PowerShell HashTable – Everything you need to know

lazyadmin.nl/powershell/powershell-hashtable

September 13, 2022

Hashtables in PowerShell can be used to store information in a structured manner. This is done by using key/value pairs allowing you to easily extract the needed information from the hashtable. You can compare a hashtable in PowerShell with an associative array if you want.

Hashtables are used quite often in PowerShell because they are a great way to store and process information. The great thing about hashtables is that the key and value can be any object type. So you are not limited to integers and strings, but you can also create nested hashtables for example.

In this article, we are going to take a look at hashtables and how to use them in your PowerShell scripts.

Hashtable vs Array

Before we are going to take look at hashtables, let’s first explain the difference between hashtables and arrays. This is a question that is often asked, and people get often confused between the two.

An array is an indexed list of values. The index is automatically generated when you add values to the array. So you don’t have control over where you store the information in the array.

```
$array = @('apple','raspberry','kiwi')
```

We can access the items in the array by looping through them or accessing the items by their index number:

```
$array | Foreach { Write-Host $_ }
# Result
apple
raspberry
kiwi
# Or by index number
write-host $array[1]
# Result
raspberry
```

We can also add or update values in an array, but we can’t change or determine the index number of the item:

```
# Replace the raspberry with melon
$array[1] = 'melon'
```

```
# Add a fruit to the array
$array += 'banana'
```

Hashtable

As we have seen an array is just a list (collection) of items, which we can add, modify or remove from the list. The index is automatically generated when you add items to it. On the contrary, with a PowerShell HashTable, we must define the index (keys) of the value that we want to add.

To create a hashtable we use `{}` instead of `()`:

```
# Creating a hashtable
$serverIps= @{}
# Or creating a pre-populate hashtable
$serverIps= @{
'la-srv-lab02' = '192.168.10.2'
'la-srv-db01' = '192.168.10.100'
}
```

To add a server's IP Address to the hashtable we will need to define a key. We can't do simply `$hashtable.add('value')`, because the hashtable won't create an index automatically.

```
$key = 'la-srv-lab01'
$value = '192.168.10.1'
$serverIps.add($key,$value)
# Or simply
$serverIps.add('la-srv-lab03','192.168.10.3')
# Result:
$serverips
Name Value
---- -----
la-srv-lab03 192.168.10.3
la-srv-lab01 192.168.10.1
```

The key in this case is the server name and the value of the IP Address of the server. If we need the IP Address of the lab03 server we can simply do the following:

Retrieving data from a Hashtable

To retrieve data from the hashtable in PowerShell we can simply specify the key, which will return the value of the key:

```
$serverIps['la-srv-lab03']
# Result
192.168.10.3
```

Another option is to iterate through the hashtable, but this works a bit differently compared to an array. If we just simply pipe a foreach behind the hashtable, you will notice that it only returns a hashtable object:

```
$serverIps | foreach {write-host $_}
System.Collections.Hashtable
```

To get only the values from the hashtable we can use the `.value` property:

```
# Return only the values
$serverIps.values
# Or pipe them in a foreach to ping the servers for example:
$serverIps.values | foreach {ping $_}
```

When working with a hashtable you probably also want to retrieve the data as key/value pairs. For this, we can use the `GetEnumerator` function:

```
$serverIps.GetEnumerator() | ForEach-Object {
$result = '{0} IP address is {1}' -f $_.key, $_.value
write-host $result
}
# Result
la-srv-lab03 IP address is 192.168.10.3
la-srv-lab01 IP address is 192.168.10.1
```

Updating values

Updating values in a hashtable can be done by using the key to select the correct key/value pair:

```
$serverIps['la-srv-lab03'] = '192.168.10.5'
```

If you need to update multiple values in a hashtable, you can't simply use a ForEach loop to iterate through the hashtable. For example, this won't work:

```
$serverIps.Keys | ForEach-Object {
$serverIps[$_] = '192.168.10.20'
}
```

You will first need to clone the keys before you can update the values:

```
$serverIps.Keys.Clone() | ForEach-Object {
$serverIps[$_] = '192.168.10.20'
}
```

Removing keys or values

To completely remove a key/value pair from the hashtable you can use the `.remove` function. You will need to specify the key that you want to remove:

```
$serverIps.Remove('la-srv-lab03')
```

It's also possible to remove only the value from a key:

```
$serverIps['la-srv-lab03'] = $null
```

To completely clear the hashtable you have two options, you can recreate the hashtable instance or use the clear function. The latter is preferred because it makes your code easier to read.

```
# Recreate the hashtable instance
$serverIps = @{}
# Better option for readability, use the clear function
$serverIps.clear()
```

Splatting hashtables

Some cmdlets in PowerShell require a lot of properties that we generally all write on a single line. This can make your PowerShell scripts harder to read because you might need to scroll horizontally to view the complete command.

For example, when you want to send an email from PowerShell, you will end up with a long line of parameters

```
Send-MailMessage -SmtpServer smtp.contoso.com -To johndoe@lazyadmin.nl -From info@contoso.com -Subject 'super long subject goes here' -Body 'Test email from PowerShell' -Priority High
```

To make this more readable we can use a hashtable and splat it. Splatting replaces all the parameters and values with a single variable. Note that we use the `@` symbol to splat the variable instead of `$`

```
$mail = @{
SmtpServer = 'smtp.contoso.com'
To = 'johndoe@lazyadmin.nl'
From = 'info@contoso.com'
Subject = 'super long subject goes here'
Body = 'Test email from PowerShell'
Priority = High
}
# Note the @ infront of mail, instead of $
Send-MailMessage @mail
```

As you can see, splatting makes your code easier to read and maintain instead of a long line of parameters.

Creating Objects from Hashtables

Hashtables are great when you need to structure data for a single object or for splatting, but they are not true objects. Hashtables only have two columns, key, and value. So we can create a hashtable with the specifications of a single server, but when we need to store data about multiple servers we run into limitations (we can’t reuse column names for example).

So a better way to structure your data when you have multiple entries is to use objects. The easiest way to create objects in PowerShell is to convert a hashtable to a custom PowerShell object. We do this by adding [pscustomobject] in front of the hashtable.

Take the following example, first we create a hashtable with server specifications:

```
$servers = @{
'name' = 'la-srv-db01'
'ip' = '192.168.10.100'
'model' = 'Hp DL380 G10'
'memory' = '96Gb'
}
# Result
Name Value
---- -----
memory 96Gb
ip 192.168.10.100
name la-srv-db01
model Hp DL380 G10
```

As you can see, we only have two columns, name and value. The hashtable represents a single server and we can’t use it to create a list of servers.

However, if we convert the hashtable to a custom PowerShell object we get an actual table, where we have multiple columns.

```
$servers = [pscustomobject]@{
'name' = 'la-srv-db01'
'ip' = '192.168.10.100'
'model' = 'Hp DL380 G10'
'memory' = '96Gb'
}
# Result
name ip model memory
---- -- ----- ------
la-srv-db01 192.168.10.100 Hp DL380 G10 96Gb
```

So now we can add another server to the `pscustomobject` as well:

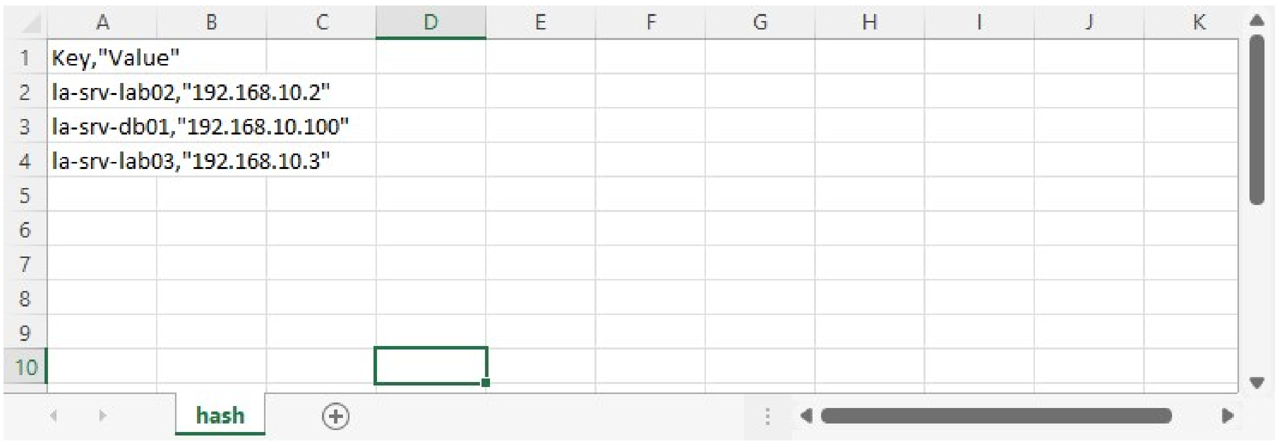
```
function servers {
[pscustomobject]@{
'name' = 'la-srv-db01'
'ip' = '192.168.10.100'
'model' = 'Hp DL380 G10'
'memory' = '96Gb'
}
[pscustomobject]@{
'name' = 'la-srv-lab03'
'ip' = '192.168.10.3'
'model' = 'Hp DL180 G9'
'memory' = '32Gb'
}
}
servers
# Result
name ip model memory
---- -- ----- ------
la-srv-db01 192.168.10.100 Hp DL380 G10 96Gb
la-srv-lab03 192.168.10.3 Hp DL180 G9 32Gb
```

Exporting Hashtable to CSV

When you export a hashtable to CSV you will notice that by default the key and values as not exported. The output will only contain information about the hashtable object, not the data inside the hashtable.

To actually export the contents of the hashtable you will need to use the GetEnumerator method, just like we did with viewing the contents of the hashtable. We only need the Key and Value from the hashtable, so you will need to add a select to the command as well.

```
$serverIps= @{
'la-srv-lab02' = '192.168.10.2'
'la-srv-db01' = '192.168.10.100'
'la-srv-lab03' = '192.168.10.3'
}
$serverIps.GetEnumerator() | Select Key, Value | Export-CSV -path c:\temp\hash.csv -No
```



```
Key,"Value"
la-srv-lab02,"192.168.10.2"
la-srv-db01,"192.168.10.100"
la-srv-lab03,"192.168.10.3"
```

Export hashtable to CSV

Wrapping Up

Hashtables in PowerShell are a great way to structure data or to use in combination with splatting. However, keep in mind that they are intended to use for only a single item. We can’t create a custom column or store multiple rows with the same key. For that, you really need to use the custom PowerShell Objects.

I hope you found this article useful, if you have any questions, just drop a comment below.

If you want to learn more about PowerShell, make sure that you also read the Top 10 PowerShell Commands that you must know

Did you **Liked** this **Article**?
Get the latest articles like this **in your mailbox**
or share this article

I hate spam to, so you can unsubscribe at any time.