirección bidireccional del texto en el documento de origen.
--------------	---

Utilizando hábilmente el método `SaveAs`, podremos grabar el documento en un formato «documento Word 97-2003»; y esto gracias al parámetro `FileFormat`, del que mostramos a continuación la lista no exhaustiva de sus valores posibles.

Formato	Extensión asociada	Valor del parámetro
<code>wdFormatDocument</code>	*.doc	0
<code>wdFormatHTML</code>	*.htm, *.html	8
<code>wdFormatPDF</code>	*.pdf	17
<code>wdFormatTemplate</code>	*.dot	1
<code>wdFormatText</code>	*.txt	2
<code>wdFormatXMLDocument</code>	*.docx	12
<code>wdFormatXMLTemplate</code>	*.dotx	14



Para poder grabar en formato PDF, la aplicación Office 2007 debe disponer de un complemento adicional (gratuito) descargable del sitio Microsoft.

Al elegir el valor 0, el archivo se grabará en modo compatibilidad Office 97-2003. Sin embargo, preste atención a especificar el nombre de archivo con su extensión «.doc» sino, el archivo mantendrá su extensión «.docx». Tenga en cuenta, que el comando `SaveAs` necesita que sus argumentos sean de tipo *PSReference* (referenciados a un objeto), preste atención por tanto en especificar correctamente el tipo `[ref]` delante de cada variable.

```
PS > $objword.ActiveDocument.SaveAs([ref]<nombre del archivo.doc>,[ref]0)
```

Finalmente, para terminar el proceso, llamamos a los métodos `Close` y `Quit` respectivamente en el miembro `Documents` y en el objeto `COM`.

```
PS > $objWord.Documents.Close()
PS > $objWord.Quit()
```

Al asociar los elementos que acabamos de presentar, obtendremos el script siguiente que permite convertir de forma masiva todo documento nativo Word 2007 de un directorio en un documento Word 97-2003.

```
Function Convert-Doc
{
param ([String]$path = '.')
$lista = Get-ChildItem $path *.docx
$objWord = New-Object -ComObject Word.Application
Foreach ($archivo in $lista)
{
[void]$objWord.Documents.Open(($archivo.FullName))
$nombre_archivo = $($archivo.FullName).replace('.docx',' .doc')
$objword.ActiveDocument.SaveAs([ref]$nombre_archivo,[ref]0)
$objWord.Documents.Close()
}
$objWord.Quit()
```

```
}
```



El script no elimina la ocurrencia del archivo .docx, pero crea un nuevo archivo .doc con el mismo nombre.

## Hablemos de seguridad

En este ejemplo, iremos un poco más lejos en la gestión de la protección de los archivos Word, puesto que vamos ahora a centrar nuestra atención en la protección de un documento.

Quizás no lo sepa, pero con Microsoft Office, es posible definir una contraseña para la lectura y la modificación de sus documentos Word y Excel.

Como hemos visto en la parte anterior, el método SaveAs para guardar un documento *dispone de los parámetros Password y WritePassword*. En estos dos parámetros vamos a definir una contraseña para un documento. Comencemos por abrir un documento Word existente:

```
PS > $objWord = New-object -ComObject Word.Application
PS > $objWord.Visible = 'MsoTrue'
PS > [void]$objWord.Documents.Open(<Nombre del archivo>)
```

Luego, procedemos a su registro, teniendo cuidado de informar los parámetros Password y WritePassword. Tenga en cuenta que para dejar vacíos los numerosos parámetros opcionales del método SaveAs, utilizamos un objeto de tipo System.Missing (Sin información).

```
PS > $m = [system.type]::missing
PS > $objWord.ActiveDocument.SaveAs([ref]<nombre del archivo>,[ref]12,
[ref]$m,[ref] <Password>,[ref]$m,[ref]<WritePassword>)
```

Así, como se muestra en la figura siguiente, en la siguiente apertura del documento, Word invitará al usuario a introducir la contraseña para la lectura y/o la modificación.



Petición de contraseña para un documento Word

La función siguiente nos permitirá automatizar esta tarea, aprovechando simplemente la ruta completa del directorio que contiene los archivos a proteger así como las contraseñas para la lectura y para la modificación.

```
Function Secure-Doc
{
param ([String]$path = '.',[String]$password='', `
[String]$WritePassword='')
$m = [system.type]::missing
$aarchivos = Get-ChildItem $path *.docx
$objWord = New-object -ComObject Word.Application
Foreach ($doc in $aarchivos)
{
[void]$objWord.Documents.Open($doc.FullName)
$nombre_archivoseguro = $doc.FullName + 's'
$objword.ActiveDocument.SaveAs($nombre_archivoseguro, `
[ref]12,[ref]$m,`
```

```
[ref]$password,[ref]$m,[ref]$writepassword)
$objWord.Documents.Close()
}
$objWord.Quit()
}
```

Por el contrario, como lo demuestran los dos líneas de PowerShell siguientes, la eliminación de la seguridad de un documento exigirá abrir el archivo con el método `Open`, teniendo cuidado de informar las contraseñas necesarias para la apertura y la modificación del documento. Después debemos guardar de nuevo el documento asignándole un valor nulo a las propiedades `Password` y `WritePassword`.

```
# Abrir el documento con las contraseñas necesarias

PS > $objWord.Documents.Open(<Nombre del archivo>,$m,$m,$m, `
<password>,$m,$m, <writepassword>)

# Grabar el documento sin contraseña

PS > $objWord.ActiveDocument.SaveAs([ref]<Nombre del archivo>, `
[ref]12,[ref]$m, [ref]','', [ref]$m,[ref]'')
```

## 2. Windows Live Messenger

Continuando nuestro viaje a través de los objetos COM, detengámonos unos instantes en los objetos COM de Windows Live Messenger. En este apartado, procederemos a mostrarle cómo interactuar con su mensajería instantánea, y únicamente con PowerShell.

No obstante tenga prudencia, ya que no todas las clases COM están disponibles en ciertas versiones de Windows. Por consiguiente, para una mejor comprensión, tenga en cuenta que todos los ejemplos que proponemos se realizaron con Windows Live Messenger 8.0 bajo el sistema operativo Windows 7.

### a. Obtener el estado de la conexión

Empezaremos por instanciar un objeto COM que nos permita de interactuar con Windows Messenger.

```
PS > $msn = New-Object -ComObject Messenger.UIAutomation.1
```

Observe ahora un poco más de cerca los miembros que contiene:

```
PS > $msn | Get-Member

TypeName: System.__ComObject#{d50c3486-0f89-48f8-b204-3604629dee10}

Name                MemberType Definition
----                -
AddContact           Method      void AddContact (int, string)
AutoSignin           Method      void AutoSignin ()
CreateGroup           Method      IDispatch CreateGroup (strin...
FetchUserTile         Method      void FetchUserTile (Varian...
FindContact           Method      void FindContact (int, ...
GetContact            Method      IDispatch GetContact (str...
HideContactCard       Method      void HideContactCard (int)
```

InstantMessage	Method	IDispatch InstantMess...
...		
MyServiceName	Property	string MyServiceName () {get}
MySigninName	Property	string MySigninName () {get}
MyStatus	Property	MISTATUS MyStatus () {get} {...
ReceiveFileDirectory	Property	string ReceiveFileDirectory ()...
Services	Property	IDispatch Services () {get}
Window	Property	IDispatch Window () {get}

Podemos ver que existen numerosos métodos y propiedades. Empecemos por aplicar la propiedad `MyStatus`, que como su nombre deja entrever devuelve el estado de su conexión.

```
PS > $msn.MyStatus
2
```

Como podrá constatar, la propiedad `MyStatus` devuelve un entero asociado a un estado. La lista de los estados definidos por Windows Live Messenger se muestra a continuación:

Valor	Estado
1	No conectado
2	En línea
6	Desconectado
10	Ocupado
14	Vuelvo en un minuto
34	Ausente
50	Al telefono
66	Salí a comer

Observemos que la propiedad `MyStatus` es accesible tanto en lectura como en escritura.

```
PS > $msn | Get-Member MyStatus | Format-List

TypeName      : System.__ComObject#{d50c3486-0f89-48f8-b204-3604629dee10}
Name           : MyStatus
MemberType    : Property
Definition    : MISTATUS MyStatus () {get} {set}
```

Esto significa que también podemos cambiar nuestro estado desde la consola PowerShell.

Ejemplo:

```
PS > If ($MSN.MyStatus -eq 2) {$MSN.MyStatus = 10}
```

También es posible averiguar información sobre contactos, y así conocer su estado. Para ello, debemos crear un objeto contacto que nos será devuelto por el método `GetContact`, y luego usar su propiedad `MyStatus`.

Para ilustrar esta propuesta, veamos un ejemplo, en forma de script, que nos permitirá obtener el estado de uno de



nuestros contactos:

```
# Get-MSNStatus.ps1
# Permite obtener el estado de un contacto

param ($mailAddress=${Throw=';Debe informar una dirección de correo!'})

$msn = New-Object -ComObject Messenger.UIAutomation.1
$contacto = $msn.GetContact($mailAddress,$msn.MyServiceId)
$estado = $Contact.Status
$nombre = $Contact.FriendlyName
switch ($estado)
{
    1 {Write-host "$nombre no está conectado"}
    2 {Write-host "$nombre está en línea"}
    6 {Write-host "$nombre está desconectado"}
    10 {Write-host "$nombre vuelve en un minuto"}
    14 {Write-host "$nombre está ausente"}
    34 {Write-host "$nombre está ocupado"}
    50 {Write-host "$nombre está al telefono"}
    66 {Write-host "$nombre salió a comer"}
    default {Write-host "Desconocido"}
}
```

Ejemplo de utilización:

```
PS > ./Get-MSNStatus.ps1 eduardobraceros@gaz.org
Edu está ocupado
```

Podemos obtener asimismo la lista de nuestros amigos conectados, como en el ejemplo siguiente:

```
#Get-OnLineFriends
#Permite listar los amigos conectados

$msn = New-Object -ComObject Messenger.UIAutomation.1
$msn.MyContacts | Where{$_.status -ne 1} | ft FriendlyName
```

Resultado:

FriendlyName	Status
-----	-----
Edu	34
Jose	2
Pol	66

## b. Abrir y cerrar sesión

Veamos ahora otras funcionalidades del objeto como son la apertura y cierre de la sesión. En lo que respecta al cierre, nos bastará simplemente con llamar al método Close del objeto COM. La apertura por su parte, requiere una etapa adicional que vamos detallar.

Para poder interactuar en la aplicación, la primera etapa consiste en instanciar el objeto `Messenger.UIAutomation.1` así como hacerlo visible.

```
PS > $MSN = New-Object -ComObject Messenger.UIAutomation.1
```

```
PS > $MSN.Window.Show()
```

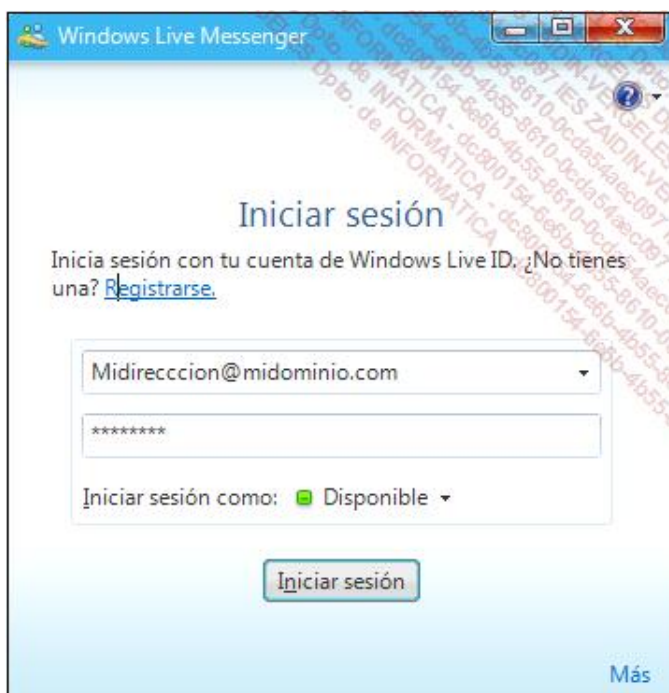
Procedemos seguidamente a iniciar la sesión con el método Signin con los parámetros:

- `HwndParent` igual a 0, para indicar que se trata de una ventana independiente,
- la dirección de correo de la cuenta Microsoft Passport,
- la contraseña asociada.

```
PS > $MSN.Signin(0,'Midirecccion@midominio.com','P@SSw0rd')
```

Estamos en presencia de la ventana de conexión precargada (véase la figura siguiente). Pero falta un detalle: iniciar la conexión pulsando la tecla [Intro]. Para ello, sencillamente, gracias a la clase `System.Windows.Forms.SendKeys` del Framework, enviamos una secuencia de teclas correspondientes a pulsar [Intro] en el teclado.

```
PS > [System.Reflection.Assembly]::LoadWithPartialName('System.windows.forms')  
PS > [System.Windows.Forms.SendKeys]::SendWait('{enter}')
```



*Ventana de conexión Windows Live Messenger*

### c. Enviar un mensaje instantaneo

Para enviar un mensaje, usamos el método `InstantMessage` al que daremos la dirección del destinatario. Después, como para iniciar la sesión, vamos a usar una vez más las secuencias de teclas para el envío de un mensaje instantáneo.

Primero, para introducir el texto en el espacio previsto para la edición del mensaje (figura siguiente). Después, para validar el envío.

```
[void]$MSN.InstantMessage('eduardobracer0s@gaz.org')  
[void][System.Windows.Forms.SendKeys]::SendWait('Buenos días !!')  
[void][System.Windows.Forms.SendKeys]::SendWait('{enter}')
```

Si lo desea, únicamente necesita introducir estas tres líneas de una función adaptada para permitirle enviar mensajes

automáticamente a algunos de sus contactos.



Espacio de edición

#### d. Exportar sus contactos

Nada más sencillo para recuperar la lista de sus contactos. La propiedad `MyContacts` nos devuelve, en formato de colección, la lista completa. Después, mediante un `Format-Table`, mostraremos las propiedades que consideremos interesantes. Ejemplo con un filtro sobre las propiedades `SigninName` y `FriendlyName`:

```
PS > $msn.MyContacts | Format-Table SigninName,FriendlyName
```

SigninName	FriendlyName
josebarranco@gaz.org	Jose
eduardobracame@gaz.org	Edu
polperez@gaz.org	Pol

No queda más que guardarlos, por ejemplo, en un archivo delimitado por comas, usando el comando `Export-Csv`, y listo.

```
PS > $msn.MyContacts | Select-Object SigninName,FriendlyName |  
Export-Csv -path 'C:\Temp\Mis_Contactos.csv'
```

Obtendremos:

```
#TYPE System.Management.Automation.PSCustomObject  
SigninName,FriendlyName  
josebarranco@gaz.org,Jose  
eduardobracame@gaz.org,Edu  
polperez@gaz.org,Pol
```

Observe que esta vez no hemos utilizado `Format-Table`, sino `Select-Object`. El motivo es bien simple: `Format-Table` es un commandlet hecho para el formateo de datos destinados a la consola y no está adaptado para utilizar otra tubería. Además `Select-Object` efectúa un filtrado de los objetos pasados a través del pipe (tubería).

### 3. Internet Explorer

Como para numerosas aplicaciones Microsoft, Windows dispone nativamente de un objeto COM que permite interactuar con el navegador Internet Explorer. Y para demostrar con qué soltura es esto posible, vamos a describir cómo navegar por Internet o bien visualizar una página HTML.

#### a. Navegador

Con el fin de poder utilizar Internet Explorer, necesitaremos en primer lugar instanciar el objeto COM `InternetExplorer.Application.1`. La sintaxis es la siguiente:

```
PS > $IE = New-Object -ComObject InternetExplorer.Application.1
```

Ahora que se ha creado el objeto, asignamos el valor `$true` a la propiedad visible con el fin de visualizar el navegador por pantalla:

```
PS > $IE.Visible = $true
```

La aplicación Internet Explorer es ahora visible. A continuación le atribuimos también una determinada altura y anchura:

```
PS > $IE.Height = 700 #Asignación de la altura
```

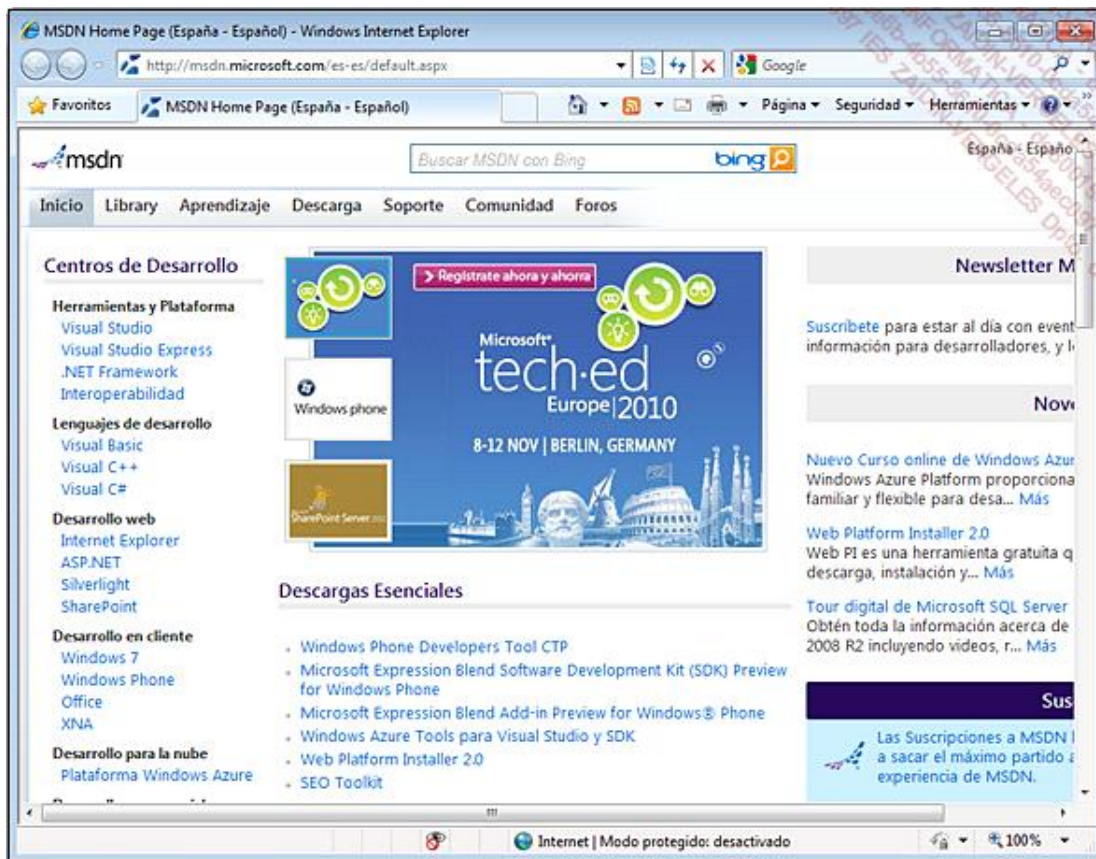
```
PS > $IE.Width = 900 #Asignación de la anchura
```



Como para los Winforms, los tamaños atribuidos están indicados en píxeles.

El navegador ya está operativo, ahora no queda más que utilizar el método `Navigate` para indicarle a qué página Web queremos acceder.

```
PS > $IE.Navigate('http://msdn.microsoft.com')
```



*Apertura de la página Web vía PowerShell*

## **b. Visualizar una página HTML**

Acabamos de ver cómo navegar por Internet utilizando el método `navigate`, pero éste también permite la visualización de sus páginas web. Supongamos que queremos mostrar una página HTML editada con Word, que contiene un mensaje de información. Pues bien, una vez más comenzaremos por crear y hacer visible nuestro objeto Internet Explorer:

```
PS > $IE = New-Object -ComObject InternetExplorer.Application.1
PS > $IE.Visible = $true
```

Seguimos atribuyendo un tamaño determinado a la ventana Internet Explorer para ajustar nuestro texto:

```
PS > $IE.Height = 190 #Atribución de la altura
PS > $IE.Width = 550 #Atribución de la anchura
```

Para depurar la ventana de Internet Explorer, podemos enmascarar la barra de herramientas. Y para ello, debemos asignar un valor de 0 a la propiedad `ToolBar`:

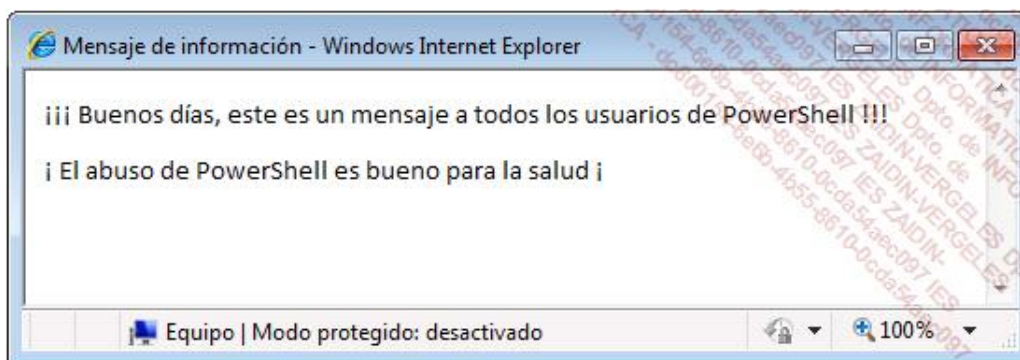
```
PS > $IE.ToolBar = 0
```

Finalmente, para visualizar el mensaje, hacemos una llamada al método `Navigate` con esta vez como argumento, no una dirección URL, sino la ruta completa a su documento HTML:

```
PS > $IE.Navigate('C:\Temp\MensajeInformacion.htm')
```

Hecho, ahora no queda más que cambiar el título de la ventana para hacerla más explícita y observar el resultado (figura inferior).

```
PS > $IE.Document.Title = "Mensaje de información"
```



*Visualización del archivo HTML*

Como puede observarse, con sólo unas líneas de código, acabamos de crear una ventana gráfica que contiene un mensaje, mientras que la misma operación con la creación de una Winform habría sido más larga y compleja. Todo y que, el uso no es el mismo.

Antes de terminar, he aquí el script completo para visualizar una página HTML.

```
#Show-HtmlFile.ps1
# Visualización de una página HTML

param([string]$archivo,[int]$altura,[int]$anchura)

$IE = New-Object -ComObject InternetExplorer.Application.1
$IE.Visible = $true

$IE.Height = $altura      #Atribución de la altura
$IE.Width = $anchura      #Atribución de la anchura
$IE.Navigate($archivo)     #Visualización de la página HTML
$IE.Document.Title = "Mensaje de información"
```

Una vez más, ha podido verificar hasta qué punto es fácil interactuar con las aplicaciones si usamos PowerShell, y todo gracias a los objetos COM.

## 4. Windows Script Host (WSH)

Windows Script Host es el intérprete de scripts nativo de los sistemas operativos desde Windows 98 hasta Windows Server 2008. Se utiliza para interpretar los scripts VBS (*Visual Basic Script*). Es una tecnología que Microsoft sigue manteniendo y que además de su papel de intérprete de scripts incorpora un conjunto de objetos COM: WshShell, WshNetwork y WshController, todos ellos se pueden utilizar, desde la consola PowerShell.

WSH es mucho más antiguo que PowerShell, por ello en la mayoría de los casos, las funciones propuestas por sus objetos COM están disponibles nativamente con objetos .NET. Sin embargo, algunos métodos siguen siendo exclusivos a los objetos COM de WSH, como la asignación de una unidad de red o la adición de una impresora también en red.

### a. WshShell

El objeto COM WshShell, es una instancia de Shell WSH. De este modo, los usuarios de PowerShell, vamos a poder, gracias a este objeto, utilizar métodos que permitirán interactuar con el registro de Windows y con el visor de eventos, enviar secuencias de teclas y ejecutar programas. Esta instancia del Shell WSH se emplea todavía mucho en los scripts VBS, ya que estos no pueden apoyarse en el Framework .NET.

Como es costumbre, la instanciación del objeto WshShell necesita el commandlet `New-Object`:

```
PS > $WshShell = New-Object -ComObject Wscript.Shell
```

WScript.Shell es el ProgID del objeto WshShell. Una vez creado el objeto, únicamente nos queda utilizar sus funcionalidades. Veamos cuáles son los miembros de los que este objeto dispone:

```
PS > $WshShell | Get-Member
```

```
TypeName: System.__ComObject#{41904400-be18-11d3-  
a28b-00104bd35090}
```

Name	MemberType	Definition
----	-----	-----
AppActivate	Method	bool AppActivate (Variant...
CreateShortcut	Method	IDispatch CreateShortcut...
Exec	Method	IWshExec Exec (string)
ExpandEnvironmentStrings	Method	string ExpandEnvironmentS...
LogEvent	Method	bool LogEvent (Variant,...
Popup	Method	int Popup (string, Varia...
RegDelete	Method	void RegDelete (string)
RegRead	Method	Variant RegRead (string)
RegWrite	Method	void RegWrite (strin...
Run	Method	int Run (string, Vari...
SendKeys	Method	void SendKeys (strin...
Environment	Parameterized Property	IWshEnvironment Environ...
CurrentDirectory	Property	string Current Directory ()...
SpecialFolders	Property	IWshCollection SpecialFol...

### Ejemplos de utilización:

Internet nos ofrece multitud de casos en materia de ejemplos de utilización del objeto WshShell, por lo que no nos detendremos hablando sobre las distintas variantes posibles de los métodos propuestos por el objeto. Sin embargo, he aquí algunos métodos interesantes, como por ejemplo `Popup` quien, como su nombre indica, permite la creación de una ventana pop-up, al igual que `Shortcut` para la creación de un acceso directo:

### **Creación de una ventana de Pop-up**

La creación de un pop-up se vuelve una tarea muy sencilla con el objeto WshShell, pues como muestra la figura siguiente, bastará con instanciar el objeto, y aplicarle el método Popup con el texto deseado.

```
PS > $WshShell = New-Object -ComObject Wscript.Shell
PS > $WshShell.Popup('Hello World')
```



*Visualización de un Popup vía WshShell*

### **Manipulación del registro de Windows**

Aunque PowerShell sepa gestionar de forma nativa el registro de Windows con los comandos `*-item` (véase el capítulo Descubra PowerShell, sección Navegación por los directorios y archivos), he aquí una segunda solución que consiste en utilizar los métodos `RegRead` y `RegWrite` para respectivamente leer y escribir en el registro de Windows.

*Ejemplo:*

```
PS > $WshShell = New-Object -ComObject Wscript.Shell

# Lectura de la clave correspondiente a la versión del cliente FTP FileZilla
PS > $WshShell.RegRead('HKLM\SOFTWARE\FileZilla Client\Version')
3.0.1

# Cambio del valor de la clave Version
PS > $WshShell.RegWrite('HKLM\SOFTWARE\FileZilla Client\
Version','4.0','REG_SZ')
```

### **Creación de un acceso directo**

Último ejemplo de utilización que proponemos: la creación de un acceso directo vía el método `CreateShortcut`:

```
$objetivo = 'C:\Temp\MiArchivo.doc'
$vinculo = 'C:\Users\Oscar\Desktop\MiAccesoDirecto.lnk'
$WshShell = New-Object -ComObject WScript.Shell
$accesodirecto = $WshShell.CreateShortCut($lien)
$accesodirecto.Targetpath = $objetivo
$accesodirecto.Save()
```

En resumen, el objeto WshShell no nos aporta nada realmente nuevo, a no ser por algunos métodos como la creación rápida de un pop-up y de un acceso directo. Sin embargo, siempre es tranquilizador saber que existe una multitud de caminos para alcanzar sus fines.

## **b. WshNetwork**

WshNetwork es el segundo objeto COM propuesto por WSH. Referenciado bajo el ProgID `WScript.Network`, permite



principalmente añadir o suprimir impresoras y unidades de red, así como la enumeración de algunas propiedades como el nombre de usuario actual, el dominio y el nombre del ordenador.

### c. Ejemplos de utilización

#### Mapeo de una unidad de red

Para mapear una unidad de red, empezaremos por crear una instancia del objeto WshNetwork.

```
$WshNetwork = New-Object -ComObject Wscript.Network
```

Después, utilizamos el método MapNetworkDrive con los argumentos que propone:

Argumento	Obligatorio	Descripción
LocalName [string]	Sí	Define la carta asignada para la unidad de red.
RemoteName [string]	Sí	Define el directorio compartido.
UdpadeteProfile [Bool]	No	Indica gracias a un booleano si la información sobre el montaje de la unidad de red deberá incluirse en el perfil. El valor por defecto es \$false.
UserName [string]	No	Especifica el nombre de usuario si el mapeo requiere una autenticación.
Password [string]	No	Especifica la contraseña asociada al nombre de usuario.

Por ejemplo, la línea de comandos siguiente asociará, gracias a su autenticación, una unidad de red al directorio \\SERVIDOR2008\partición.

```
PS > $WshNetwork.MapNetworkDrive('x:', '\\SERVIDOR2008\particion',$false,
'Nombre usuario', 'P@ssw0rd')
```

WshNetwork propone de igual forma un método inverso para permitir la desconexión de una unidad de red. Para ello, será necesario utilizar el método RemoveNetworkDrive.

Ejemplo:

```
PS > $WshNetwork.RemoveNetworkDrive('x:')
```

#### Conectar una impresora de red

Para conectar una impresora de red, empezaremos por crear una instancia del objeto WshNetwork.

```
PS > $WshNetwork = New-Object -ComObject Wscript.Network
```

Después, utilizaremos el método AddWindowsPrinterConnection. Por ejemplo, supongamos que una impresora de red llamada Impresora\_1 está situada en el servidor SERVIDOR2008. El comando que permitirá conectarla es:

```
PS > $WshNetwork.AddWindowsPrinterConnection('\\\\SERVIDOR2008\Impresora_1')
```

Y al igual que las unidades de red, el objeto nos ofrece igualmente la posibilidad de eliminar una conexión de impresora por el método RemovePrinterConnection.

Ejemplo:



```
PS > WshNetwork.RemovePrinterConnection('\\SERVIDOR2008\Impresora_1')
```