Sistemas Operativos en Red

UD 04. Introducción a PowerShell - Parte 1

Autores: Sergi García, Gloria Muñoz

Actualizado Septiembre 2025

Licencia

Nomenclatura

A lo largo de este tema se utilizarán diferentes símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:

📖 **Importante**

❕ **Atención**

💬 **Interesante**

ÍNDICE

Unidad 04. Introducción a PowerShell - Parte 1

1. Introducción a PowerShell

PowerShell es una poderosa herramienta de automatización y scripting diseñada principalmente para administradores de sistemas. Desarrollada por Microsoft, se basa en la plataforma .NET y ofrece una interfaz de línea de comandos y un lenguaje de scripting que es especialmente útil para la automatización de tareas administrativas y la gestión de configuraciones. PowerShell permite a los usuarios controlar y automatizar la administración de sistemas operativos (Windows, Linux y macOS) y aplicaciones que están diseñadas para ser gestionadas a través de la línea de comandos.

Características clave de PowerShell:

- **Automatización:** A través de scripts y cmdlets (comandos nativos de PowerShell), los usuarios pueden automatizar tareas repetitivas, lo que aumenta la eficiencia y reduce la posibilidad de errores humanos.
- **Gestión basada en objetos:** A diferencia de otros shells que devuelven texto, PowerShell trabaja con objetos .NET. Esto significa que los resultados de los comandos son objetos que pueden ser manipulados directamente en sus propiedades y métodos.
- **Interoperabilidad:** PowerShell permite la integración con otros lenguajes y procesos, ampliando sus capacidades y facilitando la ejecución de tareas complejas.
- **Seguridad:** Incluye características como la ejecución restringida de scripts, que ayuda a prevenir la ejecución de código no deseado.

PowerShell es una herramienta esencial para los profesionales de TI modernos, ofreciendo capacidades extensas para la gestión y automatización de sistemas a gran escala.

2. Variables en PowerShell

En PowerShell, las variables se utilizan para almacenar información que puede ser utilizada y manipulada en el script. Se definen utilizando el signo $ seguido del nombre de la variable. PowerShell es un lenguaje de tipado dinámico, lo que significa que no necesitas declarar el tipo de datos que una variable almacenará, aunque puedes hacerlo si deseas.

2.1 Declaración de Variables

Aquí hay algunos ejemplos básicos de cómo puedes declarar y utilizar variables en PowerShell:

Aquí te ofrezco una explicación detallada de cada ejemplo proporcionado, lo cual te ayudará a comprender cómo funcionan las variables y operaciones en PowerShell:

Variables Numéricas

```
$numero = 25
$pi = 3.14159
```

Explicación:

- $numero es una variable que almacena el valor entero 25.
- $pi es una variable que guarda el valor decimal 3.14159, que es una aproximación del número pi.

Variables de Cadena

```
$saludo = "Hola, mundo"
$nombre = 'Juan'
```

Explicación:

- $saludo contiene la cadena de texto "Hola, mundo".
- $nombre guarda la cadena "Juan". PowerShell permite el uso de comillas simples o dobles para definir cadenas de texto.

Variables Booleanas

```
$verdadero = $true
$falso = $false
```

Explicación:

- $verdadero es una variable booleana que almacena el valor True.
- $falso almacena el valor False. Estas variables se usan comúnmente para controlar el flujo de ejecución en scripts.

Array

```
$numeros = 1, 2, 3, 4, 5
$nombres = "Ana", "Beto", "Carlos"
```

Explicación:

- $numeros es un array que contiene cinco elementos numéricos.
- $nombres es un array de cadenas que incluye los nombres "Ana", "Beto", y "Carlos". Los arrays son útiles para almacenar colecciones de datos.

Hashtable

```
$persona = @{
    Nombre = "Elena"
    Edad = 32
    Ciudad = "Madrid"
}
```

Explicación:

- $persona es una hashtable que almacena datos en pares clave-valor. Aquí, Nombre, Edad, y Ciudad son las claves, y cada una tiene un valor asociado, permitiendo un acceso rápido y organizado a cada elemento.

2.2 Operaciones con Variables

Concatenación de Cadenas

```
$nombreCompleto = $nombre + " Perez"
Write-Host "Nombre completo: $nombreCompleto"
```

Explicación:

Concatena el valor de $nombre con la cadena " Perez" para formar el nombre completo y luego lo imprime.

Operaciones Matemáticas

```
$suma = $numero + 15
$multiplicacion = $numero * 2
```

Explicación:

- $suma calcula la suma de 25 y 15, resultando 40.
- $multiplicacion realiza la multiplicación de 25 por 2, dando como resultado 50.

Accediendo a Elementos de un Array

```
$primerElemento = $numeros[0]
$ultimoElemento = $nombres[-1]
```

Explicación:

- $primerElemento extrae el primer elemento del array $numeros, que es 1.
- $ultimoElemento obtiene el último elemento del array $nombres, que es "Carlos".

Modificando y Añadiendo Datos a una Hashtable

```
$persona.Edad = 33
$persona["Profesion"] = "Ingeniero"
```

Explicación:

- Modifica la edad en la hashtable $persona a 33.
- Añade un nuevo par clave-valor, con la clave "Profesion" y el valor "Ingeniero".

Interpolación de Cadenas

```
$mensaje = "Tu nombre es $nombre y tienes $($persona.Edad) años."
Write-Host $mensaje
```

Explicación:

Forma una cadena interpolada que incluye el nombre y la edad almacenada en la hashtable $persona, y luego la imprime.

Variables de Entorno

```
$path = $env:PATH
Write-Host "El PATH actual es: $path"
```

Explicación:

Recupera la variable de entorno PATH y la imprime. Esta variable de entorno contiene las rutas de los directorios donde el sistema operativo y otros programas buscan ejecutables.

3. Manejo de Excepciones en PowerShell

El manejo de excepciones en PowerShell se realiza principalmente a través de los bloques try, catch, y finally. Aquí te explico cada uno:

- **try**: Aquí colocas el código que puede causar una excepción o error durante su ejecución.
- **catch**: Este bloque captura la excepción lanzada en el bloque try. Puedes manejar el error o registrar información sobre el error aquí.
- **finally**: Este bloque se ejecuta siempre, independientemente de si se produjo una excepción o no, y es útil para limpieza o liberación de recursos, por ejemplo.

3.1 Ejemplos de Manejo de Excepciones

Captura Simple

```
try {
    $resultado = 1 / 0
} catch {
    Write-Host "Ocurrió un error: $_"
}
```

Explicación:

- Este código intenta realizar una división por cero, lo cual es matemáticamente imposible y genera una excepción en la mayoría de los lenguajes de programación, incluido PowerShell.
- La cláusula `catch` captura la excepción generada y muestra un mensaje de error genérico junto con la descripción del error.

Captura Específica de Tipo de Excepción

```
try {
    [int]$numero = "abc"
} catch [System.FormatException] {
    Write-Host "Formato incorrecto."
} catch {
    Write-Host "Error desconocido."
}
```

Explicación:

- Aquí, el script intenta convertir la cadena "abc" a un entero, lo que fallará y lanzará una `System.FormatException` porque la cadena no tiene el formato correcto.
- El primer bloque `catch` está diseñado para manejar específicamente errores de formato, mientras que el segundo `catch` captura cualquier otro tipo de excepción no especificada anteriormente.

Uso del Bloque Finally

```
try {
    $stream = [System.IO.StreamWriter] "archivo.txt"
    $stream.WriteLine("Hola mundo")
} catch {
    Write-Host "No se pudo escribir en el archivo."
} finally {
    $stream.Close()
}
```

Explicación:

- Este código intenta abrir un archivo y escribir en él. Si algo sale mal durante el proceso (por ejemplo, el archivo no tiene permisos de escritura), la excepción se captura y se maneja.
- El bloque `finally` se ejecuta siempre, independientemente de si se lanzó una excepción, asegurando que el recurso (en este caso, un archivo abierto) se cierre correctamente.

Propagando Excepciones

```
try {
    throw "¡Algo salió mal!"
} catch {
    Write-Host "Capturado: $_"
    throw
}
```

Explicación:

- Aquí se lanza manualmente una excepción con `throw "¡Algo salió mal!"`.
- La excepción es capturada, se imprime un mensaje, y luego se relanza la misma excepción para que pueda ser capturada por un manejador de excepciones de nivel superior, si lo hay.

Acceso a la Excepción

```
try {
    Get-Item "path/que/no/existe"
} catch {
    Write-Host "Error: $($_.Exception.Message)"
}
```

Explicación:

- Intenta obtener un elemento de un camino que no existe, lo que causa una excepción.
- La excepción es capturada y se accede a su mensaje a través de `$_`, que es la variable automática que contiene información sobre la excepción actual en el bloque `catch`.

Condiciones Específicas

```
try {
    $numero = 10 / 0
} catch {
    if ($_.Exception -is [System.DivideByZeroException]) {
        Write-Host "División por cero detectada."
    } else {
        Write-Host "Otro tipo de error."
    }
}
```

Explicación:

- Este código también intenta una división por cero.
- Cuando se captura la excepción, se verifica si es una `System.DivideByZeroException` usando `-is`. Si es así, se maneja de manera específica; de lo contrario, se maneja como un error genérico.

Captura con Varios Tipos de Excepciones

```
try {
    [int]$numero = "no es un número"
} catch [System.FormatException], [System.OverflowException] {
    Write-Host "Error de formato o desbordamiento."
}
```

Explicación:

- Intenta convertir una cadena claramente no numérica a un entero, lo cual generará una `System.FormatException`.
- Este `catch` está configurado para capturar tanto `System.FormatException` como `System.OverflowException`, proporcionando un manejo común para ambos tipos de error.

4. MOSTRAR MENSAJES EN LA CONSOLA EN POWERSHELL

Los cmdlets más comunes para mostrar mensajes en la consola son Write-Host, Write-Output, Write-Verbose, Write-Warning, y Write-Error. Aquí te explico cómo y cuándo usar cada uno:

- **Write-Host**: muestra mensajes en la consola, pudiendo personalizar colores. No pasa los objetos a la tubería si no se especifica con -NoNewline.
- **Write-Output:** envía objetos por el pipeline, ideal para continuar procesamientos.
- **Write-Verbose:** muestra mensajes detallados usualmente para depuración, que se muestran solo si se habilita el parámetro -Verbose.
- **Write-Warning:** muestra advertencias en amarillo, útil para alertas no críticas.
- **Write-Error**: muestra errores en rojo, útil para destacar fallos críticos.

4.1 Ejemplos de Impresión en Consola

Imprimir Texto Simple

```
Write-Host "Hola, mundo."
```

Explicación:

Write-Host es usado para imprimir el mensaje "Hola, mundo." directamente en la consola. Este es el uso más básico y directo de Write-Host para mostrar información simple al usuario.

Imprimir con cambio de línea personalizado

```
Write-Host "Hola," -NoNewline; Write-Host " mundo."
```

Explicación:

Aquí se utiliza Write-Host con la opción -NoNewline para evitar que se añada un salto de línea después de "Hola,". Inmediatamente después, se imprime " mundo." Esto resulta en la salida "Hola, mundo." en la misma línea, demostrando cómo controlar los saltos de línea en la impresión.

Imprimir con Colores

```
Write-Host "Texto en verde." -ForegroundColor Green
```

Explicación:

Este comando cambia el color del texto a verde usando el parámetro -ForegroundColor. Es útil para destacar mensajes importantes o para diferenciar tipos de mensajes por color.

Imprimir Salida de un Comando

```
Write-Output (Get-Date)
```

Explicación:

Write-Output envía la salida del comando Get-Date, que obtiene la fecha y hora actual, a la siguiente etapa en el pipeline. Aunque no se ve un efecto visual directo en la consola, Write-Output es más adecuado para pasar datos entre cmdlets en un pipeline.

Usar Write-Verbose

```
Write-Verbose "Mensaje detallado del proceso." -Verbose
```

Explicación:

Write-Verbose proporciona información detallada sobre la ejecución de un script, útil para la depuración y el seguimiento. Sin embargo, estos mensajes solo se muestran si se activa el parámetro -Verbose.

Mostrar una advertencia

```
Write-Warning "Advertencia: Queda poco espacio en disco."
```

Explicación:

Write-Warning emite una advertencia, que se muestra en color amarillo por defecto. Este tipo de mensajes sirve para alertar al usuario sobre condiciones que no son necesariamente errores pero que podrían requerir atención.

Emitir un error

```
Write-Error "Error crítico: Fallo de operación."
```

Explicación:

Write-Error muestra un mensaje de error en color rojo, indicando problemas que han ocurrido durante la ejecución del script. A diferencia de Write-Host o Write-Output, Write-Error también puede ser capturado y manejado dentro de bloques try-catch.

Combinar textos y variables

```
$nombre = "Ana"
Write-Host "Hola, $nombre"
```

Explicación:

Aquí se demuestra cómo incluir el contenido de una variable ($nombre) en un mensaje impreso. Este es un ejemplo de interpolación de cadenas en PowerShell.

Imprimir múltiples líneas con un solo comando

```
"Primera línea", "Segunda línea" | Write-Output
```

Explicación:

Este ejemplo usa un array de cadenas, cada elemento del cual es pasado a Write-Output a través del pipeline. Cada cadena se imprime en su propia línea.

Imprimir listado de objetos

```
$colores = "Rojo", "Verde", "Azul"
$colores | Write-Host
```

Explicación:

Similar al ejemplo anterior, pero utilizando Write-Host para imprimir cada color en una línea nueva, demostrando la capacidad de Write-Host para manejar múltiples entradas del pipeline.

Imprimir Texto Alineado

```
Write-Host "Inicio de línea" -NoNewline; Write-Host " y fin de línea."
```

Explicación:

Combina dos Write-Host con -NoNewline para mantener la salida en la misma línea, útil para situaciones donde se requiere un control preciso del formato del texto.

Imprimir con Formato Tabular

```
$info = [pscustomobject]@{Nombre="Luis"; Edad=30},
[pscustomobject]@{Nombre="Marta"; Edad=25}
$info | Format-Table | Write-Output
```

Explicación:

Crea objetos personalizados con propiedades Nombre y Edad, los organiza en una tabla mediante Format-Table, y luego pasa esta tabla a Write-Output. Es una manera efectiva de presentar datos estructurados de manera clara y organizada.

Imprimir y redirigir a un Archivo

```
Write-Output "Este es un mensaje" > mensaje.txt
```

Explicación:

Redirige la salida de Write-Output a un archivo llamado mensaje.txt, demostrando cómo se pueden guardar mensajes o datos en archivos directamente desde la consola de PowerShell.

5. Estructuras condicionales If-Else en PowerShell

Las estructuras if-else permiten ejecutar bloques de código de manera condicional. Funcionan evaluando una expresión que devuelve true o false y ejecutando el bloque de código correspondiente basado en el resultado de esa evaluación.

Sintaxis Básica de If-Else en PowerShell

```
if (condición) {
    # código si la condición es verdadera
} elseif (otra condición) {
    # código si la otra condición es verdadera
} else {
    # código si ninguna de las condiciones anteriores es verdadera
}
```

Explicación:

Esta es la estructura básica para crear una lógica condicional en PowerShell. Comienza con un if que evalúa una condición. Si la condición es verdadera, ejecuta el código dentro del bloque. Si no, pasa a evaluar una condición en un elseif (si existe), y finalmente tiene un bloque else para ejecutar un código si todas las condiciones anteriores son falsas.

5.1 Ejemplos de estructuras condicionales If-Else

If Simple

```
$edad = 20
if ($edad -ge 18) {
    Write-Host "Eres mayor de edad."
}
```

Explicación:

Verifica si la variable $edad es mayor o igual a 18. Si es cierto, imprime "Eres mayor de edad." Este tipo de condicional es útil para decisiones rápidas basadas en un único criterio.

If-Else

```
$edad = 16
if ($edad -ge 18) {
    Write-Host "Eres mayor de edad."
} else {
    Write-Host "No eres mayor de edad."
}
```

Explicación:

Similar al anterior pero con un bloque else que cubre el caso en que la condición no se cumpla. Aquí, si $edad no es al menos 18, se ejecuta el código en else, imprimiendo "No eres mayor de edad."

If-ElseIf-Else

```
$calificacion = 85
if ($calificacion -ge 90) {
    Write-Host "Excelente"
} elseif ($calificacion -ge 80) {
    Write-Host "Bueno"
} else {
    Write-Host "Necesita mejorar"
}
```

Explicación:

Este ejemplo maneja múltiples condiciones para clasificar una calificación. Dependiendo del rango en el que la calificación se encuentre, imprime un mensaje diferente.

Condiciones Compuestas

```
$edad = 25
$tieneLicencia = $true
if ($edad -ge 18 -and $tieneLicencia) {
    Write-Host "Puede conducir"
```

```
} else {
    Write-Host "No puede conducir"
}
```

Explicación:

Evalúa dos condiciones simultáneamente ($edad debe ser al menos 18 y $tieneLicencia debe ser verdadero). Solo si ambas condiciones son verdaderas, se permite conducir.

Negación de Condición

```
$usuarioLogueado = $false
if (-not $usuarioLogueado) {
    Write-Host "Usuario no logueado"
}
```

Explicación:

Utiliza el operador de negación -not para verificar si $usuarioLogueado es falso, es decir, si el usuario no está logueado.

Operador -or

```
$temperatura = 30
if ($temperatura -lt 15 -or $temperatura -gt 25) {
    Write-Host "Temperatura fuera del rango cómodo"
}
```

Explicación:

Verifica si la temperatura está fuera de un rango cómodo (menor que 15 o mayor que 25). Si alguna de las condiciones es verdadera, ejecuta el bloque de código.

Comparación de cadenas

```
$usuario = "admin"
if ($usuario -eq "admin") {
    Write-Host "Usuario es administrador"
}
```

Explicación:

Compara directamente la variable $usuario con la cadena "admin" para verificar si el usuario tiene privilegios de administrador.

Verificación de nulos

```
$producto = $null
if ($null -eq $producto) {
    Write-Host "Producto no definido"
}
```

Explicación:

Verifica si la variable $producto es nula, lo que indica que no ha sido definida o inicializada.

Evaluación de variables booleanas directamente

```
$activo = $true
if ($activo) {
    Write-Host "El sistema está activo"
}
```

Explicación:

Dado que $activo es una variable booleana, se puede evaluar directamente en el if sin necesidad de compararla explícitamente con true o false.

Evaluación de arrays

```
$numeros = 1, 2, 3
if ($numeros.Contains(2)) {
    Write-Host "El número 2 está en el array"
}
```

Explicación:

Utiliza el método .Contains() para verificar si el array $numeros contiene el elemento 2.

Uso de funciones en condiciones

```
function EsPar($numero) {
    return $numero % 2 -eq 0
}

$miNumero = 4
if (EsPar $miNumero) {
    Write-Host "$miNumero es par"
}
```

Explicación:

Define una función EsPar que determina si un número es par. Luego, usa esta función dentro de una condición if para evaluar si $miNumero es par.

Comparaciones con tipos de datos específicos

```
$fecha = Get-Date
if ($fecha -is [DateTime]) {
    Write-Host "Fecha es de tipo DateTime"
}
```

Explicación:

Utiliza el operador -is para verificar si la variable $fecha es del tipo [DateTime].

Estos ejemplos demuestran cómo puedes utilizar las estructuras if-else para realizar decisiones

basadas en condiciones complejas, manipulación de tipos de datos, y lógica de negocio en tus scripts de PowerShell.

6. Lectura de la entrada estándar en PowerShell

En PowerShell, la entrada estándar se puede leer a través del cmdlet Read-Host, que permite a los usuarios interactuar con scripts mediante la introducción de datos en tiempo de ejecución. Esta capacidad es esencial para scripts que requieren input dinámico o personalización basada en el usuario.

6.1 Ejemplos de lectura de la entrada estándar

Leer una Cadena Simple

```
$nombre = Read-Host "Introduce tu nombre"
Write-Host "Hola, $nombre!"
```

Explicación:

Este ejemplo pide al usuario que introduzca su nombre. El valor introducido se almacena en la variable $nombre y luego se usa para saludar al usuario.

Convertir la Entrada en un Número Entero

```
$edad = Read-Host "Introduce tu edad"
$edad = [int]$edad
Write-Host "Tienes $edad años."
```

Explicación:

Aquí se solicita la edad del usuario. La entrada, que por defecto es una cadena, se convierte explícitamente a un entero ([int]) antes de ser almacenada y utilizada.

Leer un Booleano

```
$respuesta = Read-Host "¿Eres mayor de 18 años? (sí/no)"
if ($respuesta -eq "sí") {
    Write-Host "Acceso concedido."
} else {
    Write-Host "Acceso denegado."
}
```

Explicación:

Se pide al usuario que confirme si es mayor de edad. Dependiendo de la respuesta, el script toma una decisión de control de acceso.

Validación de entrada

```
do {
    $email = Read-Host "Introduce tu correo electrónico"
} while (-not $email -match "^\S+@\S+\.\S+$")
Write-Host "Email válido: $email"
```

Explicación:

Solicita al usuario su correo electrónico repetidamente hasta que introduzca un formato válido (simple validación con expresión regular).

Lectura de contraseña

```
$password = Read-Host "Introduce tu contraseña" -AsSecureString
Write-Host "Contraseña guardada de forma segura."
```

Explicación:

Pide al usuario que introduzca una contraseña, utilizando el parámetro -AsSecureString para asegurar que la entrada se maneje de manera segura en memoria.

Elección múltiple

```
$colorFavorito = Read-Host "Elige tu color favorito (Rojo, Verde, Azul)"
switch ($colorFavorito) {
    "Rojo" { Write-Host "Has elegido Rojo." }
    "Verde" { Write-Host "Has elegido Verde." }
    "Azul" { Write-Host "Has elegido Azul." }
    default { Write-Host "No has elegido un color válido." }
}
```

Explicación:

Este script le permite al usuario elegir entre tres opciones. Utiliza un switch para manejar la respuesta.

Ejemplo 7: Confirmación Sí/No

```
$confirmacion = Read-Host "¿Deseas continuar? (sí/no)"
if ($confirmacion -eq "sí") {
    Write-Host "Proceso continuado."
} else {
    Write-Host "Proceso detenido."
}
```

Explicación:

Pide al usuario que confirme si desea continuar con el proceso. La acción se toma en base a su respuesta.

Entrada de número con validación

```
do {
    $numero = Read-Host "Introduce un número (0-100)"
} while (-not ($numero -as [int]) -or $numero -lt 0 -or $numero -gt 100)
Write-Host "Número válido: $numero"
```

Explicación:

Pide un número entre 0 y 100, asegurando que la entrada sea un entero válido dentro del rango especificado.

Ejemplo 9: Lectura de fecha

```
$fecha = Read-Host "Introduce tu fecha de nacimiento (formato:
YYYY-MM-DD)"
$fecha = [datetime]::Parse($fecha)
Write-Host "Fecha de nacimiento: $fecha"
```

Explicación:

Solicita la fecha de nacimiento del usuario y la convierte a un objeto de tipo DateTime.

Lista de valores separados por comas

```
$items = Read-Host "Introduce una lista de artículos, separados por
comas"
$arrayItems = $items -split ","
Write-Host "Has introducido $($arrayItems.Count) artículos."
```

Explicación:

Pide al usuario que introduzca una lista de artículos separados por comas y luego divide la cadena en un array de artículos individuales.

7. EJERCICIOS RESUELTOS DE EJEMPLO

Ejercicio 1: Cálculo de Descuentos de Tienda

Desarrolla un script en PowerShell para ayudar a una tienda a calcular descuentos en el punto de venta. El script debe:

- Solicitar al usuario el precio original de un producto.
- Solicitar al usuario el porcentaje de descuento a aplicar.
- Calcular y mostrar el precio final después del descuento.
- Manejar cualquier error de entrada (por ejemplo, entradas no numéricas o porcentajes inválidos).

Solución:

```
try {
    $precioOriginal = [double](Read-Host "Ingrese el precio original
del producto")
    $descuento = [double](Read-Host "Ingrese el porcentaje de descuento
(sin '%')")

    if ($descuento -lt 0 -or $descuento -gt 100) {
        throw "El porcentaje de descuento debe estar entre 0 y 100."
    }

    $precioFinal = $precioOriginal - ($precioOriginal * $descuento /
```

```
100)
    Write-Host "El precio final después del descuento es: $precioFinal"
} catch {
    Write-Host "Error: $_"
}
```

Explicación:

- El script pide al usuario que introduzca el precio original y el porcentaje de descuento. Ambos valores se convierten a tipo double.
- Si el porcentaje de descuento no es válido (menor que 0 o mayor que 100), se lanza una excepción.
- Se calcula el precio final aplicando el descuento al precio original.
- Las excepciones se manejan para capturar errores en la conversión de tipos o en los valores del porcentaje.

Ejercicio 2: Registro de Participantes en Evento

Crea un script en PowerShell para registrar participantes en un evento. El script debe:

- Solicitar el nombre, edad y correo electrónico de un participante.
- Verificar que la edad es mayor o igual a 18 años.
- Verificar que el correo electrónico tenga un formato válido.
- Imprimir un mensaje de confirmación con los datos del participante o un mensaje de error si hay problemas con la entrada.

Solución:

```
try {
    $nombre = Read-Host "Ingrese su nombre"
    $edad = [int](Read-Host "Ingrese su edad")
    if ($edad -lt 18) {
        throw "Debe ser mayor de 18 años para registrarse."
    }

    $email = Read-Host "Ingrese su correo electrónico"
    if (-not $email -match "^\S+@\S+\.\S+$") {
        throw "Formato de correo electrónico no válido."
    }

    Write-Host "Registro completado con éxito. Nombre: $nombre, Edad:
$edad, Email: $email"
} catch {
    Write-Host "Error: $_"
}
```

Explicación:

El script recoge información básica del participante y verifica la edad para asegurarse de que cumpla con un mínimo requerido.

También verifica que el correo electrónico introducido tenga un formato adecuado usando una expresión regular.

Si alguna validación falla, se lanza una excepción y se muestra un mensaje de error correspondiente.

Si todo es correcto, se imprime un mensaje de confirmación.

Ejercicio 3: Simulador de Autenticación

Implementa un script en PowerShell que simule el proceso de autenticación de un usuario en un sistema. El script debe:

- Solicitar un nombre de usuario y una contraseña.
- Verificar que el nombre de usuario y la contraseña no estén vacíos.
- Simular la verificación de las credenciales (por ejemplo, nombre de usuario "admin" y contraseña "admin123").
- Imprimir un mensaje de éxito o de error en la autenticación.

Solución:

```
$username = Read-Host "Ingrese su nombre de usuario"
$password = Read-Host "Ingrese su contraseña"

if ([string]::IsNullOrEmpty($username) -or
[string]::IsNullOrEmpty($password)) {
    Write-Host "Ni el nombre de usuario ni la contraseña pueden estar
vacíos."
} else {
    if ($username -eq "admin" -and $password -eq "admin123") {
        Write-Host "Autenticación exitosa. Bienvenido $username!"
    } else {
        Write-Host "Nombre de usuario o contraseña incorrectos."
    }
}
```

Explicación:

- Este script pide al usuario que introduzca su nombre de usuario y contraseña.
- Verifica que ninguno de los campos esté vacío.
- Comprueba si las credenciales coinciden con un conjunto predefinido ("admin" y "admin123").
- Dependiendo de la validez de las credenciales, imprime un mensaje apropiado.

8. BIBLIOGRAFÍA

https://somebooks.es/scripts-powershell-guia-principiantes/
https://learnxinyminutes.com/docs/es-es/powershell-es/
https://github.com/fleschutz/PowerShell
https://learn.microsoft.com/es-es/powershell/scripting/overview?view=powershell-7.4