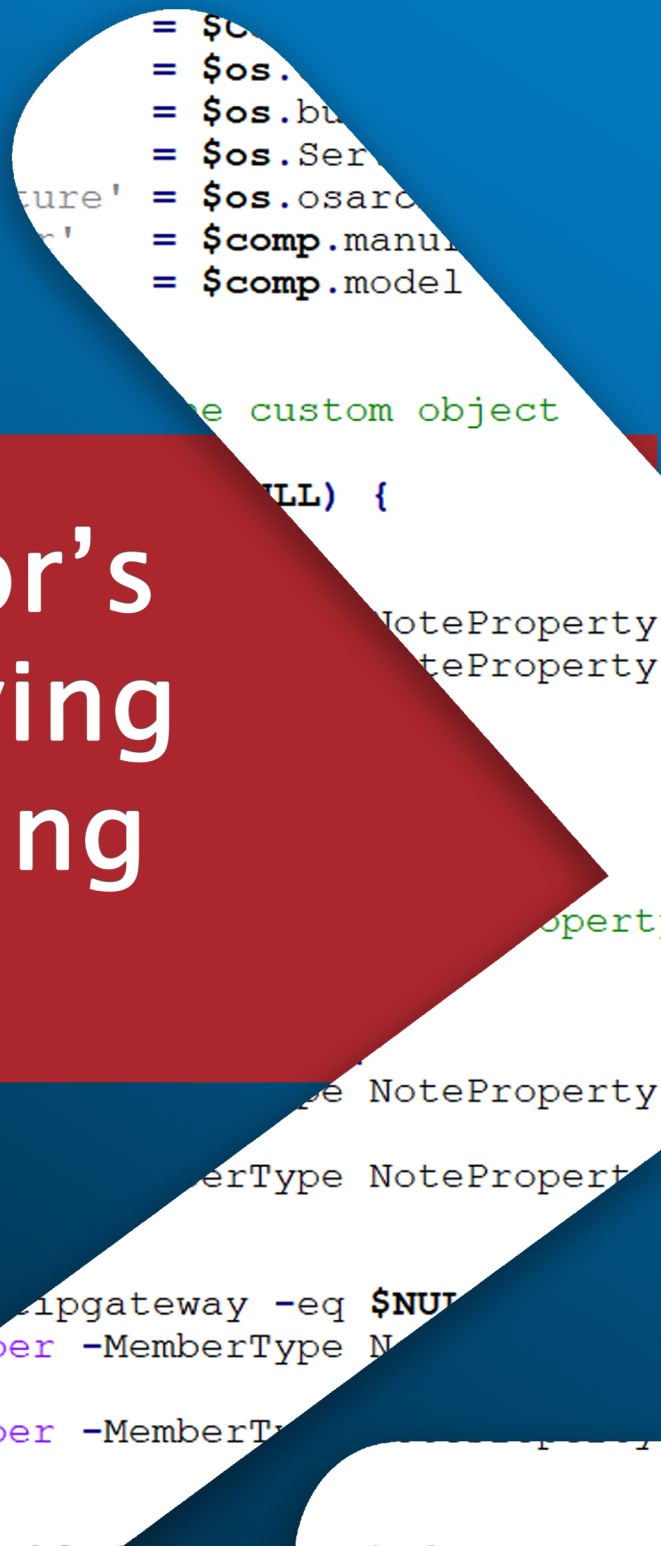
ConfigMgr



An Administrator's Guide to Deploying Applications Using PowerShell

Owen Smith

ConfigMgr - An Administrator's Guide to Deploying Applications using PowerShell

Owen Smith

This book is for sale at http://leanpub.com/configmgr-DeployUsingPS

This version was published on 2020-02-21

Just a quick thank you to my family who put up with me taking time away to write this book. A big shout-out to my colleague Jack, who designed the book cover for me; my original one was horrific! And finally to you, dear reader, for taking the leap of faith and purchasing the book.

I could not have done it without you. Thank you.

Contents

Introduction

Recently I have found myself deploying a lot of applications. Some complicated and some not so much. More often than not, I've been using PowerShell for both the actual deployment and the detection rule.

While I've been doing this, I've noticed a few things that I always did; cmdlets that I always used; particular lines of code etc. The more I used PowerShell, the more gotcha's I had to figure out too.

I thought I'd write about my findings on my blog, and indeed that's exactly what I was going to do until I started writing down some of the topics that I wanted to blog about. The more I thought about it, the longer the topic list grew until really, it was impractical to blog about. It made more sense to gather all of the information into one cohesive volume. And that's what we have here! It's even better than I had hoped as it's a 'living' book meaning that I can amend it and update it as and when. It's good for you too as your single purchase will give you access to a lifetime of updates!

Thanks for purchasing this book. I appreciate it. It takes time to write a book, even one with not many pages like this one. Your purchase keeps me spurred on to do more, add more examples and expand.

Who This Book Is For

I am going to make a few assumptions here:

1. You already know how to deploy a standard application using Configuration Manager.
2. You have *some* PowerShell experience.

Number 2 is important because, fundamentally, this is a book about deploying applications using PowerShell. Although you don't need a great deal of experience, it would be advantageous to have some PowerShell knowledge and a bonus would be some scripting experience too.

I'll cut to the chase here: this book is not going to teach you PowerShell or how to script.

Having said that, even if you do not possess either of these skills I am certain that there is still a great deal of value to be had out of a read-through of this book.

So, do you take the blue pill or the red pill?

How This Book is Organised

Code

All PowerShell code used is presented in this book so that you don't have to 'jump out of the moment' however the downside to this is that the formatting may be a little off! To see the actual code-as-is, take a look here on my Github repository.[1]

All code examples used in this book will have the .ps1 filename just above it so you can easily find the matching PowerShell file in the Github repository.

I should point out that I'm English and therefore some spellings may look odd to you: summarise instead of summarize, colour instead of color, organise instead of organize etc. You say potato I say potaaaaaato!

The Sum of Its Parts

I recommend reading this book in order, however, it is not necessary to do that and should you wish to then feel free to jump to any section that takes your fancy. **Part 1**, "Why Use PowerShell?," explains why you may choose this method and takes you through the various PowerShell cmdlets that I find I use all the time when deploying applications.

Part 2, "MSIEXEC" covers some of the fundamentals of this command; how to get help, how to extract from a setup.exe, installing and uninstalling, using parameters as well as explaining where uninstall GUIDs are stored in the registry and how you can use them to your advantage. This part also explains how to easily find out not only which properties are available for you to set on your msi but crucially, how to find out what the allowed values are that you will pass to the property. You are then shown an example of a 'problem' deployment: a Setup.exe that has user input, and step-by-step I explain how the msi is extracted, how I discover the available properties and their allowed values and build the final command line.

Part 3, "Detection Rules" explains why you should use PowerShell for your detection rules and how they work. I then demonstrate some of the more traditional ways that you can use PowerShell for your detection rules; such as checking the registry for certain values or interrogating the file system. You will also learn how to create your own files or registry keys \ values to be used as part of the detection when all else fails.

Part 4, "Location, Location, Location" is where you learn how to identify where your script is running from, how to use this to reference your files and where to place your script and files.

[1] https://github.com/ozthe2/ConfigMgr-Book-Code

Part 5, "Installing the Program" is the meat and bones of our deployment script. Here you will learn how to script our msi or exe installation as well as deal with common issues such as passing parameters and dealing with spaces in file names.

Part 6, "Deploying the Script" explains how to deploy your finished script using the ConfigMgr 'application method.' We will learn the different methods we need to use depending on what type of script we have eg a function or a plain 'top-to-bottom' script.

Part 7, "A Deployment Template" is a rough and ready template I created that can be used as the basis for all of your deployments; you will just need to season to taste.

Part 8, "Useful Code Snippets" gives examples of re-usable code I use frequently in my scripts, such as detecting Microsoft Office 'bitness' or obtaining the current logged in user as well as many more.

Part 9, "Real World Examples" shows ready-to-go-code that can be used to deploy popular applications such as Java, Adobe Acrobat Reader, resizing the CCMCache and more.

Part 1: Why Use PowerShell?

My manager said to me, "Hey, I have an application that needs deploying to the Accounts department, it should be straight forward. Oh, by the way, you'll need to customise an xml file which needs to be copied to a specific folder structure that may or may not already exist.

There's also an Excel add-in, so you need to ensure that it's automatically picked up by Excel and activated so that the tab appears with no end-user config. Should be a piece of cake."

Believe it or not, that was a true story and something that I had to deploy just the other day!

Or how about that good old favourite: Java! How can you be certain that all previous versions of Java have been removed and that only the new version gets installed? Better yet how can this be made an easy, repeatable procedure for when the next version is released?

Or how about an easy and repeatable method of uninstalling old versions of Adobe Reader followed by installing the new version as well as any version patches?

What if you need to read or write to the registry? Start or stop some processes as part of the deployment? Register or unregister a dll file? How about if you need to do anything pre or post deployment?

Using PowerShell allows for rapid, repeatable deployments. Using PowerShell allows you to do more than deploy a simple msi. That's why I use PowerShell.

PowerShell Cmdlets

There are a small set of PowerShell cmdlets that you will find yourself using over and over again when creating deployment scripts. Below are all of the cmdlets that you will tend to use along with a brief description of how they are used. The idea here is not to give you a step-by-step walk-through on exactly how these cmdlets are used in scripting, it's more of a guided tour of the ones you should investigate and prioritise your time on learning and exploring.

Later in the book, you will use these in real-world deployment scenarios so that you can see how they all fit in, so don't worry if you can't see the 'context' at the moment.

Write-Host

Used for detecting if the deployment was successful or not.

Get-Location

This cmdlet gets an object that represents the current directory. This is used to establish the directory that the deployment script is being called from.

Set-Location

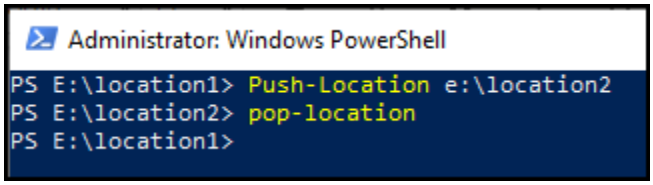
Sets the current working location to a specified location. It's useful if changing to a different PSDrive, for example from the file system to the registry.

Push-Location \ Pop-Location

The *Push-Location* cmdlet adds or pushes, the current location onto a location stack. A location stack functions like your location history. If you specify a path, this cmdlet pushes the current location onto a location stack and then changes the current location to the location specified by the path. You can use the Pop-Location cmdlet to get locations from the location stack.

This can sound confusing when you first try to get your head around it so let's take a moment to break it down: Let's say your current location is *E:\Location1*. You can then change the location to somewhere else, for example, E:Location2 by doing this: Push-Location E:Location2. The first location is held in memory, so you can now reference whatever needs doing from the second location

and when you wish to go back to the first location, you can do so without having to know explicitly where that first location was. How? By doing this: *Pop-Location.*



How is this useful? As you know, when you deploy an application using ConfigMgr it downloads the application content to the client in *.\Windows\CCMCache\<unknown folder name>*

In some deployments, you may need to change the location to reference other files more easily. Once you have finished referencing those files how do you know which file path to change back to? With Push Pop-Location, this is handled for you: simply push to the new location and then pop right back!

Get-Process \ Stop-Process

Get-Process is handy for seeing if a process is running that would potentially result in an unsuccessful deployment unless the process is stopped. Closely followed by its big brother, Stop-Process where if the desired process is found running, it can be stopped in its tracks.

Start-Process

This cmdlet is used to do the actual installation of the executable or MSI files.

New-Item

The New-Item cmdlet creates a new item and sets its value. The types of items that can be created depend on the location of the item. For example, in the file system, New-Item creates files and folders. In the registry, New-Item creates registry keys and entries.

New-Item can also set the value of the items that it creates. For example, when it creates a new file, New-Item can add initial content to the file.

New-ItemProperty

The New-ItemProperty cmdlet creates a new property for a specified item and sets its value. Typically, this cmdlet is used to create new registry values, because registry values are properties of a registry key item.

You will tend to use this to create a registry value which is then used as a detection rule.

Get-Item

This is very useful in detection rules. It can be used to determine and compare version numbers of executables.

Test-Path

Used to determine if a folder or file in a given file path exists or not.

Try \ Catch

Used for error handling. This can be used to determine if the deployment really succeeded or not by setting an error variable to true if the catch is activated.

Copy-Item

The Copy-Item cmdlet copies an item from one location to another location in the same namespace. For instance, it can copy a file to a folder. This is useful for copying files from the CCMCache to other file locations on the target computer.

And there you have it. Of course, you may use more or fewer cmdlets than those listed above, but generally, nine-times-out-of-ten, these are the cmdlets that you will keep coming back to and I recommend that you have a read of the official PowerShell documentation for each one.

Try It Out

Take an hour out, right now, to try a few examples out on your own computer and really familiarise yourself with them in order to give yourself even more of a heads-up as you go through this book. I promise you it will be worth your time.

The Story so Far

In Part 1 you have learned that with PowerShell comes great flexibility in your ConfigMgr deployments.

No longer will you be confined to simple MSI deployments; instead, the chains will be broken, and you will soon be able to deploy even the most complex scenarios that the workplace continuously and unreasonably demands.

You also looked at the small handful of PowerShell cmdlets that you will find yourself using all of the time when scripting deployments and you were encouraged to take a look at the documentation for each one and to try to familiarise yourself with them.

As Laozi once said, “A journey of a thousand miles begins with a single step” and you are now well on your way.

Part 2: MSIEXEC

Msiexec.exe provides the means to install, modify, and perform operations on Windows Installer from the command line. This is the tool that, where possible, you should try to use all of the time to install applications. Msiexec can install .msi and .msp (patches) as well as accept parameters and mst (transform) files in order to customise the deployment.

It's the easiest 'tool' to use to deploy .msi files because it "just works" and it provides the greatest flexibility.

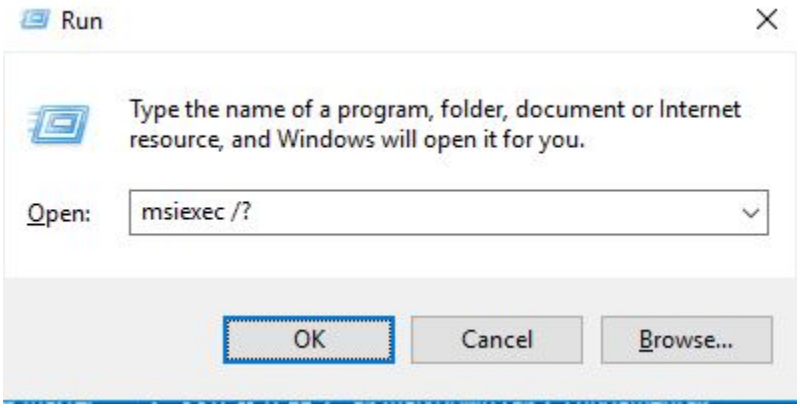
So, "Msiexec, you say?" Why yes, I do. Let's dive in…

Fundamentals

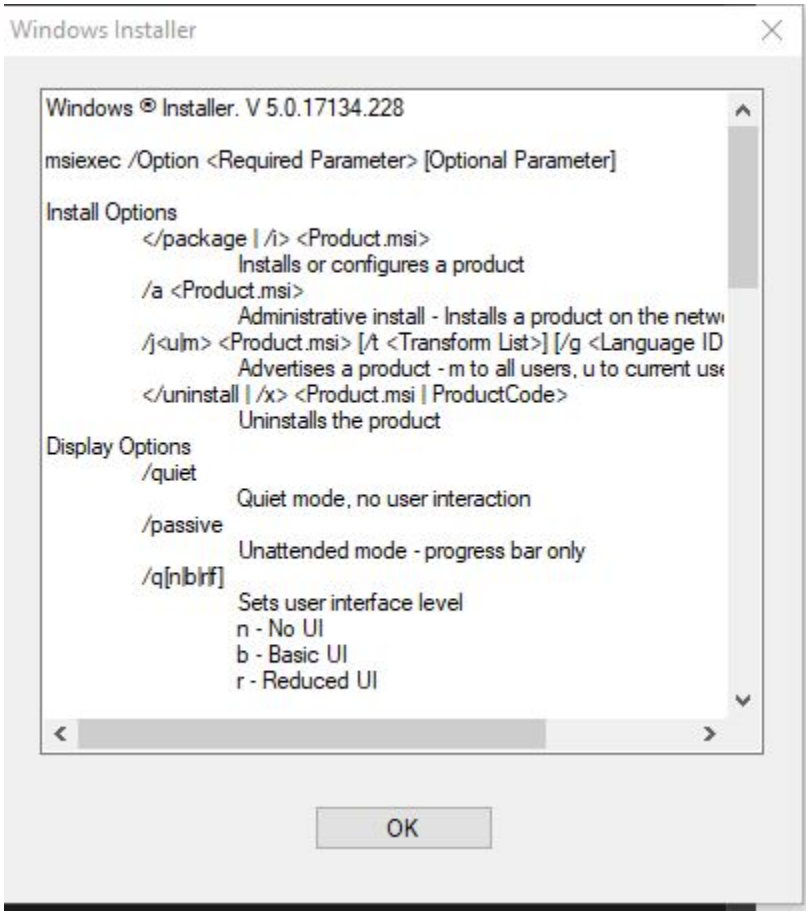
Msiexec.exe is found natively on all modern Microsoft Windows operating systems today. The way that you will be using it in your scripts is to call using another PowerShell cmdlet along with appropriate parameters. But first-things-first…

View the help

Open a run menu (Windows key + R) and type: *msiexec /?* to produce the in-built help dialog.



A run box



In-built help

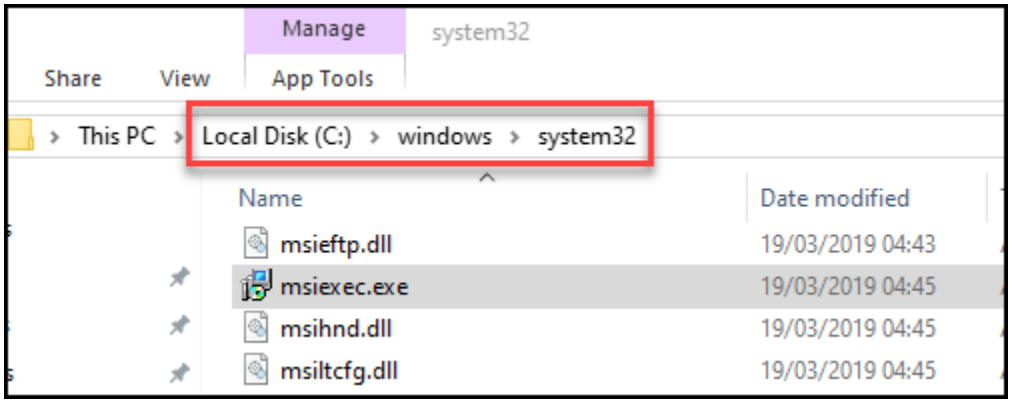
However, I find the help to be more detailed online, and you can view it here:

https://docs.microsoft.com/en-gb/windows/desktop/Msi/command-line-options

Where Is It?

As you've seen when you typed msiexec into the run menu, you don't need to type the complete file path to call it; simply reference it by calling the exe directly: msiexec; however, I strongly recommend that you do use the full path to the exe when calling it from your deployment scripts for clarity and certainty.

Here's where it usually lives if C: is your Operating System drive: *C:\Windows\System32*



Better to use $Env:

"Better to use say what?" Yup, as you have just read, we are 'assuming' that Windows was installed and accessed via the drive letter of 'C:' But what if it was installed on drive 'D:' or 'E:' or any other drive letter? If we have hard-coded our call to msiexec using *C:\WindowsSystem32\msiexec.exe* and Windows was on 'D:' then our deployment script would fail.

Luckily enough, PowerShell includes many environment variables which come in handy all of the time when writing scripts.

In a PowerShell window simply type *$ENV:* and then tab through to see what environment variables are available. In this case, we will be using the environment variable of *$ENV:SystemRoot* as this will always resolve to the Windows directory where-ever it may be installed.

With that in mind, here is how you should always reference any calls to msiexec.exe in your deployment scripts: *"$ENV:SystemRoot\System32\msiexec.exe"*

Even if you know that all your computers have a standard Windows installation using the drive letter "C:", it is very good practice to get into the habit of using environment variables wherever it's appropriate. After all, you may write a script for yourself now that you end up sharing later and who knows where the Windows installation resides in another organisation?

Parameters

Before we begin our journey in earnest, you must understand the fundamentals of building up a command line to install or uninstall using msiexec.exe.

The parameters that you will mostly use with msiexec are: */i* , */x*, */qn*, */norestart* and sometimes *PROPERTY=* and */update*.

I rarely find myself using the *TRANSFORMS=* switch though there are certainly great use-cases for it.

Let's see how to build up a typical command line to install *MyProduct.msi*.

Installation

To install a msi, use the */i* as your first switch followed by your .msi file:

msiexec /i MyProduct.msi

Silent Install

To make this installation silent i.e. no GUI or user interaction, add */qn*

msiexec /i MyProduct.msi /qn

No Restart

It's important that the install does not restart the computer that you deploy the msi to so let's suppress that too by adding */norestart*

msiexec /i MyProduct.msi /qn /norestart

Uninstall

To uninstall you use the same process but use */x* as our first parameter.

Here we perform a silent uninstall and suppress any restarts:

msiexec /x MyProduct.msi /qn /norestart

You will see later in this book that it's better to use the uninstall GUID instead of the msi file name.

That's essentially it. Of course, there are other parameters you can use as you saw from looking at the help, but these are the common ones that you will find yourself using.

Now that you have the basics under your belt, when it comes to scripting this later it will be a breeze.

Note

You cannot simply type: *msiexec /i MyProduct.msi /qn /norestart* in your PowerShell deployment script and expect it to work - it must have a small tweak made first!

This is covered this later in the book.

Properties

Some msi's have properties that you can modify to customise the installation. For instance, automatically accepting an End User License Agreement (EULA) or preventing the creation of a desktop shortcut. To do this you would simply reference the property with the associated value when you create your command line.

Something like this: *msiexec /i MyProduct.msi PROPERTY=value /qn /norestart* where *PROPERTY* is the name of the property you wish to modify, and value is the value for that property.

You may or may not already know what the properties and values are that you require for your command line. On very rare occasions a product will come with some great documentation and you know ahead of time exactly what you need.

But what if you don't know this information? How can you find out which properties are available, and then, what the permitted values are for them?

Which Properties Can I Set?

There is more than one way to find out which properties can be configured with your msi. The easiest method, as it does not require any extra 'tools', is to simply run an admin installation (this way the msi is not actually installed on your computer), and add some logging parameters. Then you can examine the resulting log output and view which properties are available.

To do this, use this command line:

msiexec /a <msi_name> /lp! <msi_property_logfile> TARGETDIR=<absolute_path_to_extract_to>

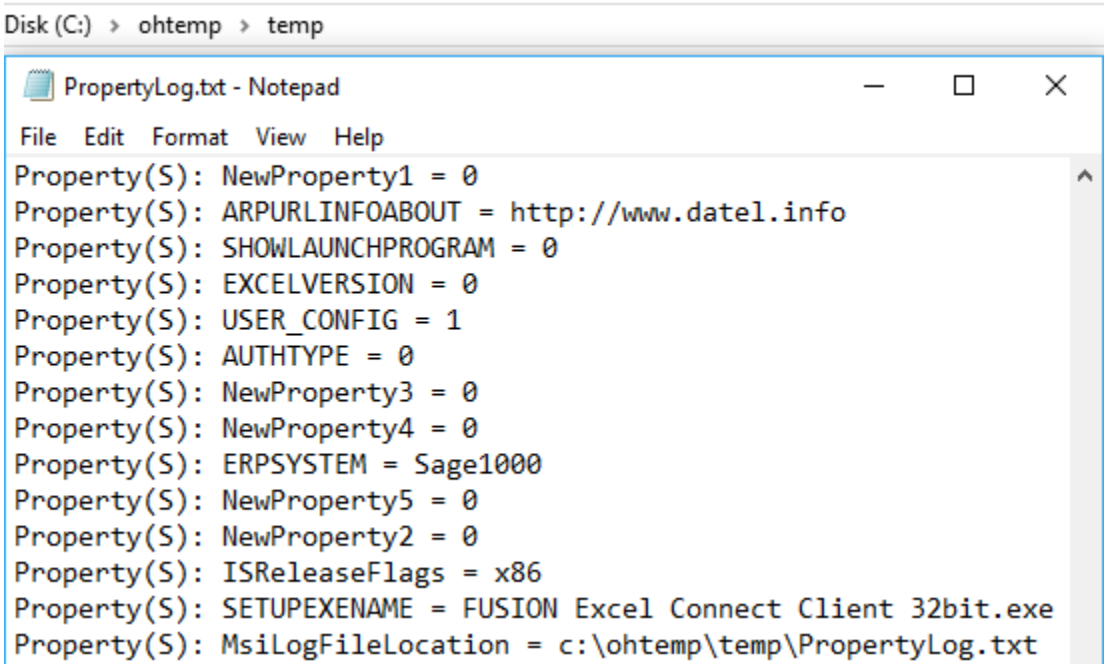
Top Tip!

Note that the *<absolute_path_to_extract_to>* can point to a non-existent directory as the command will create the directories for you. Now that's handy!

For example: *msiexec /a MyProgram.msi /lp! PropertyLog.txt TARGETDIR="c:\tempFiles"*

Once you've done that, look at: *PropertyLog.txt*

Any Property(S) or Property(C) in the log that is in capitals is a public property and can usually be configured.



MSI Log file

In the above screenshot, it can be seen that the property: *ERPSYSTEM* can be configured, and by default, it has a value of Sage1000

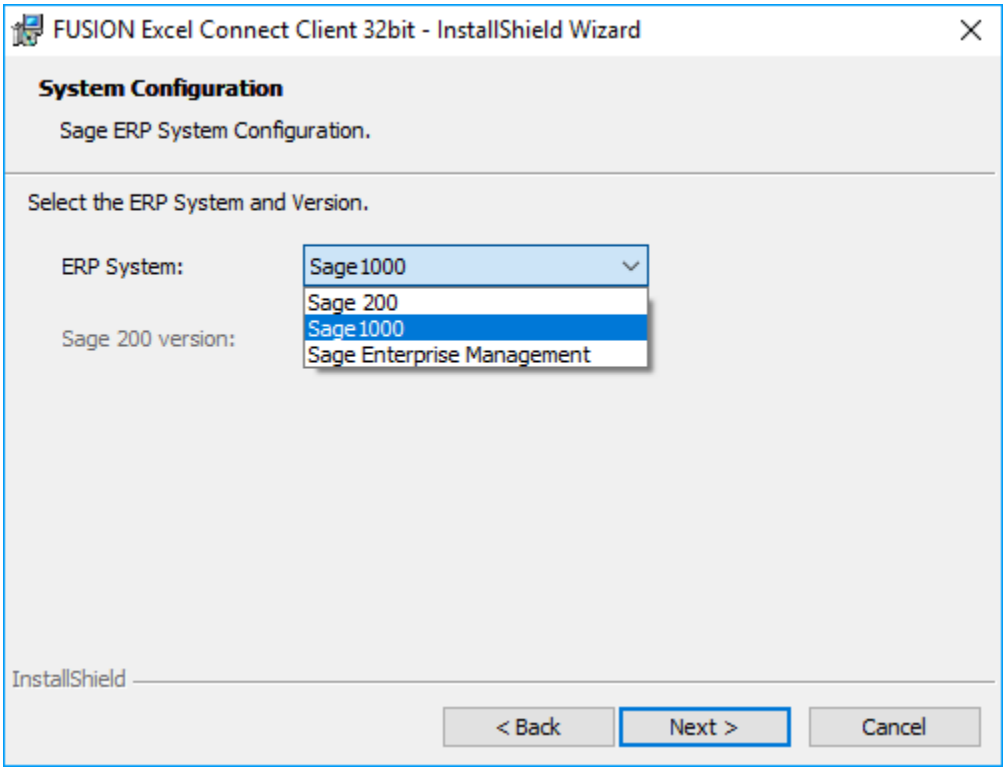
How to Find Valid Property Values

Now you know which properties can be modified, how do you know which values can be used?

Using logging, that's how!

As an example, I had to deploy an application recently: *FUSION Excel Connect Client 32bit.msi* that requires user input.

When I manually ran the msi, here is what I saw:



User Selection

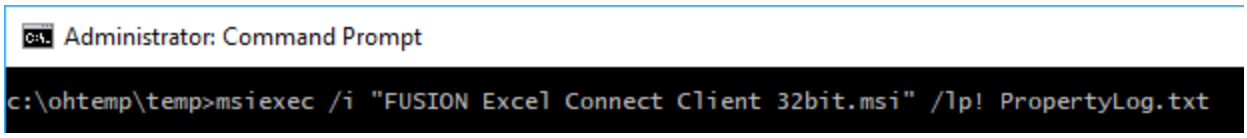
By default, it had Sage 1000 selected. Our organisation required using the drop-down menu and choosing: Sage Enterprise Management.

I had used the previously shown technique to view the available properties from the log file and saw that the property I needed was ERPSYSTEM but what would the correct value to pass to the property be?

To find out, I would have to install the msi with property logging turned on.

I used the following command:

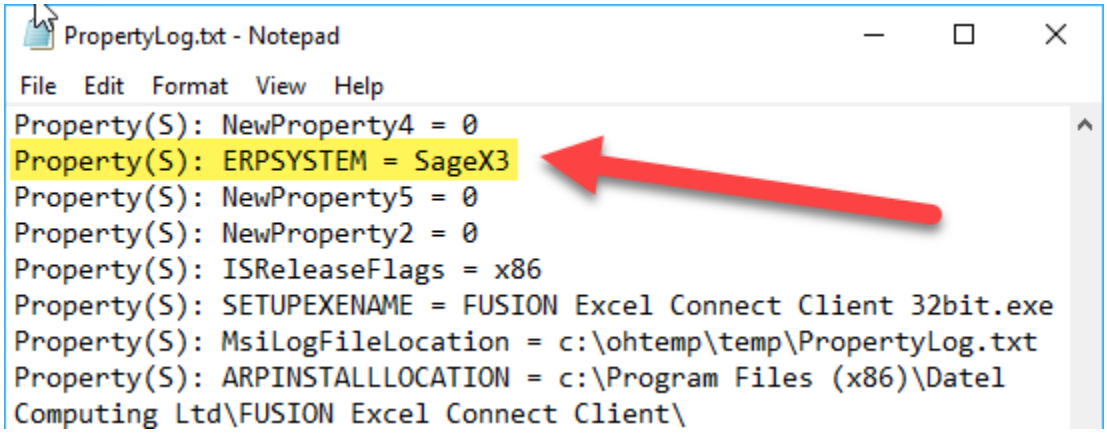
msiexec /a "FUSION Excel Connect Client 32bit.msi" /lp! PropertyLog.txt as shown in the following screenshot:



Install the msi

This command-line started the installation with a full GUI. I then clicked through the GUI, ensuring that I selected *Sage Enterprise Management* when prompted and completed the installation.

This time, when I checked the *PropertyLog.txt* I saw this:



Property and value

Success! Now I knew exactly what the value should be for the property for me to automate this selection. With this information to hand, my final command line for a silent installation, suppressing restarts and ensuring that Sage Enterprise Management was selected, looked like this:

msiexec /i "FUSION Excel Connect Client 32bit.msi" ERPSYSTEM=SageX3 /qn /norestart

Tip!

Don't forget to uninstall the program you just installed! (Unless of course you want it to remain on that computer.)

Uninstall GUIDs

When you install an msi, it stores various useful information, including its display name and uninstall GUID in the registry.

Where this useful information is stored depends on whether it was a 32-bit or 64-bit installation.

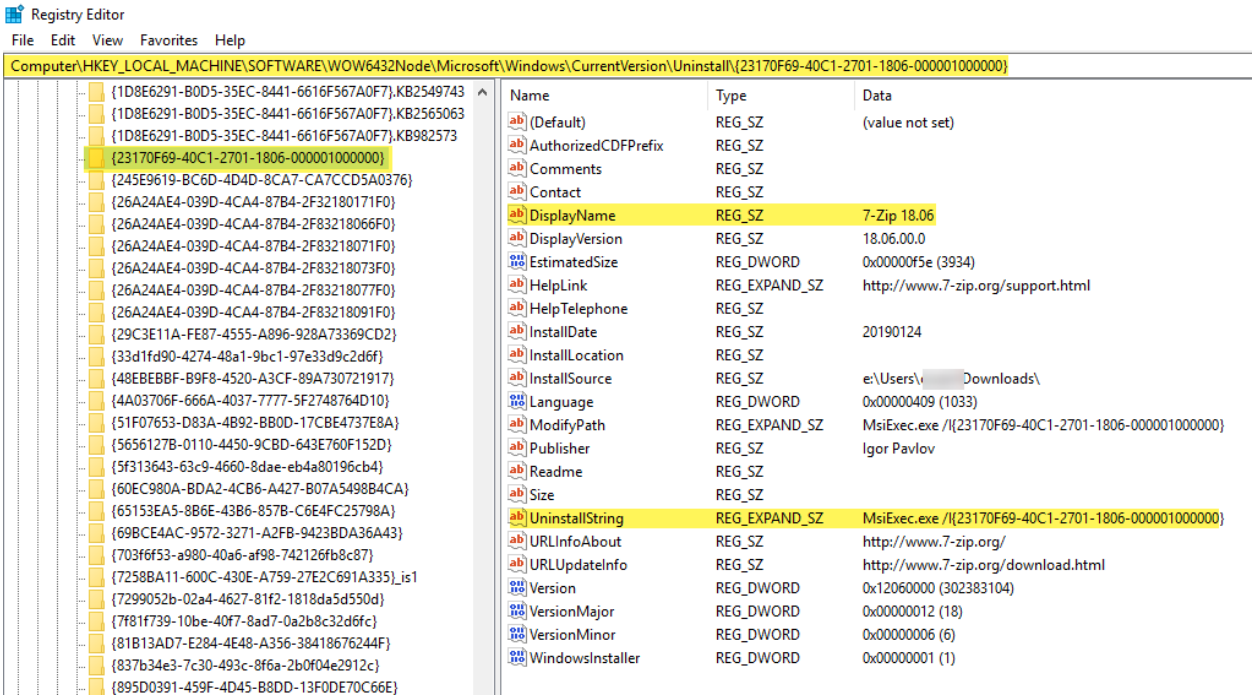
32-bit Installations

Here is the location in the registry for 32-bit installations:

HKLM:\SOFTWARE\Wow6432Node\Microsoft\Windows\CurrentVersion\Uninstall

In the screenshot below, you can see the details for the 32-bit version of 7-Zip that I have installed which includes version information too.

The most important parts as far as deployment scripting goes, is the Display Name and Uninstall String.



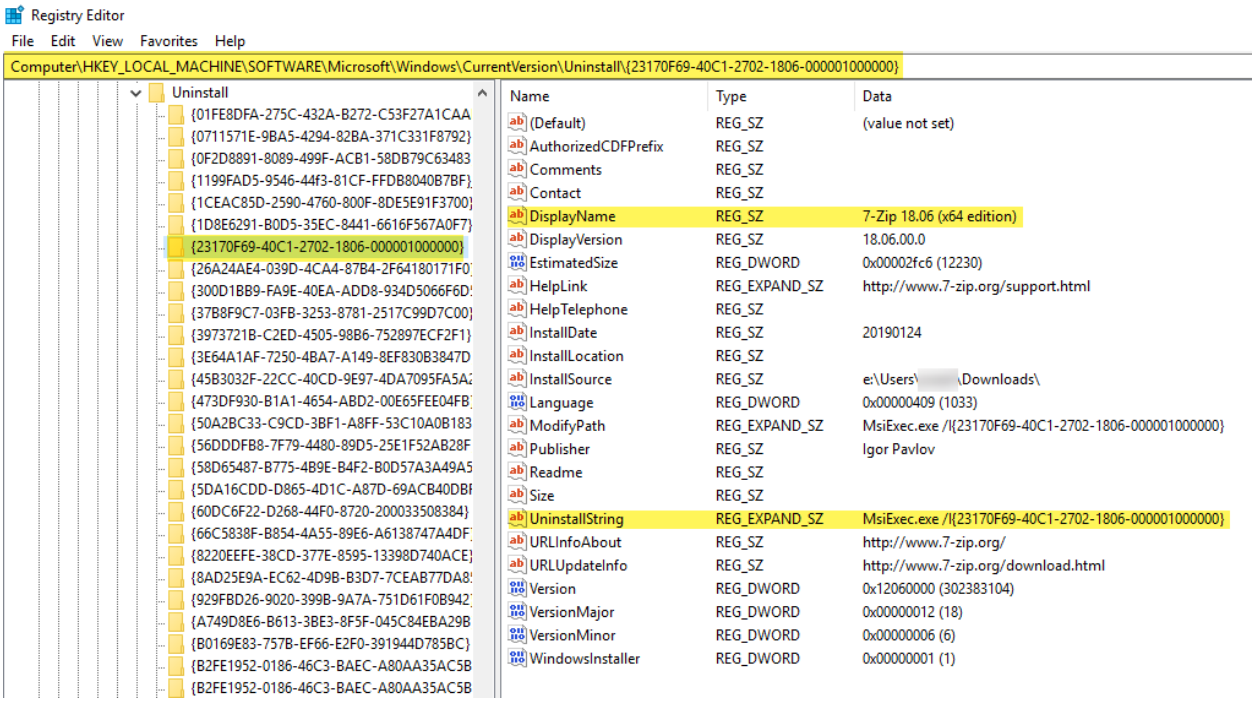
Registry 32-Bit

64-bit Installations

Here is the location in the registry for 64-bit installations:

HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall

And here are the details for the 64-bit version of the same program. As you can see, we are presented with the same information:



Registry 64-Bit

Knowing where this information lives is crucial to our deployment scripts as we can parse the registry in these two locations searching for the display name of the application that we wish to uninstall, then grab the uninstall GUID and pass it to msiexec.exe to start the uninstall.

TIP!

Before any deployment of a new application, I like to search for any existing older versions first and ensure that they are uninstalled before the new installation takes place.

Uninstall previous versions I have written a PowerShell function that uninstalls applications for you based on the display name and I tend to add this function to all my deployment scripts, calling the function to ensure removal of all older versions before the new installation. Don't concern yourself with how to use this script just yet - this is covered later.

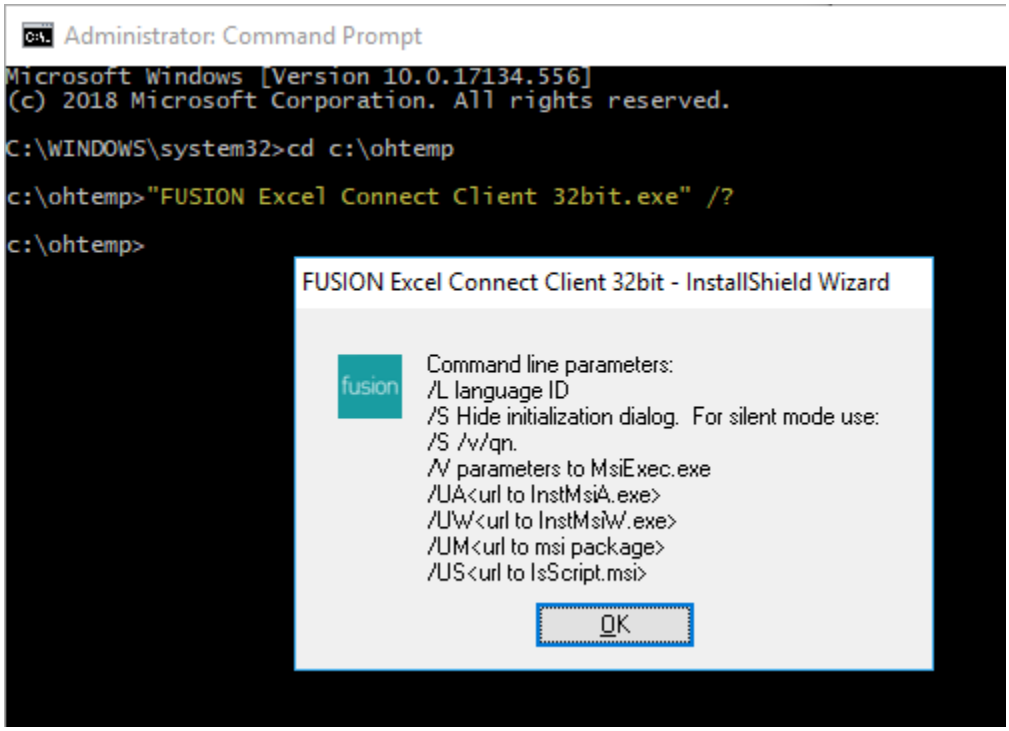
You can find the script here:

https://github.com/ozthe2/ConfigMgr-Book-Code/blob/master/Code/Invoke-ApplicationRemoval.ps1

Setup.exe

Some of the time you won't have a msi to deploy and instead you'll be faced with the dreaded Setup.exe.

Try typing: *setup.exe /?* at a command prompt on the off-chance it provides you with the command-line switches you are going to need:



A Lucky Find

If not, try experimenting to see if you can figure out the silent switch or search online for it.

(You will see later exactly how to easily deploy a setup.exe if the correct parameters are known.)

I like to do my best to get to the msi and you can sometimes extract the msi from setup.exe using a program like 7-zip. (Right-click the setup.exe and extract the contents)

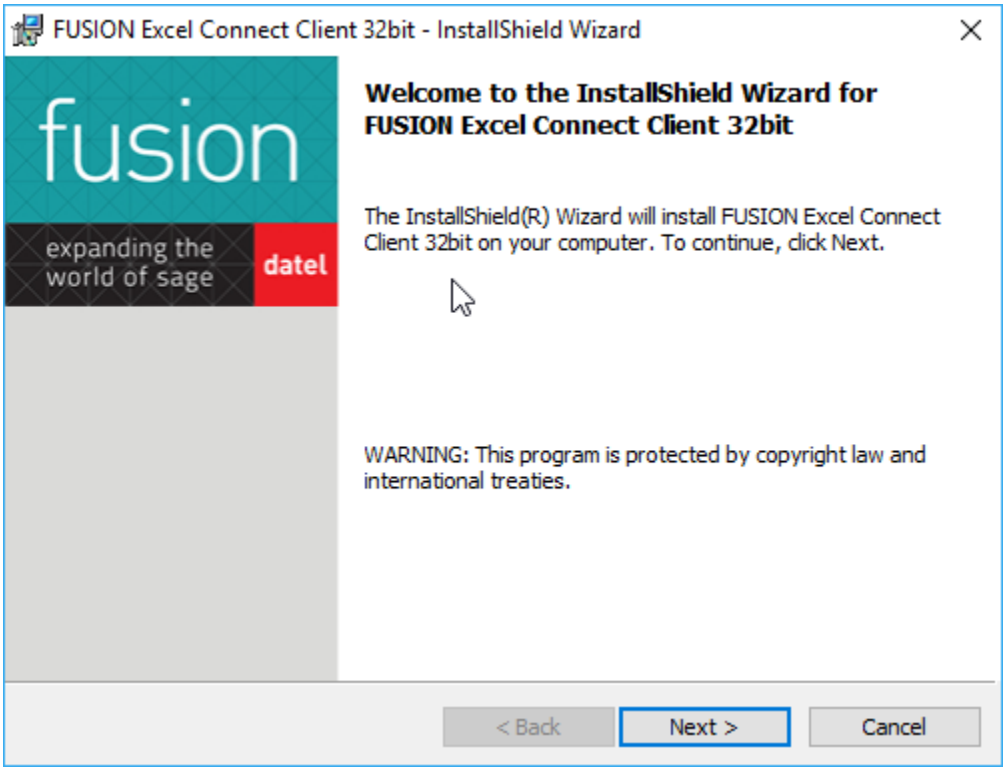
Other sneaky methods are to run the setup.exe manually and just before you click next to install it, browse to *%APPDATA%* (in the run menu type *%appdata%*), and check the Local or LocalLow directory where you may be able to grab the extracted msi from the relevant sub-directory. Once you've done that, you can cancel the manual install process you started.

Sometimes though, no matter how hard you try, you will have no choice but to deploy an exe, and later in this book, we will do just that.

Example MSI Extraction

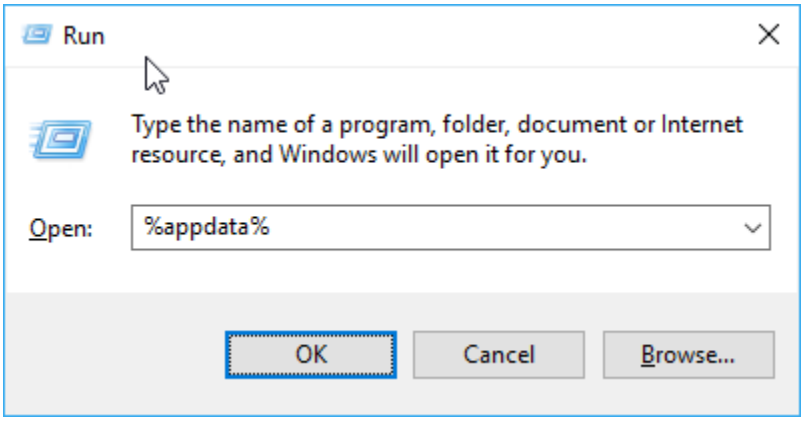
I had to deploy an application recently: *FUSION Excel Connect Client 32bit.exe* and to extract the msi I used the following procedure:

1. Double-click the exe installer to start the installation:



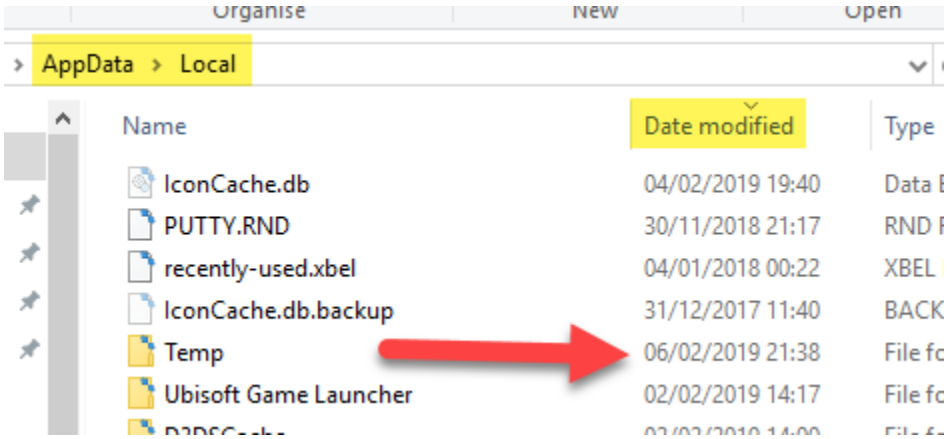
Launch Setup.exe

1. Do not click Next! Leaving the Installation GUI on screen, I then typed the following in the run menu: *%appdata%*



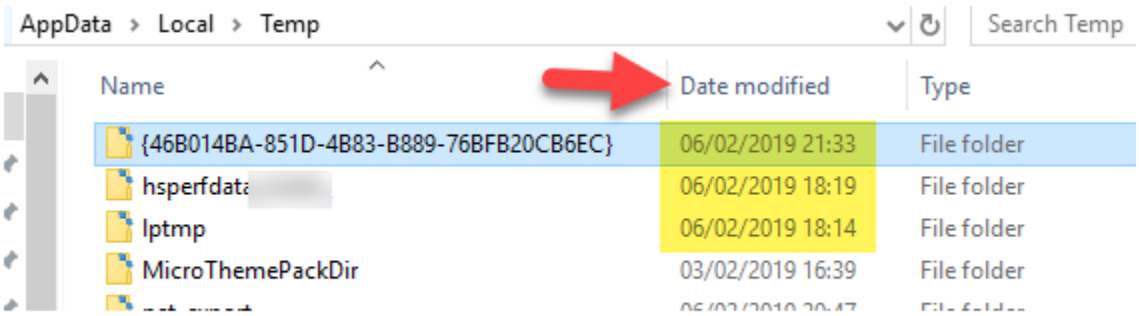
Launch Setup.exe

1. I changed to the *Local* directory (going back one level from the current directory I was in), and sorted by *Date Modified*. This showed me that the Temp directory was last modified:



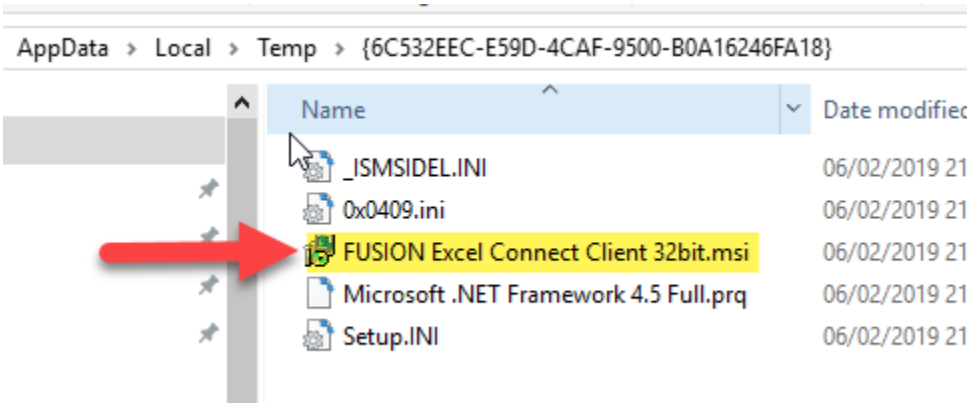
Launch Setup.exe

1. In the Temp directory I could see under the *Last Modified* title there was only one entry that matched the relevant date and time that I ran the setup.exe:



Temp Directory

1. I double-clicked the {….GUID….} folder…and there it was! I copied the msi to a different location and then cancelled the exe installation:



GUID Folder

And just like that, I had an msi!

The Story so Far

You have discovered why it's better to use *$ENV:SystemRoot* (and PowerShell environment variables in general), to determine where the Windows directory may be on a given computer.

You have learned the basic parameters required when using msiexec.exe and also how to build up a command line step-by-step.

You have also realised that many msi's have properties (both documented and undocumented), that may help to better customise your deployment, and you have learned how to easily view the public properties available and how to figure out what some of the values are that can be passed to them.

You've also discovered that a msi installation registers the uninstall GUID and its display name (among other properties) in different parts of the registry. (Depending on if it is a 32-bit application or 64-bit application.)

Finally, you have seen that, given a setup.exe file, various methods can be leveraged to attempt to extract the msi. If all fails though, it may still be possible to natively deploy a setup.exe.

Phew! That was tough. We've covered a lot of ground in this one. Let's keep pushing forward.

Part 3: Detection Rules

How can you be sure that your application has deployed? Are you certain that the application that you have just deployed has the correct version number? Detection rules are what determines whether a successful deployment has occurred or not. Quite important I'm sure you would agree! Let's dive in…

Why Use PowerShell?

Much like the reason for using PowerShell to deploy in the first place, using PowerShell as a detection rule shares the same reasoning: Speed of implementation and flexibility.

Detection Fundamentals

Let's start off with figuring out exactly how to tell Configuration Manager what constitutes a successful deployment.

The Microsoft "Rules"

Here is a table that can be found on the Microsoft documentation website:

https://docs.microsoft.com/en-us/previous-versions/system-center/system-center-2012-R2/gg682159(v=technet.10)#use-a-custom-script-to-determine-the-presence-of-a-deployment-type

Script exit code	Data read from STDOUT	Data read from STDERR	Script result	Application detection state
0	Empty	Empty	Success	Not installed
0	Empty	Not empty	Failure	Unknown
0	Not empty	Empty	Success	Installed
0	Not empty	Not empty	Success	Installed
Non-zero value	Empty	Empty	Failure	Unknown
Non-zero value	Empty	Not empty	Failure	Unknown
Non-zero value	Not empty	Empty	Failure	Unknown
Non-zero value	Not empty	Not empty	Failure	Unknown

In the above table, it can be seen (especially since I have rather handily highlighted the row), that to signify a successful installation, you simply have to write to the STDOUT data stream. In other words, if the STDOUT data stream is not empty (i.e. something has been written to it), then this will signify a successful installation \ detection.

In Practice

The PowerShell cmdlet *Write-Host* is what will be used to write to STDOUT. The PowerShell help files for Write-Host state: "Writes customized output to a host." Sounds perfect.

Write-Host will simply display some text to the screen, and you use it something like this:

Write-Host "Hello Reader, are you enjoying this book?"

Use the Docs

As always, I encourage you to take a quick "Time-Out" and explore the documentation for the Write-Host cmdlet[2].

Perhaps, fire up the PowerShell console too and try it out.

To signify a successful deployment, you use the PowerShell cmdlet: *Write-Host* to send a message to the screen via STDOUT but what message do you send?

Well, quite frankly, anything! You could use: *Write-Host "Success!"* or: *Write-Host "Installed"* just as much as you could use: *Write-Host "What a nice day!"* So long as you write something it will be interpreted as a success.

Note

Even though you are using Write-Host to write to the screen, the user will not see this.

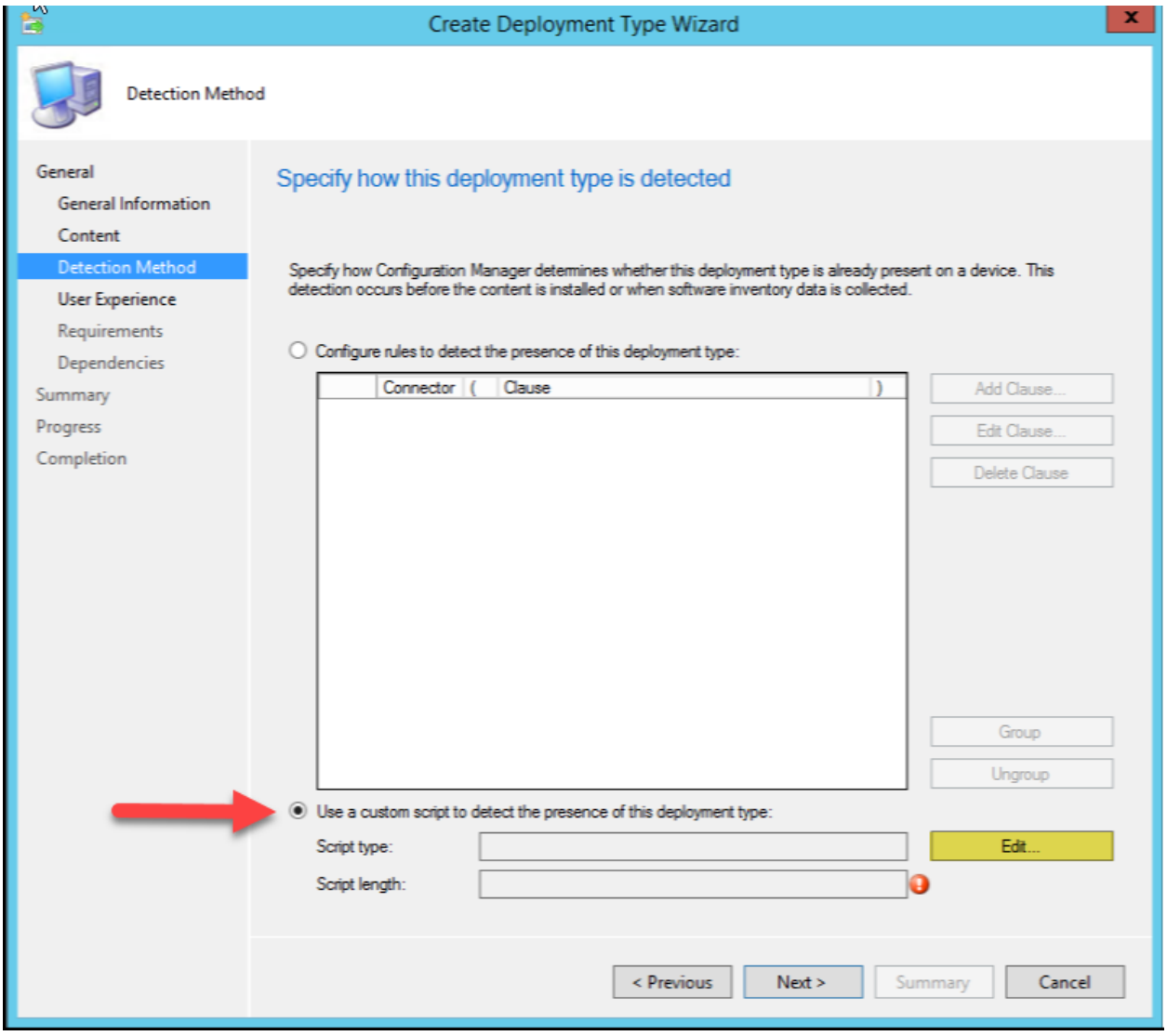
[2] https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/write-host?view=powershell-6

Where Do I Put My Detection Rules Anyway?

OK, you've got your detection rules written and tested (you did test them, right?), and you have copied them from the PowerShell ISE ready to paste into…

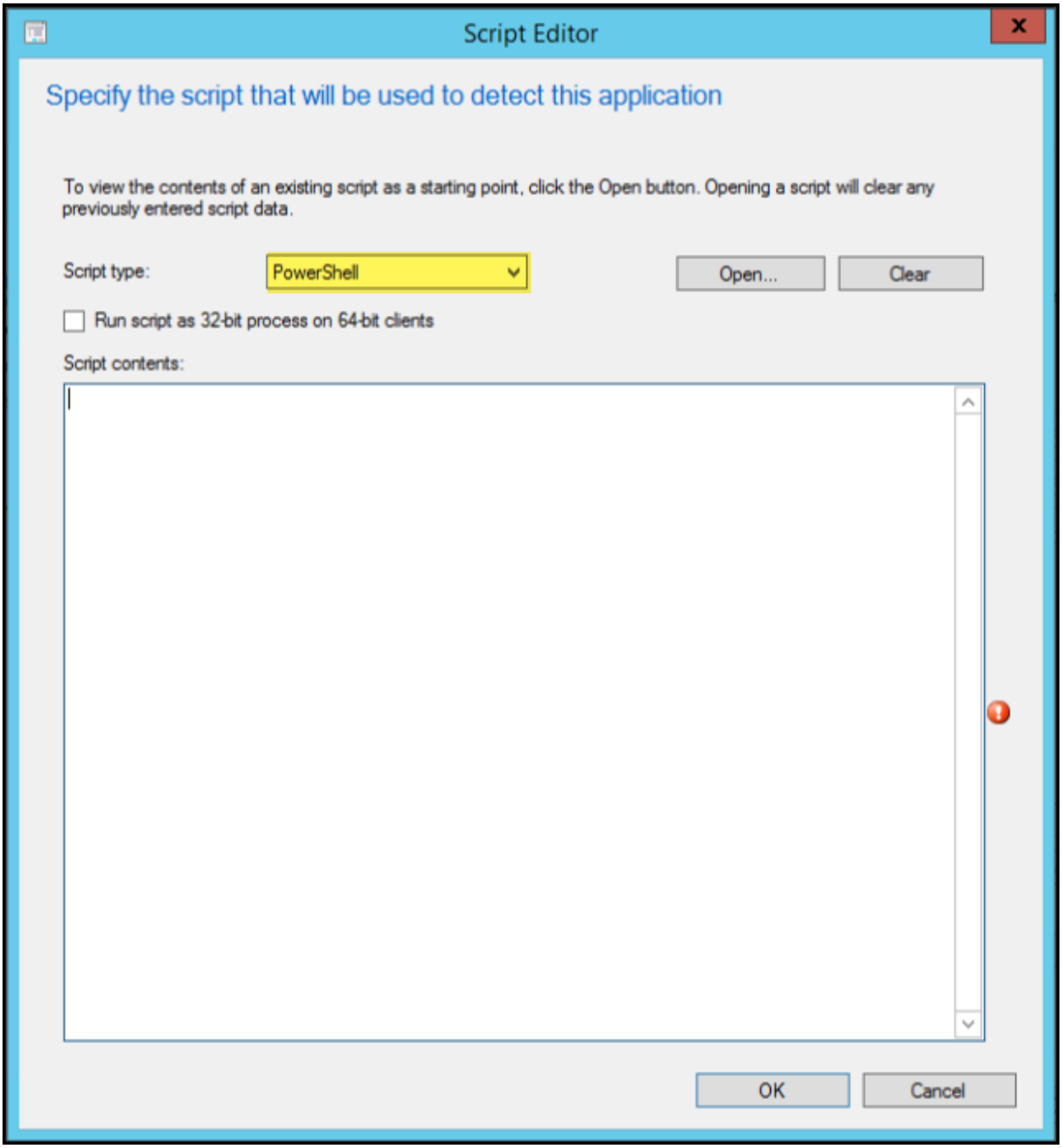
…into the same place (sort of), where you would normally add your detection rules.

It's in your Application that you lovingly crafted in the Configuration Manager console, except instead of selecting *Configure rules* to *detect the presence of this deployment type*, look further down and select Use a custom script to detect the presence of this deployment type followed by a swift click of the *Edit* button:



The Script Editor will then open. Select *PowerShell* from the dropdown menu where it says, *Script Type*. Then simply paste, (or write) your PowerShell detection script into the *Script Contents* text

box:


Script Editor
Specify the script that will be used to detect this application
To view the contents of an existing script as a starting point, click the Open button. Opening a script will clear any previously entered script data.
Script type:
PowerShell
Open...
Clear
Run script as 32-bit process on 64-bit clients
Script contents:
OK
Cancel

Silently Continue

In a lot of the PowerShell you are about to see, you will notice the following switch being used quite a frequently: *-ErrorAction SilentlyContinue*

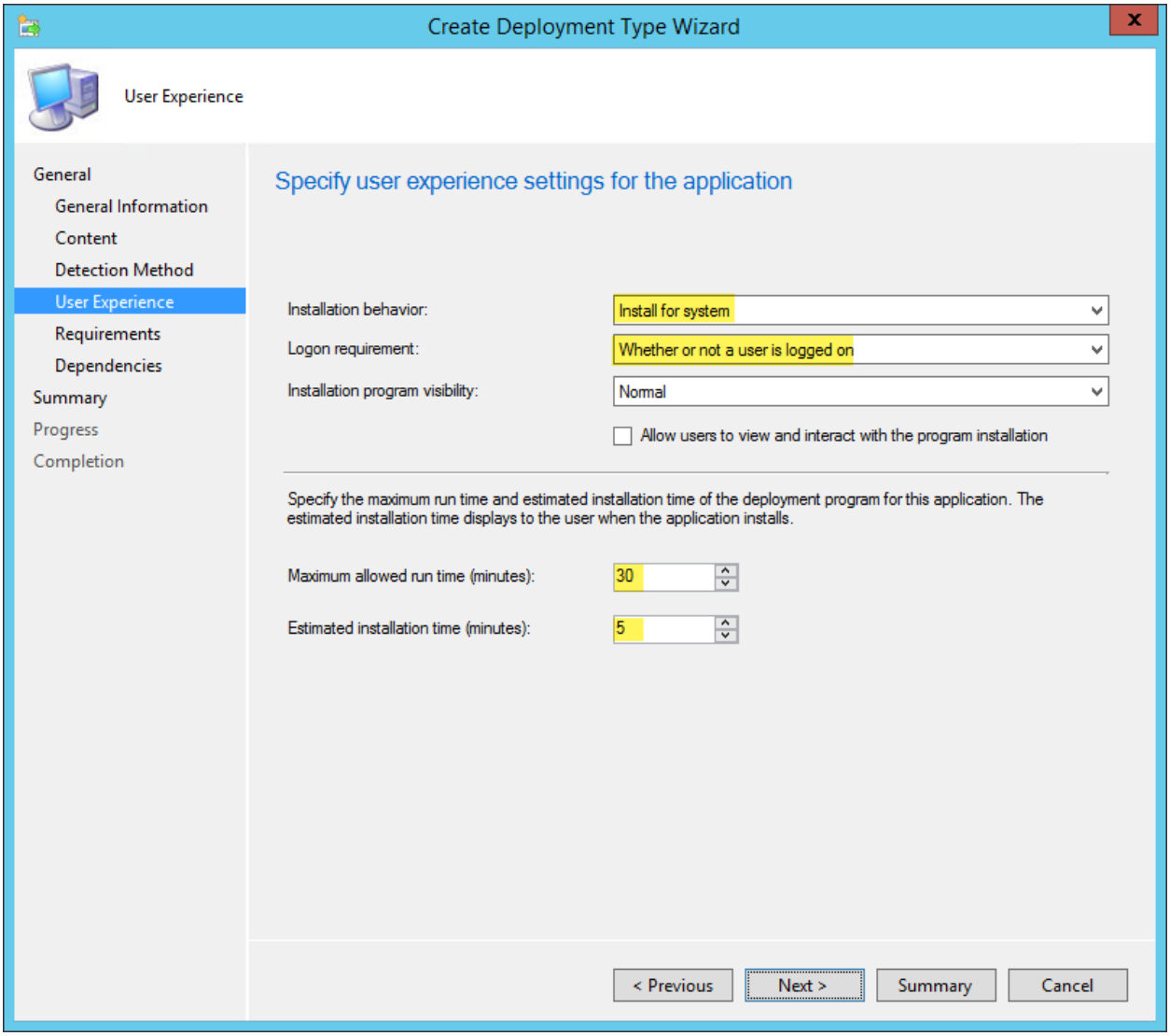
This is used to suppress non-terminating errors that would otherwise prevent your detection script from doing its job properly.

For example, if you were trying to validate a registry key that didn't exist, a terminating error would be produced resulting in script failure and the application detection state of *Unknown*. By adding this switch, the non-terminating error is suppressed, should it occur.

Detection Rule Context

You must be mindful of context when using PowerShell in detection rules. Does the detection rule run in the system context or the user context? And why does it matter?

One might assume that if you set your deployment type installation behaviour to 'System' then the detection script will run in the system context, as shown in the screenshot below:

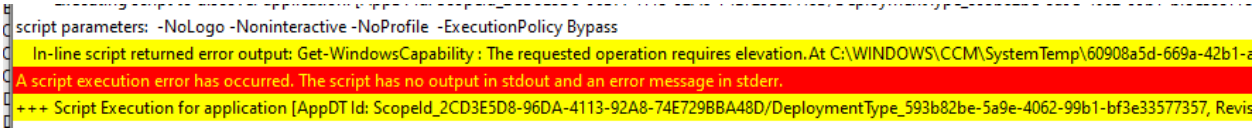


Unfortunately, this is not necessarily the case. If you set the Installation behaviour to *Install for system* and then deploy your application to users, then the detection script will run in the *logged-in user* context. This could cause you a problem.

Why Context Matters

If you have any PowerShell in your detection rule that needs to run elevated, for example, the *Get-WindowsCapability* cmdlet, then your detection script will fail. (I have no idea why this cmdlet will only run elevated!)

If you check the *AppDiscovery.log* on the client computer (the log files are usually found in *C:WindowsCCMLogs*), where you tried to install the deployed application, you may see something like this:



Where it can be seen: *Get-WindowsCapability: The requested operation requires elevation.*

This will cause your deployment to fail and the end-user will be presented with a message along the lines of: *"This software is not applicable to your device."*

The Solution

To ensure that the detection script runs elevated, you would need to deploy the application to a computer collection, not a user collection.

The table below shows you the various contexts that your detection script will run in depending on the Application Deployment's Installation behaviour and whether you have deployed your application to a user collection or a computer collection:

Installation Behaviour	Logon Requirement	Deployed to	Script Context
Install for user	Only when a user is logged on	User	User
Install for user	Only when a user is logged on	Computer	User
Install for system	Only when a user is logged on	User	User
Install for system	Only when a user is logged on	Computer	Computer
Install for system	Whether or not a user is logged on	User	User
Install for system	Whether or not a user is logged on	Computer	Computer

To Summarise

1. When the 'Application installation behaviour' is set to 'Install for system' and then deployed to a user collection then the PowerShell detection script for that Application is run as the logged-in user context.

2. When the ‘Application installation behaviour’ is set to ‘Install for system’ and then deployed to a computer collection then the PowerShell detection script for that Application is run in the system context. (Elevated).
3. When the ‘Application installation behaviour’ is set to ‘Install for user’, a PowerShell detection script for that Application is run in that user’s context.

Nine-times-out-of-ten the detection rule context won’t affect you at all, but every so often you’ll use something that can only be run elevated and it catches you out. (See the bonus chapter in this book, Step-by-Step: Deploying RSAT for Windows 10, as an example of this.)

Detection Types

Now that you understand the fundamentals of detection rules, let's look at some of the common (and not so common), detection rules that you will tend to use in your deployments.

File \ Folder Presence

One of the more simpler detection methods is to verify the existence of a File or Folder. This can be useful if you are copying files and folders as a part of your deployment and you want to be certain that they are there.

To use the code examples below, all you need to do is set the *$FileorFolderName* variable to the file or folder name you wish to detect, and then set the variable: *$Path* to the location of the file or folder. (You can add or leave trailing backslashes to your path as *Join-Path* sorts this out for you.)

This example detects for the presence of the file *e:\Users\Fred\Downloads\configmgr-DeployUsingPS.pdf*

DetectFile.ps1

```
$FileorFolderName = "configmgr-DeployUsingPS.pdf"
$Path = "e:\Users\Fred\Downloads"

If (Test-Path (Join-Path -Path $Path -ChildPath $FileorFolderName)) {
    Write-Host "Detected!"
}
```

This example detects for the presence of the folder: *c:\temp\A folder to detect*

DetectFolder.ps1

```
$FileorFolderName = "A folder to detect"
$Path = "c:\temp"

If (Test-Path (Join-Path -Path $Path -ChildPath $FileorFolderName)) {
    Write-Host "Detected!"
}
```

Executable Presence

It's pretty much identical code to file or folder detection - I've just changed one of the variable names to clarify that we are looking to detect an executable name.

In the following example, we are detecting for the presence of *C:\Program Files(x86)\Java\jre1.8.0_-201\bin\java.exe*

Note the use of the PowerShell environment variable: *${env:ProgramFiles(x86)}* in the code.

DetectExe.ps1

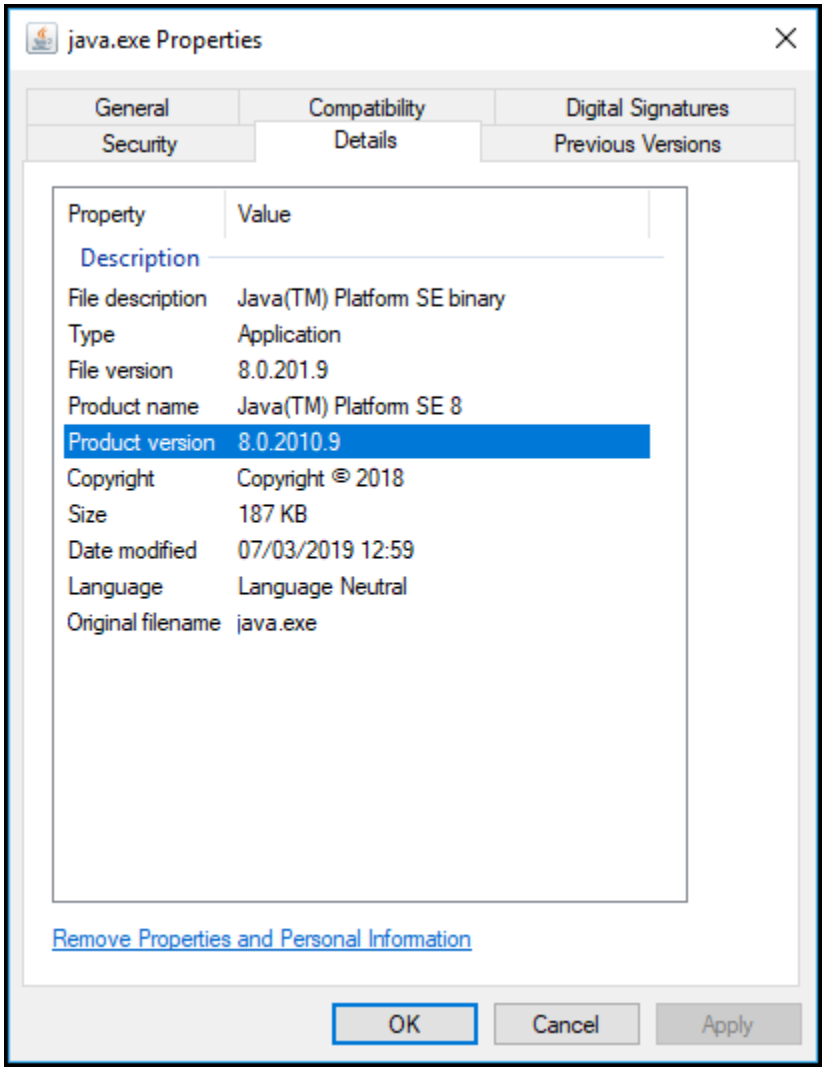
```
$Executable = "java.exe"
$Path = "${env:ProgramFiles(x86)}\Java\jre1.8.0_201\bin"

If (Test-Path (Join-Path -Path $Path -ChildPath $Executable)) {
    Write-Host "Detected!"
}
```

Executable Version

The example above, detecting an executable, is all fine and dandy, but more often than not you want to detect that the executable (be that an actual .exe, msi or something else), is the correct version number.

Continuing the example above of detecting the *java.exe*, let's now make sure it's also the version number that you were expecting.

In this instance, I want to ensure the version number of the .exe is: *8.0.201.9* as shown by right-clicking the *java.exe* file and looking at the *Details* tab in properties:



To obtain the file version info, use the PowerShell cmdlet: *Get-Item*.

DetectVersion.ps1

```
$VersionNumber = '8.0.2010.9'
$Executable = "java.exe"
$Path = "${env:ProgramFiles(x86)}\Java\jre1.8.0_201\bin"

If ((Get-item (Join-Path -Path $Path -ChildPath $Executable) -ErrorAction SilentlyCo\
ntinue).VersionInfo.ProductVersion -eq $VersionNumber) {
    Write-Host "Detected!"
}
```

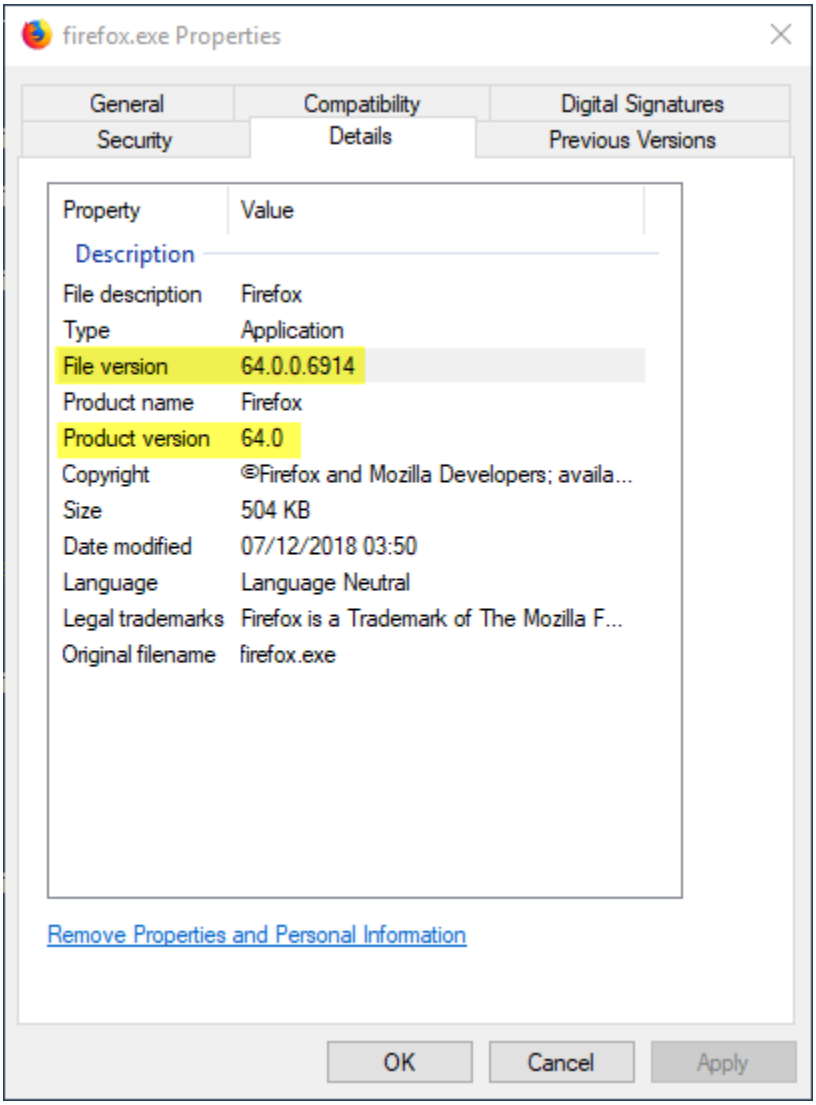
Top Tip!

You can also use the *FileVersion* property with *Get-Item* instead of *ProductVersion*. For example:
(Get-item (Join-Path -Path $Path -ChildPath $Executable)).VersionInfo.FileVersion Try them both out to see what results you get.

Hey! Where's the Build Number?

Sometimes though, you want the complete build number. For example, the Firefox.exe properties page displays the following:

File version: *64.0.0.6914* **Product version**: *64.0*



Using the *Get-Item* cmdlet to retrieve either the *FileVersion* or *ProductVersion* only returns: *64.0* as shown in the two screenshots below:

```
PS C:\> (get-item "C:\Program Files\Mozilla Firefox\firefox.exe").VersionInfo.FileVersion
64.0
```

```
PS C:\> (get-item "C:\Program Files\Mozilla Firefox\firefox.exe").VersionInfo.productversion

64.0
```

But…what if we want to detect just the build number: *6914*?

In this instance, you would need to use the rudely named property of *FilePrivatePart* to retrieve it and compare against as shown in the screenshot below:

```
PS C:\> (get-item "C:\Program Files\Mozilla Firefox\firefox.exe").VersionInfo.FilePrivatePart
6914
```

In your detection script, you could then detect for the product number *and* the build number being correct to signify a successful installation:

DetectVersionAndBuild.ps1

```
$VersionNumber = '64.0'
$BuildNumber = '6914'
$Executable = "firefox.exe"
$Path = "$Env:ProgramFiles\Mozilla Firefox"

If ((Get-item (Join-Path -Path $Path -ChildPath $Executable) -ErrorAction SilentlyCo\
ntinue).VersionInfo.ProductVersion -eq $VersionNumber -and (Get-item (Join-Path -Pat\
h $Path -ChildPath $Executable) -ErrorAction SilentlyContinue).VersionInfo.FilePriva\
tePart -eq $BuildNumber) {
    Write-Host "Detected!"
}
```

Registry Key

It's rare to only wish to detect for the presence of a registry key. (Usually, you would want to detect for a specific value.)

Should you need to though, the detection rule would resemble this: (By now I'm sure I don't need to tell you to adjust the variable: $RegistryKey in the code to meet your needs.)

DetectRegistryKey.ps1

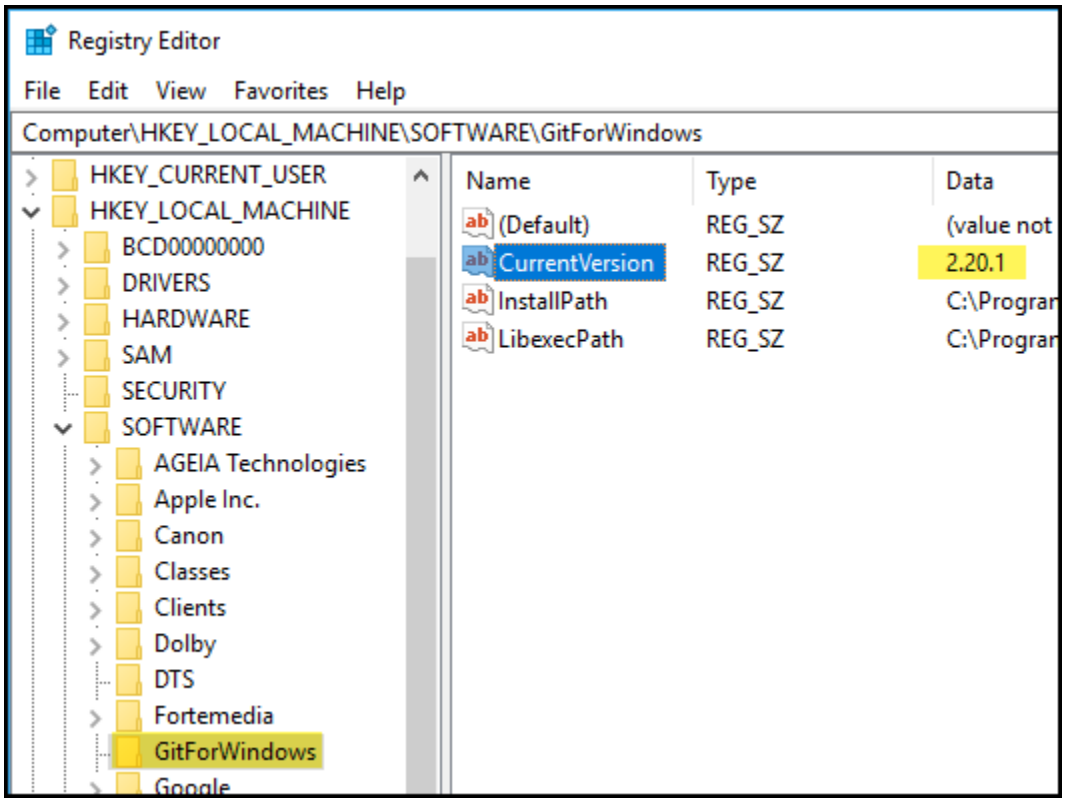
```
$RegistryKey = "HKLM:\SOFTWARE\Google\Chrome"

If (Test-Path $RegistryKey) {
    Write-Host "Detected!"
}
```

Registry Value

Now, this is more like it. Often, when you are using the registry for detection rules, you will be trying to detect a specific value and its corresponding data to signify a successful deployment.

Let's pretend that you have deployed the application *Git for Windows* and decide that you will use the registry as the detection rule.

You want to be certain that version *2.20.1* has been deployed and so you will detect the registry value *CurrentVersion* and its corresponding data as shown in the image below:



The detection rule for this example looks as follows:

DetectRegistryValue.ps1

```
$RegistryKey = 'HKLM:\SOFTWARE\GitForWindows'
$Value = 'CurrentVersion'
$Data = '2.20.1'

if ((Get-ItemProperty -Path $RegistryKey | Select-Object $Value -ExpandProperty $Val\
ue -ErrorAction SilentlyContinue) -eq $data) {
    Write-Host "Detected!"
}
```

Why Don't You Just Silently Continue?

One might wonder why you don't simply use the following PowerShell:

Get-ItemPropertyValue -Path $RegistryKey -Name $Value -ErrorAction SilentlyContinue

The reason is that if you enter an invalid registry path, *Get-ItemPropertyValue* will consider it to be a *terminating* error and the parameter *-SilentlyContinue* will not suppress terminating errors.

Hence the convoluted, yet effective workaround used in the detection rule for a registry value. Now the script is nice and clean: either it is successful and writes to the screen (Write-Host), or it is not successful and does nothing. (Signifying a detection failure.)

Custom Detection

Why Use Custom Detection?

There may come a time where you can't find anything to detect or simply don't care too much and wish to signify that the deployment was successful regardless of any *true* verification.

In ~~lazy~~ edge cases like these, (and there should be very few instances where you need to do this), you can either create a file or registry key \ value to detect as part of your deployment script and then validate its existence in the detection rules to signify a successful deployment.

By File

You have already discovered how to detect a file for your detection rules, so let's see how you could create a file as part of your deployment that you can then detect in a detection rule.

Remember!

Just to clarify, you would add this PowerShell code to your *application deployment script*, not the detection rule script. You are going to create a file as part of the deployment script and then in the detection script validate that the file you wrote exists to signify a successful deployment. Sneaky, eh?

Writing a file is extremely straightforward, although a good practice to follow is to always create your custom files in a specific folder in a specific location. (Away from users' prying eyes.)

I usually write to the Windows directory, in a subdirectory named after the company I work for, followed by a directory called: *CustomDetection*.

For example, let's pretend I work for Yum Yum Dog Foods Inc then I would create a file here:

$Env:SystemRoot\Yum Yum Dog Foods Inc\CustomDetection

The PowerShell code would check for the folder structure existence first, and if it didn't exist it would create it; then it writes the file.

Let us continue our pretence and assume that I want to detect the deployment of *MyApp*. I would write a file called *MyApp-001.log*. If I needed to make any changes to my deployment, I could adjust the deployment script to write the file *MyApp-002.log* etc. and alter my detection script to match.

Sure, it's messy. But that's why it should only be used in edge-cases.

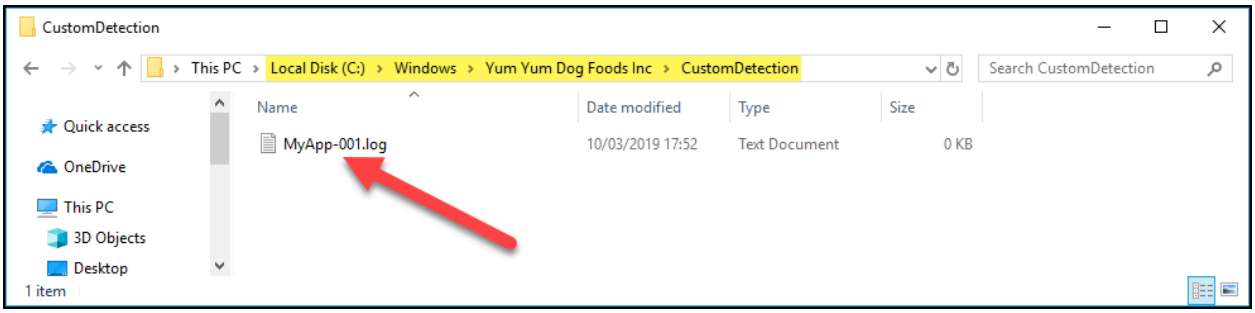
Write to the File Contents

If you were so inclined, you could write to the file instead. i.e. create and \ or open the file and write *001* to the file contents or something else to detect. Then you could detect what was inside the file instead of detecting the name of the file. But to be honest, I don't find it's worth the bother. Detecting the file name is just fine.

To use this code, simply replace the values for the variables, *$CompanyName* and *$FileName* where '*Yum Yum Dog Foods Inc*' is the name of the company you work for and '*MyApp-001.log*' is the name of the file that you are going to detect to signify a successful deployment.

DetectCustomFile.ps1

```
$CompanyName = 'Yum Yum Dog Foods Inc'
$FileName = 'MyApp-001.log'

$DetectionPath = Join-Path -Path  "$env:SystemRoot" -ChildPath "$CompanyName\CustomD\
etection"

if (!(Test-Path $DetectionPath)) {
    New-Item -Path $DetectionPath -ItemType Directory | Out-Null
}

New-Item -Path $DetectionPath -Name $FileName -ItemType File -Force | Out-Null
```



By Registry

For the same reasons that you may want to detect a custom file, you may prefer to write to the registry instead and detect a value there. Particularly if you are writing a bunch of custom registry keys and values as part of your deployment anyway, it may make sense to add one more registry keys and values and read the associated value data in your detection rule.

Once again, you already know how to read from the registry as part of your detection rules so let's see how to write to the registry to be detected.

As a best practice, you should try to always write to a specific part of the registry using a specific structure. This keeps everything neat and tidy.

Continuing the example of working for Yum Yum Dog Foods Inc, here is where you could to write to:

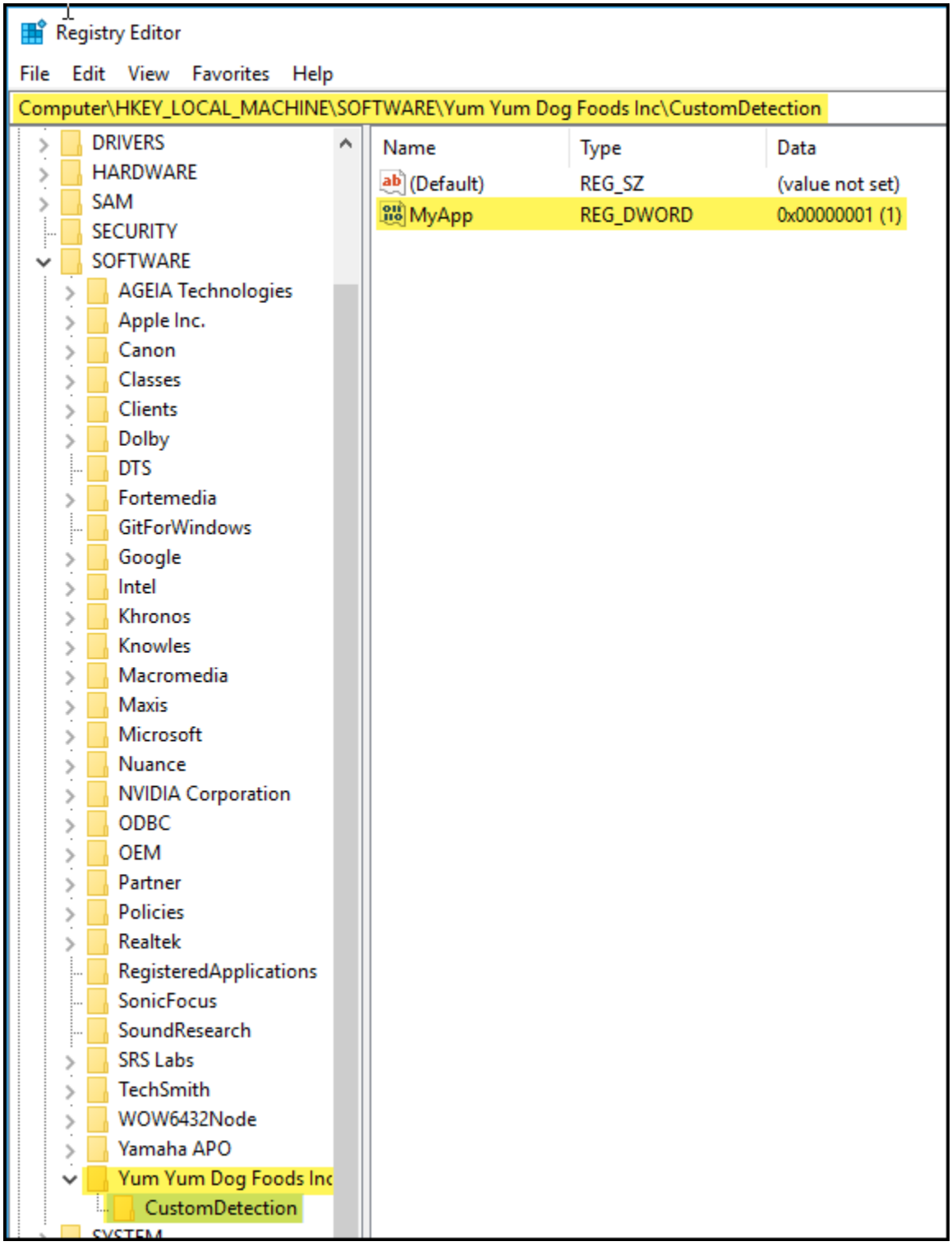
HKEY_LOCAL_MACHINE\SOFTWARE\Yum Yum Dog Foods Inc\CustomDetection

Once again, the PowerShell code will first check to see if this location exists in the registry already, and if not, creates it before writing the Value and Data that will be used for detection.

To use this code, simply replace the values for the variables, *$CompanyName* with the name of the company that you work for, *$Name* with the name of the application (in reality, this can be anything), and *$Value* with the value that you will detect in the detection rule. This can be anything but for simplicity, it's best to stick with sequential numbers starting from 0 or 1.

DetectCustomRegistry.ps1

```
$CompanyName = 'Yum Yum Dog Foods Inc'
$Name = 'MyApp'
$Value = '1'

$RegistryPath = Join-Path -Path "HKLM:\SOFTWARE" -ChildPath "$CompanyName\CustomDete\
ction"

if (!(Test-Path $RegistryPath)) {
    New-Item -Path $RegistryPath -Force | Out-Null
}

New-ItemProperty -Path $RegistryPath -Name $Name -Value $Value -PropertyType DWORD -\
Force | Out-Null
```


Registry Editor
File Edit View Favorites Help
Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Yum Yum Dog Foods Inc\CustomDetection
DRIVERS
HARDWARE
SAM
SECURITY
SOFTWARE
AGEIA Technologies
Apple Inc.
Canon
Classes
Clients
Dolby
DTS
Fortemedia
GitForWindows
Google
Intel
Khronos
Knowles
Macromedia
Maxis
Microsoft
Nuance
NVIDIA Corporation
ODBC
OEM
Partner
Policies
Realtek
RegisteredApplications
SonicFocus
SoundResearch
SRS Labs
TechSmith
WOW6432Node
Yamaha APO
Yum Yum Dog Foods Inc
CustomDetection
Name
Type
Data
(Default)
REG_SZ
(value not set)
MyApp
REG_DWORD
0x00000001 (1)

Branching

Because you are using PowerShell in our detection rule, the sky's the limit in so far as what can be achieved. Sometimes. you will want to detect either in one location or another, for example in *Program Files (x86)* or *Program Files* depending on if a 32-bit or 64-bit installation has occurred. Or perhaps you wish to detect in different locations depending on Office bitness (i.e. a 32-bit Office installation or a 64-bit Office installation.)

Here are some code examples that can be used in your detection rules that do just that:

By Office Bitness

You may need to branch one way or another depending on the installed Microsoft Office 'bitness'. To detect if Office is either a 32-bit or 64-bit installation I check the registry value *Bitness* for either *x86* or *x64* at the following location:

HKLM:\Software\Microsoft\Office\<office version>\Outlook

DetectOfficeBitness.ps1

```
$OfficePath = 'HKLM:\Software\Microsoft\Office'
$OfficeVersions = @('14.0','15.0','16.0')

foreach ($Version in $OfficeVersions) {
    try {
        Set-Location "$OfficePath\$Version\Outlook" -ea stop -ev x
        $LocationSet = $true
        break
    } catch {
        $LocationSet = $false
    }
}

if ($locationSet) {
    #Check for bitness then check correct file version
    switch (Get-ItemPropertyValue -Name "Bitness") {
        "x86" { if ( <# Detect something here - file existence, file version etc #> \
) { Write-host "Installed!"}}
        "x64" { if ( <# Detect something here - file existence, file version etc #> \
) { Write-host "Installed!"}}
```

```
21          }
22  }
```

Examples

What follows is some 'real-world' examples of branching detection rules:

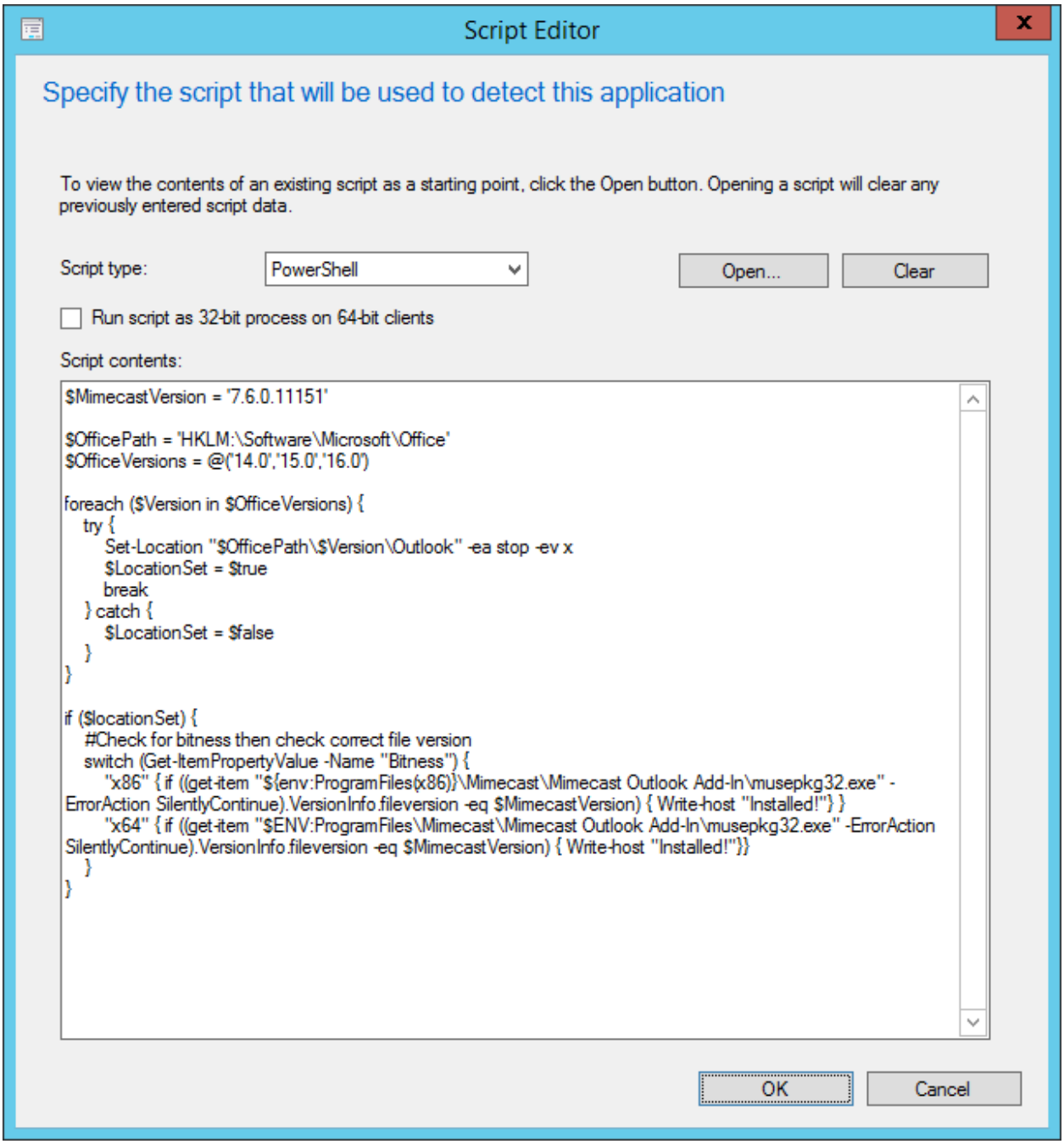
Mimecast Detection (Branching Example)

In this example, the Microsoft Office bitness is detected to branch to either *Program Files (x86)* or *Program Files*. From there, the version of *musepkg32.exe* is checked to ensure that it matches *7.6.0.11151*. If it does: it's a successful deployment.

DetectMimecast.ps1

```
$MimecastVersion = '7.6.0.11151'

$OfficePath = 'HKLM:\Software\Microsoft\Office'
$OfficeVersions = @('14.0','15.0','16.0')

foreach ($Version in $OfficeVersions) {
    try {
        Set-Location "$OfficePath\$Version\Outlook" -ea stop -ev x
        $LocationSet = $true
        break
    } catch {
        $LocationSet = $false
    }
}

if ($locationSet) {
    #Check for bitness then check correct file version
    switch (Get-ItemPropertyValue -Name "Bitness") {
        "x86" { if ((get-item "${env:ProgramFiles(x86)}\Mimecast\Mimecast Outlook Ad\
d-In\musepkg32.exe" -ErrorAction SilentlyContinue).VersionInfo.fileversion -eq $Mime\
castVersion) { Write-host "Installed!"} }
        "x64" { if ((get-item "$ENV:ProgramFiles\Mimecast\Mimecast Outlook Add-In\mu\
sepkg32.exe" -ErrorAction SilentlyContinue).VersionInfo.fileversion -eq $MimecastVer\
sion) { Write-host "Installed!"}}
    }
}
```

Here's what it looks like 'pasted' into the ConfigMgr Script Editor:



Java Detection (This *and* This)

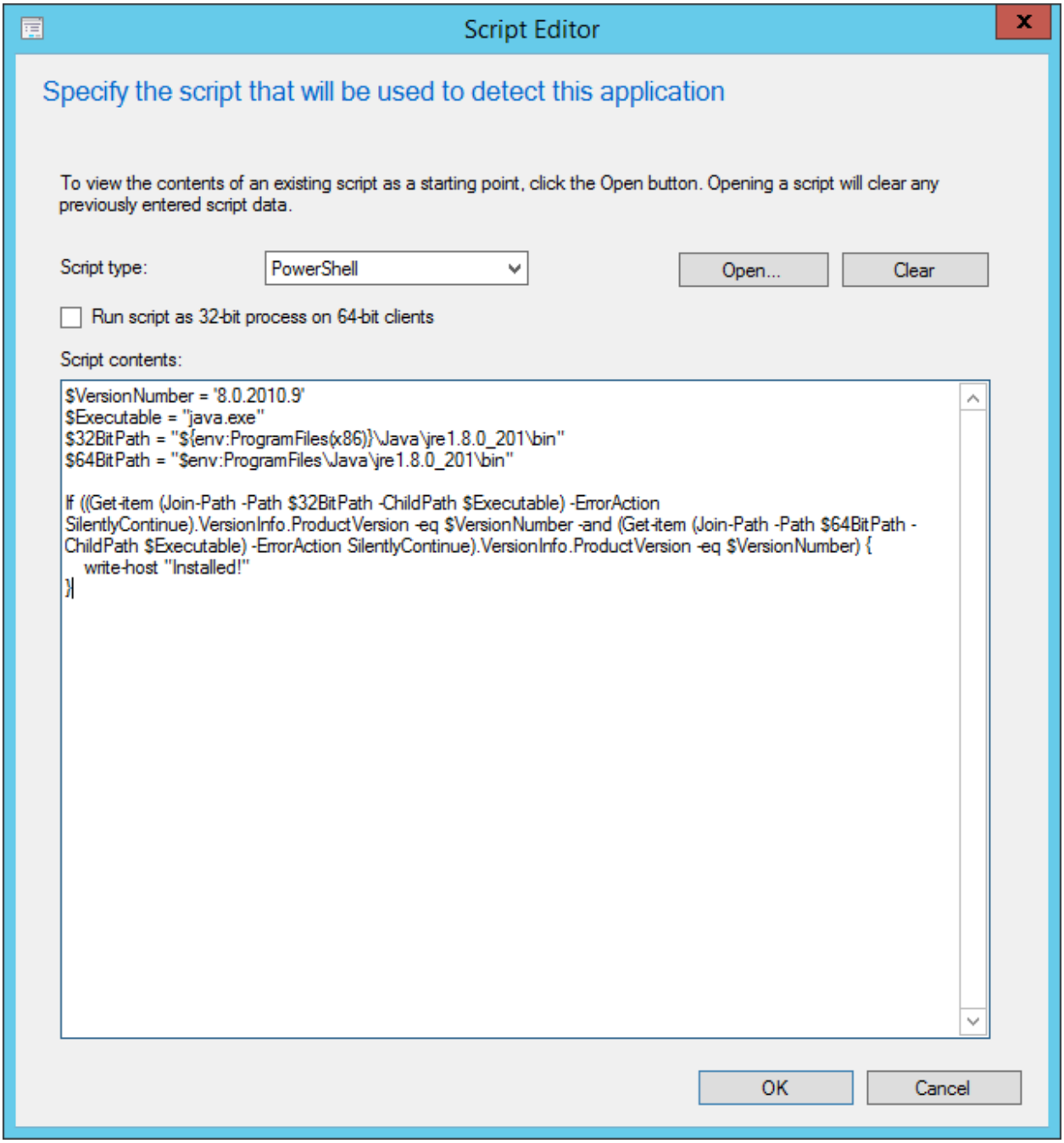
In this example, both versions of Java (32-bit and 64-bit) have been deployed and therefore for the deployment to be considered successful, both versions of Java needed to be detected.

To do this, simply check that the version number of *Java.exe* in both *Program Files (x86)* and *Program Files* match the required value for the deployment to signify success.

DetectJava.ps1

```
$VersionNumber = '8.0.2010.9'
$Executable = "java.exe"
$32BitPath = "${env:ProgramFiles(x86)}\Java\jre1.8.0_201\bin"
$64BitPath = "$env:ProgramFiles\Java\jre1.8.0_201\bin"

If ((Get-item (Join-Path -Path $32BitPath -ChildPath $Executable) -ErrorAction Silen\
tlyContinue).VersionInfo.ProductVersion -eq $VersionNumber -and (Get-item (Join-Path\
 -Path $64BitPath -ChildPath $Executable) -ErrorAction SilentlyContinue).VersionInfo\
.ProductVersion -eq $VersionNumber) {
    write-host "Installed!"
}
```

Here's what it looks like 'pasted' into the ConfigMgr Script Editor:


Script Editor
Specify the script that will be used to detect this application
To view the contents of an existing script as a starting point, click the Open button. Opening a script will clear any previously entered script data.
Script type:
PowerShell
Open...
Clear
Run script as 32-bit process on 64-bit clients
Script contents:
$VersionNumber = '8.0.2010.9'
$Executable = "java.exe"
$32BitPath = "${env:ProgramFiles(x86)}\Java\jre1.8.0_201\bin"
$64BitPath = "$env:ProgramFiles\Java\jre1.8.0_201\bin"
If ((Get-item (Join-Path -Path $32BitPath -ChildPath $Executable) -ErrorAction SilentlyContinue).VersionInfo.ProductVersion -eq $VersionNumber -and (Get-item (Join-Path -Path $64BitPath -ChildPath $Executable) -ErrorAction SilentlyContinue).VersionInfo.ProductVersion -eq $VersionNumber) {
write-host "Installed!"
}
OK
Cancel

The Story so Far

Well, who knew there would be so much to cover about detection rules?

You learned about how to determine what signifies a successful detection through a handy table discovered on the Microsoft website as well as how to signify that detection in ConfigMgr by using the Write-Host PowerShell cmdlet.

More importantly, you discovered whereabouts in the Configuration Manager console to place the PowerShell detection rules.

Whilst size may or may not matter, the context that your detection rule runs in most certainly does. You have learned not to worry too much though, as thankfully it's generally rare to require running detection rules under the system context, but if required, you know how to do it.

Various methods were discovered that can be used for detection such as by an executable, a file, a folder or registry; and how to detect them. You then went to the Dark-Side by writing custom files and registry keys to detect.

Following all of this up was how you can use branching: if this then that type of thing to direct your PowerShell code detection down a certain path e.g. turn left if it's a 64-bit detection and turn right if it's a 32-bit detection.

Lastly, there's a couple of examples that I have used in a production environment that you can use as a reference.

Part 4: Location, Location, Location

So, you have your lovingly crafted PowerShell deployment script that installs the XYZ.msi. However, after the msi installs you need to copy some jpg files and a couple of ini files. Let's cover some essential ground that you need to know about to do this correctly.

Where Is This Script Running from Anyway?

If you always knew the location that your script was in and the files it referenced, it would be a fantastic thing indeed. If you knew that the script was running from *c:\SomeDirectory\MyScript.ps1* and in that script you referenced files that were located in the same directory as your script then life would be magic.
But alas, when deploying scripts using Configuration Manager, you have no idea of the name of the folder in the *ccmcache* that will contain our script and files essential to the deployment. You need to find out how to tell where the script is running from so that you can reference the files that it deploys; be that a msi or exe for installation or any other file that gets referenced from the same directory. In other words, you need some sort of method that says, in your deployment script, "Deploy the MSI from this location: *WhereMyScriptIsRunningFrom\MyMSI.msi*"

How We Used to Do Things

In the good old days, you would write a batch file and use *%~dp0* The *%~dp0* variable, when referenced within a Windows batch, file will expand to the drive letter and path of that batch file. This meant that you would never need to know where the batch file was running from. The task now is to find a solution that does the same thing but using PowerShell.

The Various Solutions

When I first started investigating this I found many solutions, all based around the same sort of PowerShell that went something like this:

InvokeScript.ps1

```
1  $WorkingDir = $MyInvocation.MyCommand.Path | Split-Path -Parent
```

This will work, but the problem I found was that I could never remember it and it just seemed downright confusing.

A Better Way

I now use the PowerShell cmdlet: *Get-Location* It's simple and elegant. But best of all, it's easy to understand and remember.

InvokeABetterWay.ps1

```
1 $WorkingDir = Get-Location
```

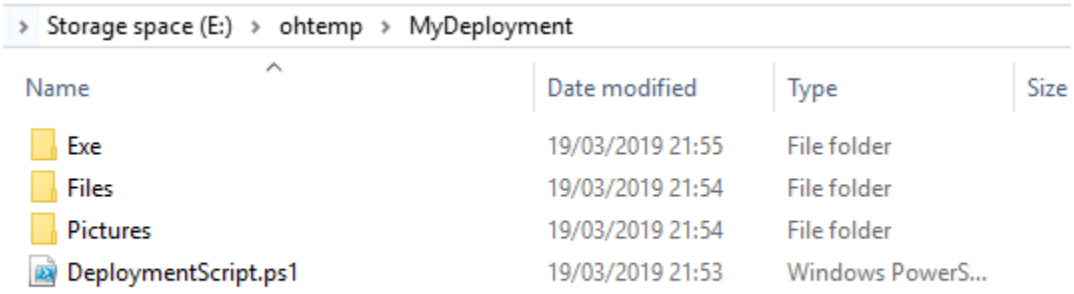
File Placement

Sometimes, you'll have some files that you either want to install or copy to the destination computer along with your deployment. These files need to be somewhere right? Right. Let's find out where to put your files.

Where to Place Your Files for Deployment

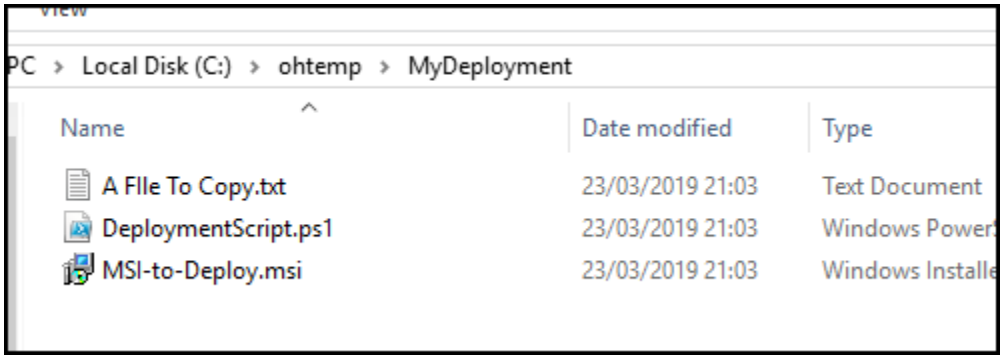
You may have an MSI to install and some files to copy all in the same deployment script. In which case, your file structure may then look like this:

Structured:



(The root folder in the above screenshot: "MyDeployment" is the folder that you would copy onto your ConfigMgr Source Files location – just as you would with any other "normal" deployment.) Most of the time though, I simply dump all files in the root directory, so it looks like this:

Flat:



The bottom line is, do what you think is necessary for the easiest deployment. There's no right or wrong way, but nine-times-out-of-ten I'll just go with the "flat" structure out of pure laziness.

Referencing Files

Now that you have given some thought about where to place your files, and you know how to get the script location (remember: *Get-Location*!), you can reference the files very easily whether it's the flat or structured method. Let's see how. To start with, in my deployment script somewhere near the start, I like to assign the script location to a variable, *$WorkingDir* like so:

AssignScriptLocation.ps1

```
1 $WorkingDir = Get-Location
```

... and then set the current location to where this script is running from like so:

SetLocation.ps1

```
1 $WorkingDir = Get-Location
2
3 Set-Location $WorkingDir
```

This way, I always know where the root location of my folder that contains all my files resides. ($WorkingDir) Sometimes, depending on my needs, I may use Push-Location along with its partner-in-crime: *Pop-Location*. (See Part 1 of this book for a handy reminder on these couple of Cmdlets)

PushLocation.ps1

```
1 $WorkingDir = Get-Location
2
3 Push-Location $WorkingDir
```

This achieves the same thing as Set-Location but gives me a bit more flexibility later down the line. Now this has been done (either *Set-Location* or *Push-Location*), not only has the location of where the script is running from been set, but there is also a handy variable that contains the same location available for potential reference later. Now when it is required to reference any files from within the deployment script, it's a piece of cake.

Referencing Files in a Flat Structure

This couldn't be easier, as we have already set the current location to where the script is running from (*Set-Location $WorkingDir*), all that is required to reference anything is precede the filename with .\ (Period backslash.) .\ means the path is relative to the current directory. For example, to reference a

msi called MyGreatProgram.msi you would use this syntax in your PowerShell deployment script: *.MyGreatProgram.msi* Remember: you can do this because you found out the directory that the script is running from (*Get-Location*), assigned that location to a variable, (*$WorkingDir*), and then set the current location to the path held in the *$WorkingDir* variable. To reference a text file would be…you guessed it: *.MyTextFile.txt*

Referencing Files in Subdirectories

To reference files that are more structured and organised is just as simple. Again, because the current location has been set to where the script is running from, you can do this:

.\Subdirectory\MyFile.txt

Simple eh?

Just because I'm feeling generous, and to clarify any potential confusion, I'll throw in an example…

Let's say I have a jpg file called: *Wallpaper.jpg* and it's in a subdirectory of the root called: *Images*. You would reference it in your PowerShell deployment script like this:

*.\Images\Wallpaper.jpg *

If You're Elsewhere…

If for some reason, you find yourself in another working location by using Set-Location to change your current working directory, you can still reference the *Wallpaper.jpg*, (as per the previous example), by using the variable that was set to the current location: *$WorkingDir* (Remember? You set this near the start of your script before all of the clever magic happens: *$WorkingDir = Get-Location*) You could then use that variable to reference files from anywhere in your script to any location from the root folder: *$WorkingDir\Images\Wallpaper.jpg* All set? Good stuff!

And Finally…

I'm sure that you realise by now that the final placement of your deployment script and other files should all live in a single root directory, (and potentially sub-directories thereof), and stored somewhere that you would normally place your files for deployment in ConfigMgr. Because later, in Part 6 of this book, you will learn how to deploy your script using ConfigMgr's 'Application' method.

The Story so Far

This was a fun one, wasn't it?

You've learned why it's essential to know where your PowerShell deployment script is running from and how it used to be done in the good old batch file days. It was discovered that the Internet is full of alternate PowerShell solutions to achieve the same thing but, you know what? There's a more elegant solution that's easy to remember.

You learned about file placement and how to reference those files be they all in the root or a neat structured set of subdirectories.

Finally, you discovered that by setting the current script location to a variable, the script and files can be referenced from wherever they may be.

This was a very important part of the journey. If you don't know how to reference the files correctly, how are you ever going to deploy the msi, the executable, or copy files? How will you navigate the file system or registry and get back to the original working directory without this important knowledge? They are important concepts and crucial to an effective script deployment. If you need to, read this part again.

Part 5: Installing the Program

We are near the end of our journey now and if you've made it this far: well done!
Up to this point, we've enjoyed a small white wine, some Dorito's and even a prawn-cocktail, but now we've reached the "main course" of deployment using PowerShell.

Calling the MSI or Setup.exe

When it comes to installing or uninstalling the msi or setup.exe you cannot just call msiexec.exe natively in the PowerShell script as if it were being typed in a command window. Likewise, if you only have a setup.exe you cannot just reference it and expect magical things to happen. So how do you call the installation or uninstallation of your program from the PowerShell deployment script? You'll be pleased to know that whether you are deploying a msi or a setup.exe the PowerShell cmdlet used is identical.

Start Your Engines Please

Well you're not starting an 'engine' per se, but: Start-Process This cmdlet is the one you will tend to use to initiate the installation of your msi or setup file. Let's begin with a quick overview of what it does.

Here's what the Microsoft PowerShell documentation has to say about it:

> **Start-Process**
>
> The Start-Process cmdlet starts one or more processes on the local computer. To specify the program that runs in the process, enter an executable file or script file, or a file that can be opened by using a program on the computer.
>
> If you specify a non-executable file, Start-Process starts the program that is associated with the file, similar to the Invoke-Item cmdlet.

If, after reading that, you're thinking it's just the ticket that we need, then you are bang-on. As an aside, I would advise that you take a few minutes out to have a quick read of the help file for the cmdlet: Invoke-Item too, just so that you are aware of the differences - it could come in handy.

Parameters

The cmdlet *Get-Process* has a few parameters however you tend to use just four.

Follow me for a closer look…

-FilePath

Depending on if you are going to deploy an MSI or Executable the FilePath parameter is going to be different:

MSIEXEC

This is where you will specify the path to msiexec.exe. Like so:

Start-Process -Filepath "$ENV:SystemRoot\System32\msiexec.exe"

EXE

If you are deploying an exe, simply specify the name of your exe installer. If you have used my recommendations earlier of already setting our current location to the script root, (See Part 4 of this book if you need a reminder), and you are using the 'lazy' flat-structure method i.e. all files are in the root directory, then you do this:

start-process -filepath ".\7z1900-x64.exe"

Top Tip

If your current location has been set to the script root through the use of a variable: a good one would be $WorkingDir (you did do that, right?), then you don't need to use .\ in front of the setup.exe. I would always recommend using *$WorkingDir\Setup.exe* instead for script clarity.

-ArgumentList

This is where you can specify any parameters that need supplying to the msi or setup.exe:

MSI

With an MSI, be sure to specify the actual .msi as well as any parameters required:

-ArgumentList "/i $WorkingDir\jre1.8.0_191.msi /qn /norestart"

Break It Down! In the above example, you can see how all of the desired parameters have been supplied on one line surrounded by quotes. It starts with /i to signify to msiexec that it should install, followed by the name of the actual msi file. Then there is the /qn switch to signify it's a silent but deadly installation, and finally, the self-explanatory /norestart switch is added. Note the use of the variable *$WorkingDir* that you would have previously set in your deployment script to the script root location using: *$WorkingDir = Get-Location*

EXE

-ArgumentList "/S"

In the above example, a single parameter has been passed for a silent installation of a Setup.exe file.

-NoNewWindow

This parameter starts the new process in the current console window. By default, PowerShell opens a new window and you don't want that.

-Wait

Indicates that this cmdlet waits for the specified process and its descendants to complete before accepting more input. This parameter suppresses the command prompt or retains the window until the processes finish. In other words, don't do anything else until this process has finished! Don't miss this parameter out or your script may go a bit screwy, especially if you are adding post-deployment tasks that are dependent on your msi or setup installation.

Dealing with Spaces

What if you have spaces in your msi file name, for example, what if your msi was named: *Mimecast for Outlook 7.6.0.26320 (32 bit).msi* (True story!) Of course, you could always rename the file and remove the spaces but luckily, the solution is rather simple: Simply enclose the path and msi name in double quotes like so:

”“$workingDir\Mimecast for Outlook 7.6.0.26320 (32 bit).msi””

Continuing the example, your complete command line would effectively look like this:

DealingWithSpaces.ps1

```
start-process -FilePath "$ENV:SystemRoot\System32\msiexec.exe" -ArgumentList "/i ""\
$workingDir\Mimecast for Outlook 7.6.0.26320 (32 bit).msi"" /qn" -NoNewWindow -Wait
```

As a habit, try to always use double-quotes regardless of whether the MSI has spaces in its name or not.

Putting It All Together

Nothing helps more than a good set of solid examples to help with the reinforcement of concepts. C'mon, I've got some....

Example 1 - Simple MSI

This example silently installs the 64-bit version of 7-zip using the msi installer:

SimpleMSI.ps1

```
Start-Process -Filepath "$ENV:SystemRoot\System32\msiexec.exe" -ArgumentList "/i $W\
orkingDir\7z1900-x64.msi /qn /norestart"
```

Note the use of the variable: *$WorkingDir* that would have previously been set in the PowerShell deployment script to the script root location using: *$WorkingDir = Get-Location*

Is this sinking in yet?

Example 2 - MSI with Properties

This example silently installs Java using the MSI installer and sets several properties:

MSIWithProperties.ps1

```
Start-Process -FilePath "$ENV:SystemRoot\System32\msiexec.exe" -ArgumentList "/i $W\
orkingDir\jre1.8.0_191.msi /qn JAVAUPDATE=0 AUTOUPDATECHECK=0 IEXPLORER=1 REBOOT=Sup\
press" -NoNewWindow -Wait
```

Example 3 - Setup.Exe

This example silently installs the 64-bit version of 7-zip using the executable installer and passing the argument for a silent installation:

SilentEXE.ps1

```
start-process -FilePath ".\7z1900-x64.exe" -ArgumentList "/S" -NoNewWindow -Wait
```

Note the use of .\ to signify that the exe file is in the current directory. (This example deliberately does not to use *$WorkingDir* as a demonstration of an alternate method.)

The Story so Far

You've learned that a single PowerShell cmdlet: Start-Process is used in the deployment script as the "engine" to install the .msi or .exe file. The most useful parameters were explained, and you finished up with a few examples for good measure. The Journey is nearly over. Hang in there. You've nearly made it.

Part 6: Deploying the Script

OK, so you have your deployment script. You've run it manually and everything works as expected. This is the moment you have been building up to; pacing up and down like an expectant father. Like some bastard child of Frankenstein's monster, it's time to give this thing some life: it's time to deploy using ConfigMgr. <Insert fanfare here.>

Calling Your Script

Your deployment script then: Is it a function? Does your function accept parameters? Does your script have an entry point that calls a function? Is it just a script that reads sequentially from top to bottom? Depending on which type of script you are deploying depends on how you reference it when creating the Application within Configuration Manager.

Let's explore each of these methods:

Standard Script (Top to bottom)

This type of script doesn't contain any functions and is read simply from top to bottom.

n example script looks like this:

Standard.ps1

```
#THis is an example of a standard script.

[int]$ValueA = 10
[int]$ValueB = 20
[Int]$Total = $ValueA + $ValueB

Write-Output "I have just done some incredible things with this script!"
Write-Output "Including adding up to values! The answer is: $Total"
```

It's easy to deploy, and your installation program command line would look like this:

powershell.exe -executionpolicy bypass -file .\MyScript.ps1

Script with Entry Point

It's similar to the Standard Script (Top to bottom) to deploy except that this type of script allows for more flexibility as you can write functions and pass parameters to them in the script entry point, or you could do all sorts of things after the function and then as the last line add the script entry point.

Script with Entry Point (No Parameters)

An example script with an entry point looks like this:

ScriptEntryPoint.ps1

```
Function Invoke-ApplicationInstall {

    # This function does some clever deployment here...

}


#Script entry point
Invoke-ApplicationInstall
```

To deploy it in ConfigMgr, your installation program command line would look like this:

powershell.exe -executionpolicy bypass -file MyDeploymentScript.ps1

Script with Entry Point (With Parameters)

Deploying a script with an entry point that includes function parameters is the same as you have just seen. A script with an entry point that uses parameters may look something like this:

ScriptEntryPointParam.ps1

```
Function Invoke-ApplicationInstall {

[CmdletBinding()]
    Param (
        [Parameter(Mandatory=$true)]
        [ValidateSet("x64","x86")]
        [String]
        $Architecture,
        [Parameter(Mandatory=$true)]
        [ValidateNotNullorEmpty()]
        [String]
        $Text
    )

    # This function does some clever deployment here...

}


#Script entry point with parameters
Invoke-ApplicationInstall -Architecture x64 -Text "Some important text"
```

To deploy this type of script within ConfigMgr I use this installation program command line:

powershell.exe -executionpolicy bypass -file MyDeploymentScript.ps1

Function

I try to avoid writing these types of scripts for ConfigMgr deployment as I'm lazy and like things to be easy. But you know, sometimes you just can't help it:

Function (No Parameters)

Let's say you have written a super-duper function called: Do-Something and it's contained in a script called: Function.ps1 Perhaps it looks like this:

Function.ps1

```
function Do-Something {

[CmdletBinding()]

    param ()


    Process {
        write-output "I'm doing something really cool"
    }
}
```

Your installation program command line in ConfigMgr would resemble the following to deploy it:

powershell.exe -executionpolicy bypass -command "& { . .\function.ps1; Do-Something}"

Calling the .ps1 file like so: *. .\thescript.ps1* is called "dot sourcing." Effectively, this means that the script is read into memory and can then simply call any function defined therein.

Watch Your Dots!

That's no typo: it's a full-stop, (period, for our American cousins), followed by a space followed by another full-stop, (period), and then a backslash!

Function Accepting Parameters

Much like you've just seen in deploying a function without parameters, it's just a case of adding the parameters to the same installation program command line.

This time let's pretend you have a function called: *Do-SomethingWithParameter* which accepts a single parameter named: *$Message* and then does something cool with that parameter in the function.

It could look like this, I suppose:

FunctionWithParam.ps1

```
function Do-SomethingWithParameter {

[CmdletBinding()]

    param (
        [parameter (Mandatory=$true)]
        [string]$Message
    )


    Process {
        write-output "I'm doing something really cool using the supplied parameter o\
f: $Message"
    }
}
```

And let's say you have saved your function in a PowerShell script named: *FunctionWithParam.ps1*

Let's deploy that script, call the function and pass the parameter: *-message* the value of: *"Important-Text!"*

In ConfigMgr, your installation program command line would look like this:

powershell.exe -executionpolicy bypass -command "& {..\FunctionWithParam.ps1; Do-SomethingWithParameter -Message "ImportantText!" }"

The Story so Far

Deploying your PowerShell script is easy. You have discovered that it's the same method for three types of deployment: Standard Script (Top to Bottom), Script with Entry Point (No Parameters) and Script with Entry Point. (With Parameters.) And two methods, (well, one really; we just add the parameters and values), if you are calling a function directly from your deployment command line: Function without parameters and Function Accepting Parameters.

It all just depends on how you write your scripts and what you are trying to achieve.

Part 7: Deployment Template

As I found myself deploying more and more applications using PowerShell, I realised that to speed things up I could write a common deployment template that included testing for the items I nearly always required. (Such as Microsoft Office 'bitness' or operating system architecture.) As well as the actual installation of an executable or msi based on the above criteria, I also wanted an easy method of optionally doing 'stuff' either before or after (or both) the actual install of the program; such as copying files or registering dll's etc. I also wanted basic logging which may help me out in the event of something going wrong!

I came up with a template that is exceptionally easy to use for multiple deployment scenarios and it uses the techniques demonstrated throughout this book.

How to Use

The PowerShell deployment template is called: *Invoke-ApplicationInstall.ps1*

As with all the code snippets and scripts used in throughout this book, you can find it on my Github repository:

https://github.com/ozthe2/ConfigMgr-Book-Code/tree/master/Code/Part7

I've kept it as simple as possible to use; there are no parameters and only a single optional switch: *-InstallBasedOnOfficeBitness*

Script Deployment Options - Do you want to install your application based on Microsoft Office 'bitness'? (i.e. 32-bit or 64-bit version of Microsoft Office.) or... - Do you want to install your application based on the operating system architecture? (i.e. 32-bit or 64-bit operating system.)

The only other decision you must make is whether you wish to add tasks pre-install, post-install or both.

I like to deploy my scripts as a "Script with Entry Point" and therefore all the following examples will reflect this method. (See Part 6 of this book for a refresher on how to do this.)

Let's cover all of this in a bit more detail now, so you can see how very easy it is to use this template and take your application deployments to the next level:

Deploying Based on Office 'Bitness'

Let's say that you want to install a different msi version depending on if the installed version of Microsoft Office is 32-bit or 64-bit. (As is often the case when deploying the Mimecast for Outlook plugin, for example.) To deploy the script this way, simply call the script like so: (Edit Line 109 in the script - your script entry point.)

Invoke-ApplicationInstall -InstallBasedOnOfficeBitness

Then add your exe or msi installation code at lines 74, 75 or 76, depending on your requirements:

```
65          # --- END PRE-INSTALL ---
66
67          if ($InstallBasedOnOfficeBitness) {
68              # Install the application depending on if Microsoft Office is 64bit or 32bit or not installed
69              # $True = Office is 32bit - Use this part to install 32bit applications
70              # False = Office is 64bit - Use this part to install 64bit applications
71              # Unknown = Office may not be installed.
72
73              switch ($Obj.OfficeIs32Bit) {
74                  $true {"Install a 32-bit application here"}
75                  $false {"Install a 64-bit application here"}
76                  'Unknown' {"Office not detected - do what you want here"}
77              }
78          }
```

Install based on 32-bit Microsoft Office

As you can see in the above screenshot, if you want to install a msi based on the installed version if Microsoft Office being 32-bit, then add your msiexec call at line 74.

Install based on 64-bit Microsoft Office

Again, using the above screenshot as a reference, if you want to install a msi based on the installed version of Microsoft Office being 64-bit, then add your msiexec call at line 75.

If Microsoft Office was not detected, then optionally you can do something in line 76.

Note that you do not have to populate all three lines (74, 75 and 76). If you are only deploying a msi based on Microsoft Office being 32-bit, then simply populate line 74 and leave the others exactly as they are.

Example

Here is an example of deploying either the 32-bit or 64-bit Mimecast plugin based on Office bitness:

```
58
59      if ($InstallBasedOnOfficeBitness) {
60          # Install the application depending on if Microsoft Office is 64bit or 32bit or not installed
61          # $True = Office is 32bit - Use this part to install 32bit applications
62          # False = Office is 64bit - Use this part to install 64bit applications
63          # Unknown = Office may not be installed.
64
65          switch ($Obj.OfficeIs32Bit) {
66              $true {start-process -FilePath "$ENV:SystemRoot\System32\msiexec.exe" -ArgumentList "/i ""$workingDir\Mimecast for Outlook 7.6.0.26320 (32 bit).msi"" /qn" -NoNewWindow -Wait}
67              $false {start-process -FilePath "$ENV:SystemRoot\System32\msiexec.exe" -ArgumentList "/i ""$workingDir\Mimecast for Outlook 7.6.0.26320 (64 bit).msi"" /qn" -NoNewWindow -Wait}
68              'Unknown' {""}
69          }
70      }
```

Note!

Note the use of double quotation marks due to the msi having spaces in the name.

Deploying Based on Operating System Architecture

You may prefer to install based on operating system architecture. This is the default option when calling the script and your entry point, (on line 109), simply calls the function like so:

Invoke-ApplicationInstall

Install based on 64-bit Operating System

As you can see in the screenshot below, if you want to install a msi based on the operating system being 64-bit, then add your msiexec call at line 85.

```
79          else {
80              # Installs a 64bt or 32bit application depending on the OS Architecture.
81              # $True = The OS Architecture is 64-bit - Use this part to install the 64-bit app
82              # default = The OS is 32-Bit - Use this part to install the 32-bit application.
83
84              switch ($obj.OSIs64Bit) {
85                  $True {start-process -FilePath "$ENV:SystemRoot\System32\msiexec.exe" -Argume
86                  default {start-process -FilePath "$ENV:SystemRoot\System32\msiexec.exe" -Argu
87              }
88          }
```

Install based on 32-bit Operating System

As you can see in the above screenshot, if you want to install a msi based on the operating system being 32-bit, then add your msiexec call at line 86.

Example Here is an example of deploying either the 32-bit or 64-bit version of 7-Zip based on operating system architecture:

```
79          else {
80              # Installs a 64bt or 32bit application depending on the OS Architecture.
81              # $True = The OS Architecture is 64-bit - Use this part to install the 64-bit application.
82              # default = The OS is 32-Bit - Use this part to install the 32-bit application.
83
84              switch ($obj.OSIs64Bit) {
85                  $True {start-process -FilePath "$ENV:SystemRoot\System32\msiexec.exe" `
86                  -ArgumentList "/i ""$workingDir\7z1900-x64.msi"" /qn" -NoNewWindow -Wait}
87                  default {start-process -FilePath "$ENV:SystemRoot\System32\msiexec.exe" `
88                  -ArgumentList "/i ""$workingDir\7z1900.msi"" /qn" -NoNewWindow -Wait}
89              }
90          }
```

Pre-Deployment Tasks

As part of your deployment, there may also be a requirement to do something before the installation of the msi or exe takes place.

To do this, simply add your required code inside the function: *Invoke-PreInstallation*

This function can be found starting on line 1 of the deployment template:

```
1  Function Invoke-PreInstallation {
2      # Anything you do here will occur **before** the installation...
3      # Perhaps copy some files, register some DLL's or anything else you can think of!
4  }
5
```

Post-Deployment Tasks

Let's say you wish to do something after the deployment takes place: all that is required is to add your code to the function: *Invoke-PostInstallation*

This function can be found starting on line 6:

```
6  function Invoke-PostInstallation {
7      # Anything you do here will occur **after** the installation...
8      # Perhaps copy some files, register some DLL's or anything else you can think of!
9  }
10
```

Logging

The deployment template also contains some basic logging. The log file is not permanent as I have chosen to place the log file in the same directory that ConfigMgr has used for the application

deployment. (You will find the log somewhere in *%windir%\ccmcache* - sort the directories by date to help you find it.)

The log file is named: *Invoke-ApplicationInstall.log*

The log contains the following information: - The current logged in username. - The detected Microsoft Office version. (32-bit or 64-bit.) - The detected Operating System Architecture. (32-bit or 64-bit.)

You can add or modify what is or isn't logged by modifying or editing between lines 99 to 104:

```
99          # Write what the discovered object values are to a log.
100         # This may help in troubleshooting.
101         # Feel free to expand this section to include anything else that you want logged.
102         "Logged In User: $($Obj.CurrentUser)" | out-file -FilePath ".\Invoke-ApplicationInstall.lo
103         "Office Version is 32Bit: $($Obj.OfficeIs32Bit)" | out-file -FilePath ".\Invoke-Applicatio
104         "Operating System is 64Bit: $($Obj.OSis64Bit)" | out-file -FilePath ".\Invoke-Application
105     }
```

How to Call the Template

Call the template within your ConfigMgr application like this:

powershell.exe -executionpolicy bypass -file .\Invoke-ApplicationInstall.ps1

Lead by Example

There's a good step-by-step example of using the deployment template to deploy EMC SourceOne in Bonus Chapter 2 at the end of this book. It's worth a read through even if you are not going to deploy SourceOne.

The Story so Far

You have seen how easy it is to use a deployment template for all your PowerShell deployments. Now that you have this template under your belt, the sky is the limit!

I'm going to be honest with you, I use this template in nine-out-of-ten of my deployments - it's that easy.

Feel free to duplicate my template, re-use, edit or use it as the basis for your template. You can also fork it from GitHub and submit pull requests should that take your fancy too.

Part 8: Useful Code Snippets

What follows are some useful bits of code that may help you in your deployment scripts. Some of these code snippets are useful for any pre or post-deployment options, such as registering a dll or copying files.

Detect Office 'Bitness'

If you are using my deployment template (see part 7 of this book), then you will already have seen this code.

TestOfficeBitness.ps1

```
$OfficePath = 'HKLM:\Software\Microsoft\Office'
$OfficeVersions = @('14.0','15.0','16.0')

foreach ($Version in $OfficeVersions) {
    try {
        Set-Location "$OfficePath\$Version\Outlook" -ea Stop -ev x
        $LocationSet = $true
        break
    } catch {
        $LocationSet = $false
    }
}

if ($locationSet) {
    #Check for bitness then check correct file version
    switch (Get-ItemPropertyValue -Name "Bitness") {
        "x86" { $Bitness = '32-Bit' }
        "x64" { $Bitness = '64-Bit' }
        default { $Bitness = 'Unknown' }
    }
}

```

```
23 if (!($Bitness)) {
24     $Bitness = 'Not Installed'
25 }
```

The code will set a variable named: $Is32Bit to one of three values:

1. $True - Microsoft Office is a 32-Bit installation.
2. $False = Microsoft Office is a 64-Bit installation.
3. "Unknown" - It could not be determined if Office is 32-bit, 64-Bit or if it is installed.

Watch Out!

This detection relies on Microsoft Outlook being installed as part of your Office installation. If Outlook has not been installed, then this code will not work.

Detect Operating System Architecture

A simple one-liner: the operating system is queried and the return value (either '64-Bit' or '32-bit'), is returned and stored in the variable: *$OSArchitecture*

GetOSBitness.ps1

```
1 $OSArchitecture = (Get-CimInstance -ClassName CIM_OperatingSystem).OSArchitecture
```

Obtaining the Current Logged in User Name

Even though scripts are deployed using ConfigMgr running under the system account, it is still possible to query who the current logged in user is:

LoggedInUser.ps1

```
1 $LoggedInUser = (Get-CimInstance -ClassName CIM_ComputerSystem).username | Split-Pa\
2 th -Leaf
```

In the above code, the currently logged in user is stored in the variable: *$LoggedInUser*

Better to Use $ENV:?

Not in this case. The more observant among you may have noticed that there is a *$ENV:UserName* PowerShell environment variable. You cannot use this to detect the current logged in user if you are deploying the script under the system context. The reason being is that the PowerShell environment variable returns the account being used that is running the PowerShell script. It's a shame as it would have made life a bit easier and I'm all for that

Copying Files

Copying files is something you may wish to achieve as part of the pre or post-application deployment.

Lazy 'Single Root Directory' Method

When I do it, I dump everything, (the deployment script itself, msi, files that need copying, dll's etc) in a root directory using my lazy, "Everything in a Root Directory", method. If you have learned anything about me so far from reading this book, then you know I'm lazy.

Let's pretend you are also lazy, and you need to copy some files to the *\Windows\Wallpaper* directory and another file to *Windows\Ini* directory.

Hang on…haven't we dumped everything in the root? So, we have the msi files, the wallpaper.jpg, the Important.ini as well as the *Invoke-ApplicationInstall.ps1* deployment script all in one root directory as per the screenshot below:

Correct.

The PowerShell cmdlet: *Copy-Item* along with *–Include* switch will allow you to achieve the aim.

Let's see some examples:

Example 1 - Copy all jpg files to \Windows\Wallpaper

CopyItemJpg.ps1

```
1  Copy-Item -Path '*.*' -Destination "Env:SystemRoot\Wallpaper" -Include '*.jpg' -Err\
2  orAction SilentlyContinue
```

As you can see, it's possible to filter and only copy jpg files by adding the parameter: -Include '*.jpg'

Example 2 - Copy all ini files to \Windows\Ini

CopyItemIni.ps1

```
Copy-Item -Path '*.*' -Destination 'C:\ohtemp\destination' -include '*.ini'
```

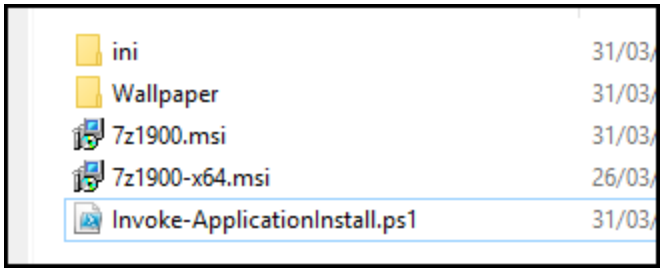
As you can see from the screenshot above, you can just as easily filter to only copy ini files by adding: -Include '*.ini'

Did you Know?

You can also specify to copy a specific file. For example, if you had more than one .ini file but you only wanted copy: *Important.ini* then you could do this: *Copy-Item -Path '.' -Destination "$Env:SystemRoot\ini" -include 'Important.ini'*

Structured Method

Of course, if you aren't lazy like me then you may have a directory structure that was more like this:



In the above screenshot, the .ini files are in the ini subdirectory and the .jpg are neatly placed in the Wallpaper subdirectory.

To use *Copy-Item*, simply change the path in the cmdlet to reflect the structure you are referencing:

Example 3 - Copy all jpg files from \Wallpaper to \Windows\Wallpaper

CopyItemJpg2.ps1

```
Copy-Item -Path '.\Wallpaper\*.*' -Destination "Env:SystemRoot\Wallpaper" -ErrorAct\
ion SilentlyContinue
```

Example 3 - Copy all ini files from \ini to \WindowsIni

CopyItemIni2

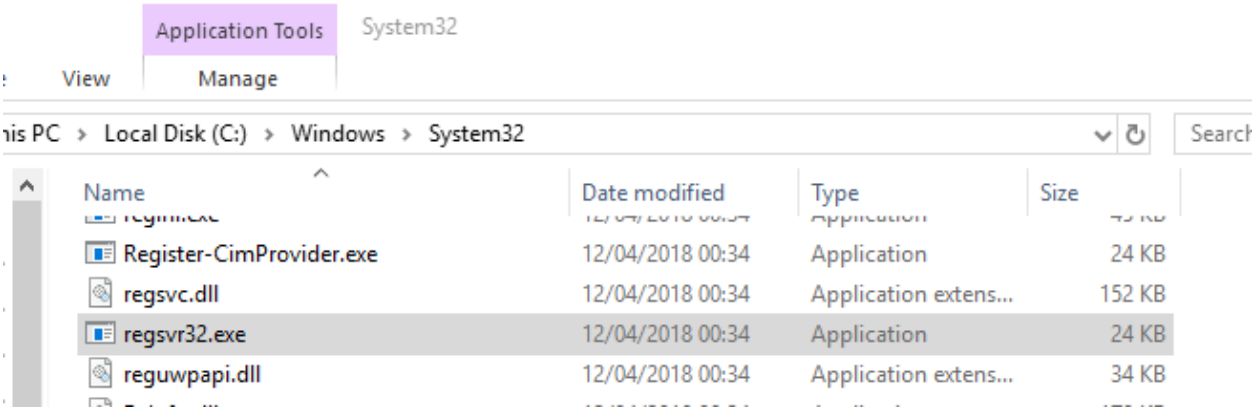
```
Copy-Item -Path '.\ini\*.*' -Destination "Env:SystemRoot\Ini" -ErrorAction Silently\
Continue
```

No Include

Note: there is no need for the -Include switch this time as the 'lazy' method is not being used (files are in a specific subdirectory by themselves.) However, the option to either –Include or –Exclude can still be used if required. (For example, copying a specific file and not all files from the subdirectory.)

Register \ Unregister DLL's

Registering or unregistering a dll uses the built-in *Regsvr32.exe* found in *\Windows\system32*



regsvr32

You can read the help for this command by reading the official Microsoft documentation here:

https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/regsvr32

If you've followed the advice in this book, then your deployment script will already contain the two lines of code necessary for this to work correctly.

Those two lines of are:

SetLocation.ps1

```
1 $WorkingDir = Get-Location
2
3 Set-Location $WorkingDir
```

You should place these lines of code somewhere near the top of your script. Alternatively, you can use ready to go Deployment Template: *Invoke-ApplicationInstall.ps1* from Part7 of this book which already contains this.

(If you need a refresher on why this code is added to the deployment script then have another read of Part 4 of this book.)

If you're all set (pun intended!) then all that's left to do is to place your .dll file in the same root folder as your deployment script and use the following code to either register or unregister the dll:

Register a DLL

Simply edit Line 1 ($DLLFileName) to reflect the name of your dll file. In this example, the dll to be registered is named: *MyDLL.dll*

RegisterDLL.ps1

```
1 $DLLFileName = "MyDLL.dll"
2 Start-Process -FilePath "$env:SystemRoot\System32\regsvr32.exe" -ArgumentList "/S ""\
3 $WorkingDir\$DLLFileName""" -Wait -NoNewWindow
```

Unegister a DLL

Unregistering a DLL file simply adds the */U* parameter as per the documentation:

https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/regsvr32

UnregisterDLL.ps1

```
1 $DLLFileName = "MyDLL.dll"
2 Start-Process -FilePath "$env:SystemRoot\System32\regsvr32.exe" -ArgumentList "/S /U\
3  ""$WorkingDir\$DLLFileName""" -Wait -NoNewWindow
```

Use the Template!

If you use the deployment template, you could register or unregister a DLL by placing the code in either the pre-deployment or post-deployment functions. (See Part 7 of the book for a reminder.)

Part 9: Real-World Examples

This part is where I will add real-world code examples - these are deployment scripts that I have written and used myself in production.

Most of these scripts were created before I wrote the deployment template, and as time goes on, I will eventually migrate the scripts to the template. For now, though, they are provided ‘as-is’ and the links point to their current locations in my various GitHub repo’s. It is still interesting to see how one can go about deploying various programs using PowerShell scripting techniques though.

You may also need to supply your own detection rules - be that by conventional means or via PowerShell, that’s up to you.

Ready to go Scripts

Although these are “ready to go” scripts, and all of them I have written and used myself in production, I would strongly advise that you test them before using in your environment so that you can ensure that you are obtaining the expected results. The usual disclaimer applies, and you use these scripts at your own risk.

Adobe Reader

This script will deploy Adobe Reader along with a transform file and then patch the install by applying the msp file.

Get the script here:

https://github.com/ozthe2/Powershell/blob/master/SCCM/AdobeReaderVersionManagement.ps1

How to use

This script is read top to bottom, so deployment is a breeze…

powershell.exe -executionpolicy bypass -file .\AdobeReaderVersionManagement.ps1

Java

Urgh - Java - I’m so glad it’s going. If you need to uninstall all found instances of Java and then install a new one, then this script is just the ticket:

https://github.com/ozthe2/Powershell/blob/master/SCCM/JavaVersionManagement.ps1

How to use

I’ve already posted step-by-step instructions on how to use this script on my blog, so I won’t repeat it here:

https://fearthepanda.com/configmgr/2019/01/02/ConfigMgr-How-to-Deploy-Java

Firefox

This script uninstalls all versions of Firefox that it can find (both 32-bit and 64-bit) and then install the new supplied version. In my workplace, we have offices in many countries and this script will install the correct language of Firefox based on the locale of the computer that the script runs on.

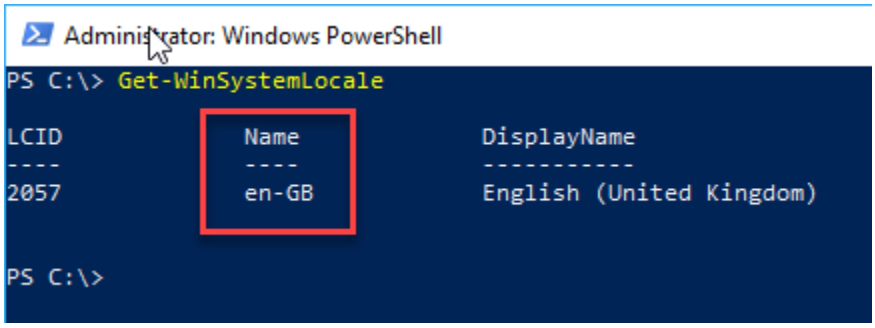
The script can be found here:

https://github.com/ozthe2/Powershell/blob/master/SCCM/FirefoxVersionManagement.ps1

How to use

To add more locales to the switch statement (Lines 32 - 36 of the script) simply run the following in a PowerShell prompt on the computer in question:

Get-WinSystemLocale



Get-WinSystemLocale

You can then add additional lines to the switch statement. (Copy and paste one of the existing lines of code in the switch statement and modify it to reflect your changes.)

For the Installation Program command line in the ConfigMgr Application that you create, use the following:

powershell.exe -executionpolicy bypass -file .\FirefoxVersionManagement.ps1

Mimecast

This script will detect the bitness of Microsoft Office and install the correct version of Mimecast accordingly. Watch out though as the script will force close Outlook as part of the installation and this may or may not be acceptable in your environment.

Find the script and the detection rule here:

https://github.com/ozthe2/Powershell/tree/master/SCCM/Mimecast

How to use

This script uses an entry point to call the function so calling the script is easy. In your Installation program command line, use the following:

powershell.exe -executionpolicy bypass -file .\Install-Mimecast.ps1

Fusion Excel Connect Client

This deployment uses the deployment template.

It will install either a 32-bit or 64-bit version of the FUSION Excel Connect Client depending on the installed Microsoft Office 'bitness.' There's also a PowerShell detection rule that you can copy \ paste.

Find the deployment script and detection rule here:

https://github.com/ozthe2/ConfigMgr-Book-Code/tree/master/Code/Part9/Fusion

How to use

This deployment uses the deployment template; therefore, it has a script entry point and you use the following for your installation program command line in your ConfigMgr application:

powershell.exe -executionpolicy bypass -file .\Invoke-ApplicationInstall.ps1

Tips

> **Gremlins (1984)**
>
> The most important rule, the rule you can never forget, no matter how much he cries, no matter how much he begs, never feed him after midnight.

Tip #1

Where possible, try and use my deployment template: *Invoke-ApplicationInstall.ps1*. It's been tested hundreds of times on production deployments. (Sure, go ahead and customise it though, if you want to.)

Tip #2

Always test your deployment script by running it manually first.

Here's how I manually test:

1. Create a directory locally on my computer: *C:\DeploymentTest*
2. Copy the deployment template and all of the relevant files, eg the msi to deploy and \ or the files that will be copied pre or post-deployment to *C:\DeploymentTest*
3. Open the PowerShell ISE as administrator (elevated) and open the deployment template from *C:\DeploymentTest*. (If you haven't renamed it then it is named *Invoke-ApplicationInstall.ps1*)
4. This part is important and is only required for testing: in the PowerShell ISE, ensure that your current location is set to *C:\DeploymentTest* (or wherever you have placed all of your source files and deployment script) by typing: *Set-Location C:\DeploymentTest* in the bottom pane of the ISE (The blue PowerShell window)
5. Now run the script and make sure that the application installs and that your pre and post-installation options have all worked. (If you included any.)
6. If required, uninstall the application that your script just deployed, adjust your script, and run it again.
7. Once you are happy that your script works as expected then you are ready to create the SCCM application.

Tip #3

Near the start of your deployment script, always use:

$WorkingDir = Get-Location followed by *Set-Location $WorkingDir*

This will ensure that your script can easily reference all files and executables, as well as make scripting easier.

Tip #4

When using the *Start-Process* parameter: *-ArgumentList*, always add the *$WorkingDir* variable to your path. This ensures that the msi will be found correctly and reduce script runtime errors.

In the event of using subdirectories, add them into your path:

"$WorkingDirMSIFilesMyProduct.msi

For example, although you can do this…

Tip2A.ps1

```
1 Start-Process -Filepath "$ENV:SystemRoot\System32\msiexec.exe" -ArgumentList "/i 7z\
2 1900-x64.msi /qn /norestart"
```

…it's far better to do this:

Tip2B.ps1

```
1 Start-Process -Filepath "$ENV:SystemRoot\System32\msiexec.exe" -ArgumentList "/i $W\
2 orkingDir\7z1900-x64.msi /qn /norestart"
```

Tip #5

Don't bother with a complex file structure that uses subdirectories for your deployment script, msi, exe's or files unless it's 100% necessary. Save yourself a lot of pain by adopting my "lazy" method and dump everything in a single root folder.

It may not look pretty but you'll thank me for it later.

Bonus Chapter 1

A Step-by-Step Guide to Deploying a CCMCache Resize

Resizing the CCMCache seems to be a common item that everyone wants to achieve.

Start by downloading the PowerShell script and Detection rule that I wrote from here.[3]

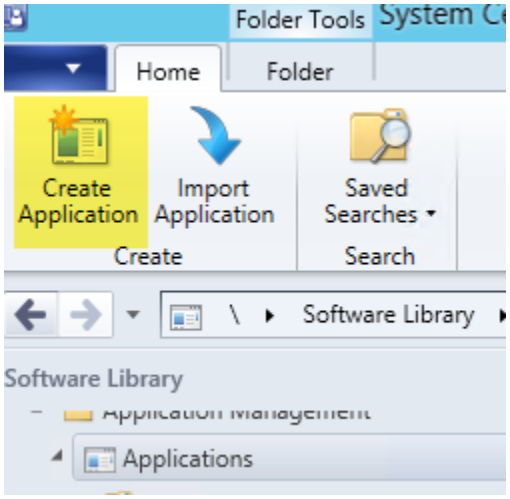
Once you've done that you will need to store the *Set-CCMCacheSize.ps1* PowerShell file in its own directory somewhere where you would normally store all of your source files used for ConfigMgr deployments.

The Scenario

In this example, we are going to resize the CCMCache to 6GB on our client computers.

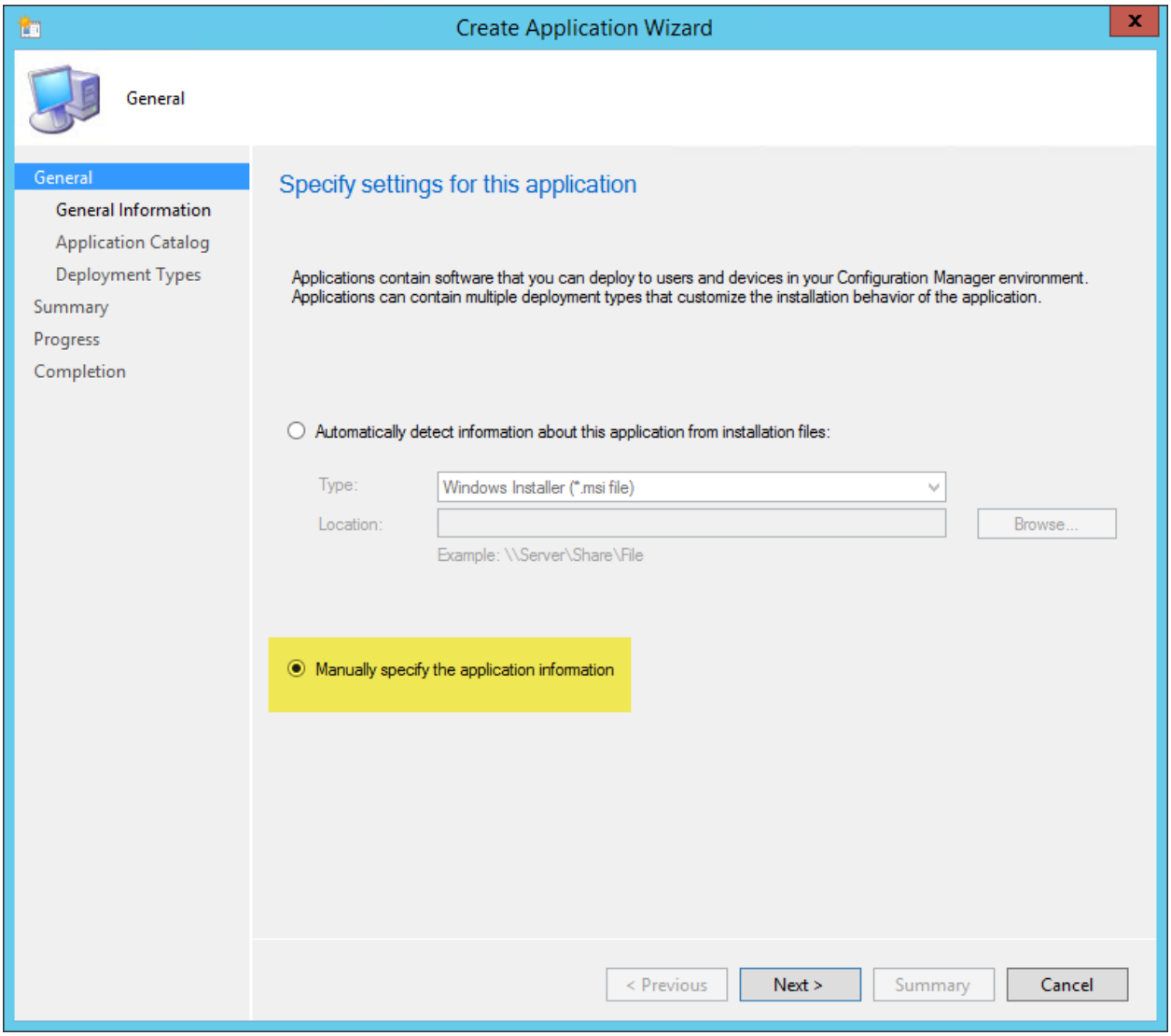
Fire up your Configuration Manager Console and let's begin…

Start by creating a new application:



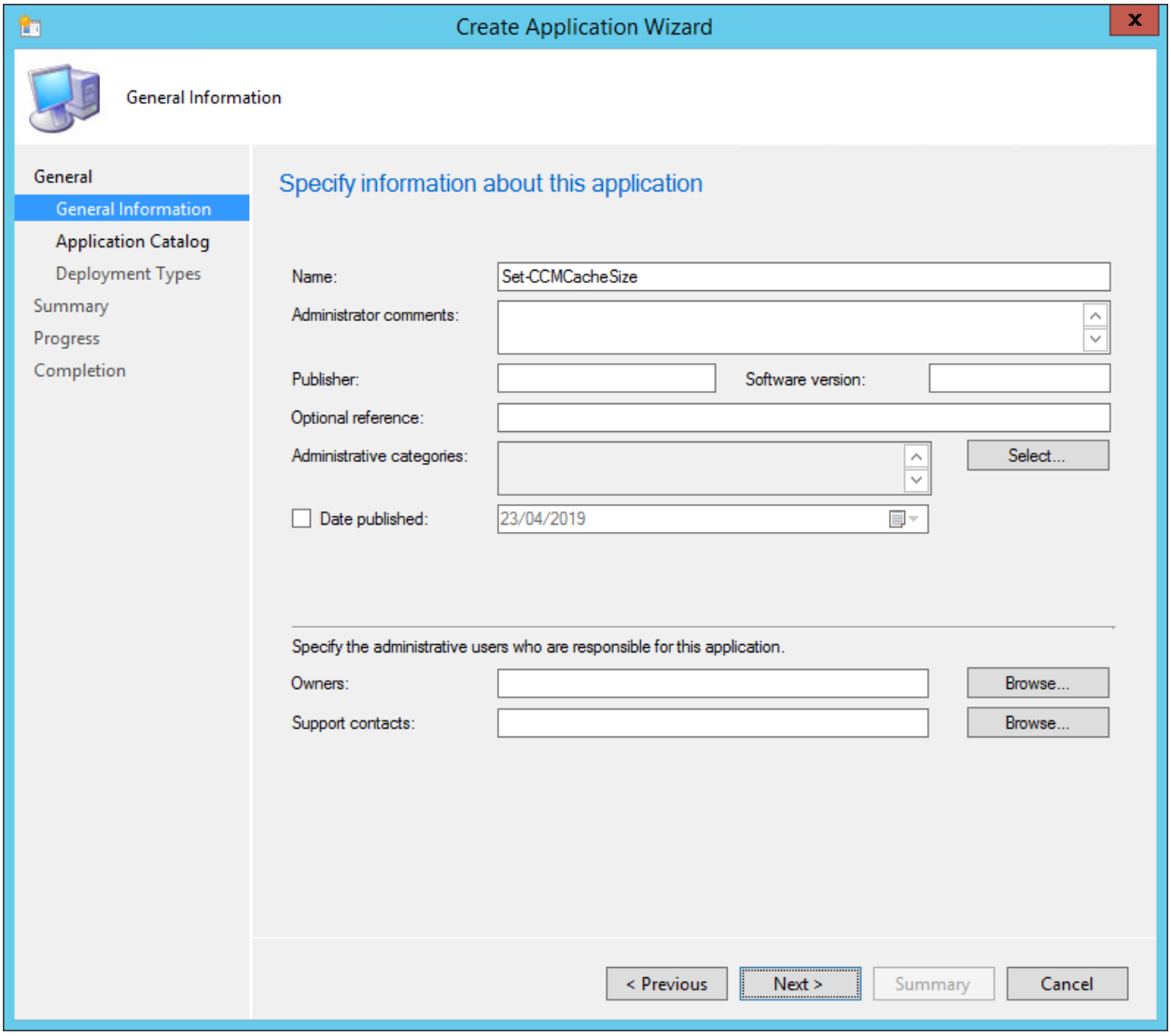
[3] https://github.com/ozthe2/ConfigMgr-Book-Code/tree/master/Code/Part9/SetCCMCache

Select to “Manually specify the application information”:

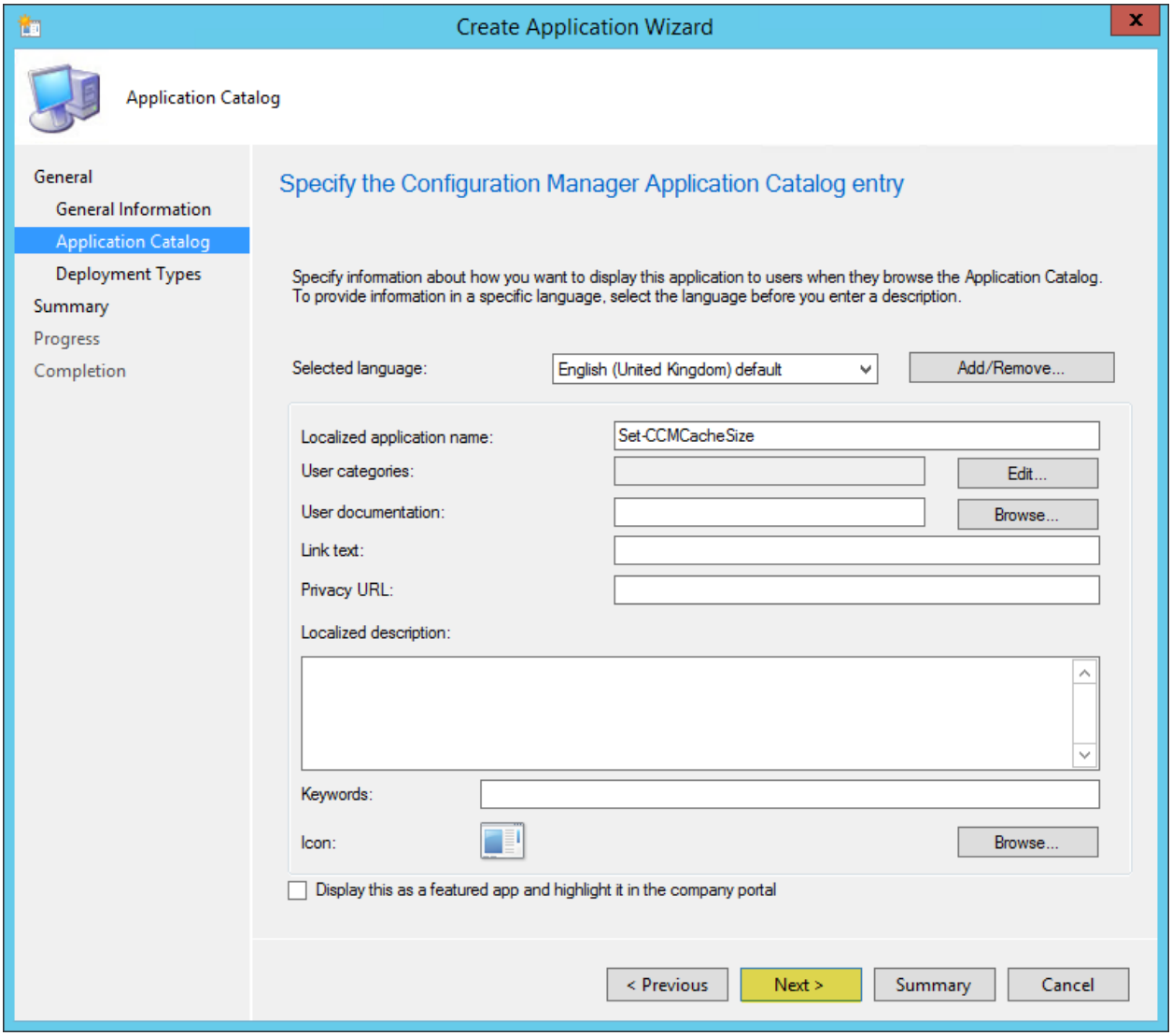


Give your application a name. I chose “Set-CCMCacheSize” as this is what the name of the function is.

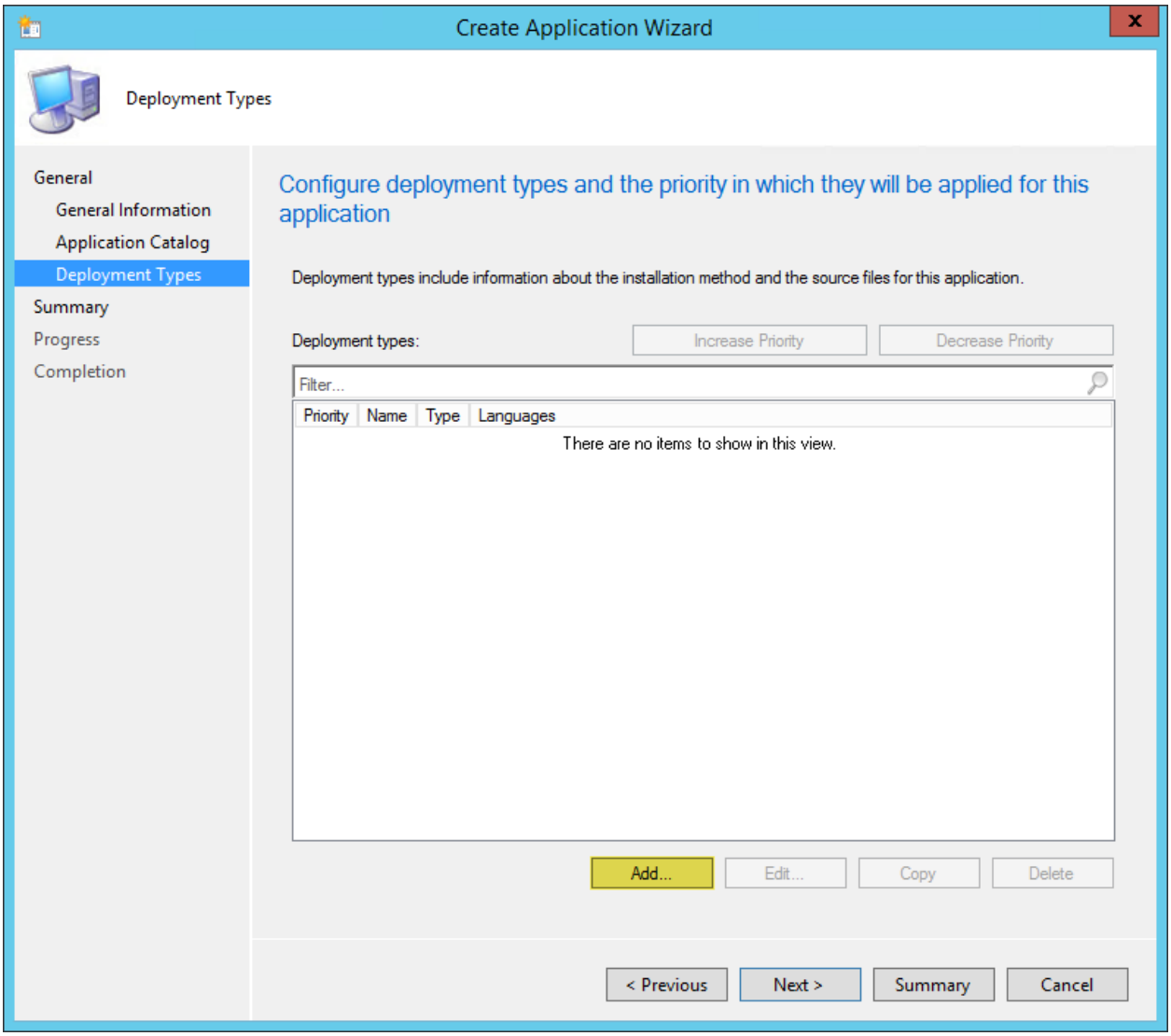
Note that I never added the size that we are setting it to in the title ie I did not called it: *Set-CCMCacheSize (6GB)* More on why later…:



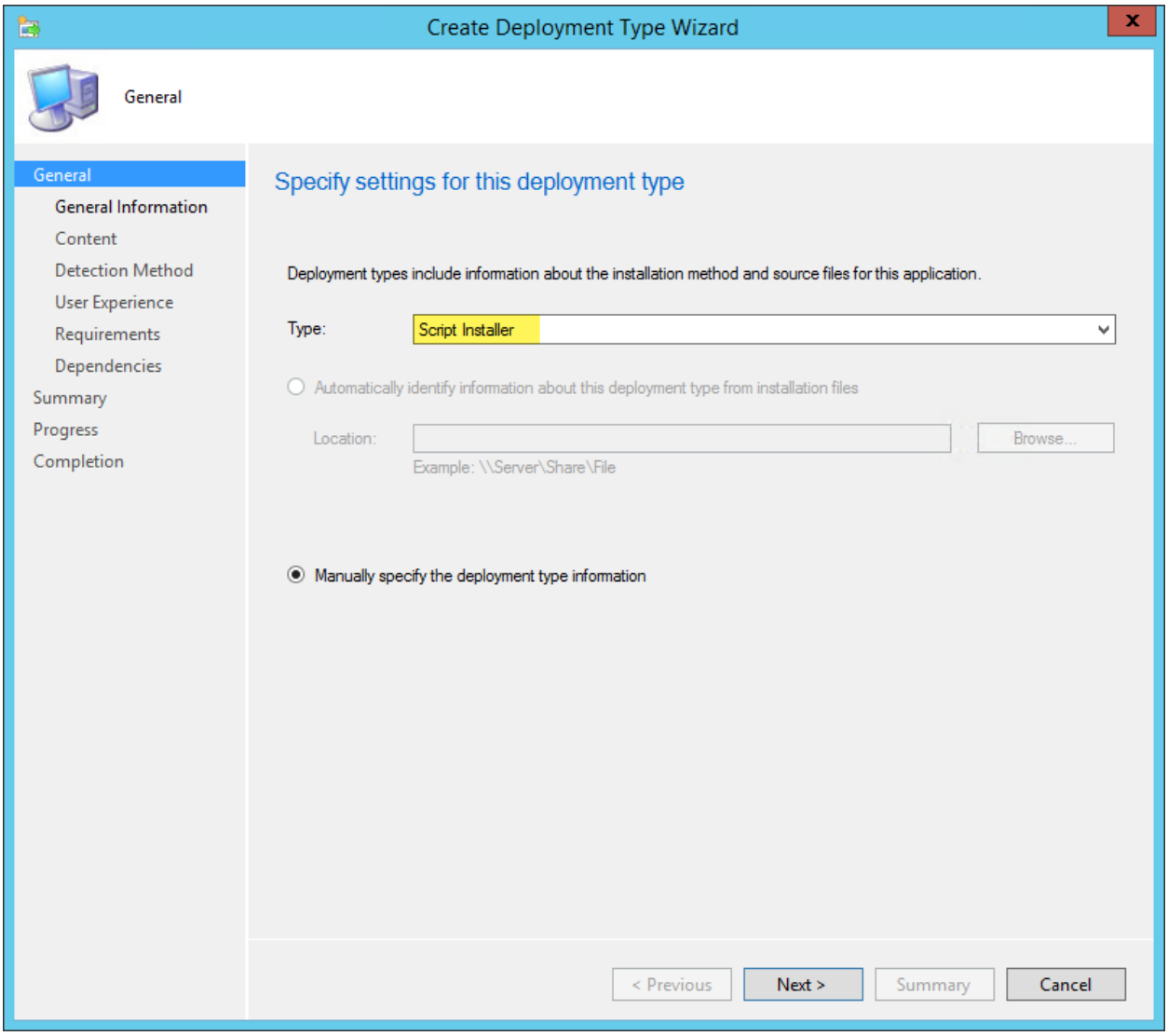
I will be deploying this application to computers and it will be force run. It's not going in the Application Catalog so I just clicked next here:



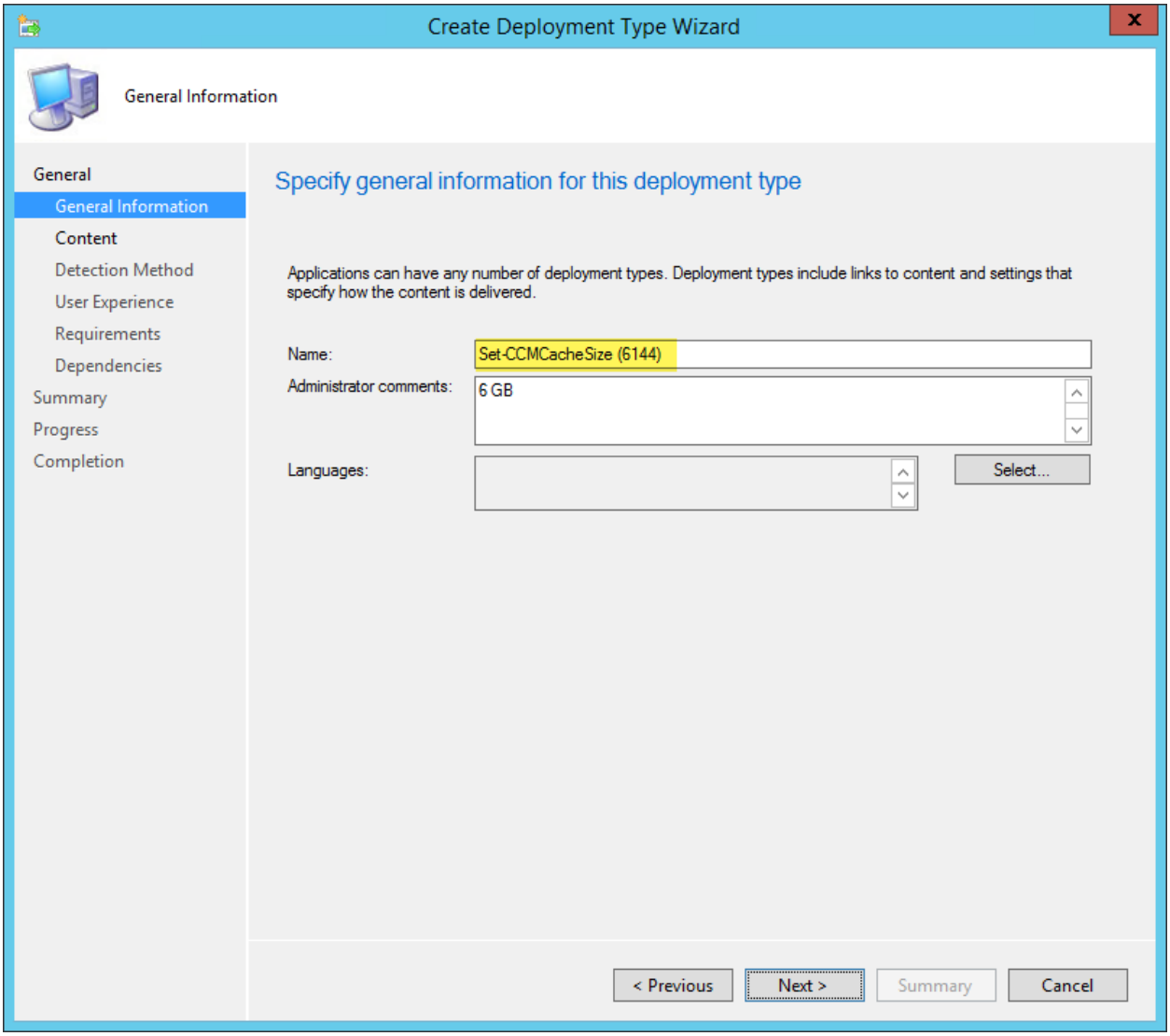
We need to add a deployment type now, so click the Add button:



Select *Script Installer*:

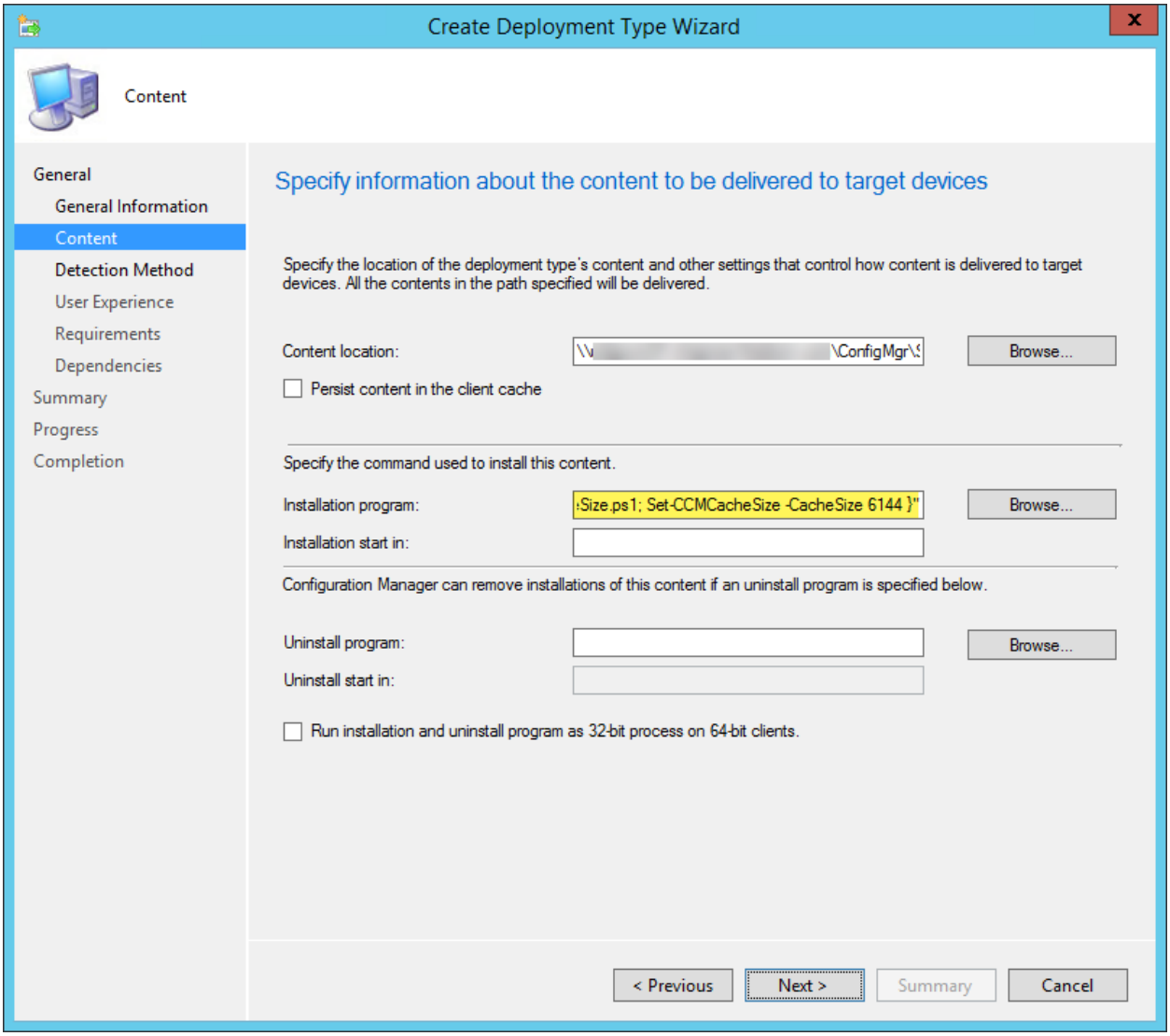


Now name the deployment type. It is here that I added the size that this deployment resizes the cache to. With that in mind, I named my deployment type: *Set-CCMCacheSize (6144)* and just to be clear, added *6GB* in the Administrator comments. I added the size to the name of the Deployment Type as I may have multiple deployment types that resize the cache to different sizes depending on the criteria:

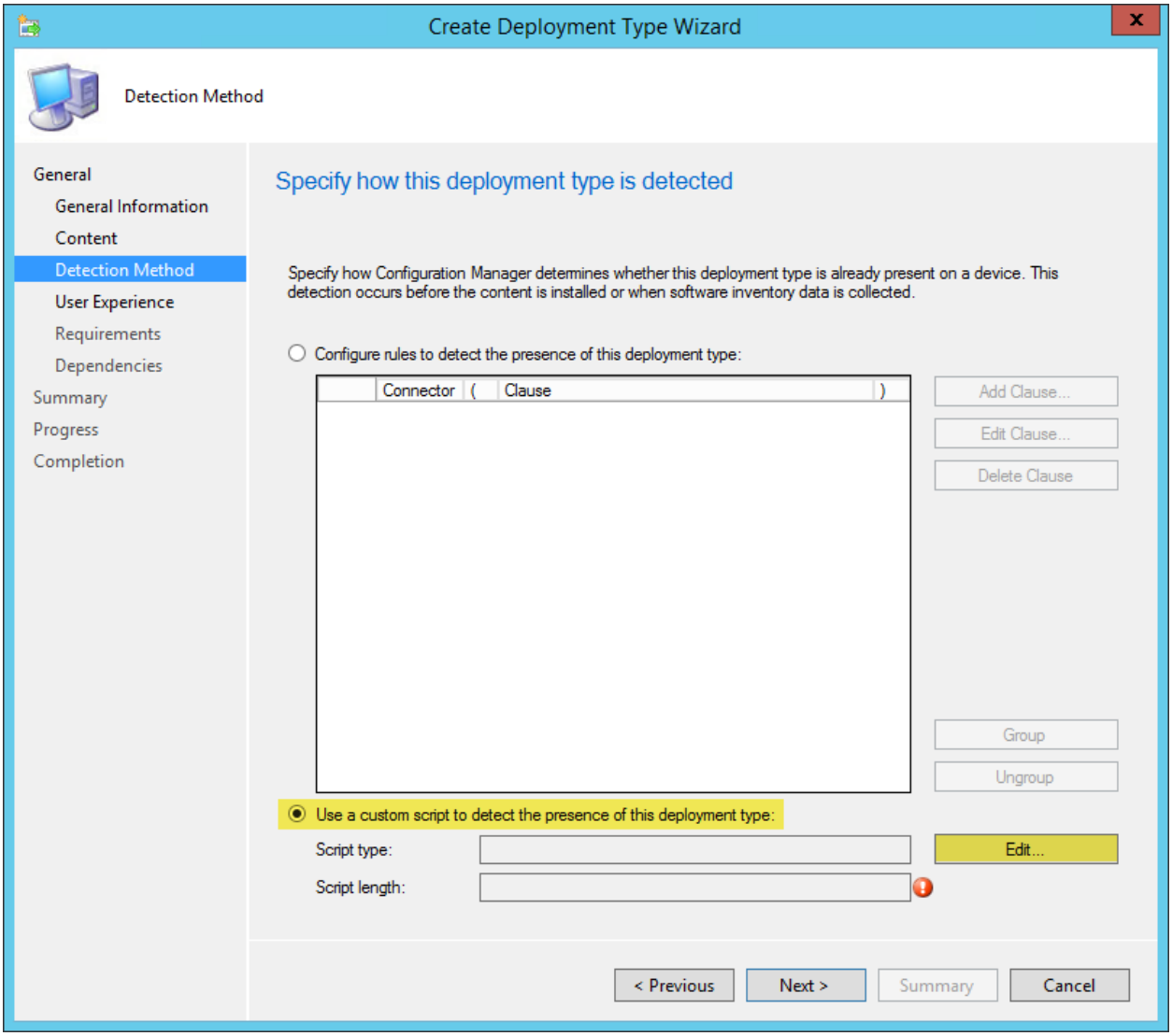


For the content location, browse to where you stored the Set-CCMCacheSize.ps1 script. Then for the installation program, we will call our function contained in the script using the parameter to signify that we want a 6GB cache resize:

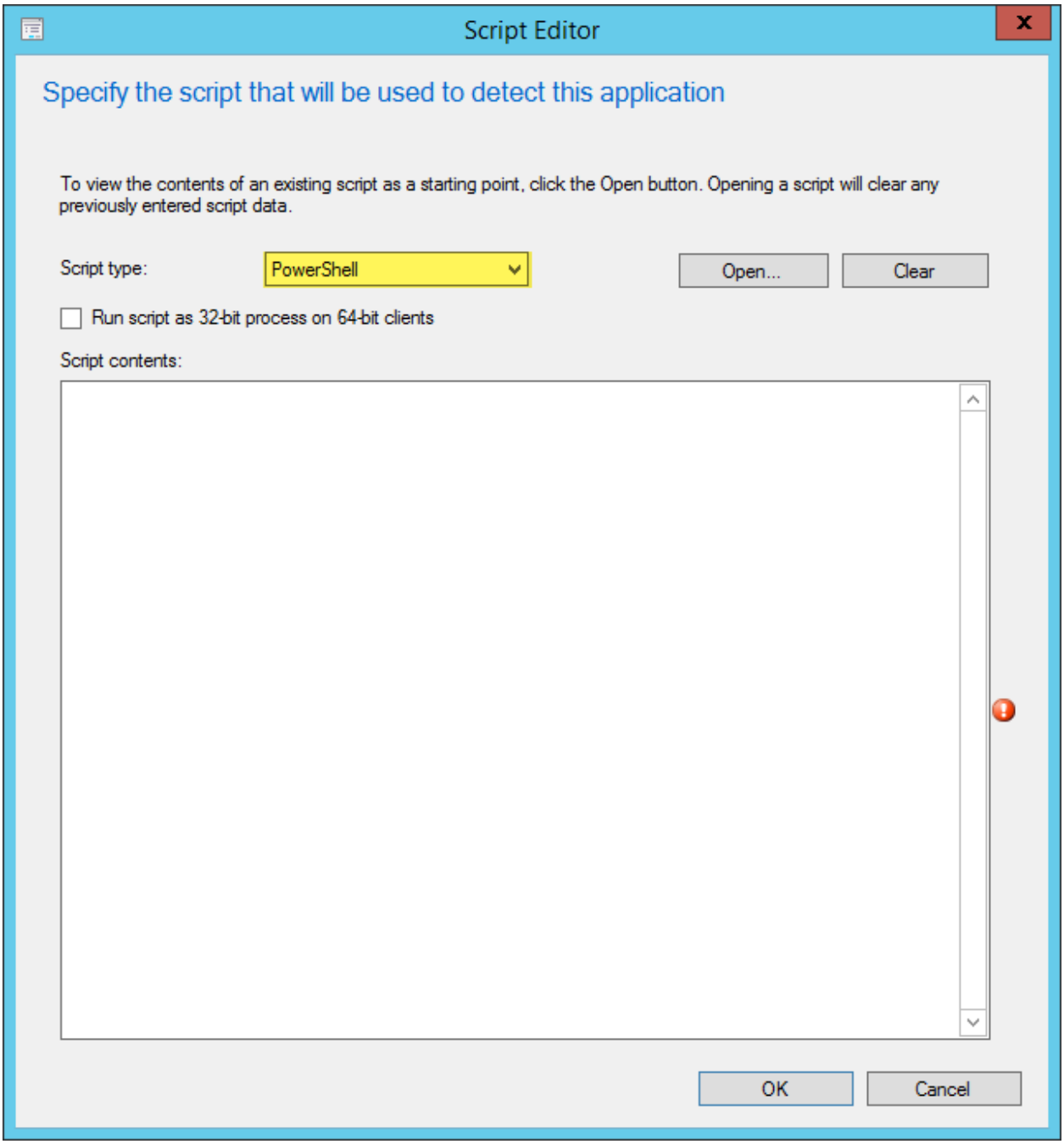
```
powershell.exe -executionpolicy bypass -command "& { . .\Set-CCMCacheSize.ps1; Set-CCMCacheSize -CacheSize 6144 }"
```



We will use PowerShell for the detection clause so select *Use a custom script to detect the presence of this deployment type* and click the Edit button:



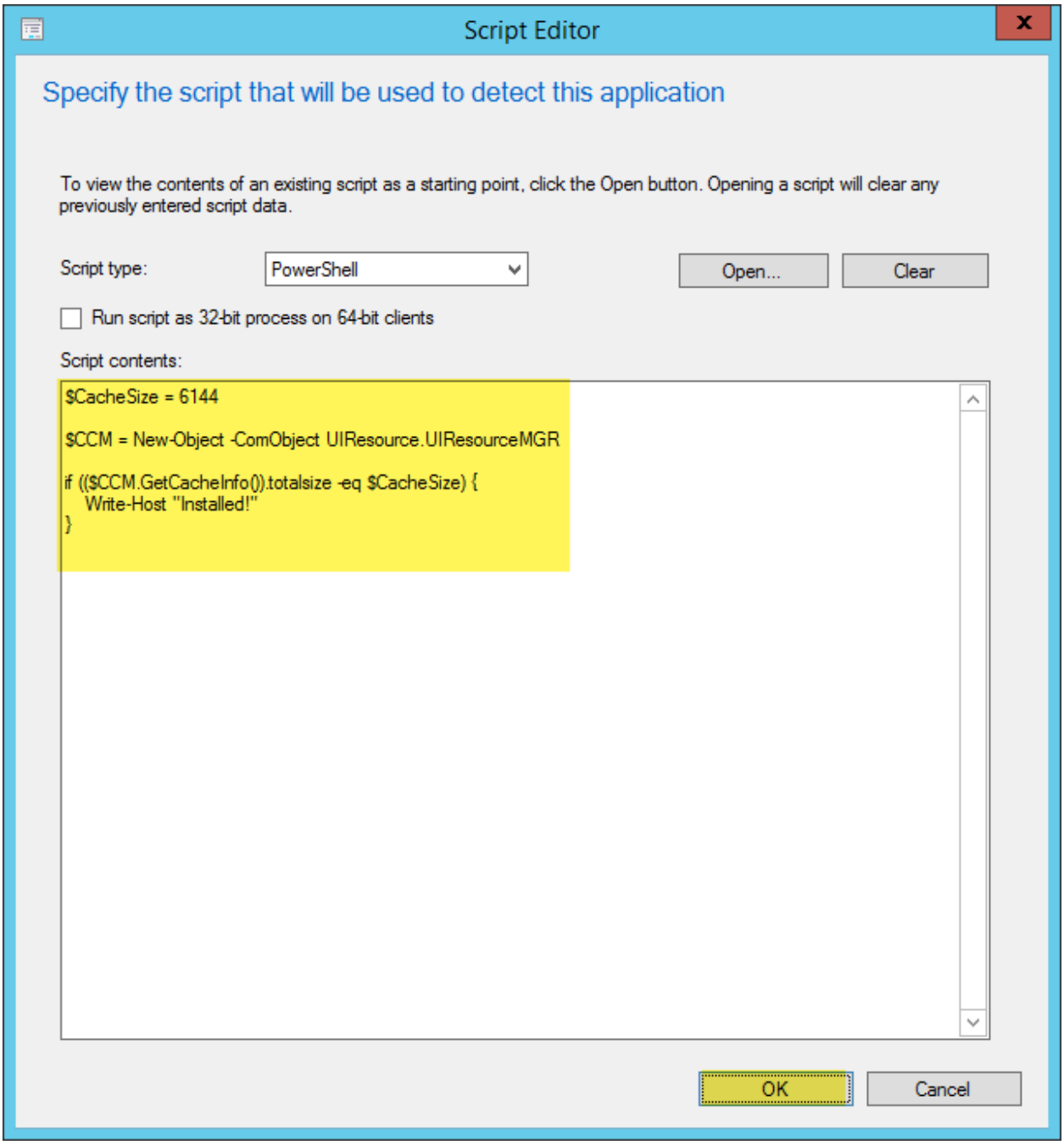
Change the script type to PowerShell:



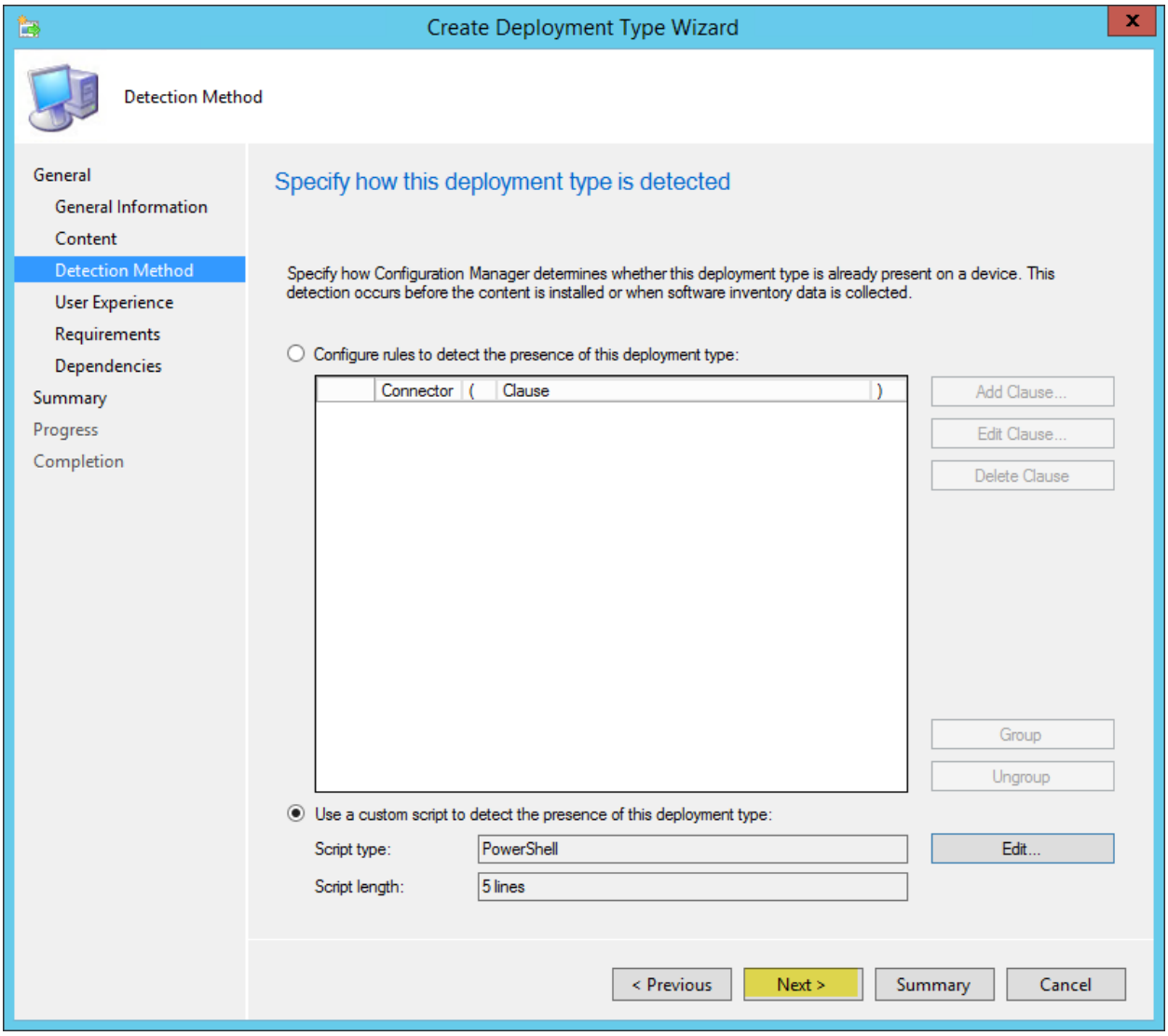
Copy the contents of the *DetectionRule.ps1* that you downloaded at the start of all this. Change line 1 to read `$CacheSize = 6144` as we are setting the cache to 6GB. (6GB x 1024MB = 6144MB)

This value needs to match the value that we set earlier in the `-CacheSize` function parameter.

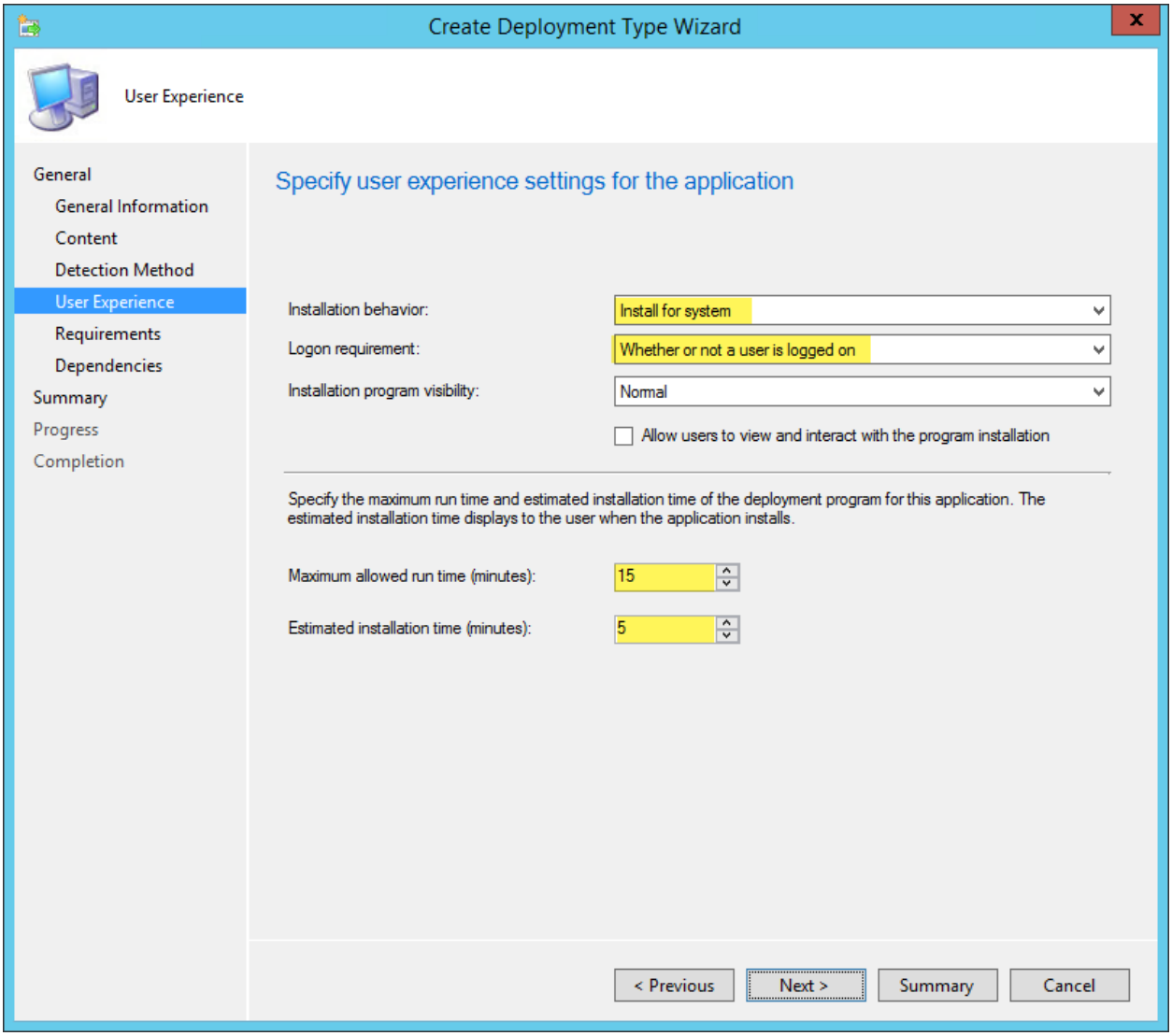
This detection rule actually queries the cache for its size value so we can be certain that the resize has been successful. (Or not!):



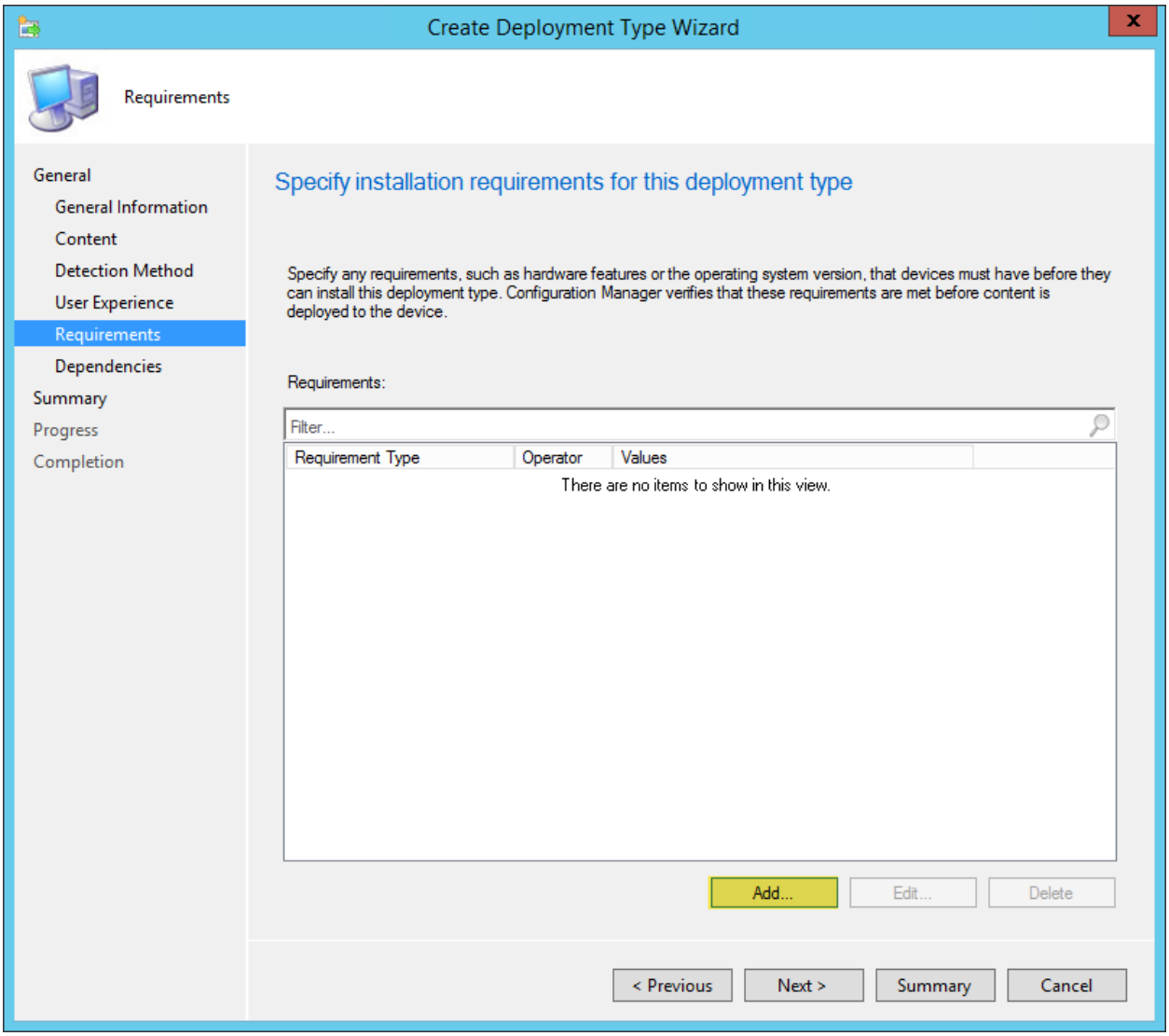
Your dialog should look pretty much like this now, so click next:



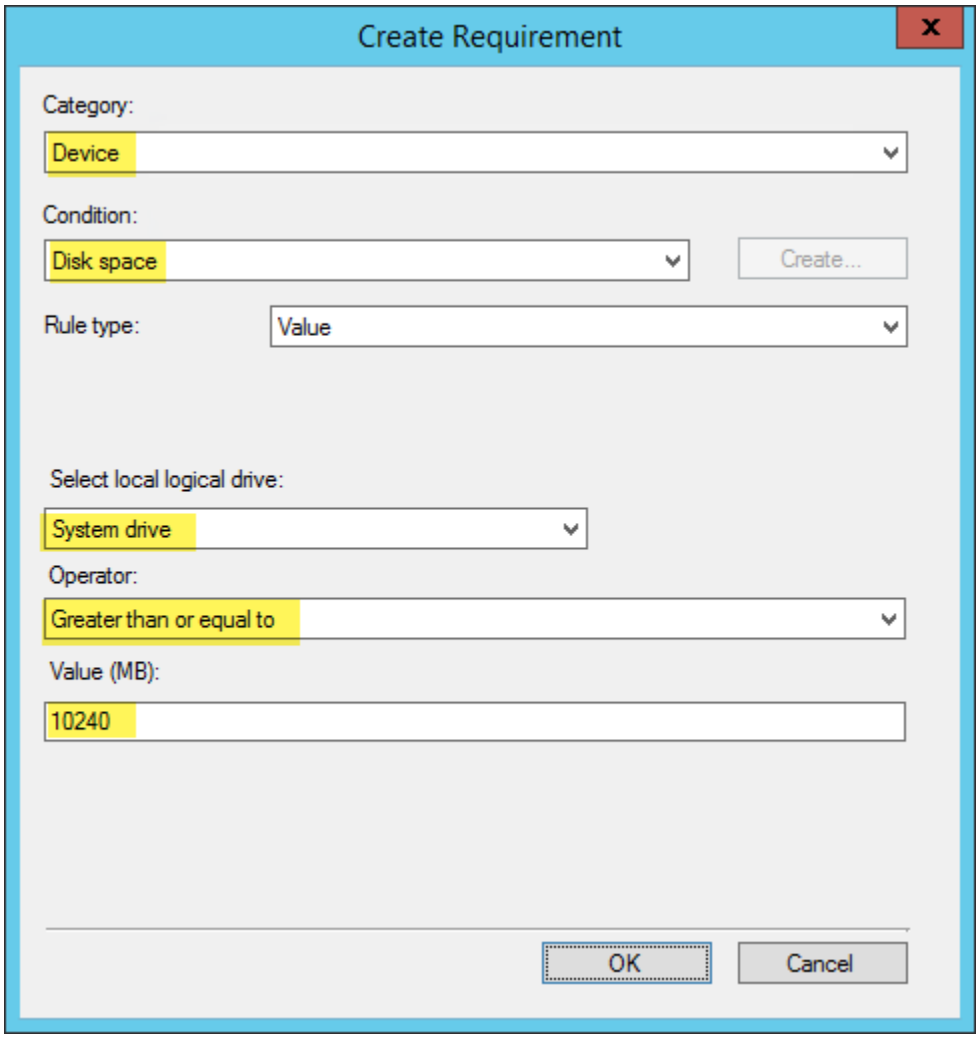
Choose to *Install for system* and *Whether or not a user is logged on* then adjust the Maximum allowed run time and estimated time to the values shown in the image below:



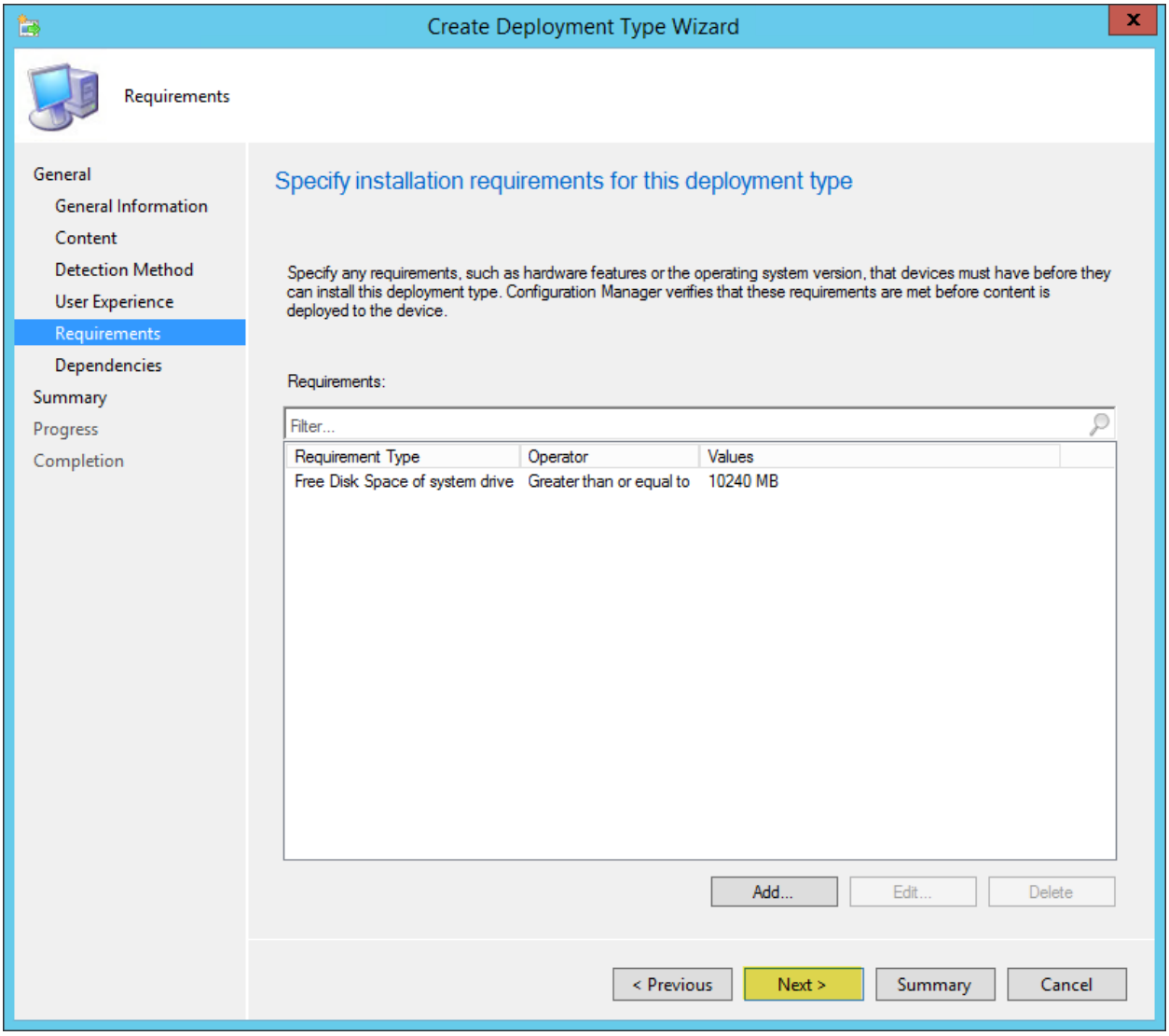
Time to add some requirements so click the Add button:



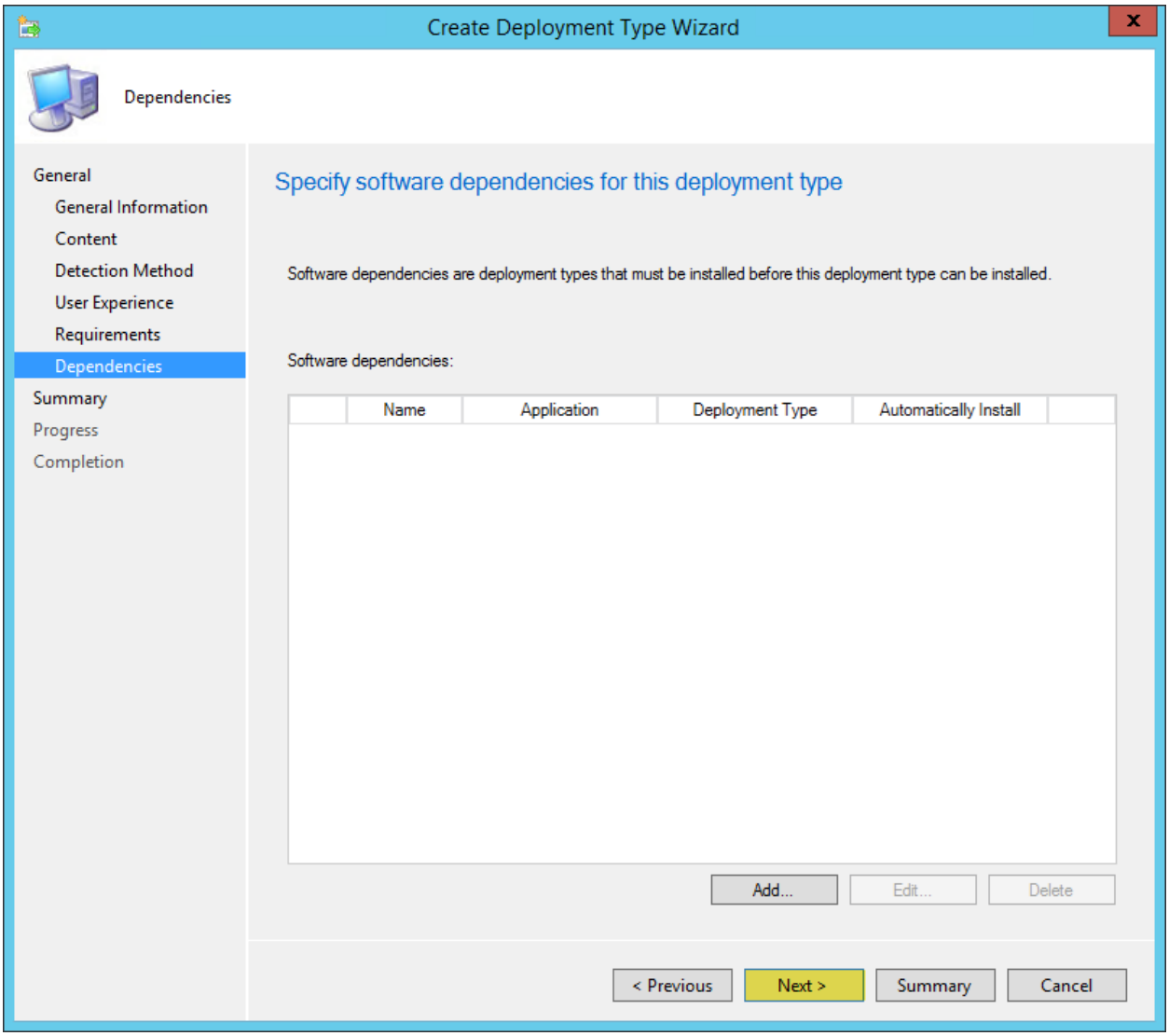
We need to ensure that that the computer in question has enough free space for the cache size increase. So season to taste here. Your needs may differ to my example. In this example, I have ensured that the cache resize will only take place if the system drive has free space greater or equal to 10GB. (I wouldn't want a 6gb cache resize to take place on a computer with only 50MB free disk space!):



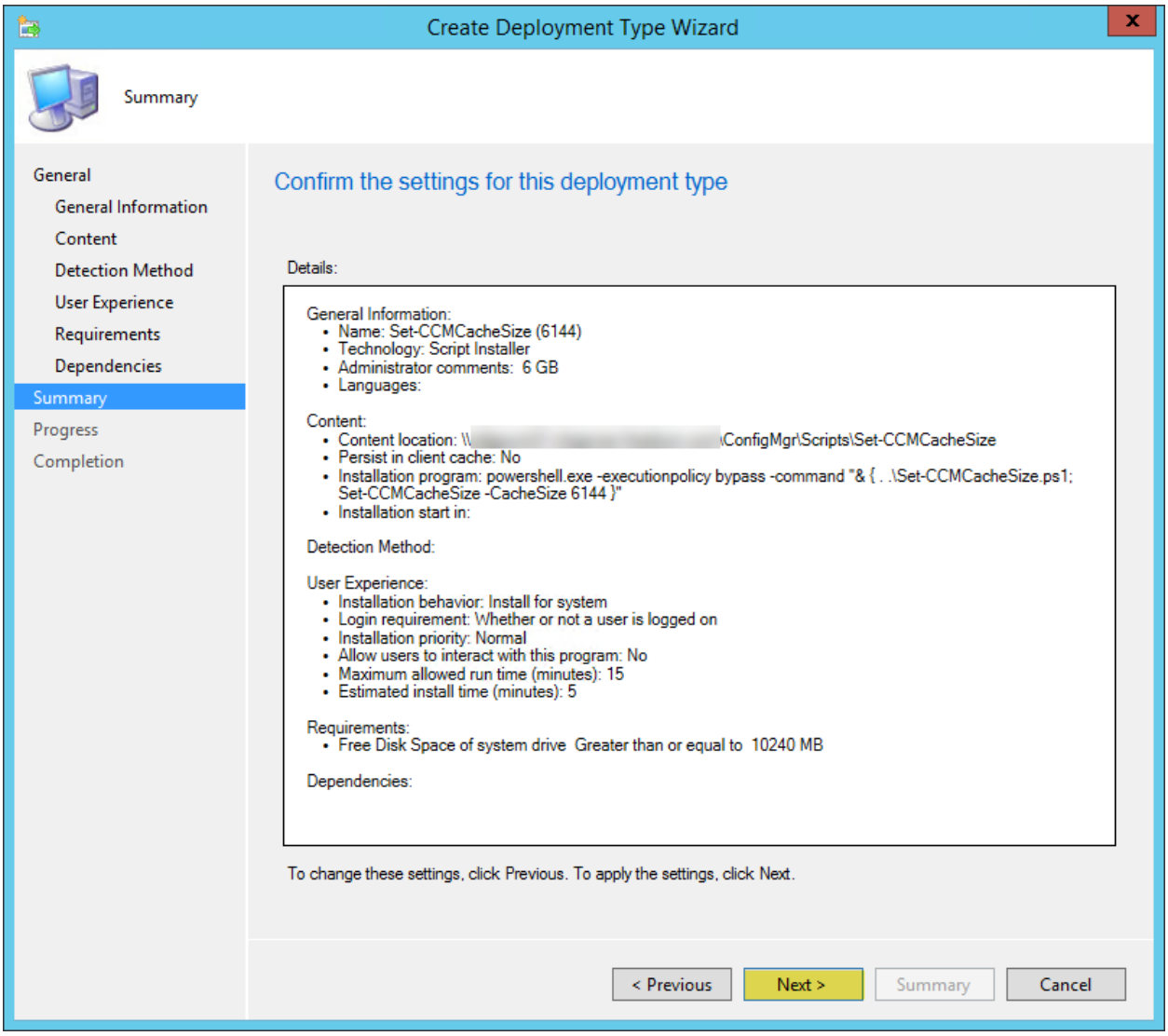
Click next here. We're almost done:



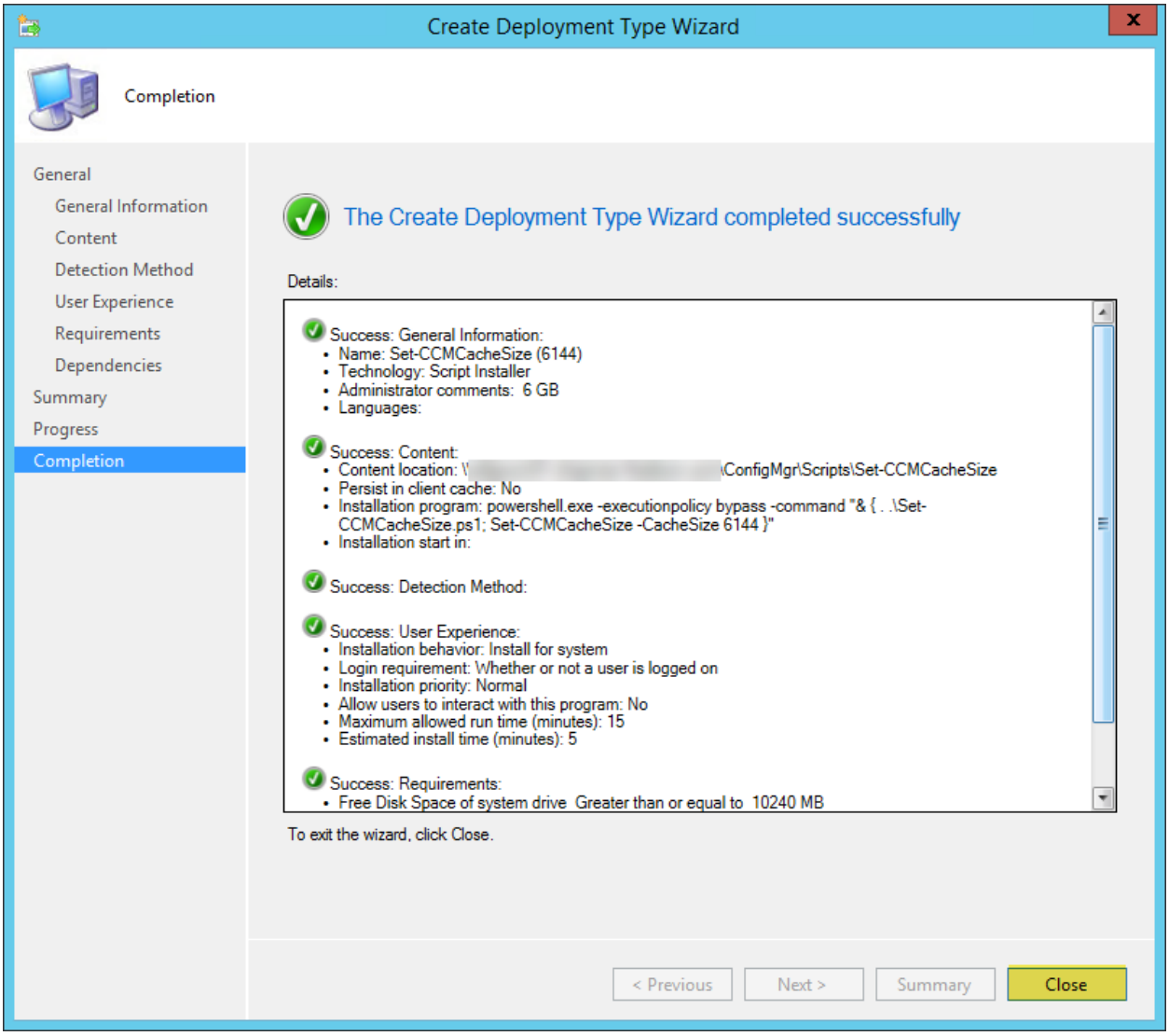
There are no dependencies for this app. If you want to add any, then go ahead, otherwise, click next:



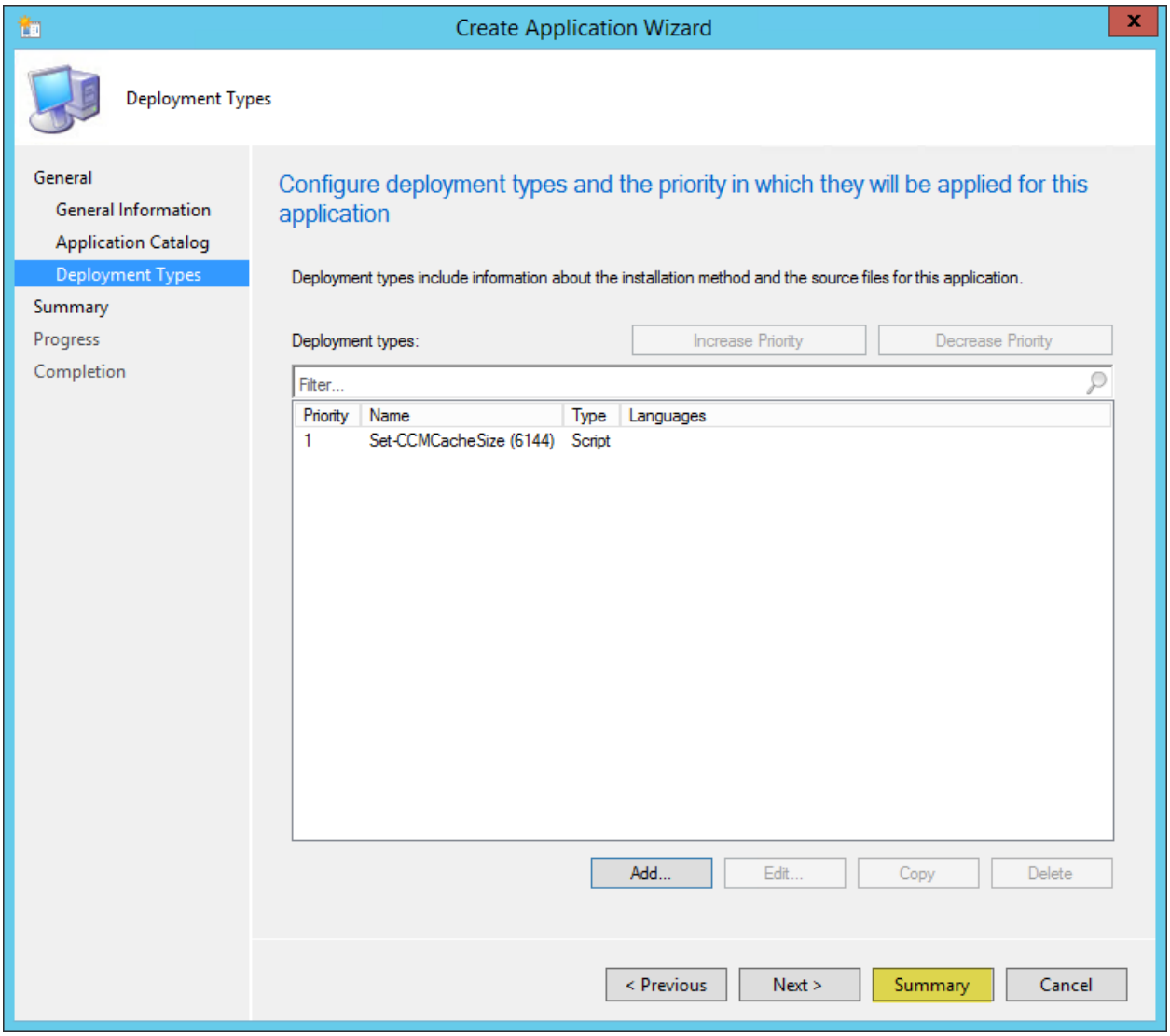
Click Next at the Summary:



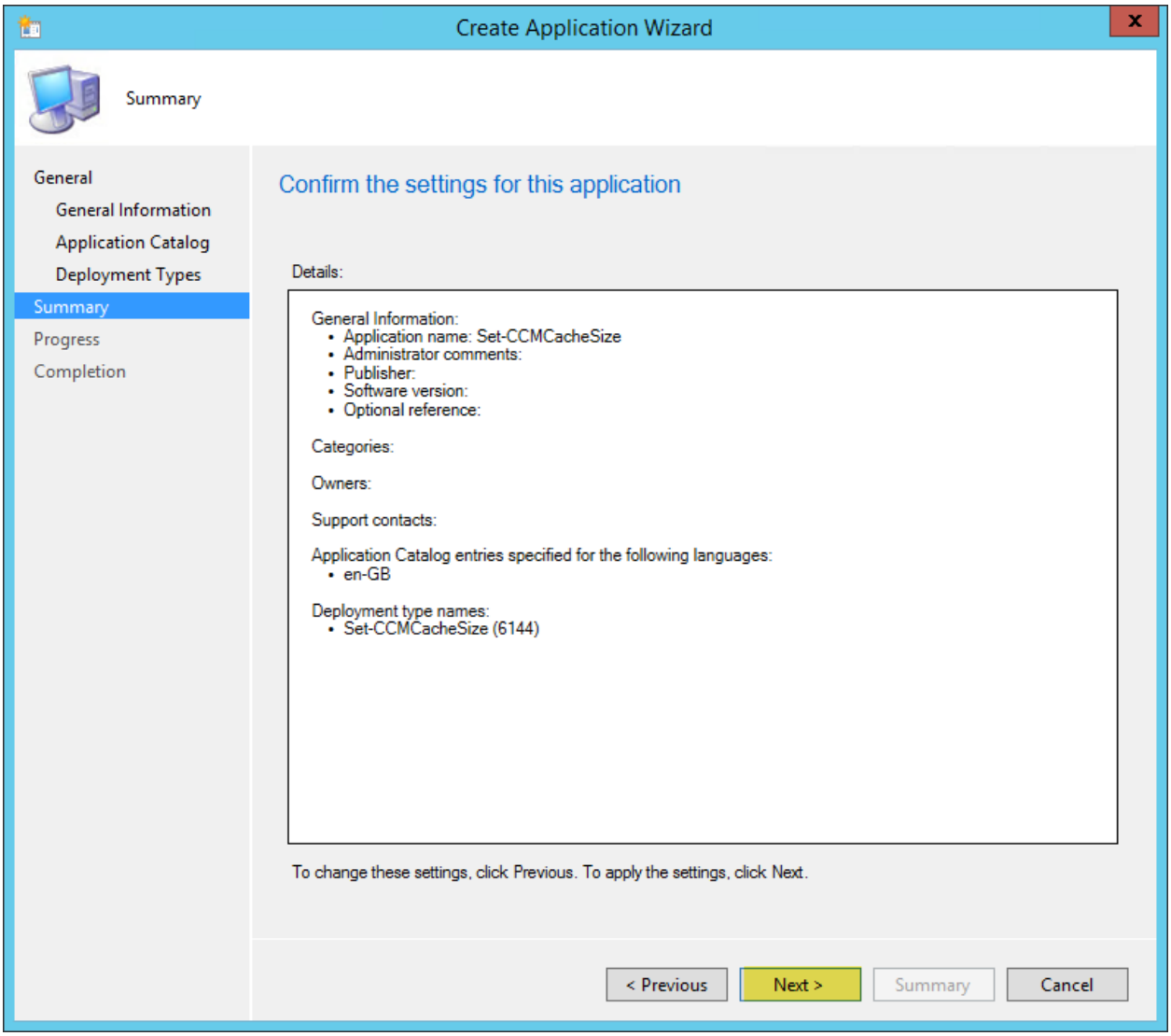
Click Close:


Create Deployment Type Wizard
Completion
General
General Information
Content
Detection Method
User Experience
Requirements
Dependencies
Summary
Progress
Completion
The Create Deployment Type Wizard completed successfully
Details:
Success: General Information:
Name: Set-CCMCacheSize (6144)
Technology: Script Installer
Administrator comments: 6 GB
Languages:
Success: Content:
Content location: \ \ConfigMgr\Scripts\Set-CCMCacheSize
Persist in client cache: No
Installation program: powershell.exe -executionpolicy bypass -command "& { . .\Set-CCMCacheSize.ps1; Set-CCMCacheSize -CacheSize 6144 }"
Installation start in:
Success: Detection Method:
Success: User Experience:
Installation behavior: Install for system
Login requirement: Whether or not a user is logged on
Installation priority: Normal
Allow users to interact with this program: No
Maximum allowed run time (minutes): 15
Estimated install time (minutes): 5
Success: Requirements:
Free Disk Space of system drive Greater than or equal to 10240 MB
To exit the wizard, click Close.
< Previous
Next >
Summary
Close

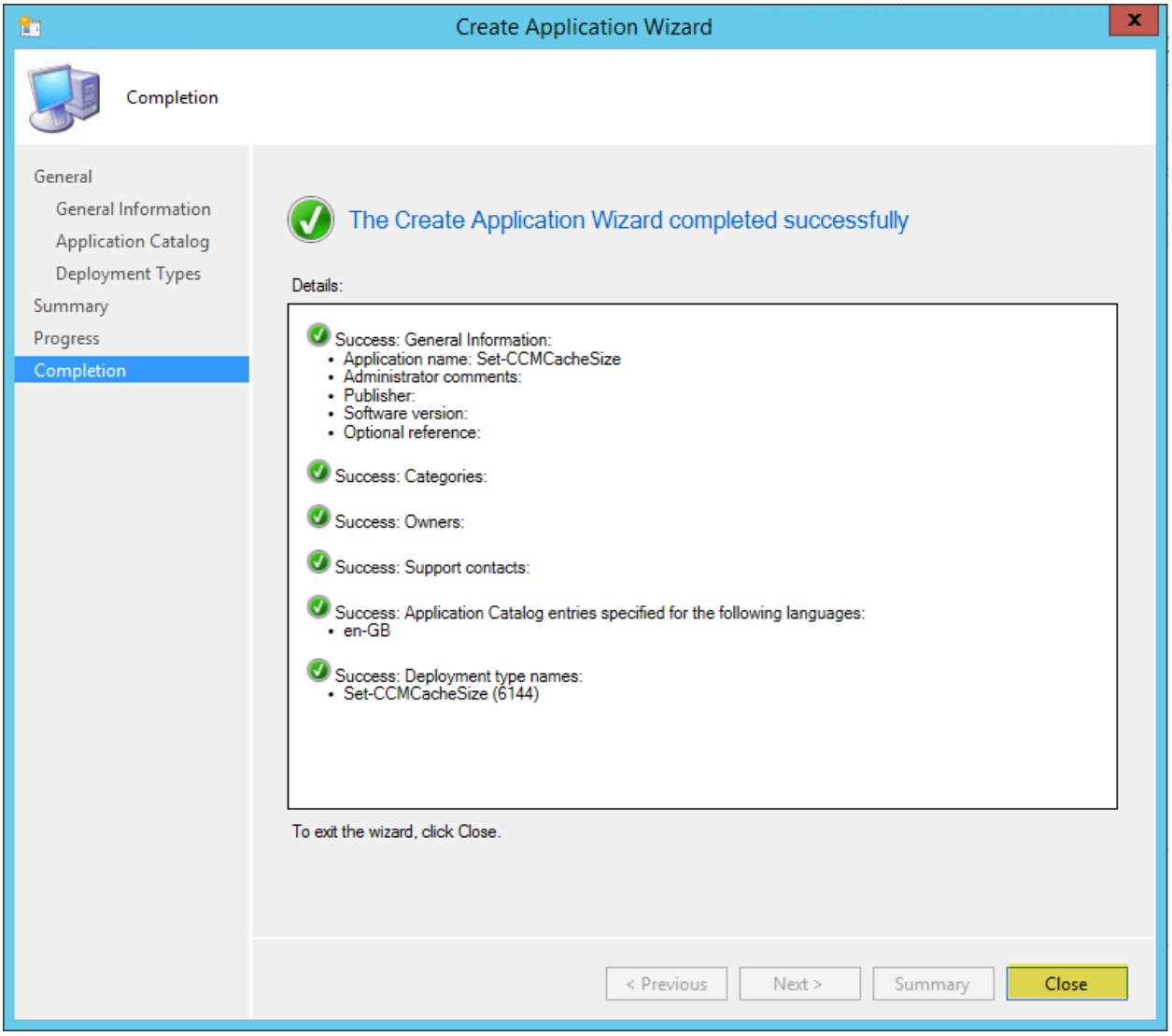
If you wanted to create another deployment type, you could do that right now. Otherwise, click the Summary button:



And it's Next again here:



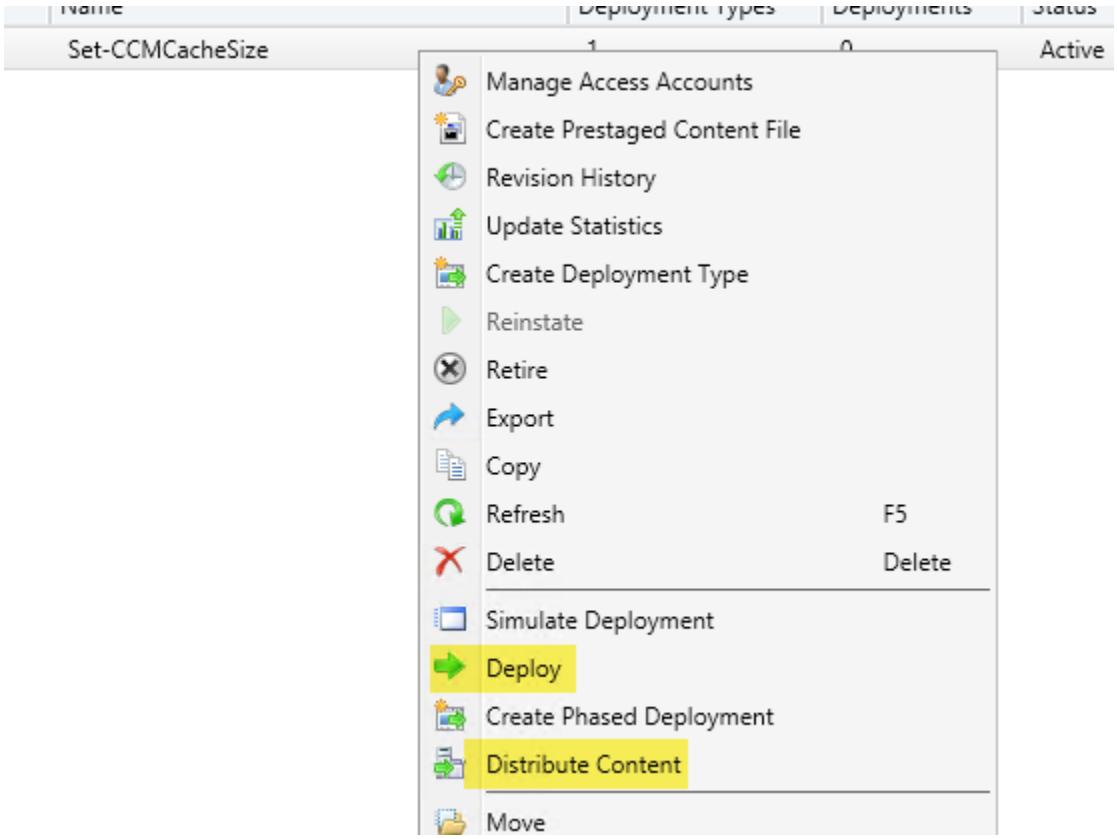
Click Close:


Create Application Wizard
Completion
General
General Information
Application Catalog
Deployment Types
Summary
Progress
Completion
The Create Application Wizard completed successfully
Details:
Success: General Information:
Application name: Set-CCMCacheSize
Administrator comments:
Publisher:
Software version:
Optional reference:
Success: Categories:
Success: Owners:
Success: Support contacts:
Success: Application Catalog entries specified for the following languages:
en-GB
Success: Deployment type names:
Set-CCMCacheSize (6144)
To exit the wizard, click Close.
< Previous
Next >
Summary
Close

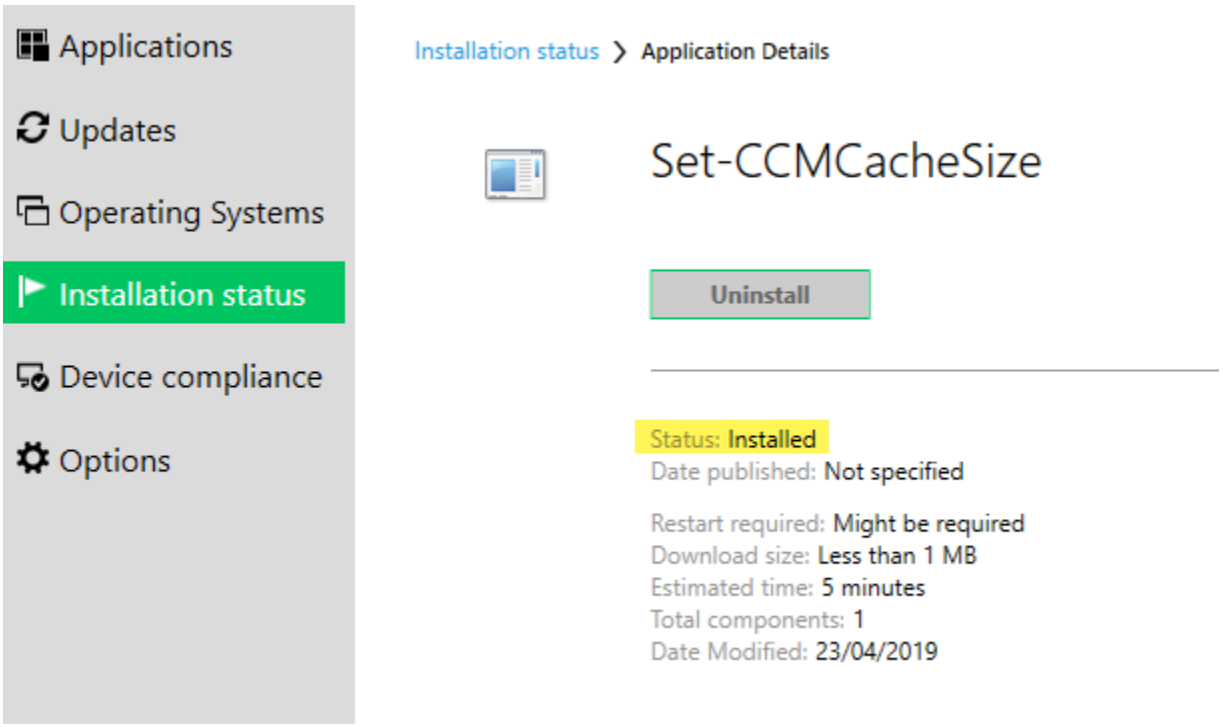
At this stage, our Application is complete. We've created an application that will deploy a PowerShell script to resize the CCMCache to 6GB and we have included a PowerShell detection rule that will query the cache for it's current size and only report a successful deployment if the size is correct. Brilliant stuff!

Right-Click the Application you've just created and select to either Distribute Content to your chosen distribution points and then Deploy it, or just select Deploy and Distribute and Deploy in the same wizard. The choice is yours.

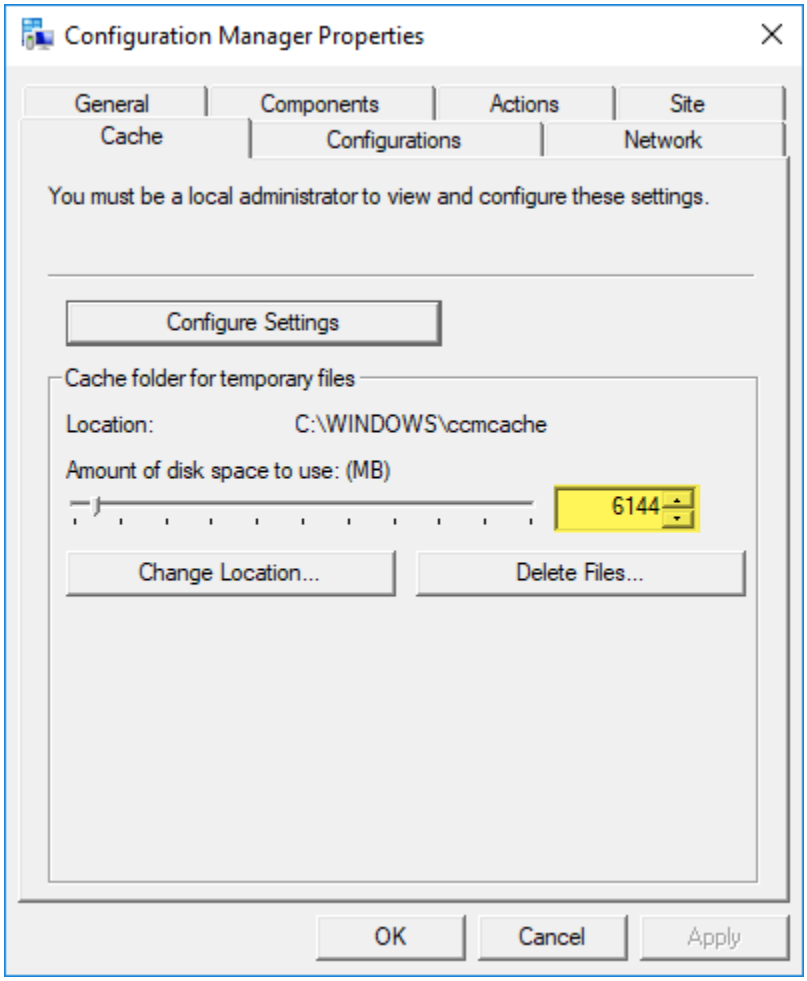
Either way, distribute and deploy to your computer collection. (I deployed to a test collection that just has one computer in it for this example.):



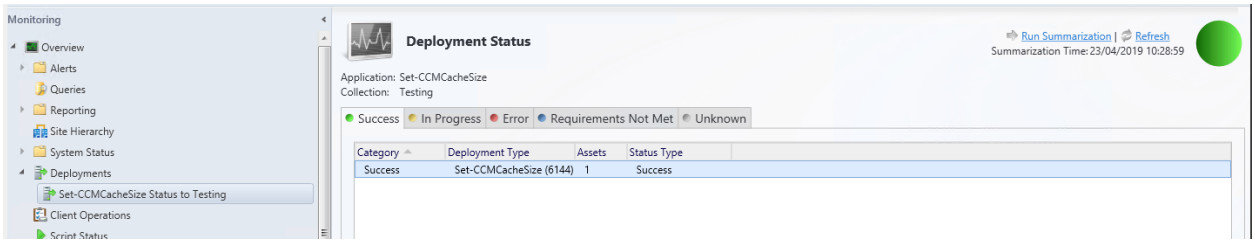
In a "blink and you'll miss it" moment when I checked the Application Catalog on my test computer I found it had installed successfully:



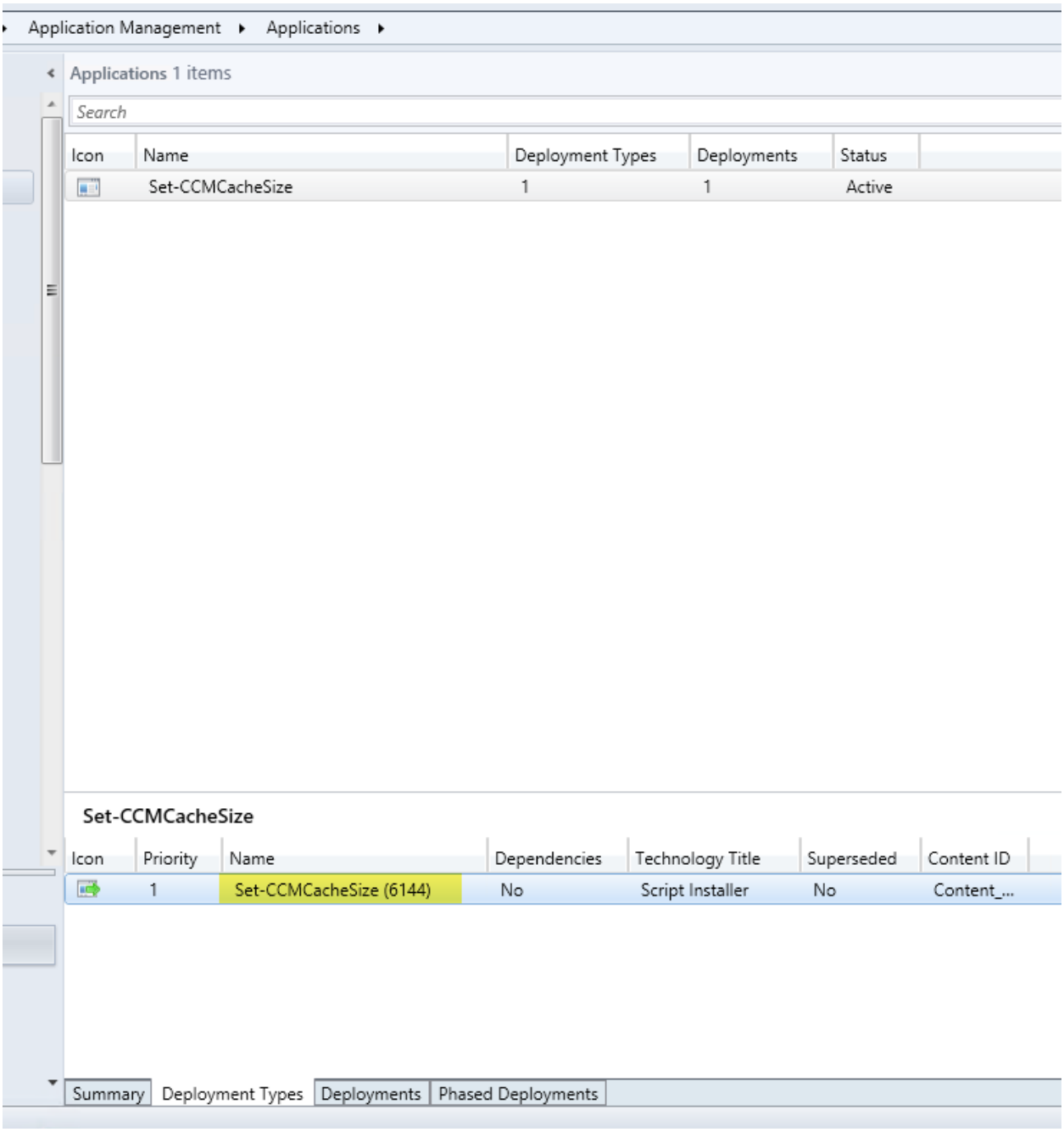
I validated this by checking the Configuration Manager Client properties and sure enough, it was sized to 6144:



I then checked the Monitoring node in the Configuration Manager Console and verified the successful installation:



And just so you can compare, here is what the application looks like in my Configuration Manager console:



Excellent!

Bonus Chapter 2

A Step-by-Step Guide to Deploying EMC SourceOne Agent for Offline Files

Deploying this application is a great showcase on how to effectively use the Deployment Template to ensure rapid and successful deployment. (As per part 7 of this book.)

When I did this, there were two exe's that I had to deploy: The main exe followed by a hotfix. At the time of writing this, they were version 7.2 SP7 and 7.2 SP7 Hotfix1.

The version of SourceOne that you install must match the installed Microsoft Office version. If you have the 32-bit version of Office installed then the 32-bit version of SourceOne must be installed. By the same token, if you have the 64-bit Office version installed then you would need the 64-bit version of SourceOne.

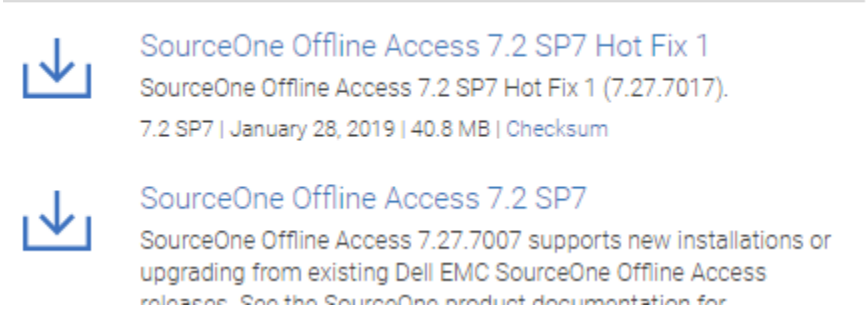
Right, before we start, let's go over the objectives so we can be clear on what we need to achieve:

Objectives

1. Install SourceOne v7.2 SP7
2. Install SourceOne v7.2 SP7 Hotfix 1
3. Ensure that the correct version of both the main installer and the hotfix are installed based on Office 'bitness'

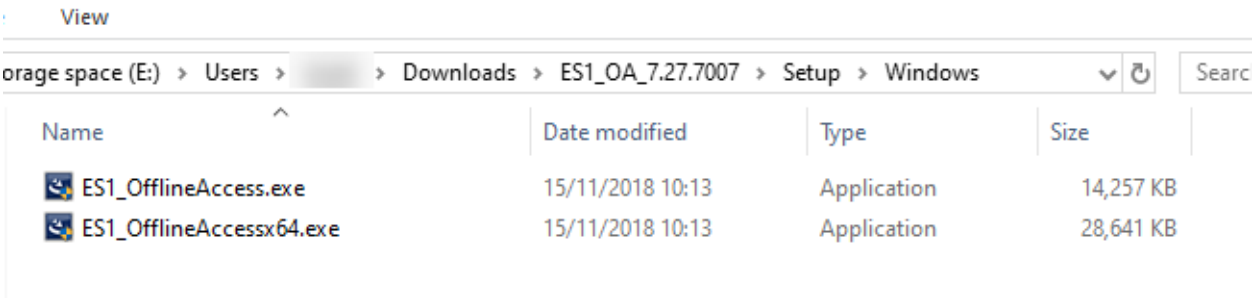
Download and Extract the Files!

The first step is to download the two exe's from the Dell EMC website.



These downloads contain both 32-bit and 64-bit versions of the exe installers we'll need to deploy.

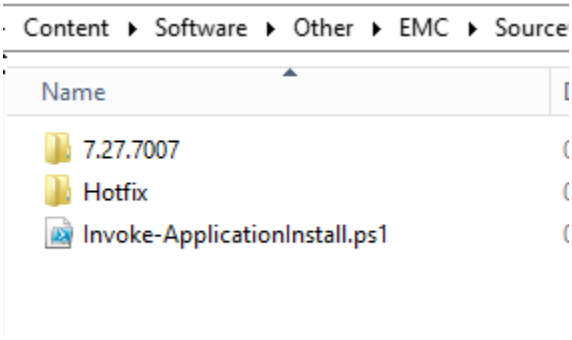
I used 7-zip and extracted the downloaded executable. This gave me access to the executables:



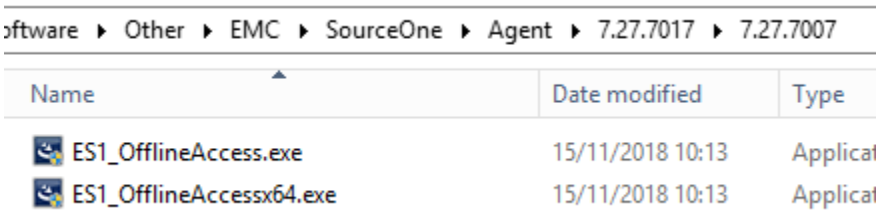
Move the Files to the SCCM Source Location

Next, move the extracted executable files to a folder where you would normally place your source files. I created a root directory entitled 7.27.7017 (the version number of the resulting source one agent deployment) that contained two other directories, one entitled 7.27.7007 which contained both the 32-bit and 64-bit versions of the executables, and a second directory entitled, 'Hotfix' which contained both the 32-bit and 64-bit versions of the hotfix executable. (The hotfix is what will bump the version number to .7017) Finally, I copied the Invoke-ApplicationInstall.ps1 deployment template to the root folder.

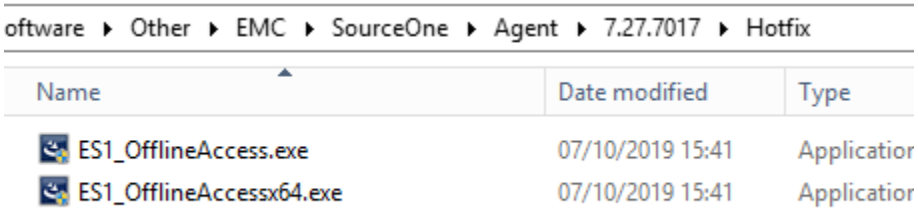
Contents of the root directory:



Contents of the 7.27.7007 sub-directory:



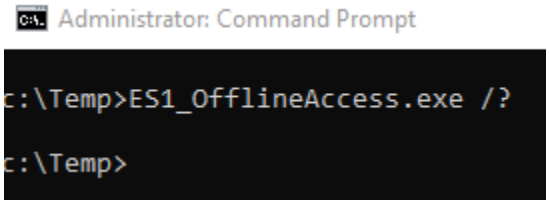
Contents of the Hotfix sub-directory:


oftware ▸ Other ▸ EMC ▸ SourceOne ▸ Agent ▸ 7.27.7017 ▸ Hotfix
Name
Date modified
Type
ES1_OfflineAccess.exe
07/10/2019 15:41
Applicatior
ES1_OfflineAccessx64.exe
07/10/2019 15:41
Applicatior

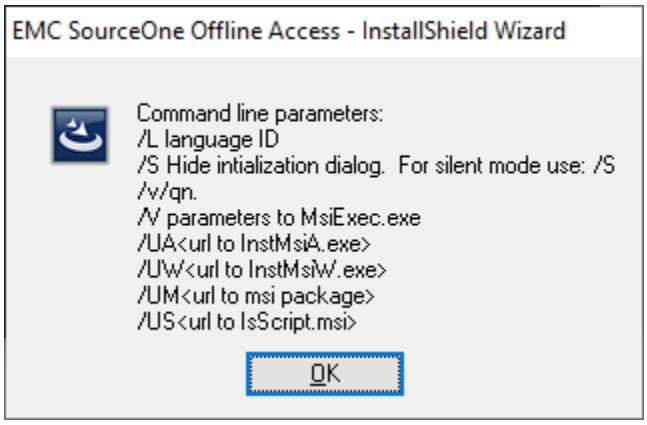
Discover the Silent Deployment Switches

To discover the switches I would need for a silent deployment I ran the following in a command prompt against one of the downloaded executables:

```
ES!_OfflineAccess.exe /?
```



Thankfully, this revealed the following dialog:



To deploy silently I now knew I would need to use the switches: `/S /v/qn`

The Deployment Template

The Deployment Template can deploy based on either operating system bitness, or office bitness.

We need this installation to deploy the correct (32-bit or 64-bit) SourceOne executable and hotfix executable that matches the installed Office version bitness (32-bit Office or 64-bit Office).

The command line I would need to install the 32-bit version of the executable would be:

```
start-process -FilePath ".\7.27.7007\ES1_OfflineAccess.exe" -ArgumentList "/S /v/qn" -NoNewWindow -Wait
```

I am simply referencing the exe file in the 7.27.7007 subdirectory and passing the silent switches that I had previously discovered.

For the 32-bit hotfix, the command line would be much the same, with the only difference being the subdirectory change: (This is because the executables had the same name, so the only difference was the file path.)

```
start-process -FilePath ".\Hotfix\ES1_OfflineAccess.exe" -ArgumentList "/S /v/qn" -NoNewWindow -Wait
```

I placed the two commands on the same line in the relevant section of the deployment template, separated by a semi-colon. (The first exe would install and because we added the -wait parameter, the second (hotfix) would install once the first one had finished.)

I repeated the process for the 64-bit version of the command lines. (All I needed to do was adjust the executable names to reflect the 64-bit exe's.)

```
    switch ($Obj.OfficeIs32Bit) {
        $true {start-process -FilePath ".\7.27.7007\ES1_OfflineAccess.exe" -ArgumentList "/S /v/qn" -NoNewWindow -Wait;start
        $false {start-process -FilePath ".\7.27.7007\ES1_OfflineAccessx64.exe" -ArgumentList "/S /v/qn" -NoNewWindow -Wait;s
        'Unknown' {"Office not detected - do what you want here"}
    }
}
```

The final edit of the Deployment Template was to ensure that it was going to deploy based on Office bitness.

I knew that I wanted to deploy this as a script with an entry point, and so I added the switch `-InstallBasedOnOfficeBitness` at the script entry point in the code.

So the very last line of the Deployment Template PowerShell script was:

```
Invoke-ApplicationInstall -InstallBasedOnOfficeBitness
```

```
            "Logged In User: $($Obj.CurrentUser)" | out-file -
            "Office Version is 32Bit: $($Obj.OfficeIs32Bit)" |
            "Operating System is 64Bit: $($Obj.OSis64Bit)" | o
        }
}

#Script entry point
Invoke-ApplicationInstall -InstallBasedOnOfficeBitness
```

That's it for the Deployment Template…all that's left is to create the application in Configuration Manager and deploy it!

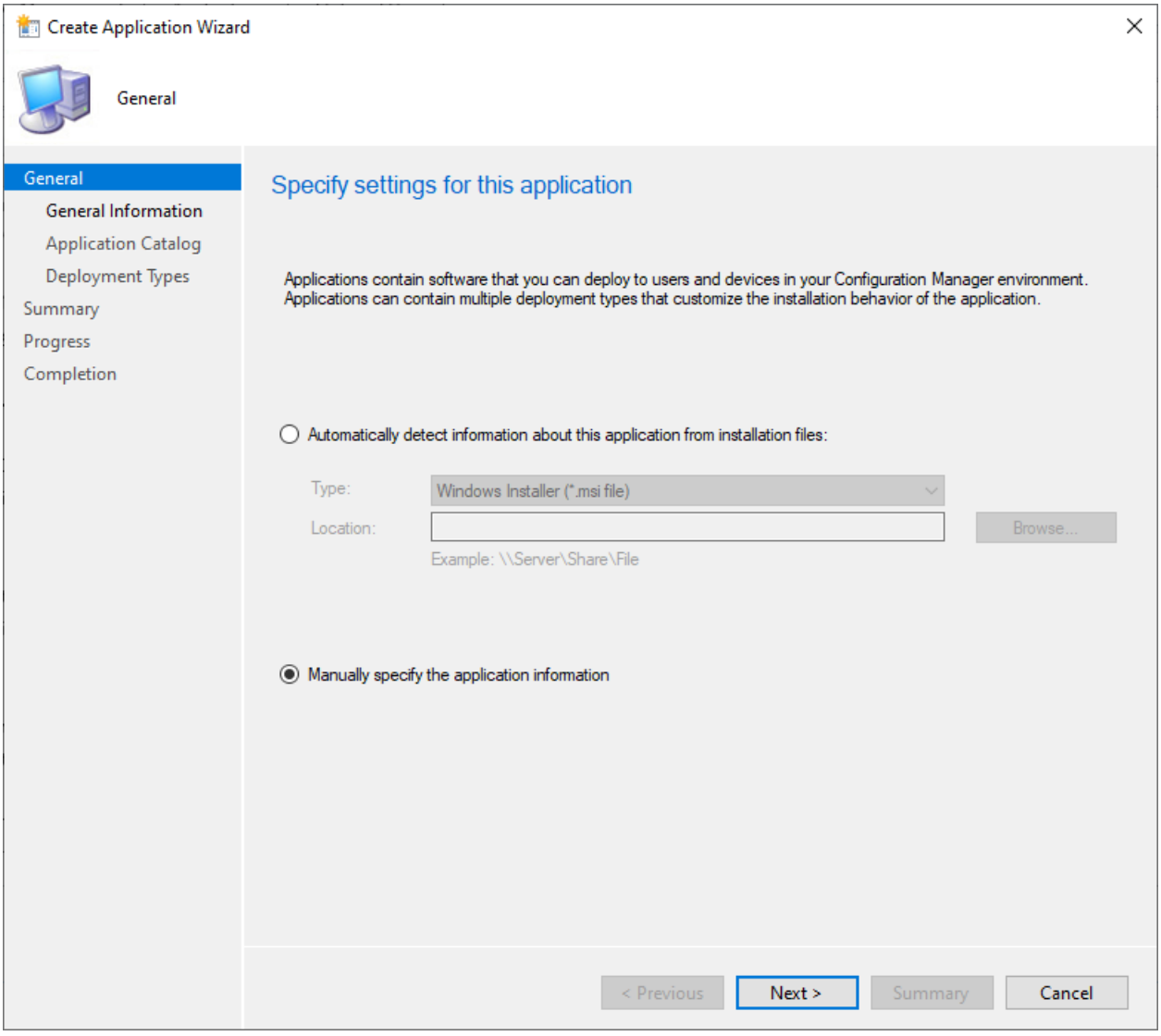
Download the Pre-Configured Deployment Template

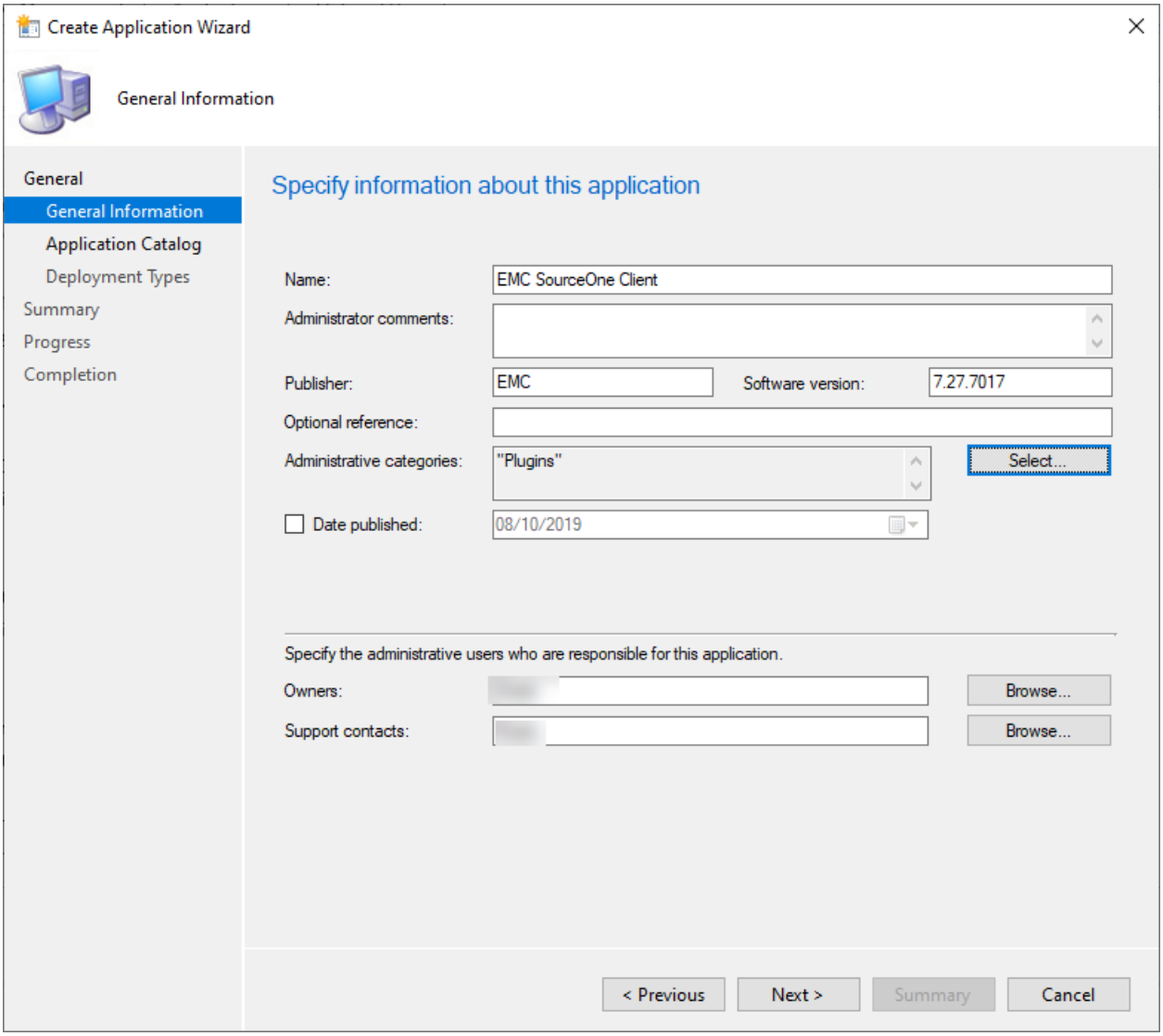
Here's where you can download the ready-to-go Deployment Template that is configured to deploy SourceOne as per all of the previous steps:

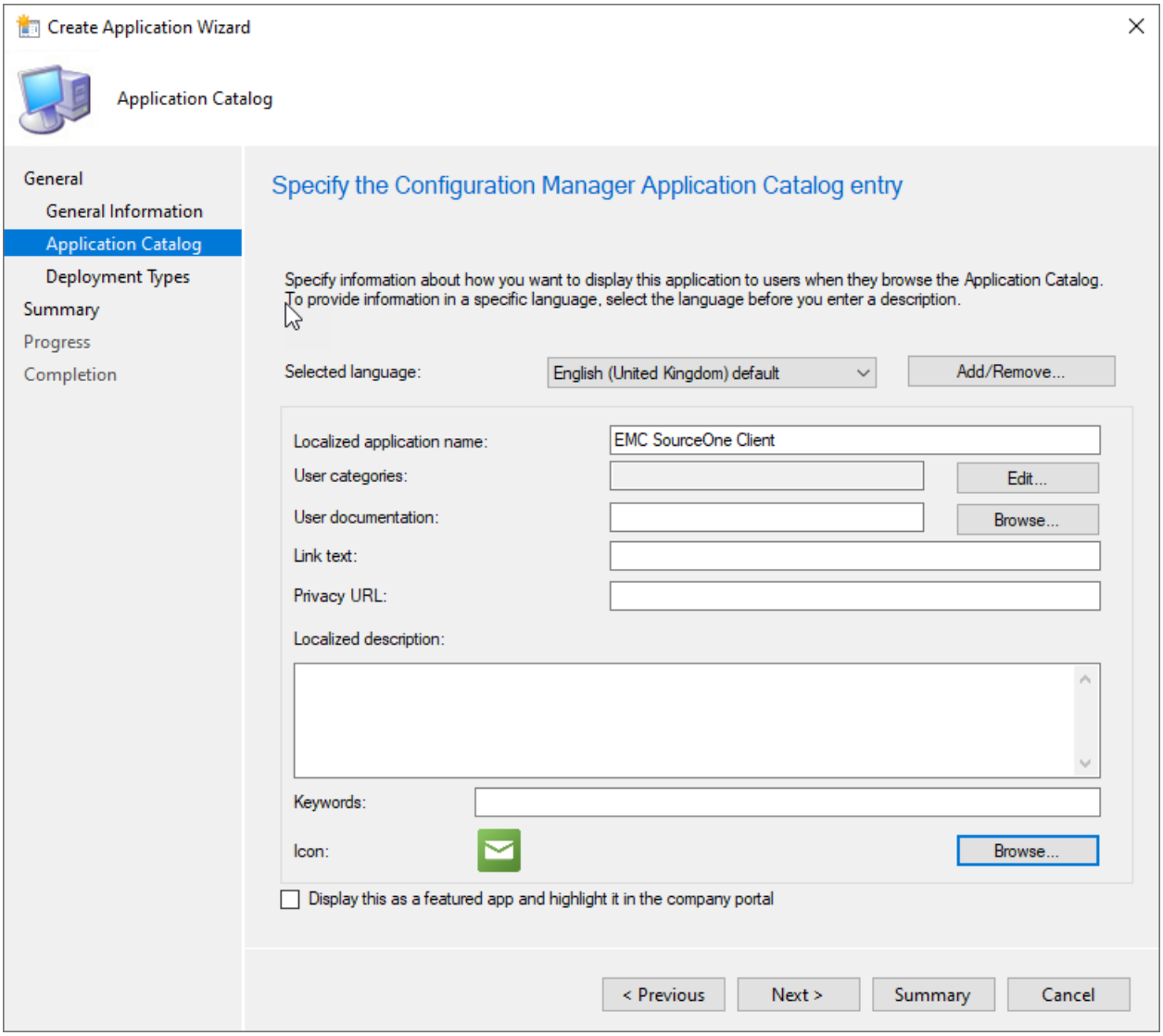
https://github.com/ozthe2/Powershell/tree/master/SCCM/SourceOne

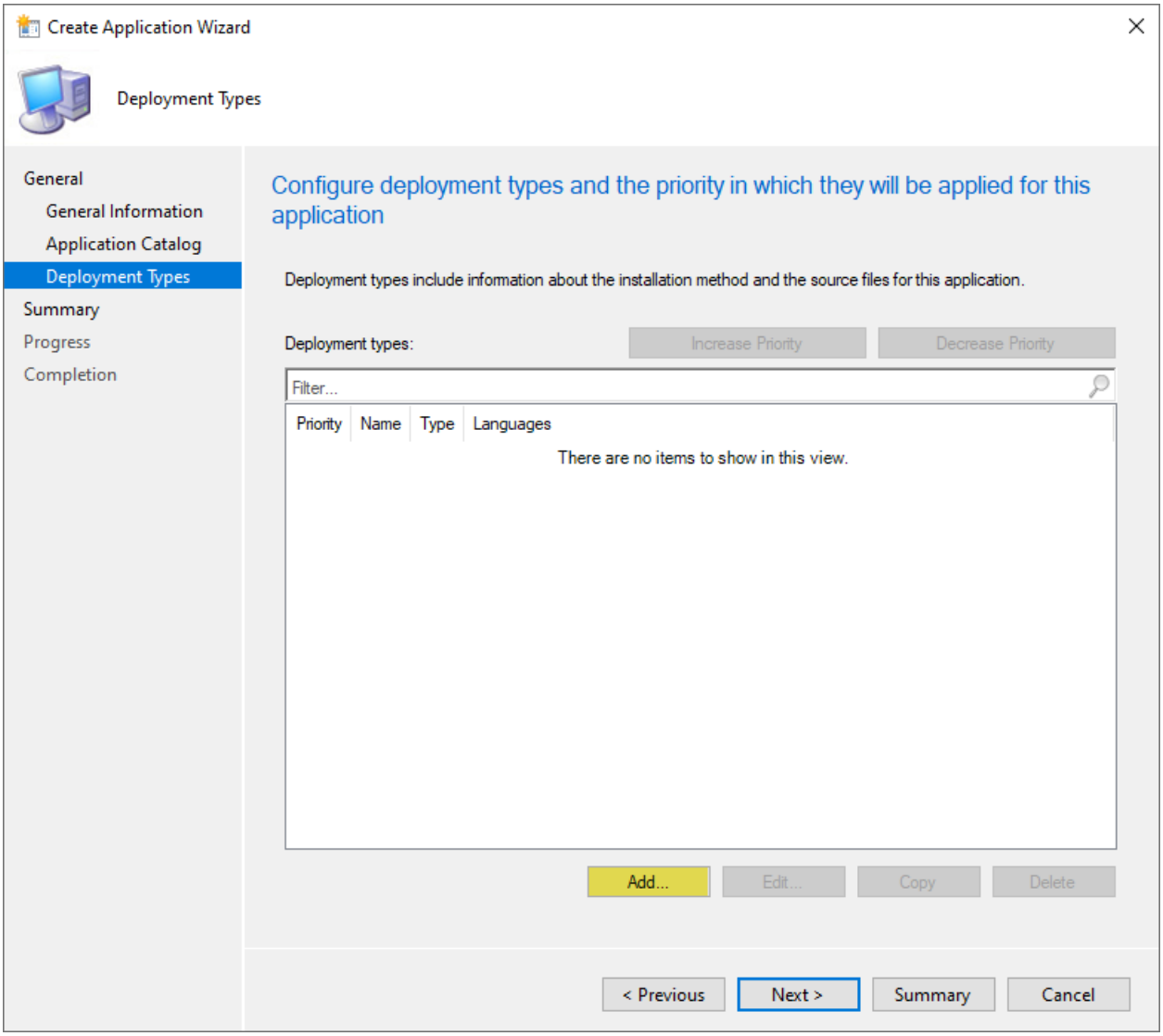
Create The Application

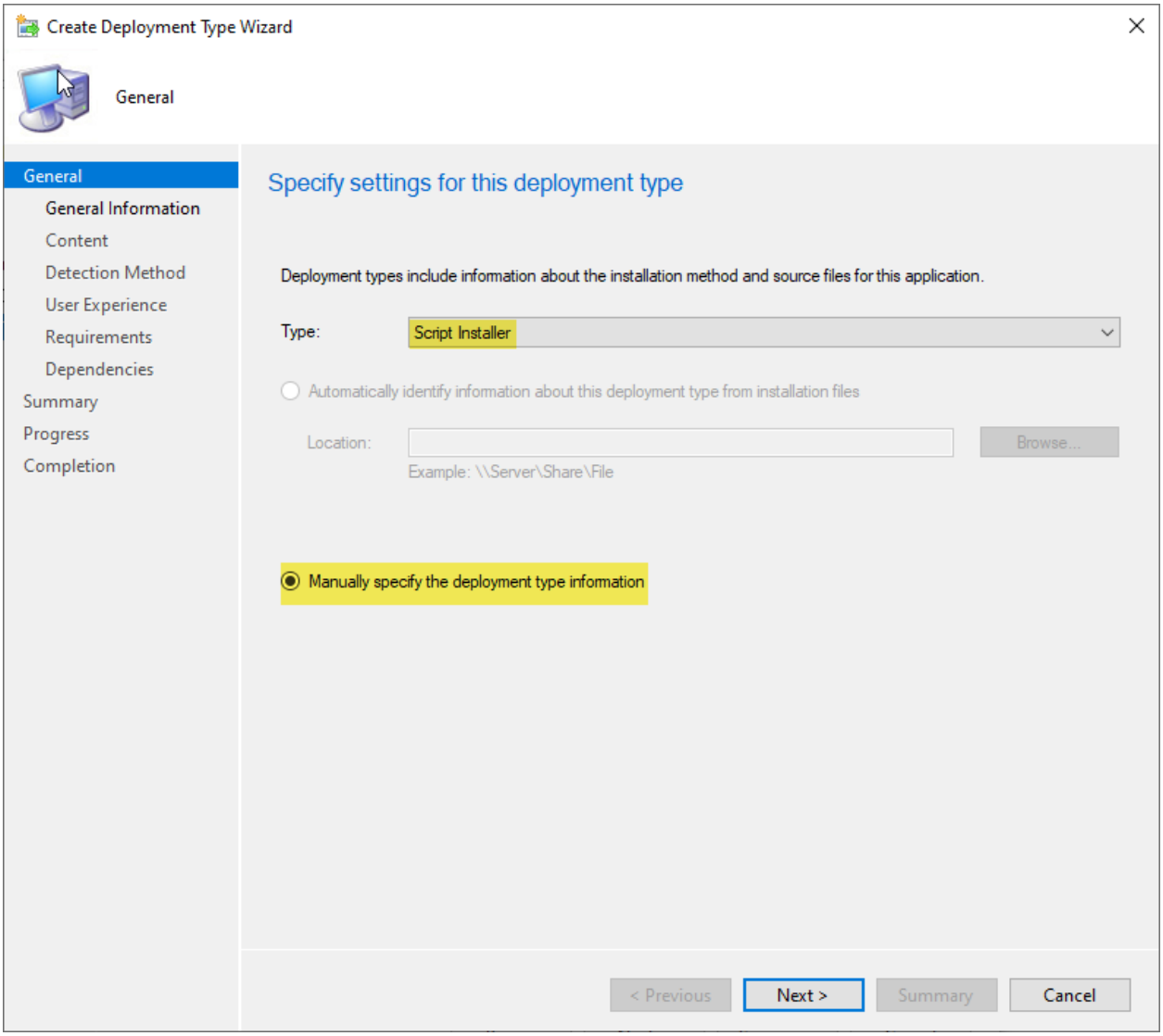
Open your Configuration Manager Console and start the wizard for creating a new application. Once you've done that, follow along..this ride won't be long…

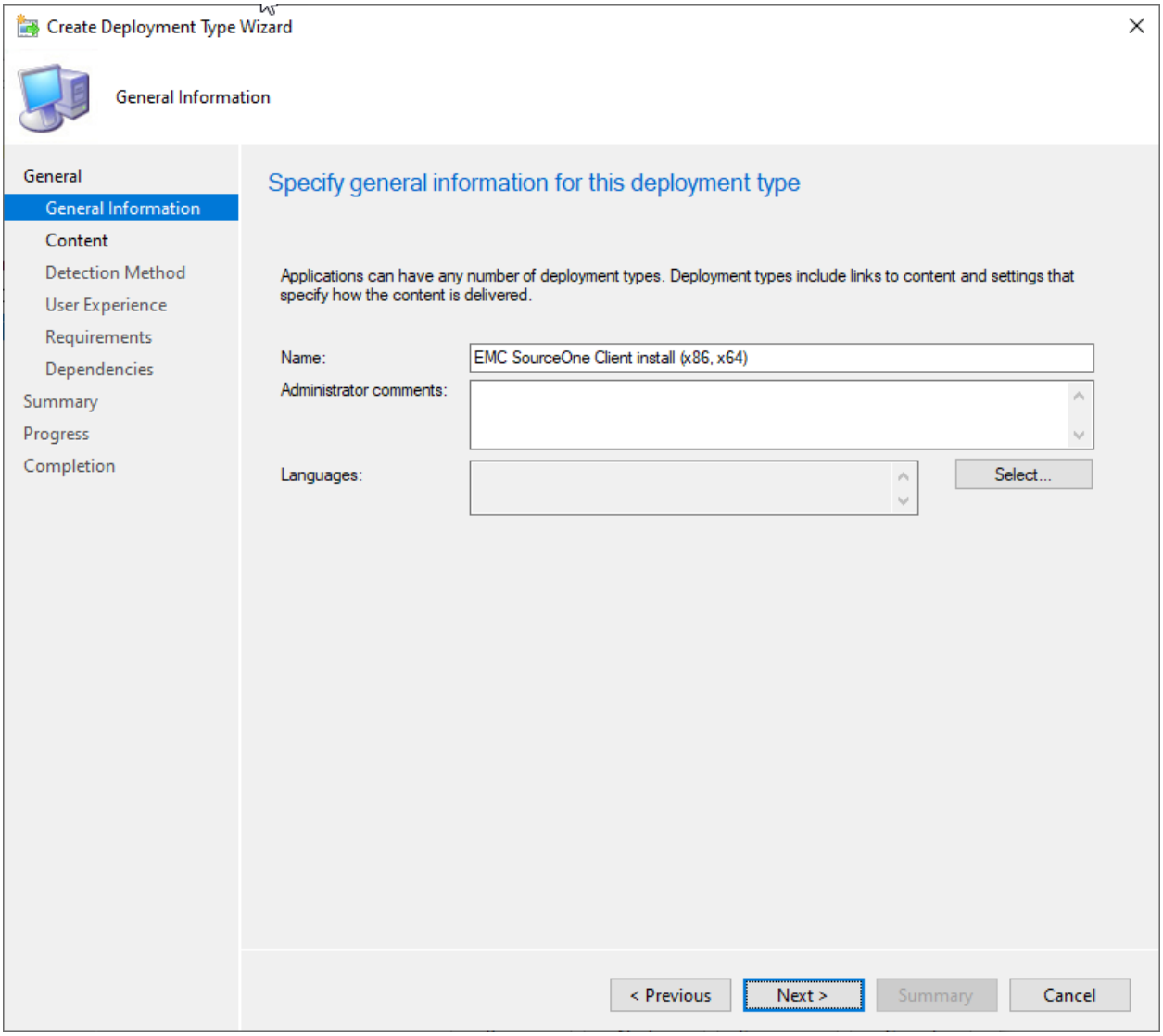



Create Application Wizard
General Information
General
General Information
Application Catalog
Deployment Types
Summary
Progress
Completion
Specify information about this application
Name:
EMC SourceOne Client
Administrator comments:
Publisher:
EMC
Software version:
7.27.7017
Optional reference:
Administrative categories:
"Plugins"
Select...
Date published:
08/10/2019
Specify the administrative users who are responsible for this application.
Owners:
Browse...
Support contacts:
Browse...
< Previous
Next >
Summary
Cancel


Create Application Wizard
Application Catalog
General
General Information
Application Catalog
Deployment Types
Summary
Progress
Completion
Specify the Configuration Manager Application Catalog entry
Specify information about how you want to display this application to users when they browse the Application Catalog. To provide information in a specific language, select the language before you enter a description.
Selected language:
English (United Kingdom) default
Add/Remove...
Localized application name:
EMC SourceOne Client
User categories:
Edit...
User documentation:
Browse...
Link text:
Privacy URL:
Localized description:
Keywords:
Icon:
Browse...
Display this as a featured app and highlight it in the company portal
< Previous
Next >
Summary
Cancel


Create Application Wizard
Deployment Types
General
General Information
Application Catalog
Deployment Types
Summary
Progress
Completion
Configure deployment types and the priority in which they will be applied for this application
Deployment types include information about the installation method and the source files for this application.
Deployment types:
Increase Priority
Decrease Priority
Filter...
Priority
Name
Type
Languages
There are no items to show in this view.
Add...
Edit...
Copy
Delete
< Previous
Next >
Summary
Cancel

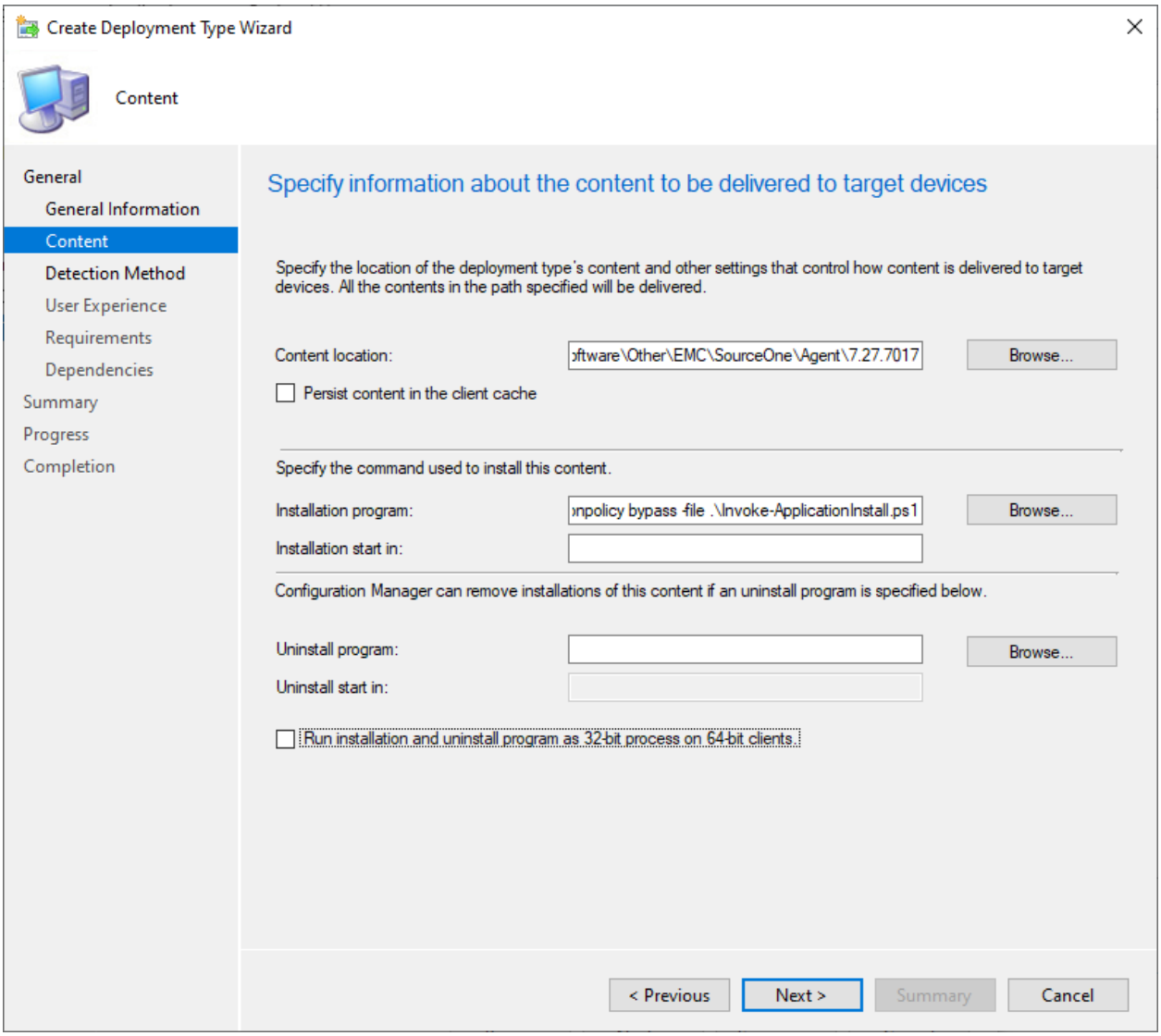

Create Deployment Type Wizard
General
General
General Information
Content
Detection Method
User Experience
Requirements
Dependencies
Summary
Progress
Completion
Specify settings for this deployment type
Deployment types include information about the installation method and source files for this application.
Type:
Script Installer
Automatically identify information about this deployment type from installation files
Location:
Browse...
Example: \\Server\Share\File
Manually specify the deployment type information
< Previous
Next >
Summary
Cancel

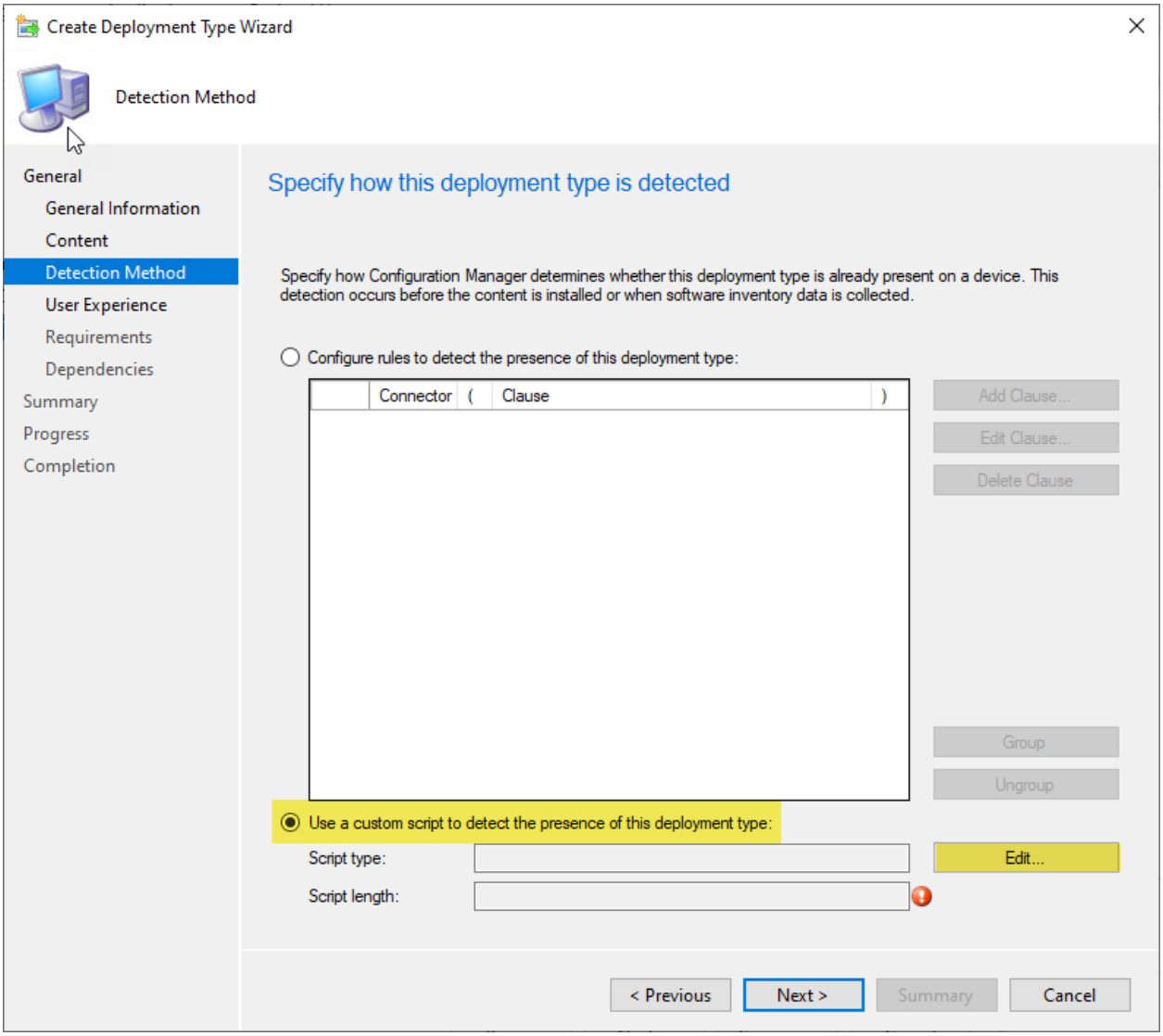

Create Deployment Type Wizard
General Information
General
General Information
Content
Detection Method
User Experience
Requirements
Dependencies
Summary
Progress
Completion
Specify general information for this deployment type
Applications can have any number of deployment types. Deployment types include links to content and settings that specify how the content is delivered.
Name:
EMC SourceOne Client install (x86, x64)
Administrator comments:
Languages:
Select...
< Previous
Next >
Summary
Cancel

Installation Program

For the *Installation Program* command line, we call the script using the techniques shown earlier in the book (Script with an entry point) and the command line used is:

```
powershell.exe -executionpolicy bypass -file .\Invoke-ApplicationInstall.ps1
```




Create Deployment Type Wizard
Detection Method
General
General Information
Content
Detection Method
User Experience
Requirements
Dependencies
Summary
Progress
Completion
Specify how this deployment type is detected
Specify how Configuration Manager determines whether this deployment type is already present on a device. This detection occurs before the content is installed or when software inventory data is collected.
Configure rules to detect the presence of this deployment type:
Connector (Clause)
Add Clause...
Edit Clause...
Delete Clause
Group
Ungroup
Use a custom script to detect the presence of this deployment type:
Script type:
Script length:
Edit...
< Previous
Next >
Summary
Cancel

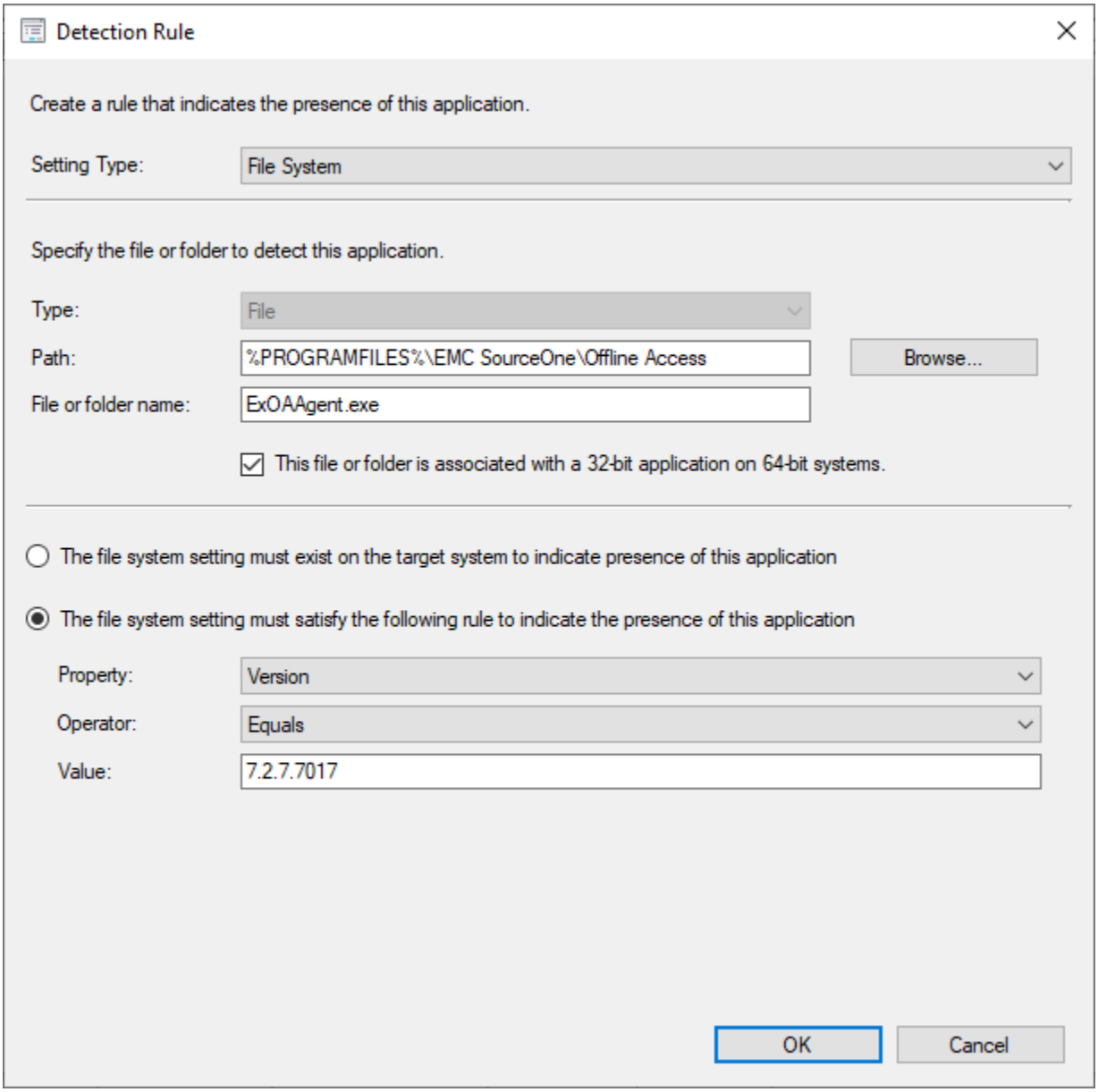
The Detection Rule

You can either use PowerShell as your detection rule, or the traditional method.

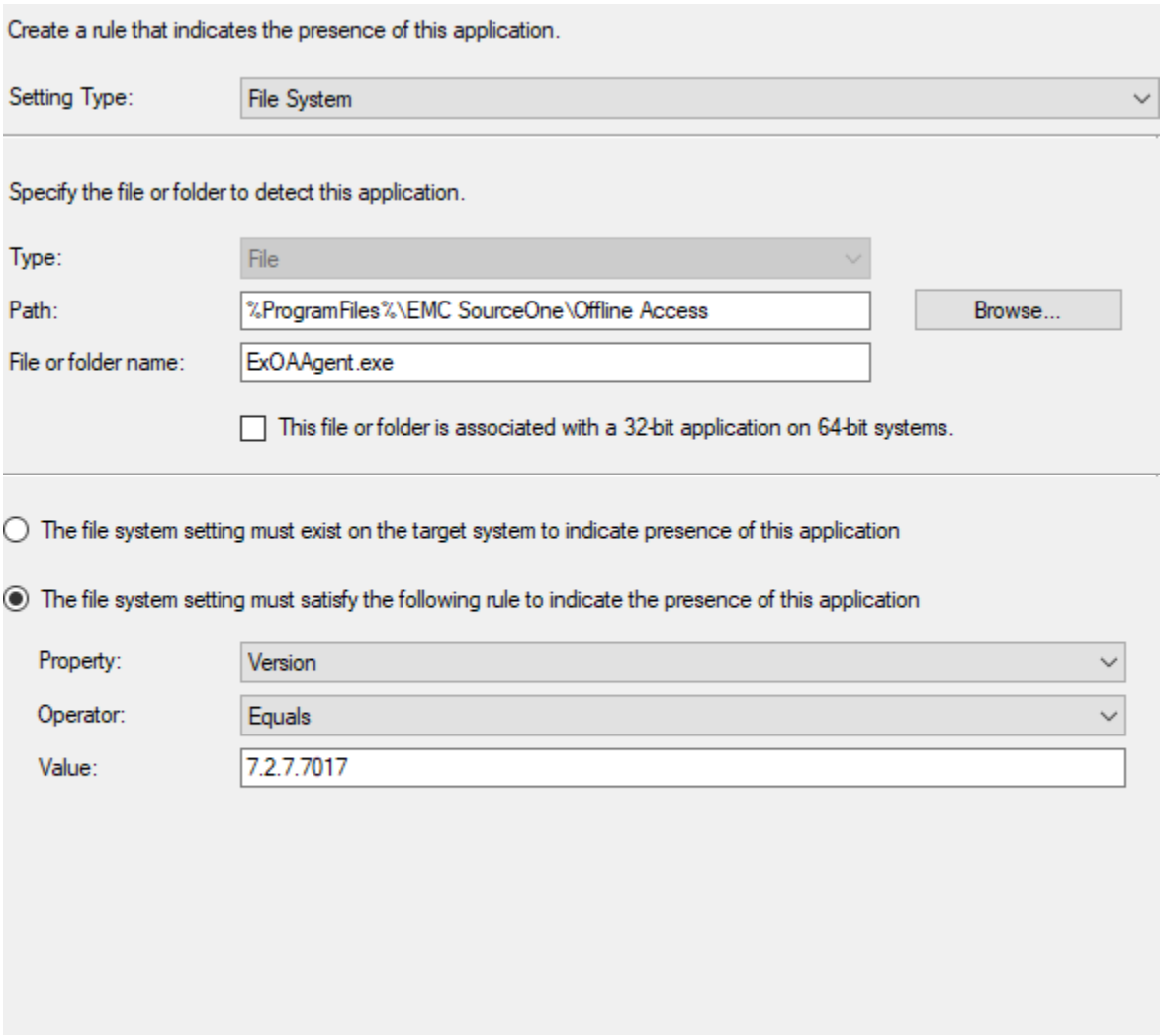
Traditional Method

Using the traditional method to detect the executable version number, you will need to create two rules: one to detect the 32-bit install and one to detect the 64-bit install. You then link them with an `OR` clause.

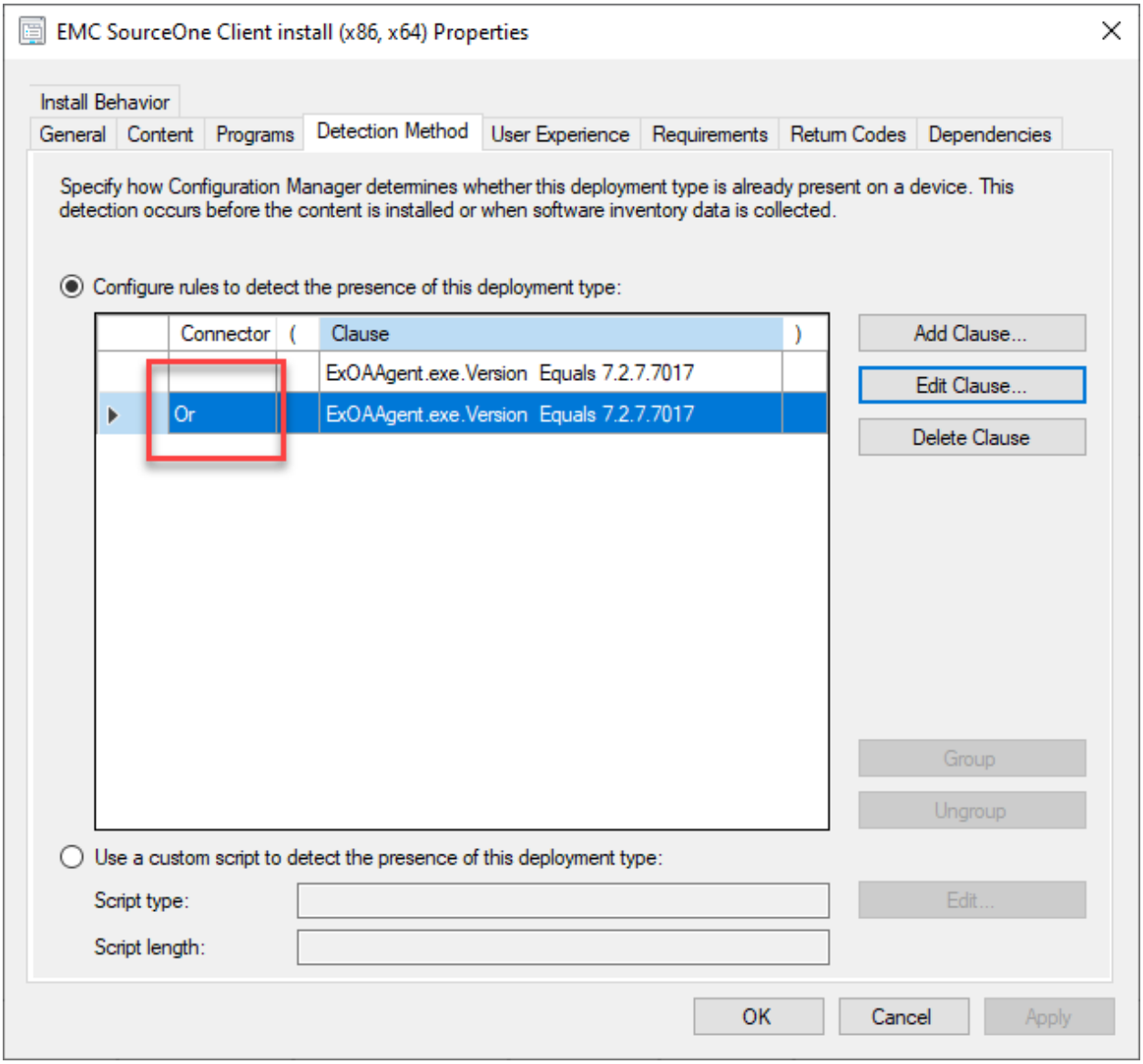
Rule1: Detect the 32-bit install:



Rule2: Detect the 64-bit install:



Link with an OR clause:



PowerShell Method

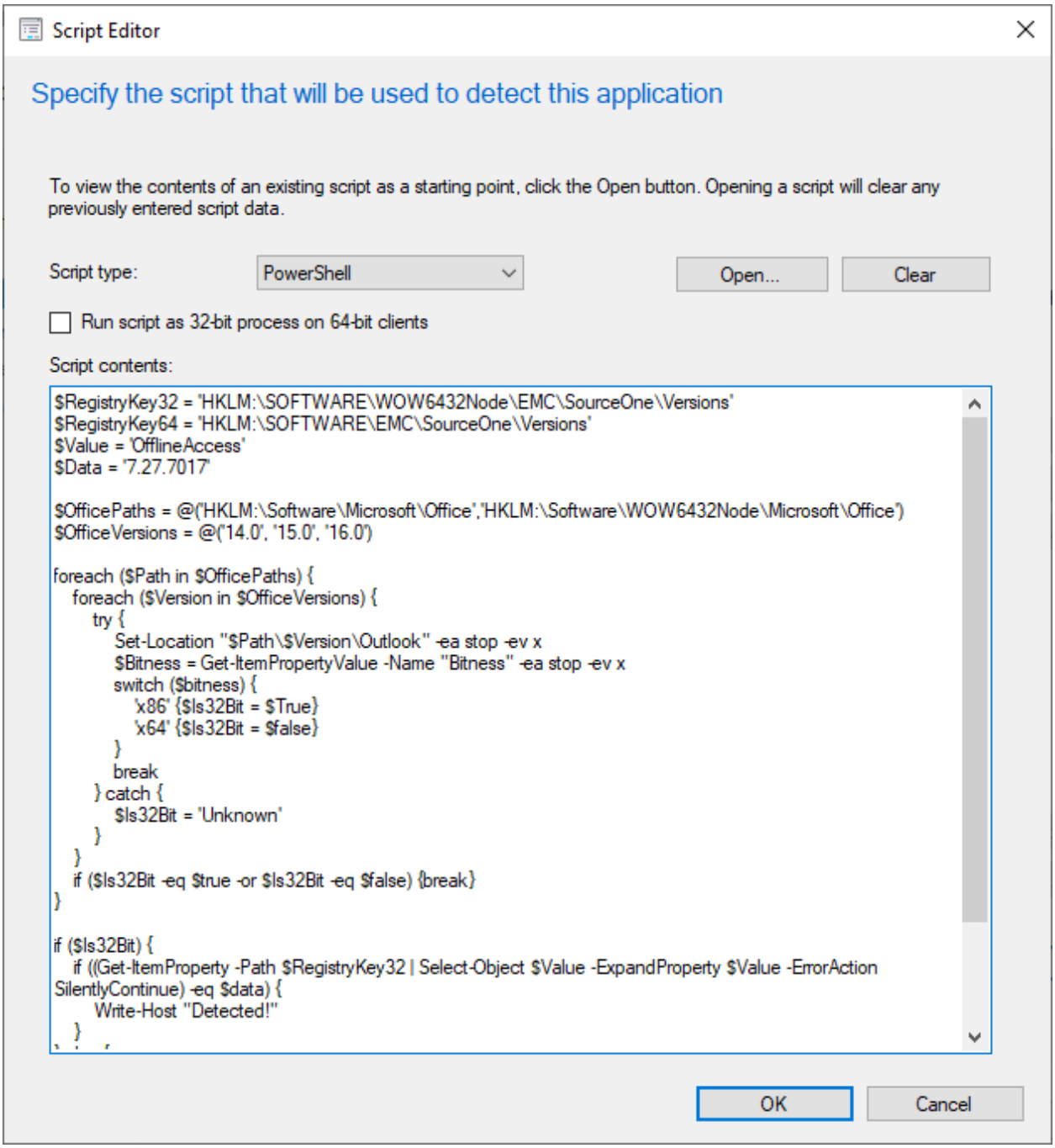
This is the method I chose, and it can be more complicated if you are unsure as to what you are doing, on the other hand, it can give you more flexibility if required.

The detection rule here is the modified code that I copied from the deployment template. You can download it from my github account here:

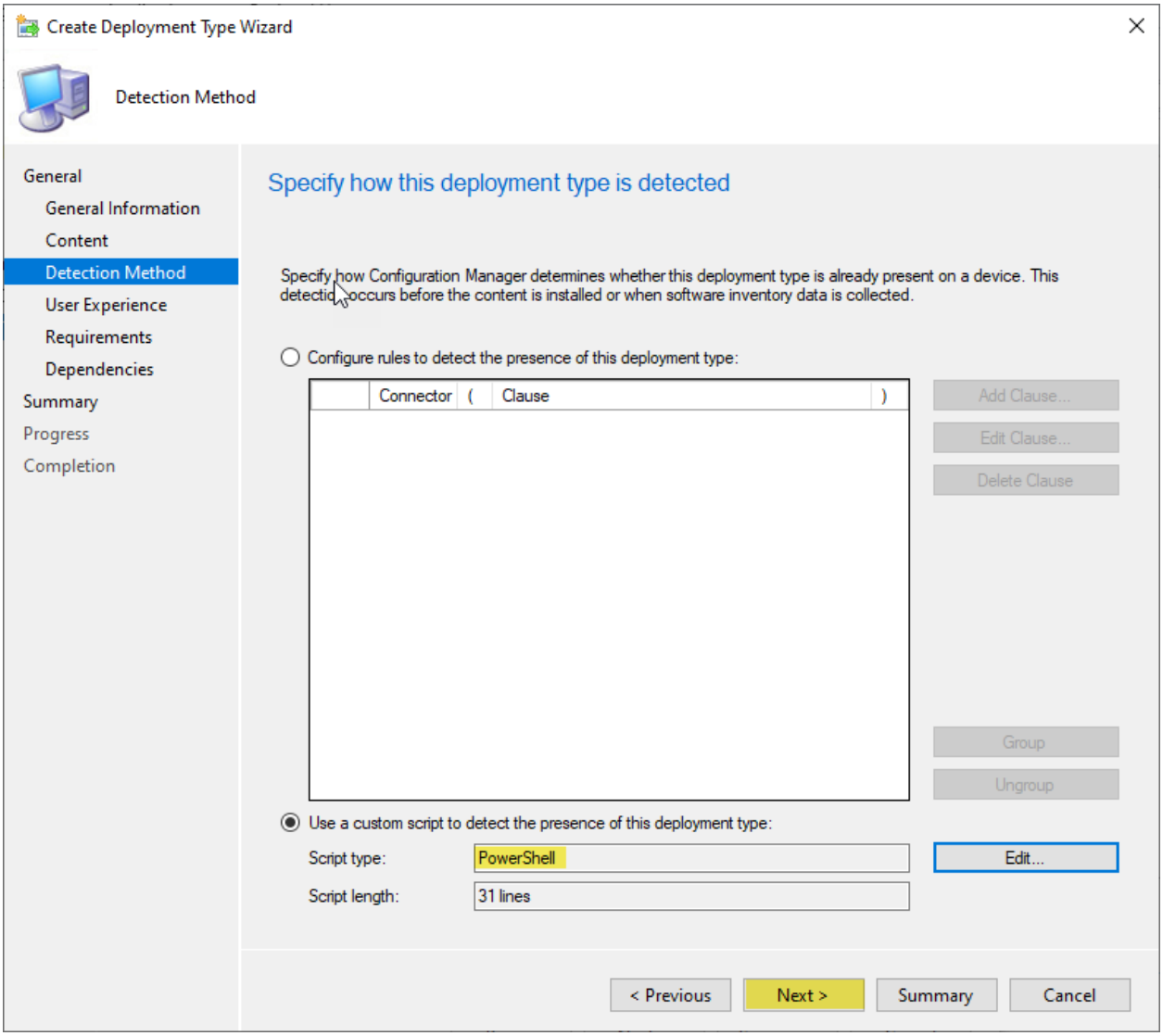
https://github.com/ozthe2/Powershell/tree/master/SCCM/SourceOne

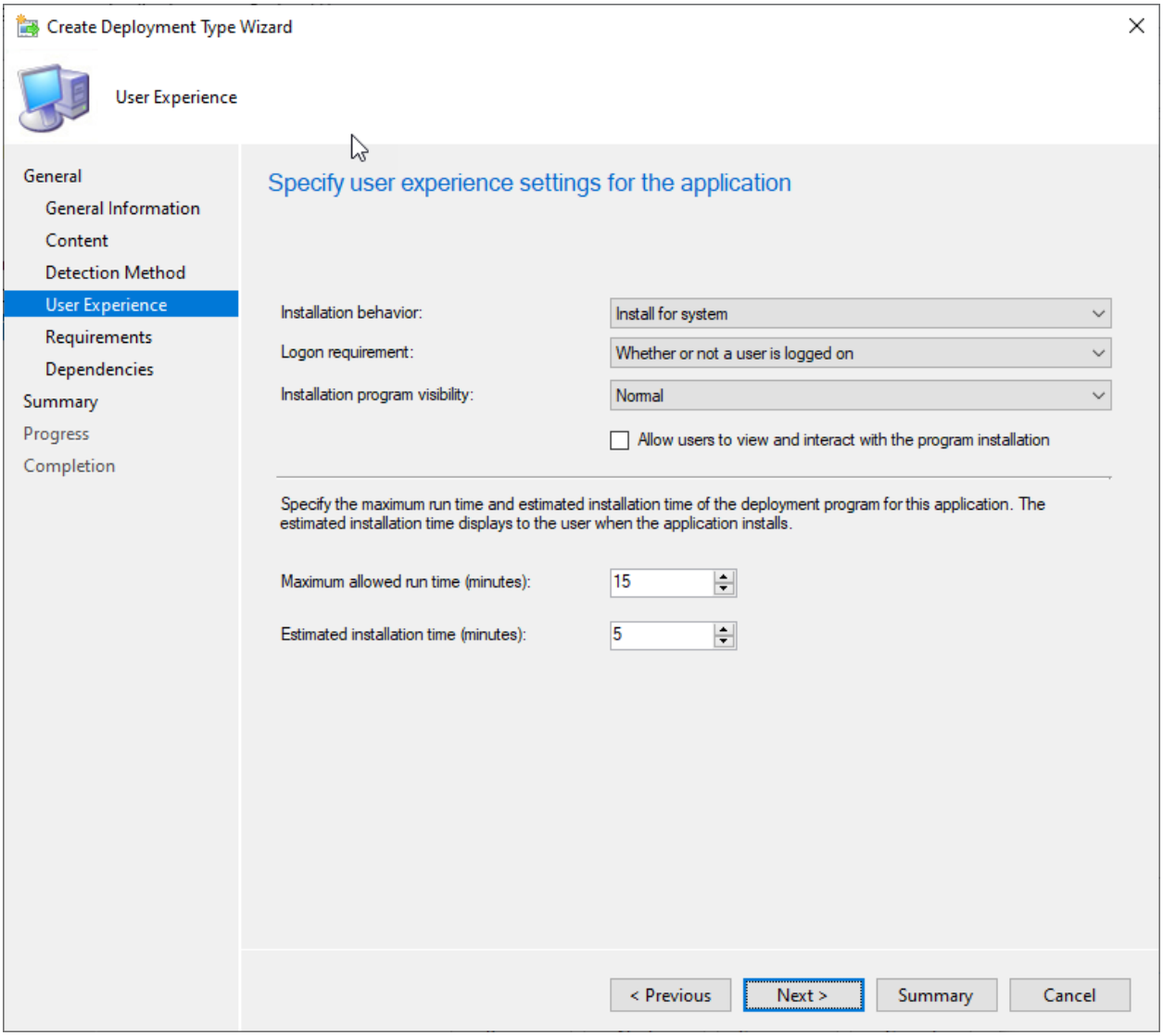
…and copy \ paste it in to the dialog box.

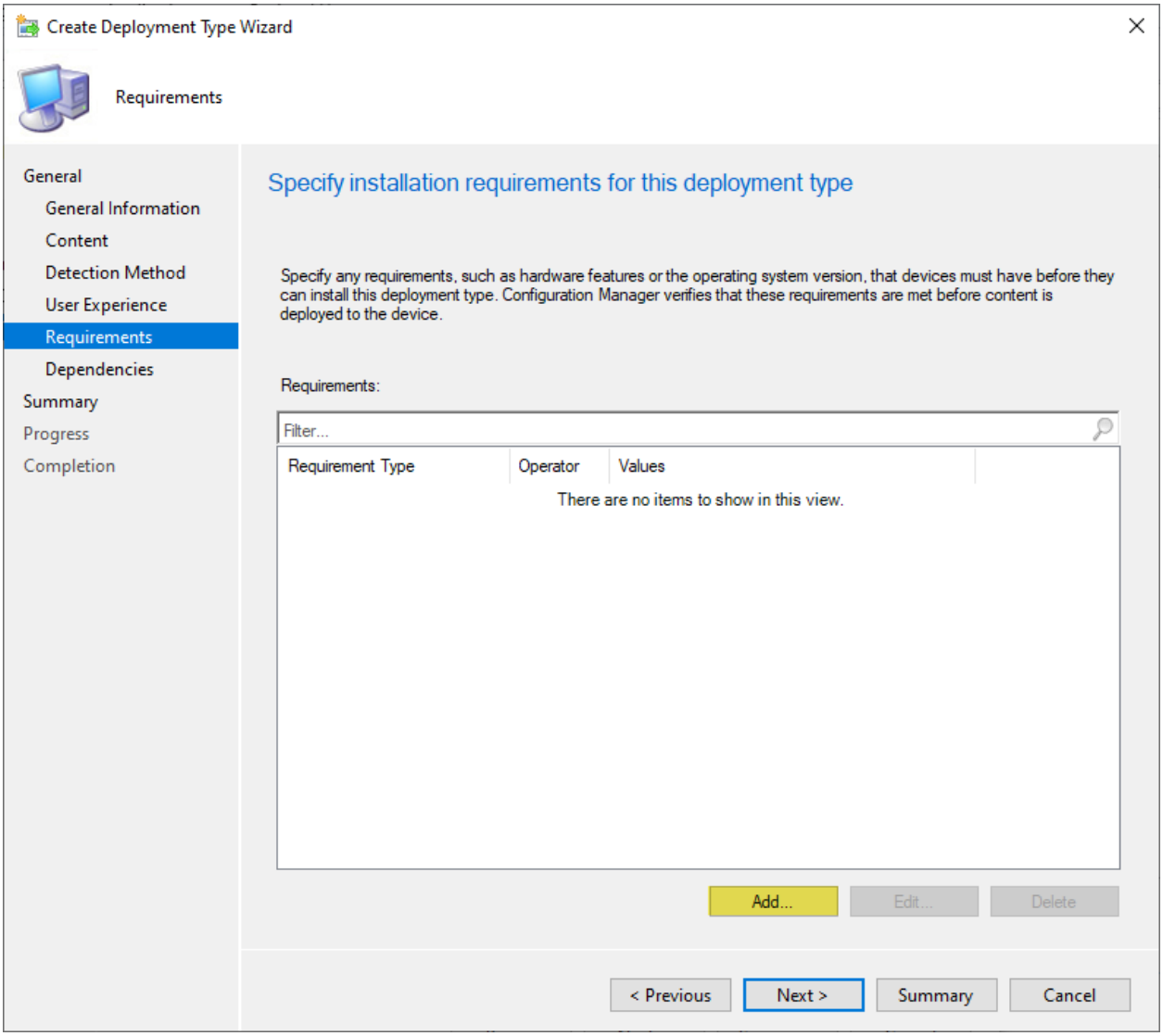
Using PowerShell Detection:

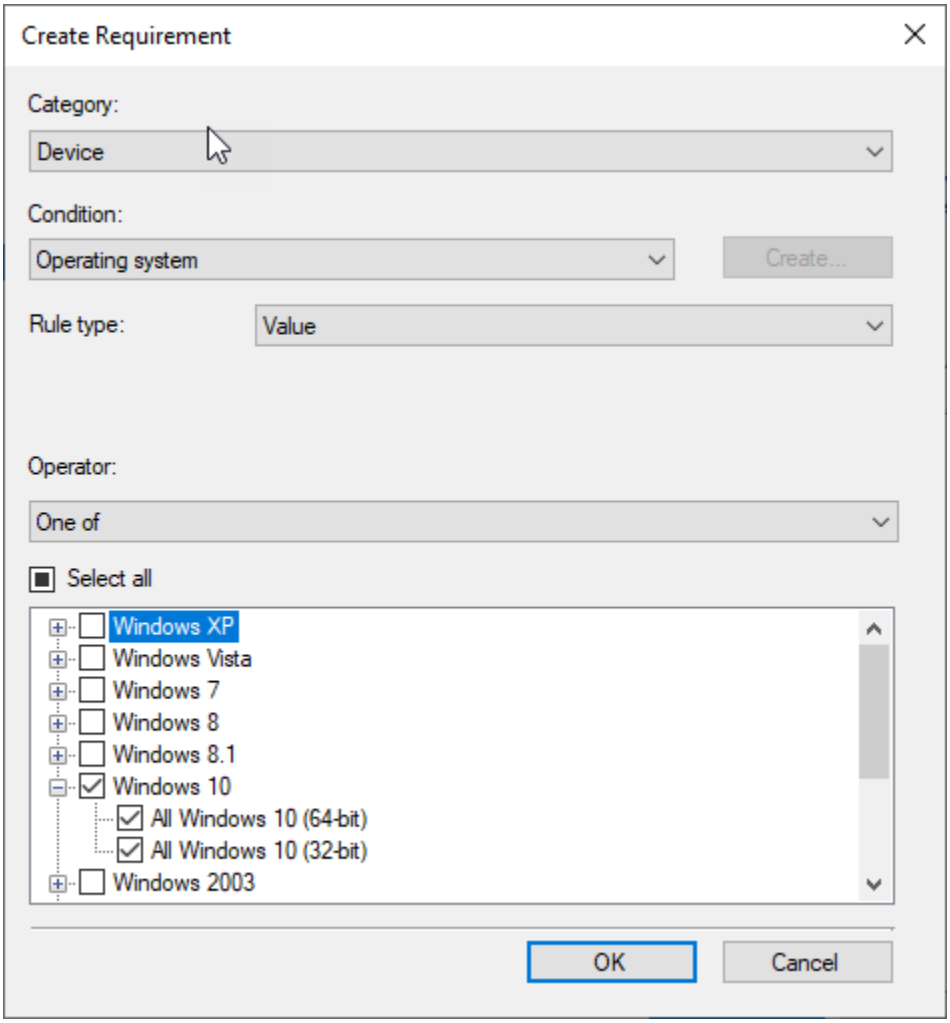


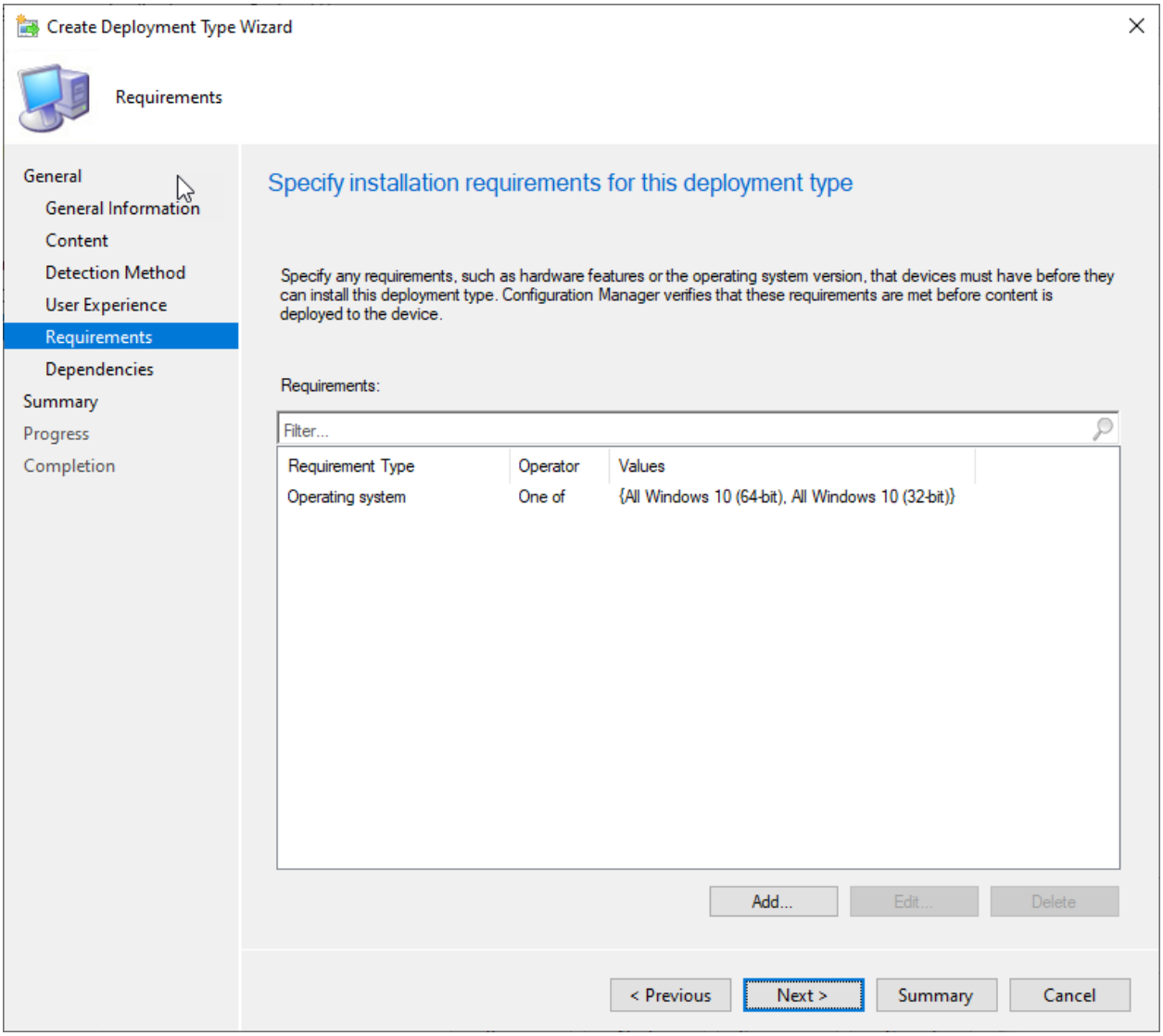
PowerShell Detection Clause

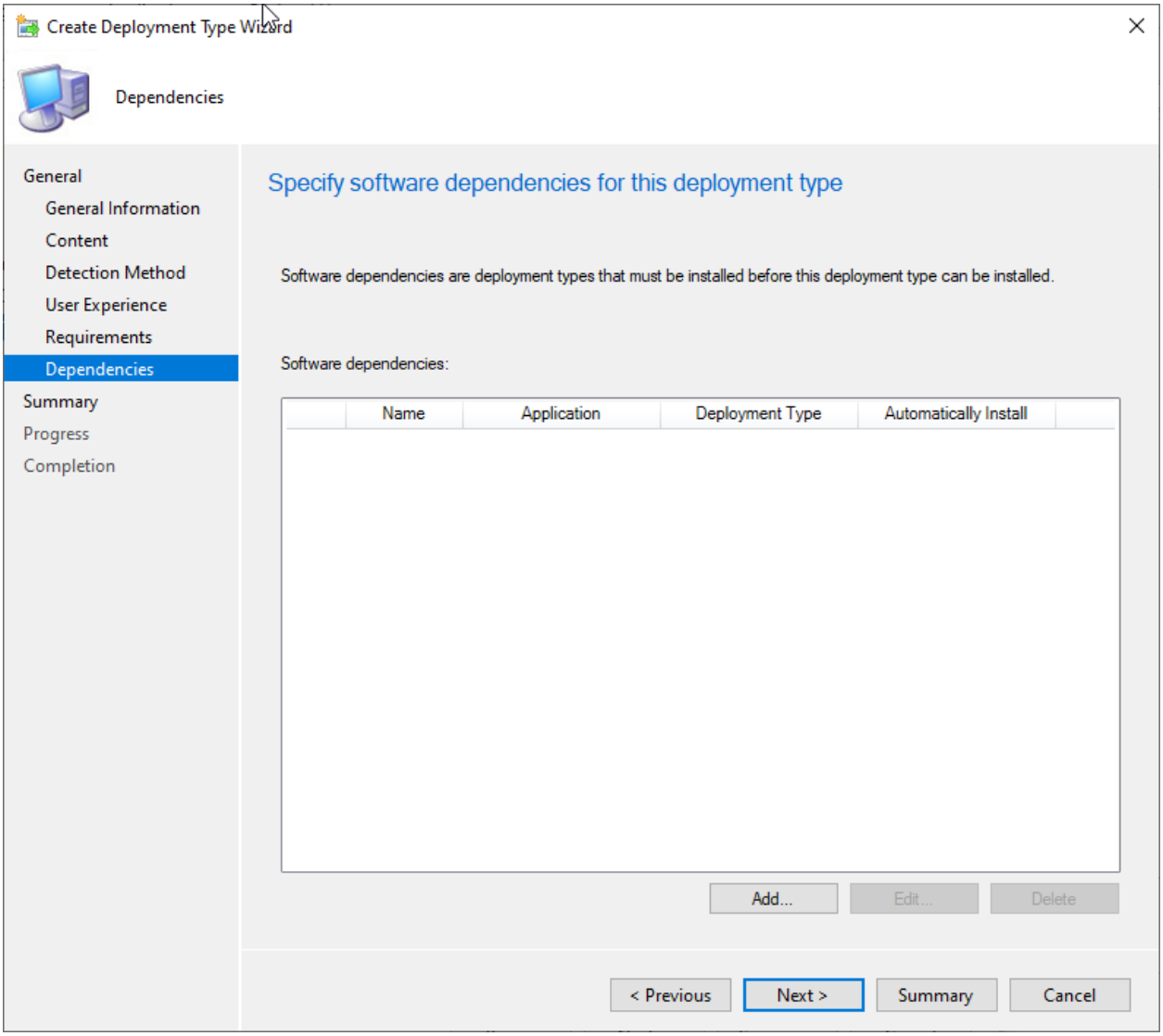

Create Deployment Type Wizard
Detection Method
General
General Information
Content
Detection Method
User Experience
Requirements
Dependencies
Summary
Progress
Completion
Specify how this deployment type is detected
Specify how Configuration Manager determines whether this deployment type is already present on a device. This detection occurs before the content is installed or when software inventory data is collected.
Configure rules to detect the presence of this deployment type:
Connector (Clause)
Add Clause...
Edit Clause...
Delete Clause
Group
Ungroup
Use a custom script to detect the presence of this deployment type:
Script type: PowerShell
Script length: 31 lines
Edit...
< Previous
Next >
Summary
Cancel

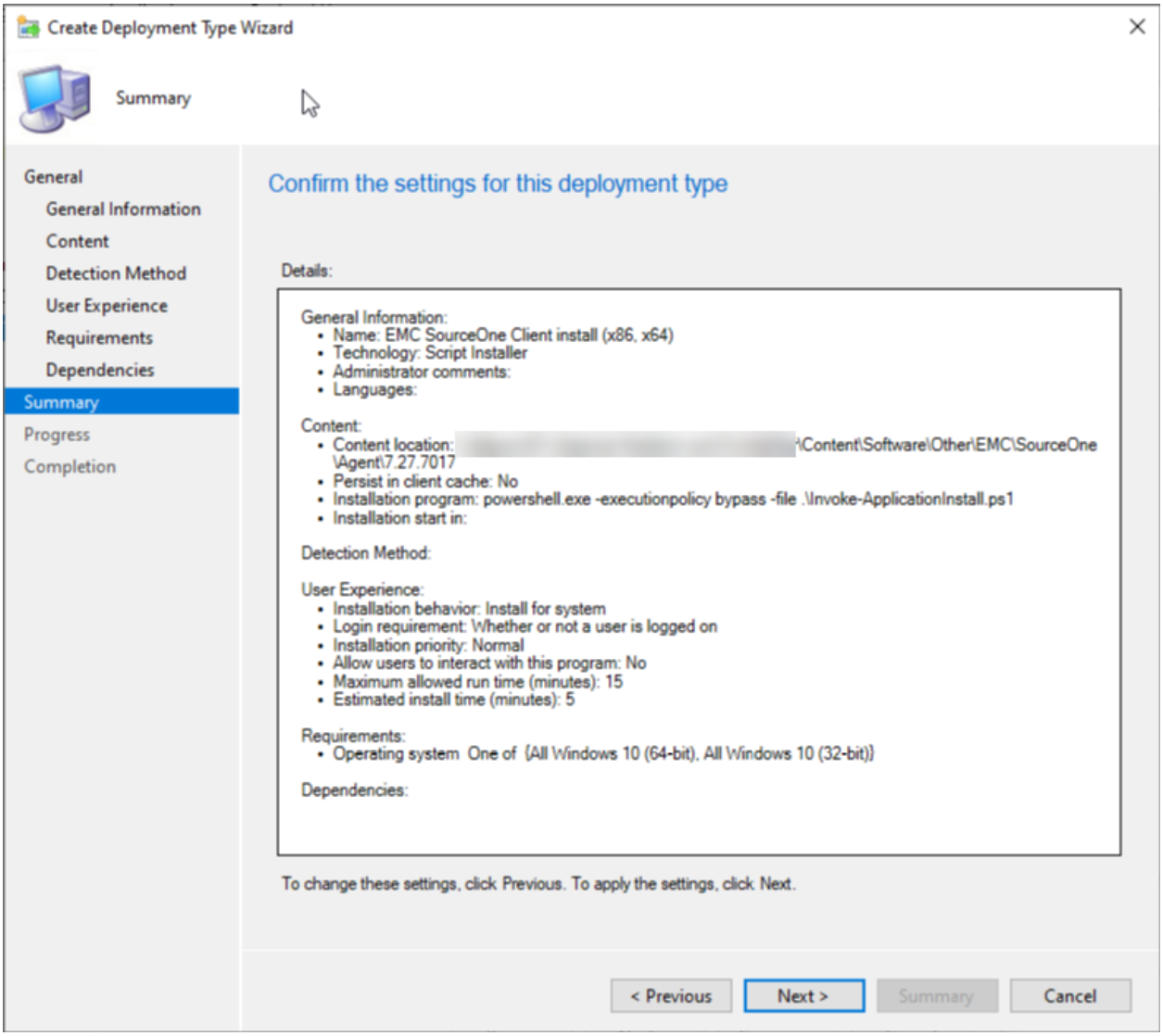

Create Deployment Type Wizard
User Experience
General
General Information
Content
Detection Method
User Experience
Requirements
Dependencies
Summary
Progress
Completion
Specify user experience settings for the application
Installation behavior:
Install for system
Logon requirement:
Whether or not a user is logged on
Installation program visibility:
Normal
Allow users to view and interact with the program installation
Specify the maximum run time and estimated installation time of the deployment program for this application. The estimated installation time displays to the user when the application installs.
Maximum allowed run time (minutes):
15
Estimated installation time (minutes):
5
< Previous
Next >
Summary
Cancel

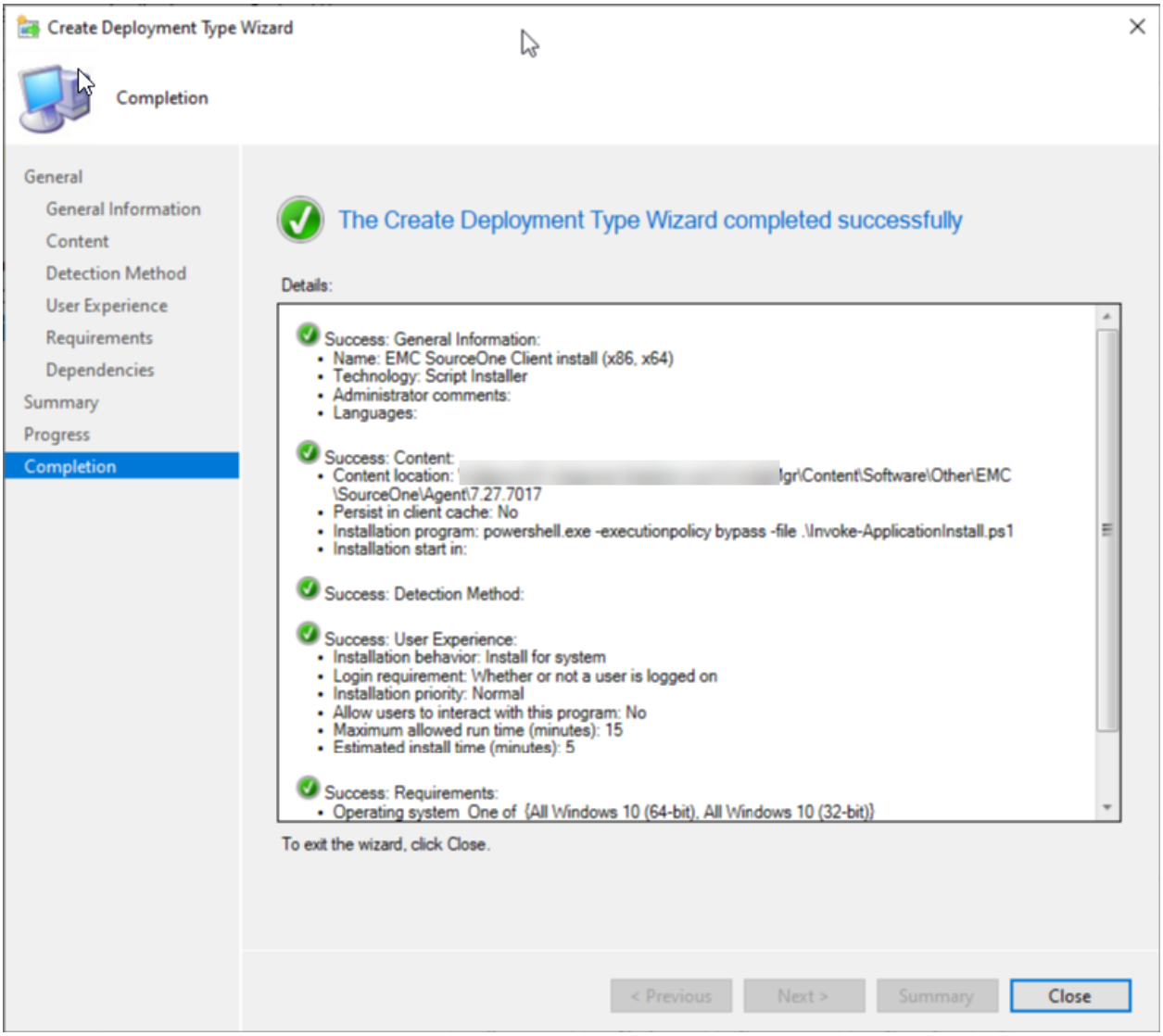

Create Deployment Type Wizard
Requirements
General
General Information
Content
Detection Method
User Experience
Requirements
Dependencies
Summary
Progress
Completion
Specify installation requirements for this deployment type
Specify any requirements, such as hardware features or the operating system version, that devices must have before they can install this deployment type. Configuration Manager verifies that these requirements are met before content is deployed to the device.
Requirements:
Filter...
Requirement Type
Operator
Values
There are no items to show in this view.
Add...
Edit...
Delete
< Previous
Next >
Summary
Cancel

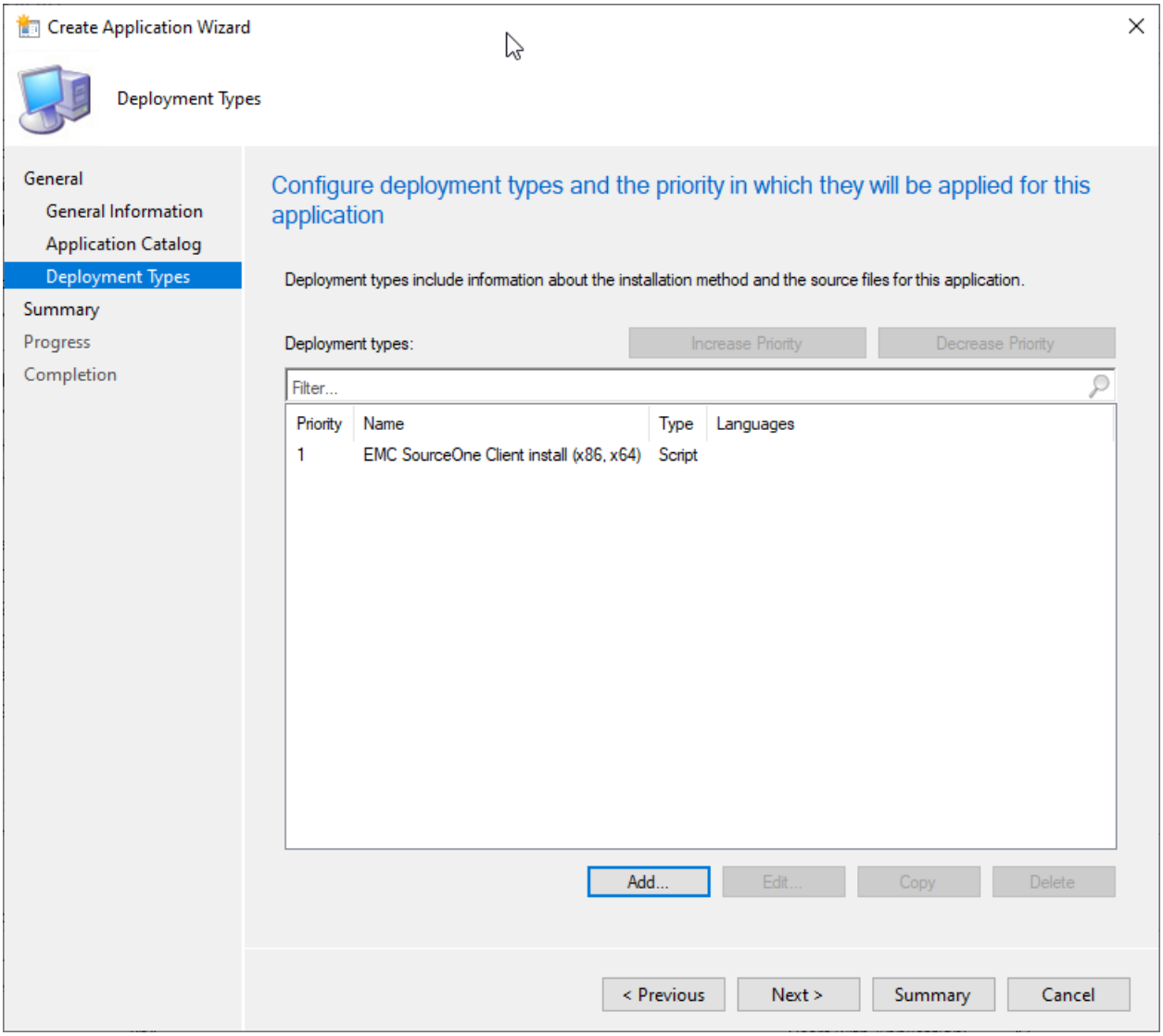

Create Requirement
Category:
Device
Condition:
Operating system
Create...
Rule type:
Value
Operator:
One of
Select all
Windows XP
Windows Vista
Windows 7
Windows 8
Windows 8.1
Windows 10
All Windows 10 (64-bit)
All Windows 10 (32-bit)
Windows 2003
OK
Cancel

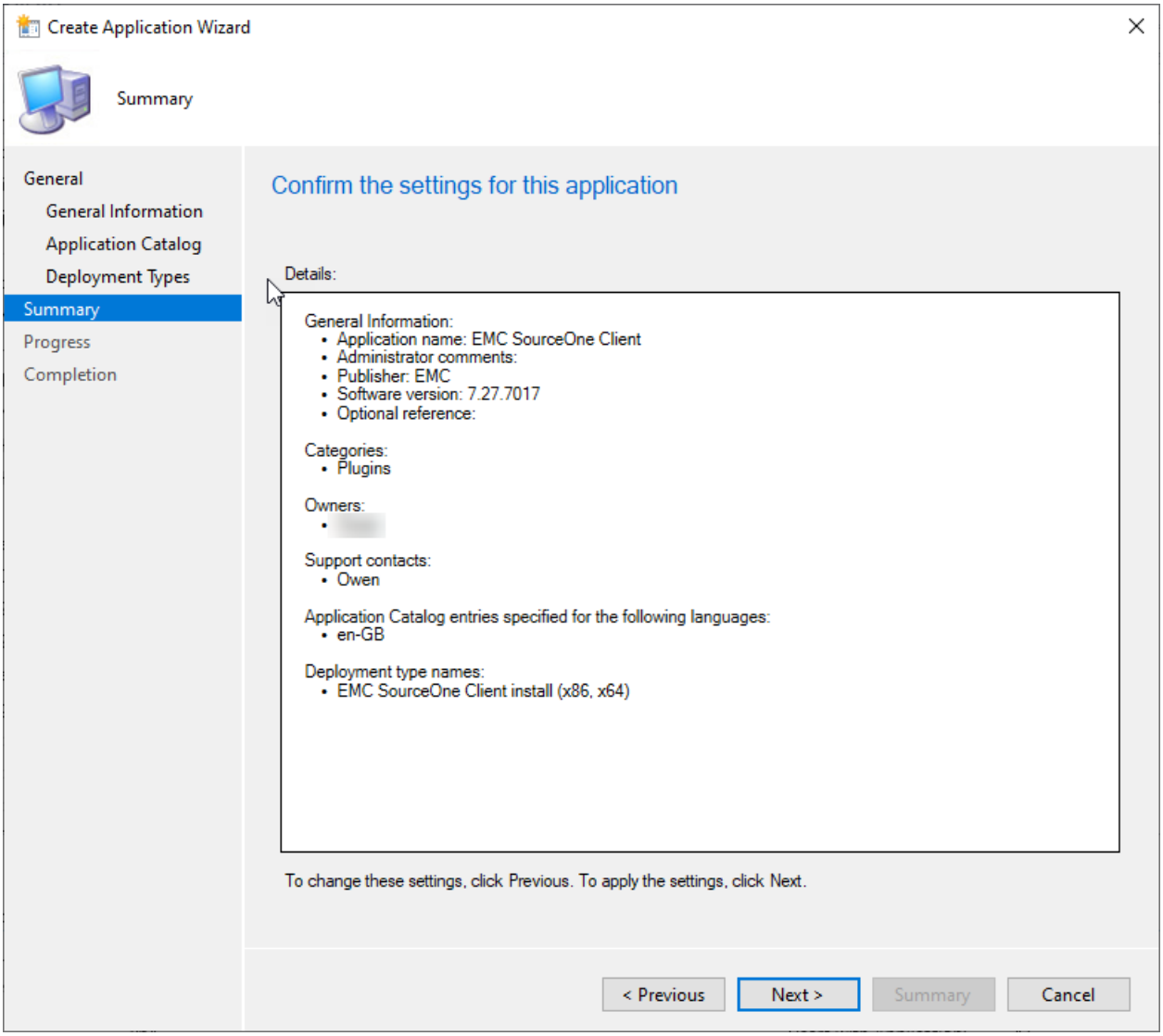

Create Deployment Type Wizard
Requirements
General
General Information
Content
Detection Method
User Experience
Requirements
Dependencies
Summary
Progress
Completion
Specify installation requirements for this deployment type
Specify any requirements, such as hardware features or the operating system version, that devices must have before they can install this deployment type. Configuration Manager verifies that these requirements are met before content is deployed to the device.
Requirements:
Filter...
Requirement Type
Operator
Values
Operating system
One of
{All Windows 10 (64-bit), All Windows 10 (32-bit)}
Add...
Edit...
Delete
< Previous
Next >
Summary
Cancel

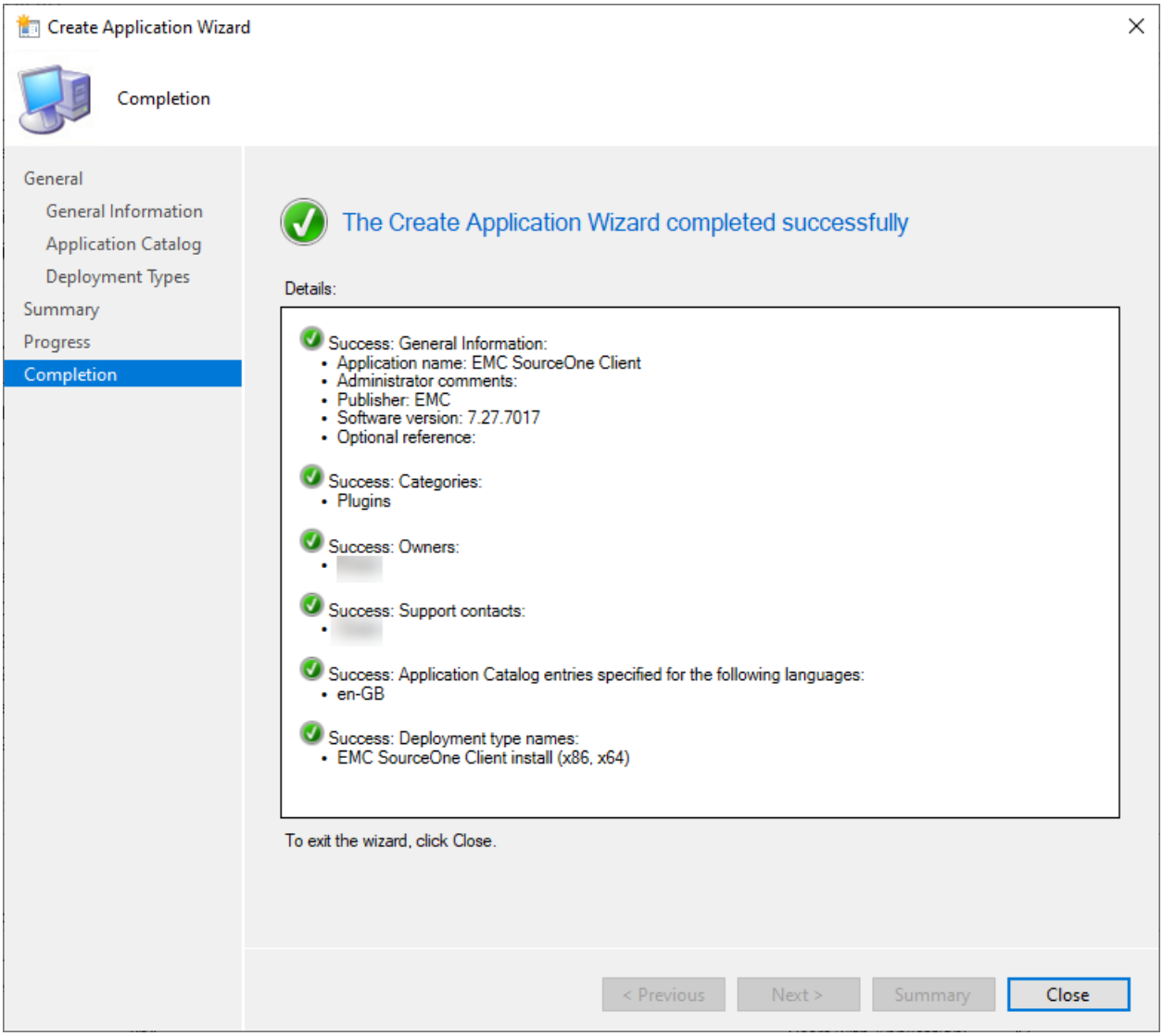

Create Deployment Type Wizard
Dependencies
General
General Information
Content
Detection Method
User Experience
Requirements
Dependencies
Summary
Progress
Completion
Specify software dependencies for this deployment type
Software dependencies are deployment types that must be installed before this deployment type can be installed.
Software dependencies:
Name
Application
Deployment Type
Automatically Install
Add...
Edit...
Delete
< Previous
Next >
Summary
Cancel


Create Deployment Type Wizard
Summary
General
General Information
Content
Detection Method
User Experience
Requirements
Dependencies
Summary
Progress
Completion
Confirm the settings for this deployment type
Details:
General Information:
• Name: EMC SourceOne Client install (x86, x64)
• Technology: Script Installer
• Administrator comments:
• Languages:
Content:
• Content location: [redacted]\Content\Software\Other\EMC\SourceOne\Agent\7.27.7017
• Persist in client cache: No
• Installation program: powershell.exe -executionpolicy bypass -file .\Invoke-ApplicationInstall.ps1
• Installation start in:
Detection Method:
User Experience:
• Installation behavior: Install for system
• Login requirement: Whether or not a user is logged on
• Installation priority: Normal
• Allow users to interact with this program: No
• Maximum allowed run time (minutes): 15
• Estimated install time (minutes): 5
Requirements:
• Operating system One of {All Windows 10 (64-bit), All Windows 10 (32-bit)}
Dependencies:
To change these settings, click Previous. To apply the settings, click Next.
< Previous
Next >
Summary
Cancel


Create Deployment Type Wizard
Completion
General
General Information
Content
Detection Method
User Experience
Requirements
Dependencies
Summary
Progress
Completion
The Create Deployment Type Wizard completed successfully
Details:
Success: General Information:
Name: EMC SourceOne Client install (x86, x64)
Technology: Script Installer
Administrator comments:
Languages:
Success: Content:
Content location: [redacted]lgr\Content\Software\Other\EMC \SourceOne\Agent\7.27.7017
Persist in client cache: No
Installation program: powershell.exe -executionpolicy bypass -file .\Invoke-ApplicationInstall.ps1
Installation start in:
Success: Detection Method:
Success: User Experience:
Installation behavior: Install for system
Login requirement: Whether or not a user is logged on
Installation priority: Normal
Allow users to interact with this program: No
Maximum allowed run time (minutes): 15
Estimated install time (minutes): 5
Success: Requirements:
Operating system One of {All Windows 10 (64-bit), All Windows 10 (32-bit)}
To exit the wizard, click Close.
< Previous
Next >
Summary
Close


Create Application Wizard
Deployment Types
General
General Information
Application Catalog
Deployment Types
Summary
Progress
Completion
Configure deployment types and the priority in which they will be applied for this application
Deployment types include information about the installation method and the source files for this application.
Deployment types:
Increase Priority
Decrease Priority
Filter...
Priority
Name
Type
Languages
1
EMC SourceOne Client install (x86, x64)
Script
Add...
Edit...
Copy
Delete
< Previous
Next >
Summary
Cancel


Create Application Wizard
Summary
General
General Information
Application Catalog
Deployment Types
Summary
Progress
Completion
Confirm the settings for this application
Details:
General Information:
• Application name: EMC SourceOne Client
• Administrator comments:
• Publisher: EMC
• Software version: 7.27.7017
• Optional reference:
Categories:
• Plugins
Owners:
•
Support contacts:
• Owen
Application Catalog entries specified for the following languages:
• en-GB
Deployment type names:
• EMC SourceOne Client install (x86, x64)
To change these settings, click Previous. To apply the settings, click Next.
< Previous
Next >
Summary
Cancel



Lock and Load

That's it! Now you just need to distribute the application and deploy it!

Summary

By using the Deployment Template we have saved ourselves a ton of time. Instead of having to create two applications and create a dependency link (The SourceOne executable *and* the Hotfix executable) we've done it in one application.

Once you have used the Deployment Template a few times, this becomes a breeze. Adding pre and post-deployment tasks…such as creating file structures in the pre-deployment phase, then deploying your application, then dumping or modifying files in the post-deployment phase…it becomes a walk in the park.

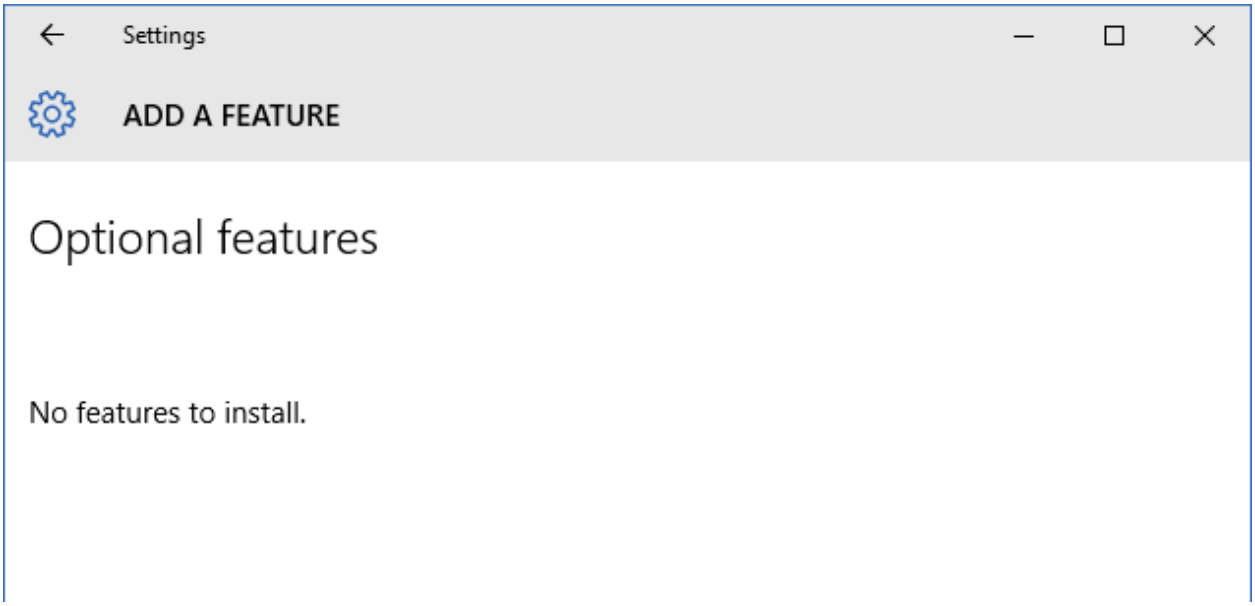
Bonus Chapter 3

A Step-by-Step Guide to Deploying RSAT Components for Windows 10

Background

If you have Windows 10 version 1809 or higher then you will know that you no longer download the Remote Server Administration Tools (RSAT) (or as we like to call it in my I.T department: "Arse-Hat"), instead it's now installed as an optional feature.

Or you may have encountered the dreaded, "No features to install." message:



Well have no fear!

Get the Script

PowerShell to the Rescue!

Download the script from my Github repository here:

https://github.com/ozthe2/Powershell/tree/master/SCCM/RSAT

The script I have written, *Manage-RSAT.ps1* contains two functions: *Install-RSATCapabilites* and *Uninstall-RSATCapabilities.*

Install-RSATCapabilities

This function, without any modification, will install the following RSAT capabilities:

- DNS Server Tools
- Group Policy Management Tools
- Active Directory Domain Services and Lightweight Directory Services Tools
- DHCP Server Tools
- File Services Tools
- IP Address Management (IPAM) Client
- Volume Activation Tools
- Active Directory Certificate Services Tools
- Bitlocker Administration Utilities

Uninstall-RSATCapabilities

This function, without any modification, will uninstall the following RSAT capabilities:

- DNS Server Tools
- Group Policy Management Tools
- Active Directory Domain Services and Lightweight Directory Services Tools
- DHCP Server Tools
- File Services Tools
- IP Address Management (IPAM) Client
- Volume Activation Tools
- Active Directory Certificate Services Tools
- Bitlocker Administration Utilities

Season to Taste

In my workplace, these are the RSAT components we require to manage our infrastructure. But what if you need more or less?

It's a simple case of modifying the variable `$Components`

Either add to or remove the capabilities you require \ do not require:

```
[CmdletBinding()]

param()

BEGIN {

    $Components = @('Rsat.Dns.Tools~~~~0.0.1.0','Rsat.GroupPolicy.Management.Tools~~~~0.0.1.0','Rsat.ActiveDirectory.DS-LDS.Too
                    'Rsat.DHCP.Tools~~~~0.0.1.0','Rsat.FileServices.Tools~~~~0.0.1.0','Rsat.IPAM.Client.Tools~~~~0.0.1.0',
                    'Rsat.VolumeActivation.Tools~~~~0.0.1.0','Rsat.CertificateServices.Tools~~~~0.0.1.0')

    $val = Get-ItemProperty -Path "HKLM:\SOFTWARE\Policies\Microsoft\Windows\WindowsUpdate\AU" -Name "UseWUServer" | select -Exp
    Set-ItemProperty -Path "HKLM:\SOFTWARE\Policies\Microsoft\Windows\WindowsUpdate\AU" -Name "UseWUServer" -Value 0 -ErrorActio
    Restart-Service wuauserv -ErrorAction SilentlyContinue
}
```

Each component to install (or uninstall) is comma separated.

To find out what the component names are, take a look here:

https://docs.microsoft.com/en-us/windows-hardware/manufacture/desktop/features-on-demand-non-language-fod#remote-server-administration-tools-rsat

And copy \ paste the *Capability name* you require into the script:

Feature: Simple Network Management Protocol (SNMP)

- **Description**: This feature includes SNMP agents that monitor the activity in network devices and report to the network console workstation
- **Capability name**: SNMP.Client~~~~0.0.1.0
- **CAB name**: Microsoft-Windows-SNMP-Client-Package~31bf3856ad364e35~amd64~en-US~.cab
- **Satellites**: Yes
- **Install size**: 1.11 MB
- **Availability**: Windows 10, version 1809 and later

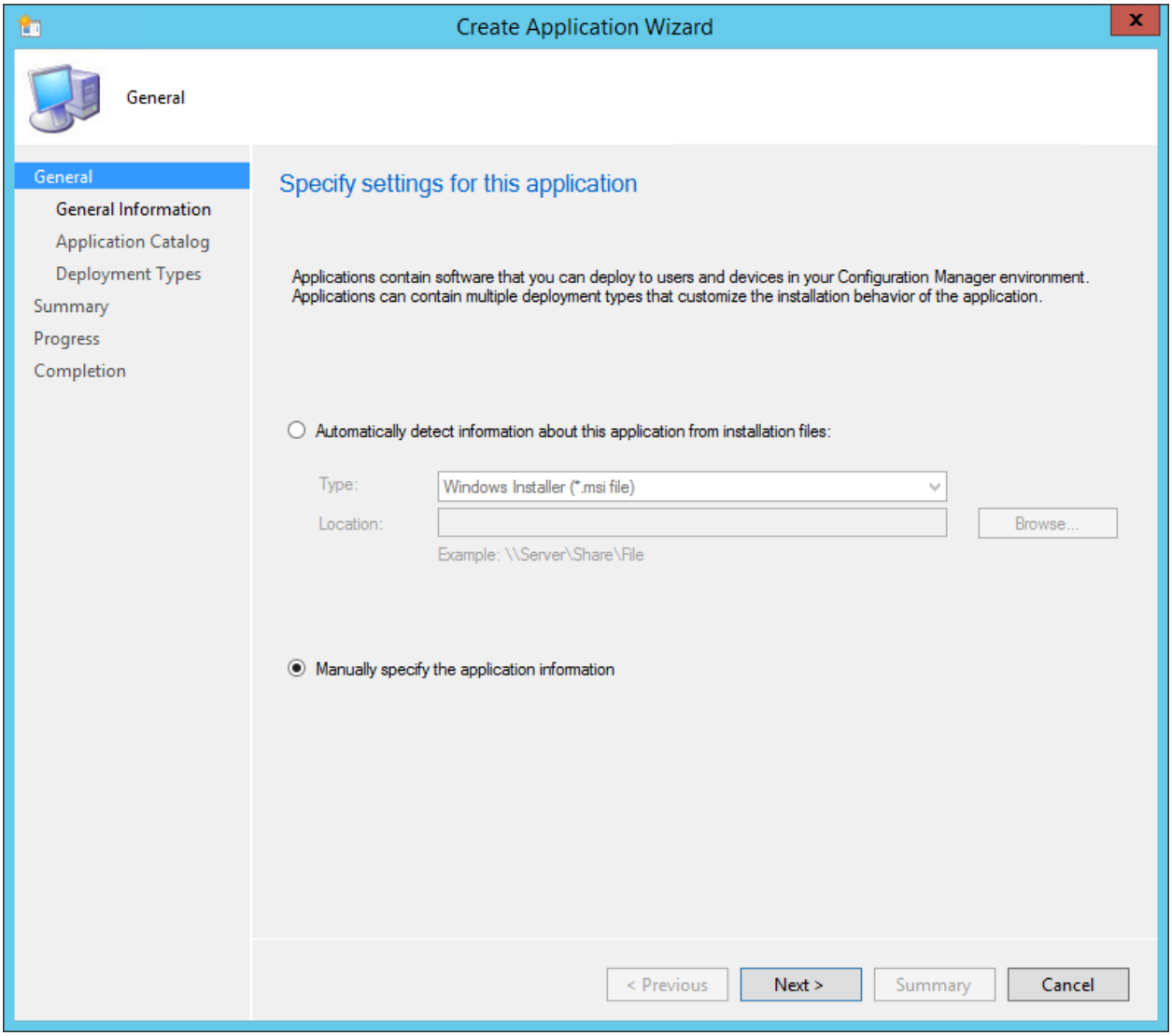
Once you've made this simple adjustment to the script, let's look at how to deploy it…

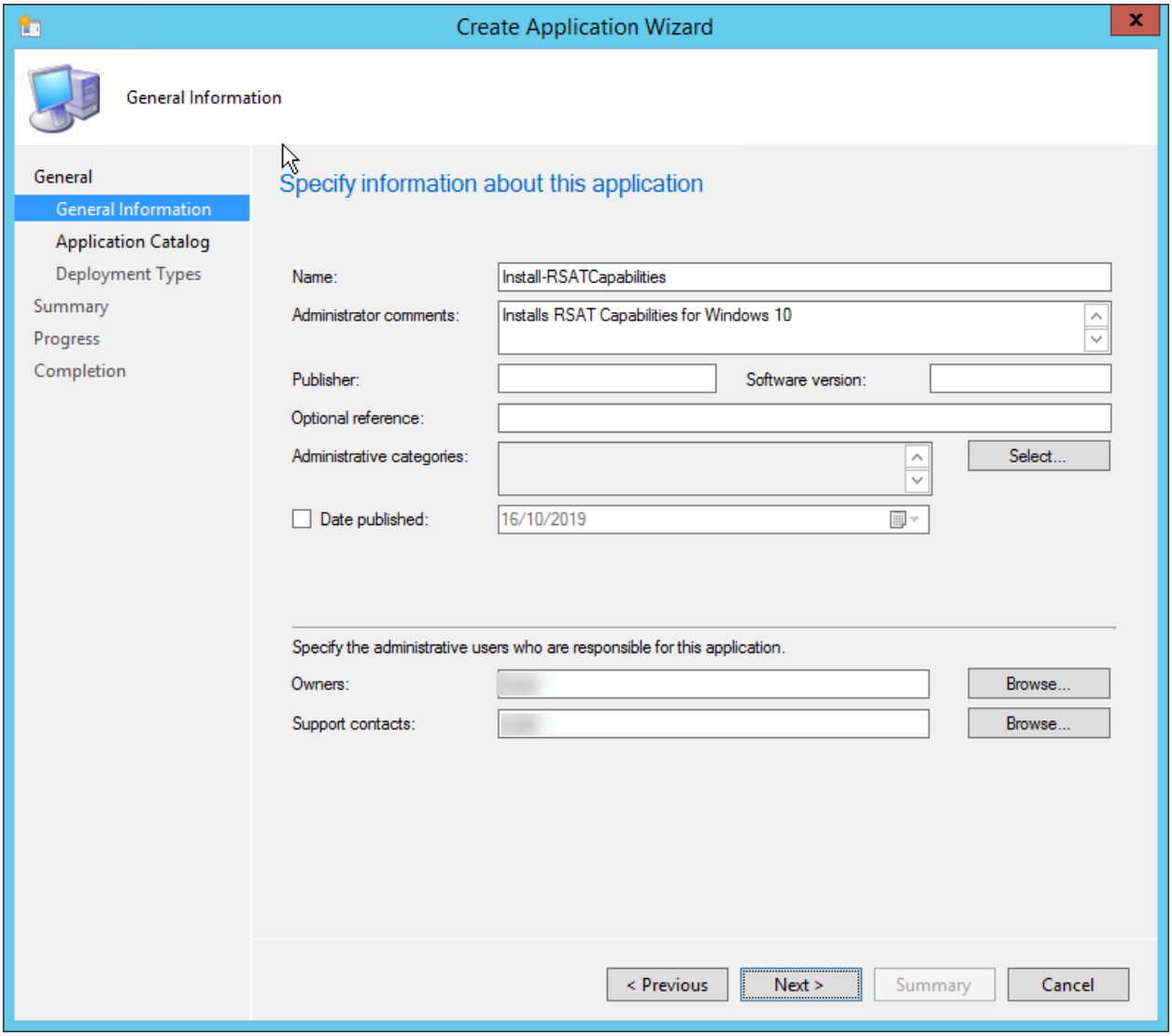
Move the Script to the SCCM Source Location

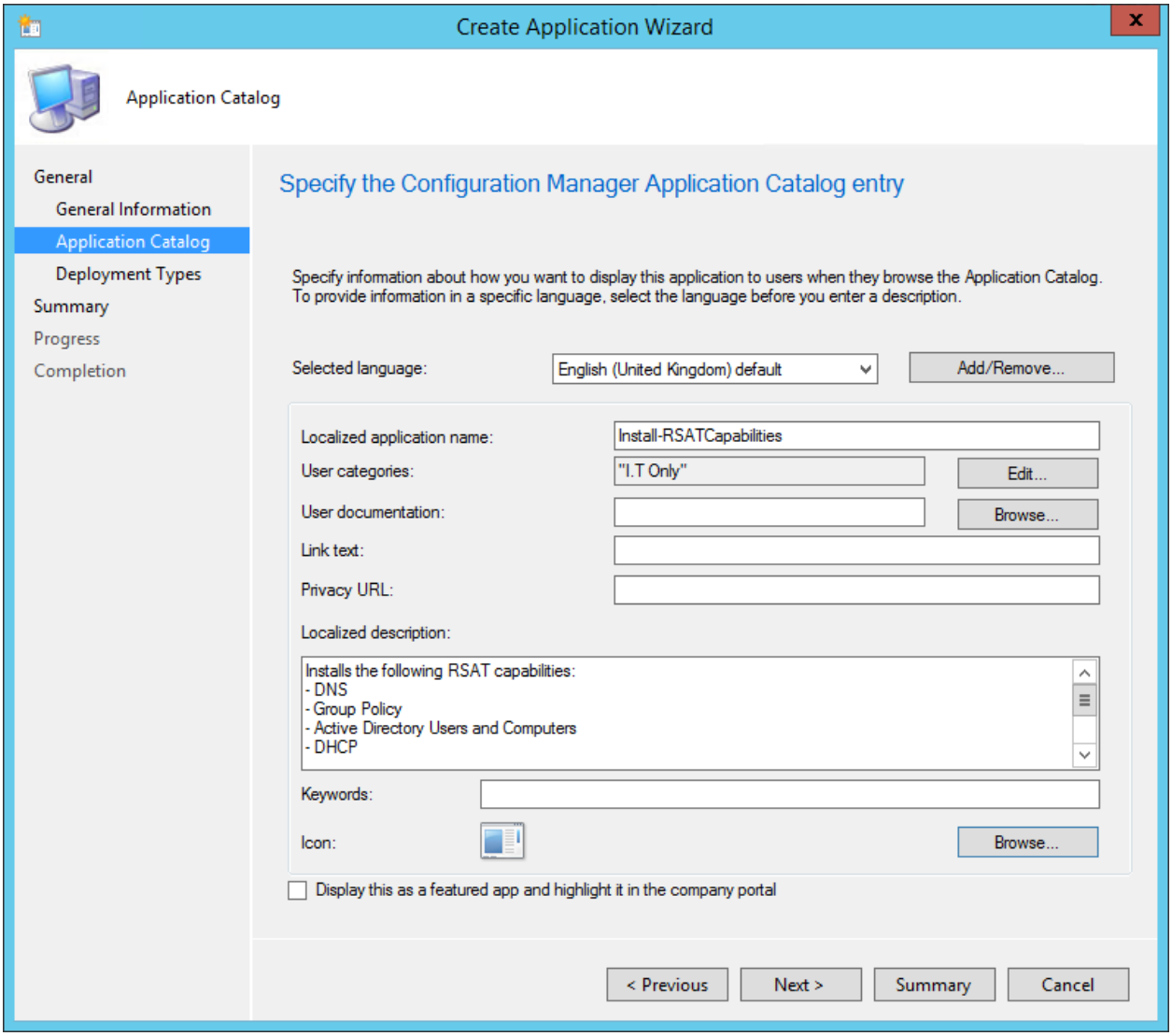
Move or copy the *Manage-RSAT.ps1* script to the location where you would normally store your source files.

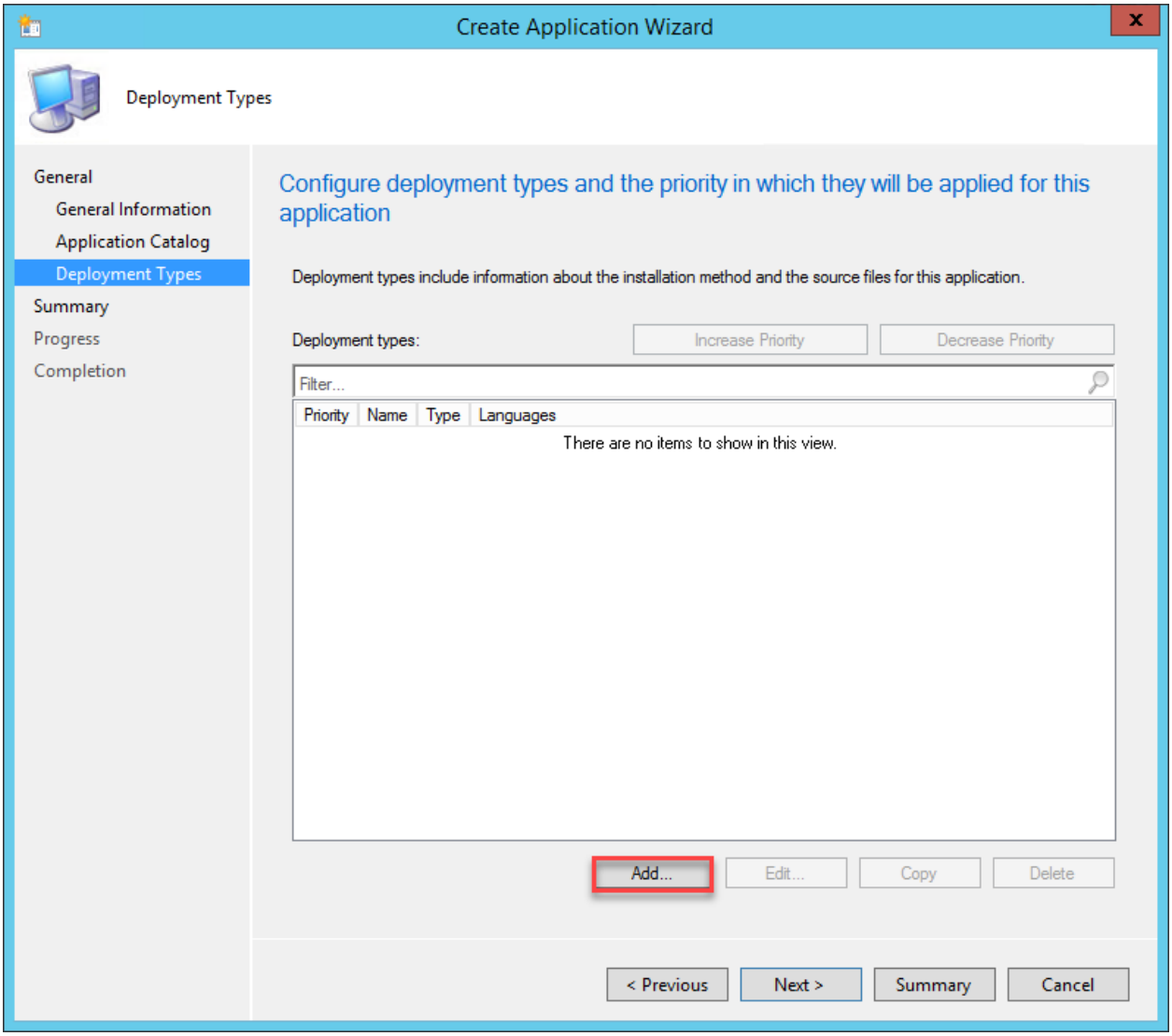
Create The Application

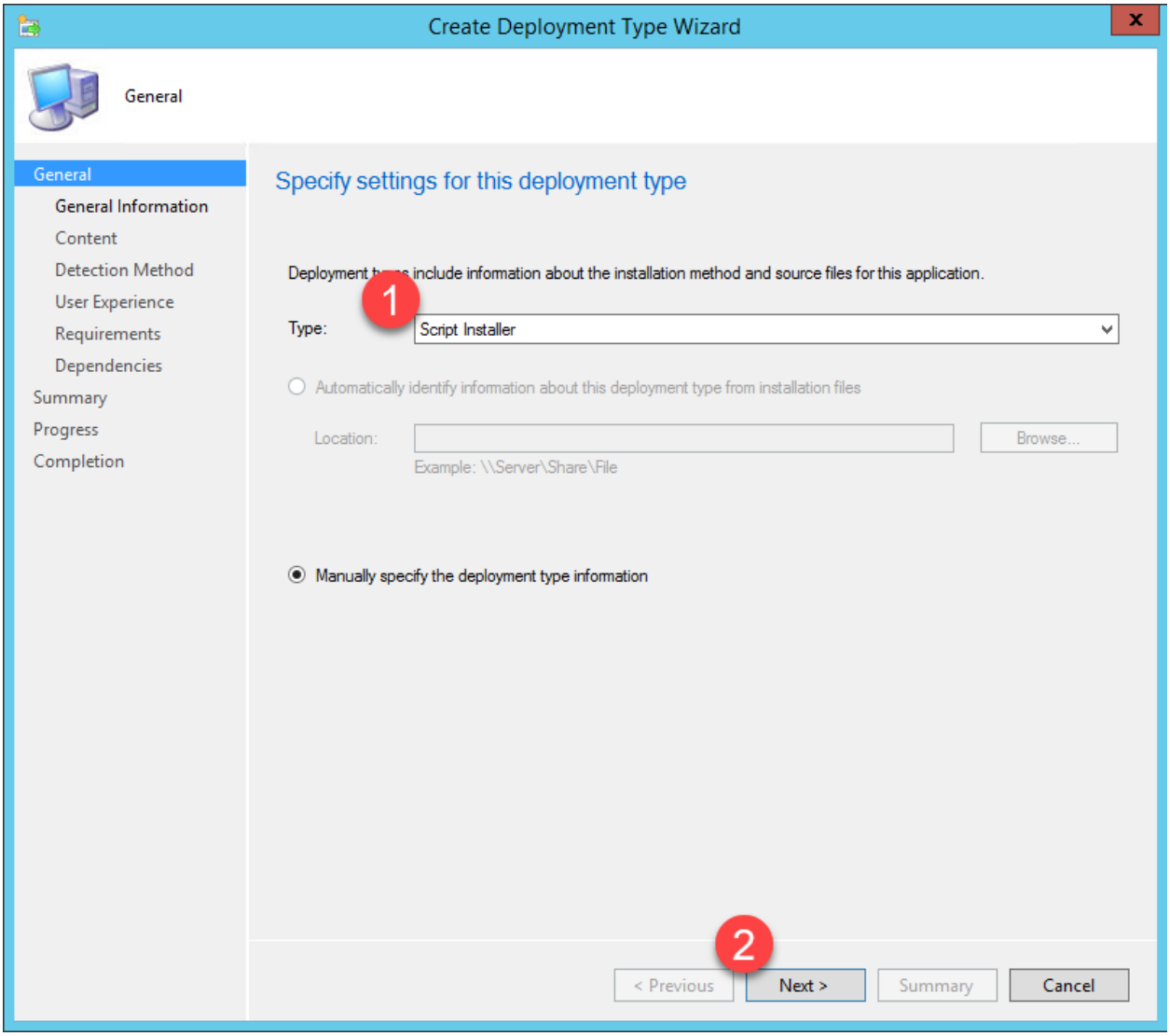
Open your Configuration Manager Console and start the wizard for creating a new application. Once you've done that, hold onto your hats; this won't take long.

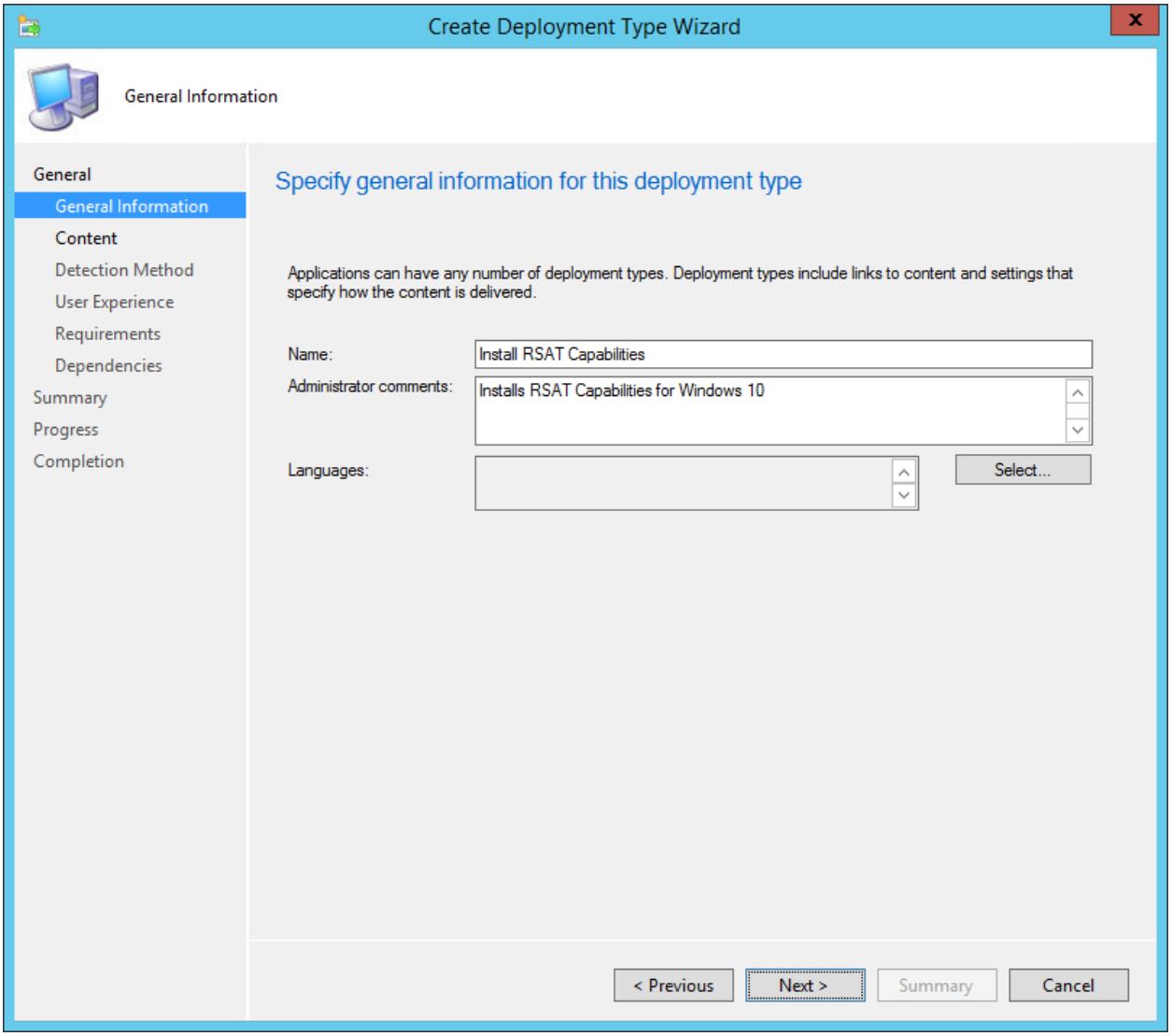

Create Application Wizard
General
General
General Information
Application Catalog
Deployment Types
Summary
Progress
Completion
Specify settings for this application
Applications contain software that you can deploy to users and devices in your Configuration Manager environment. Applications can contain multiple deployment types that customize the installation behavior of the application.
Automatically detect information about this application from installation files:
Type:
Windows Installer (*.msi file)
Location:
Browse...
Example: \\Server\Share\File
Manually specify the application information
< Previous
Next >
Summary
Cancel


Create Application Wizard
General Information
General
General Information
Application Catalog
Deployment Types
Summary
Progress
Completion
Specify information about this application
Name:
Install-RSATCapabilities
Administrator comments:
Installs RSAT Capabilities for Windows 10
Publisher:
Software version:
Optional reference:
Administrative categories:
Select...
Date published:
16/10/2019
Specify the administrative users who are responsible for this application.
Owners:
Browse...
Support contacts:
Browse...
< Previous
Next >
Summary
Cancel


Create Application Wizard
Application Catalog
General
General Information
Application Catalog
Deployment Types
Summary
Progress
Completion
Specify the Configuration Manager Application Catalog entry
Specify information about how you want to display this application to users when they browse the Application Catalog.
To provide information in a specific language, select the language before you enter a description.
Selected language:
English (United Kingdom) default
Add/Remove...
Localized application name:
Install-RSATCapabilities
User categories:
"I.T Only"
Edit...
User documentation:
Browse...
Link text:
Privacy URL:
Localized description:
Installs the following RSAT capabilities:
- DNS
- Group Policy
- Active Directory Users and Computers
- DHCP
Keywords:
Icon:
Browse...
Display this as a featured app and highlight it in the company portal
< Previous
Next >
Summary
Cancel


Create Application Wizard
Deployment Types
General
General Information
Application Catalog
Deployment Types
Summary
Progress
Completion
Configure deployment types and the priority in which they will be applied for this application
Deployment types include information about the installation method and the source files for this application.
Deployment types:
Increase Priority
Decrease Priority
Filter...
Priority
Name
Type
Languages
There are no items to show in this view.
Add...
Edit...
Copy
Delete
< Previous
Next >
Summary
Cancel


Create Deployment Type Wizard
General
General
General Information
Content
Detection Method
User Experience
Requirements
Dependencies
Summary
Progress
Completion
Specify settings for this deployment type
include information about the installation method and source files for this application.
1
Type:
Script Installer
Automatically identify information about this deployment type from installation files
Location:
Browse...
Example: \\Server\Share\File
Manually specify the deployment type information
2
< Previous
Next >
Summary
Cancel


Create Deployment Type Wizard
General Information
General
General Information
Content
Detection Method
User Experience
Requirements
Dependencies
Summary
Progress
Completion
Specify general information for this deployment type
Applications can have any number of deployment types. Deployment types include links to content and settings that specify how the content is delivered.
Name:
Install RSAT Capabilities
Administrator comments:
Installs RSAT Capabilities for Windows 10
Languages:
Select...
< Previous
Next >
Summary
Cancel

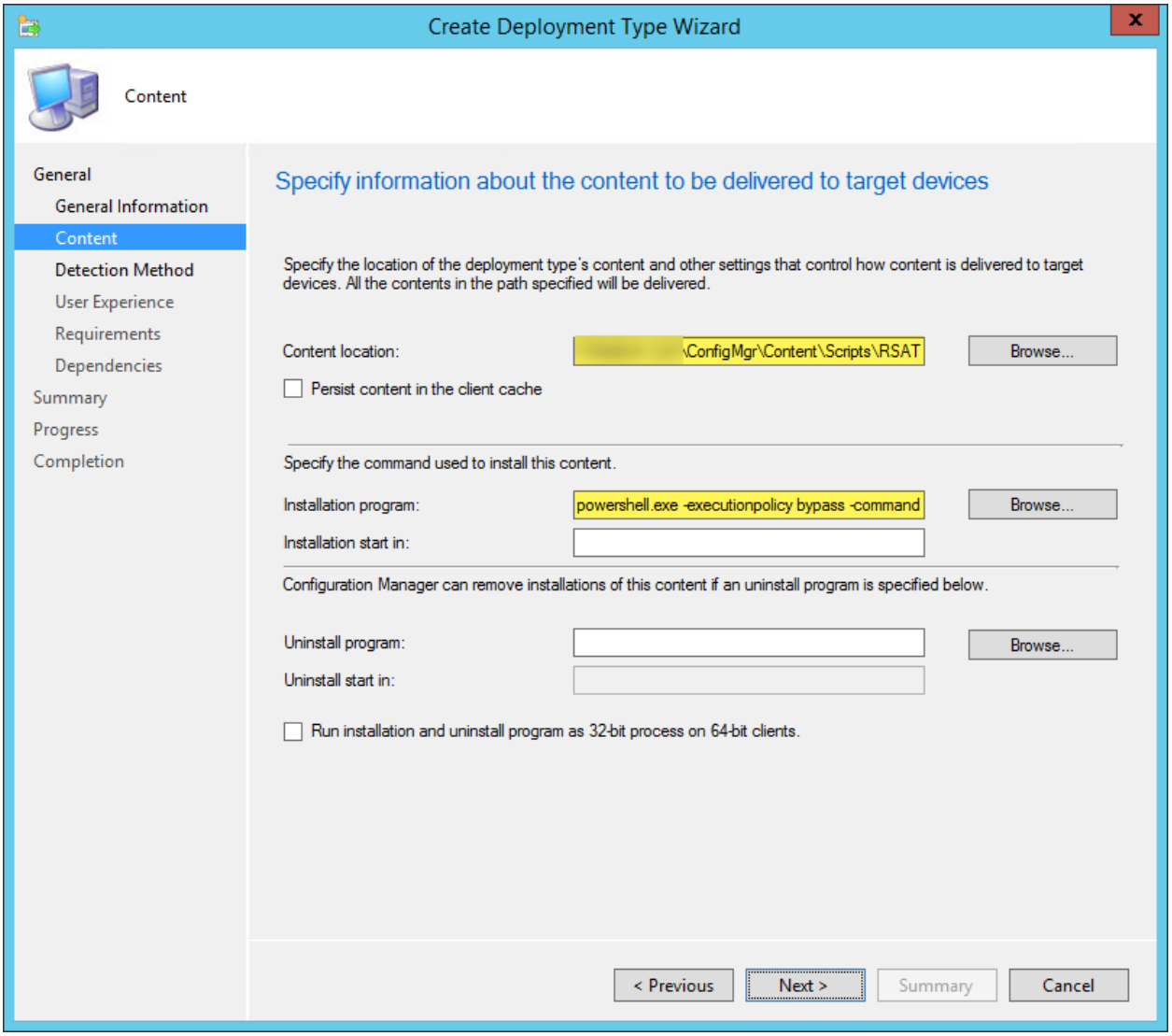
Installation Program

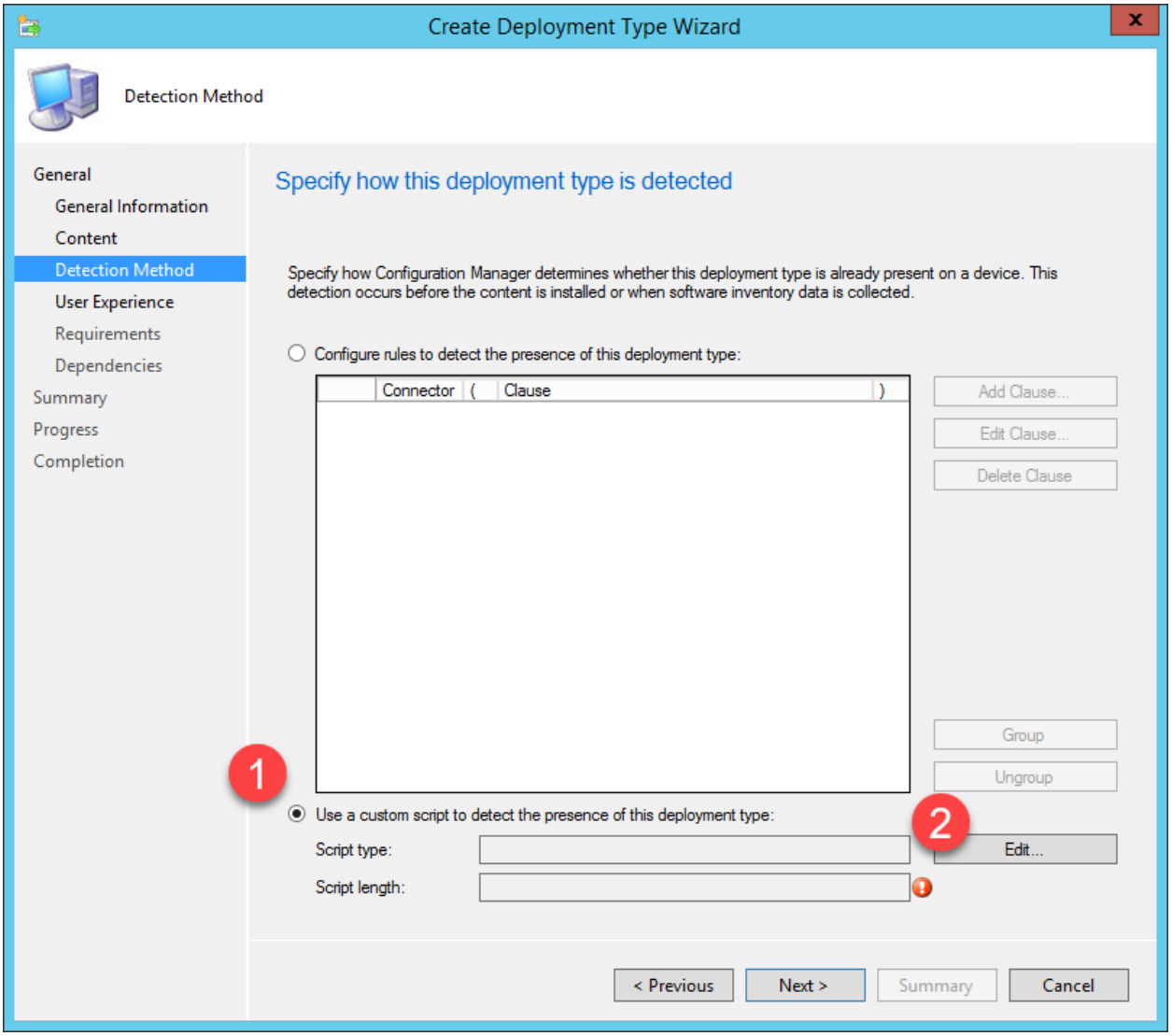
For the *Installation Program* command line, we call the script using the techniques shown earlier in the book (Script with Functions) and the command line used is:

```
powershell.exe -executionpolicy bypass -command "& { . .\ManageRSAT.ps1; Install-RSATCapabilities }"
```

Uninstalling RSAT

If I wanted to uninstall RSAT capabilities, I would call the other function instead: `powershell.exe -executionpolicy bypass -command "& { . .\ManageRSAT.ps1; Uninstall-RSATCapabilities }"`)


Create Deployment Type Wizard
Content
General
General Information
Content
Detection Method
User Experience
Requirements
Dependencies
Summary
Progress
Completion
Specify information about the content to be delivered to target devices
Specify the location of the deployment type's content and other settings that control how content is delivered to target devices. All the contents in the path specified will be delivered.
Content location:
\ConfigMgr\Content\Scripts\RSAT
Browse...
Persist content in the client cache
Specify the command used to install this content.
Installation program:
powershell.exe -executionpolicy bypass -command
Browse...
Installation start in:
Configuration Manager can remove installations of this content if an uninstall program is specified below.
Uninstall program:
Browse...
Uninstall start in:
Run installation and uninstall program as 32-bit process on 64-bit clients.
< Previous
Next >
Summary
Cancel


Create Deployment Type Wizard
Detection Method
General
General Information
Content
Detection Method
User Experience
Requirements
Dependencies
Summary
Progress
Completion
Specify how this deployment type is detected
Specify how Configuration Manager determines whether this deployment type is already present on a device. This detection occurs before the content is installed or when software inventory data is collected.
Configure rules to detect the presence of this deployment type:
Connector
(
Clause
)
Add Clause...
Edit Clause...
Delete Clause
Group
Ungroup
1
2
Use a custom script to detect the presence of this deployment type:
Script type:
Script length:
Edit...
< Previous
Next >
Summary
Cancel

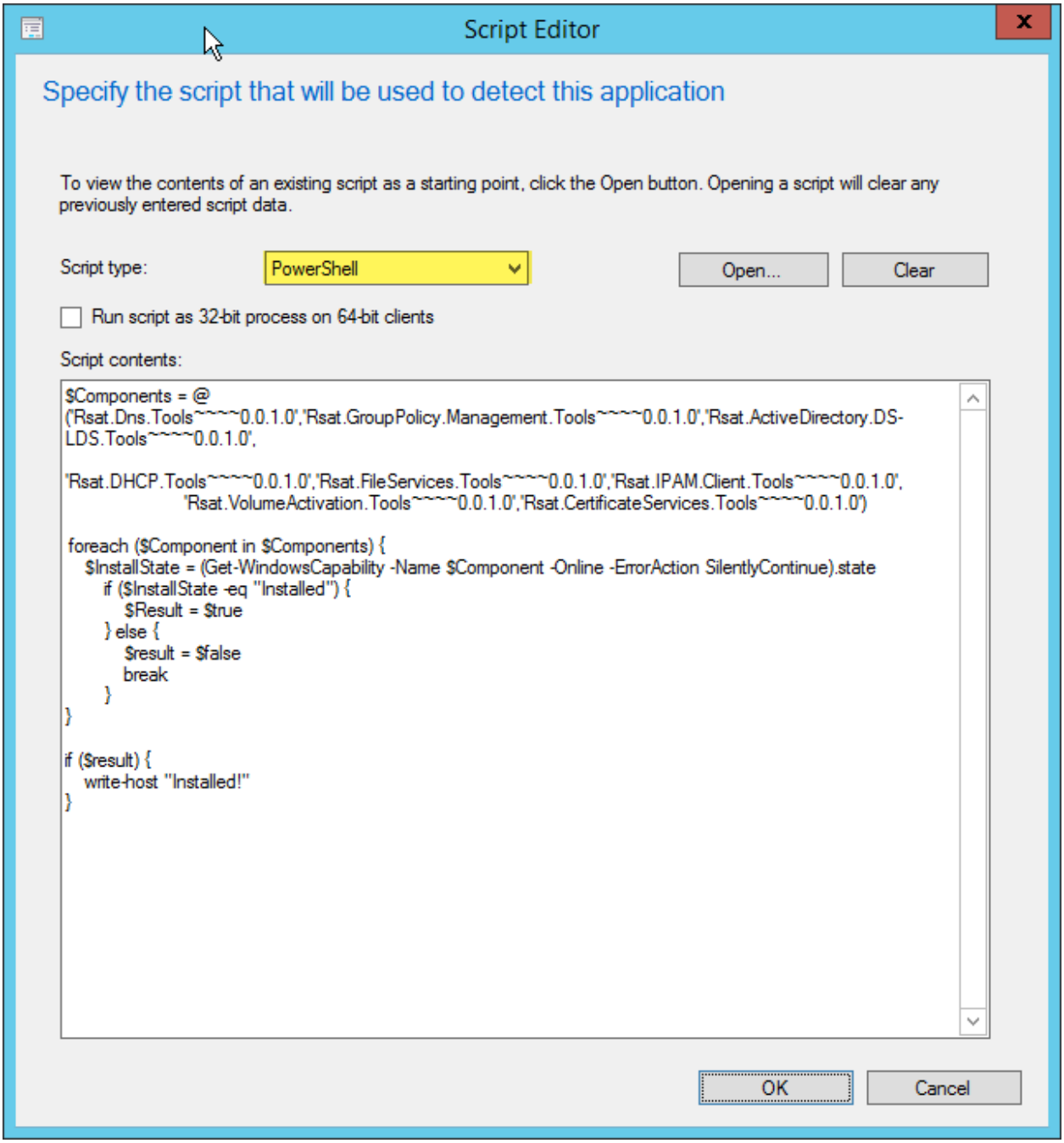
The Detection Rule

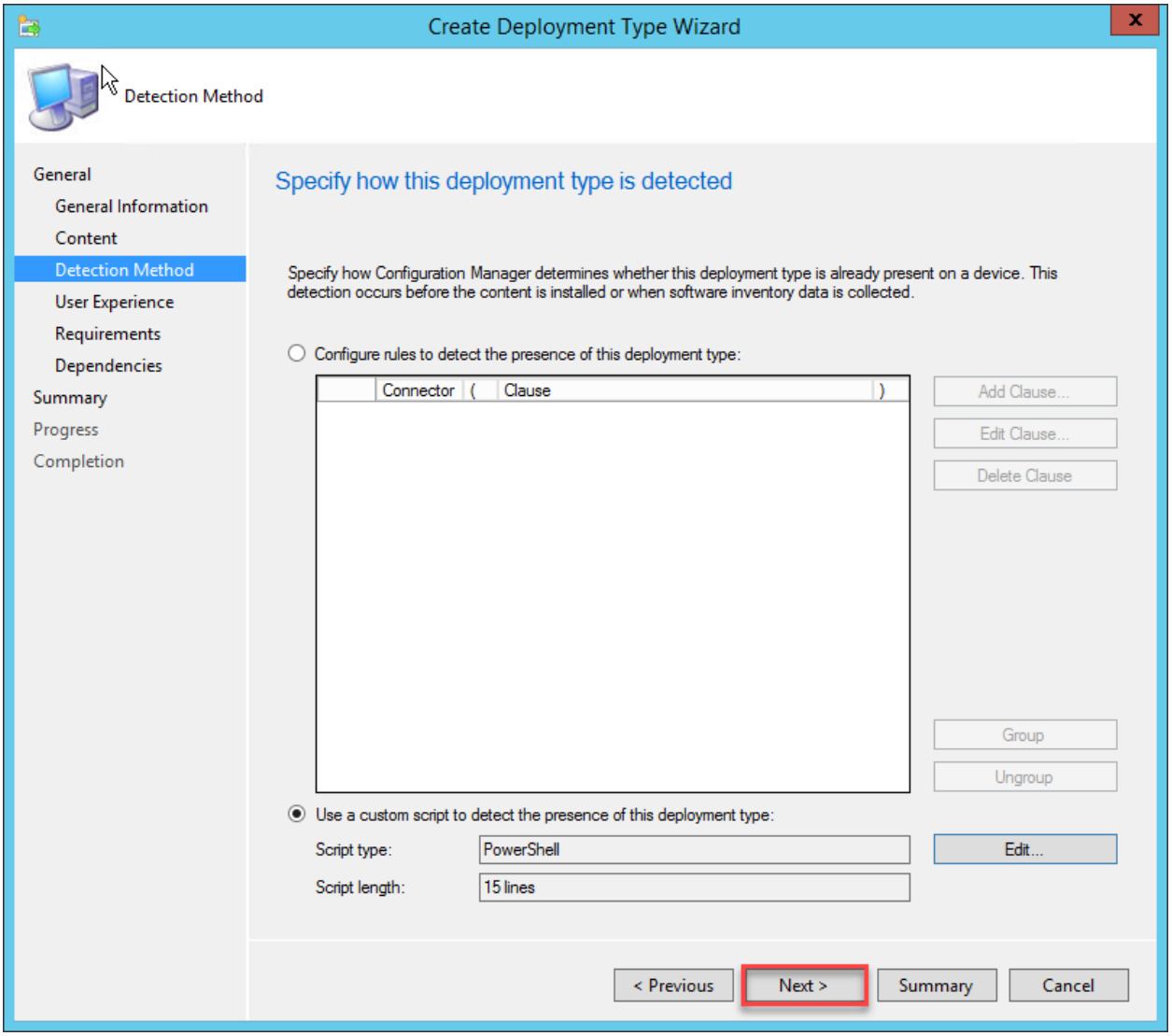
You can download the detection rule to copy \ paste into the script editor from my Github repository here: https://github.com/ozthe2/Powershell/tree/master/SCCM/RSAT

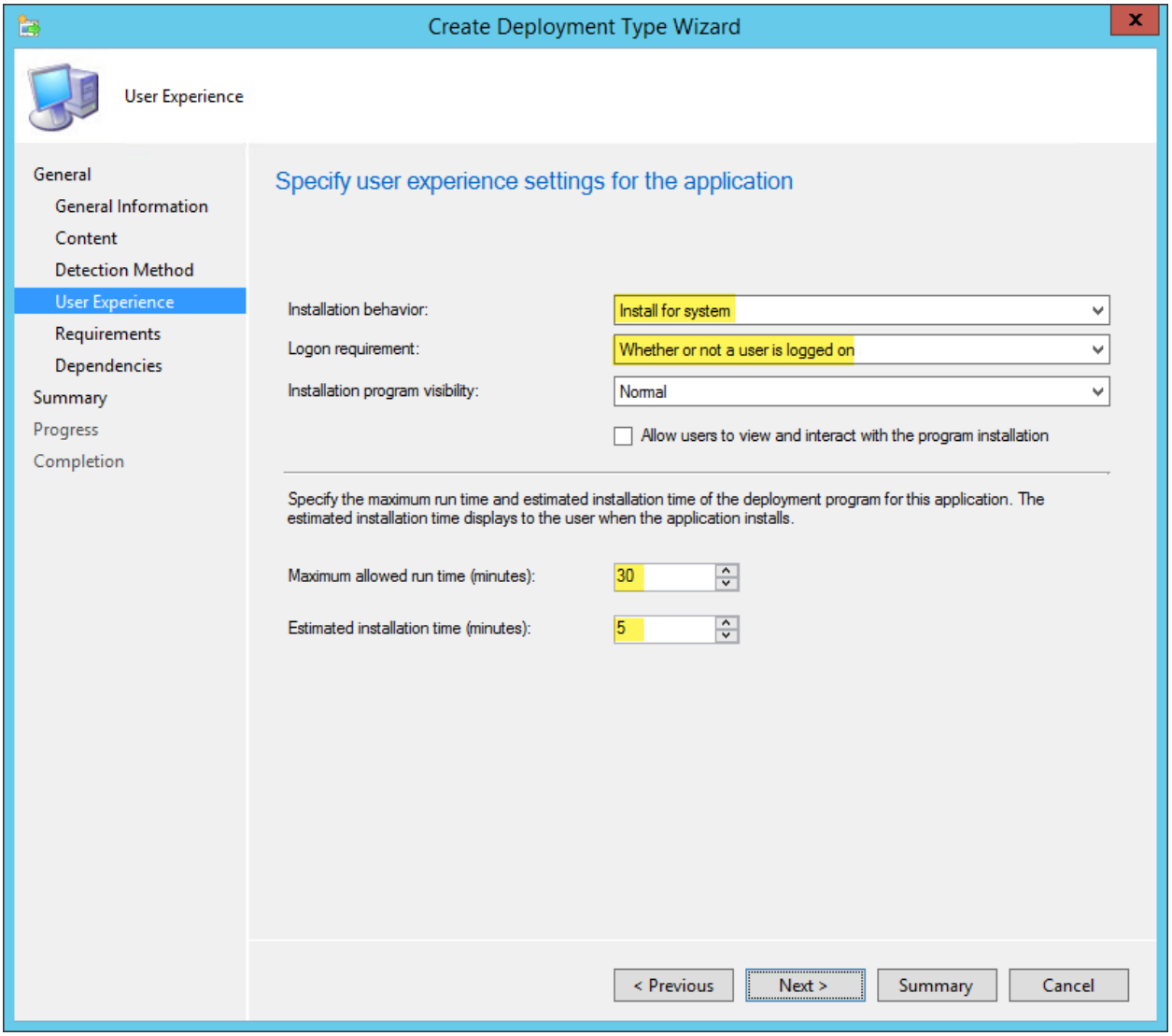
Watchout!

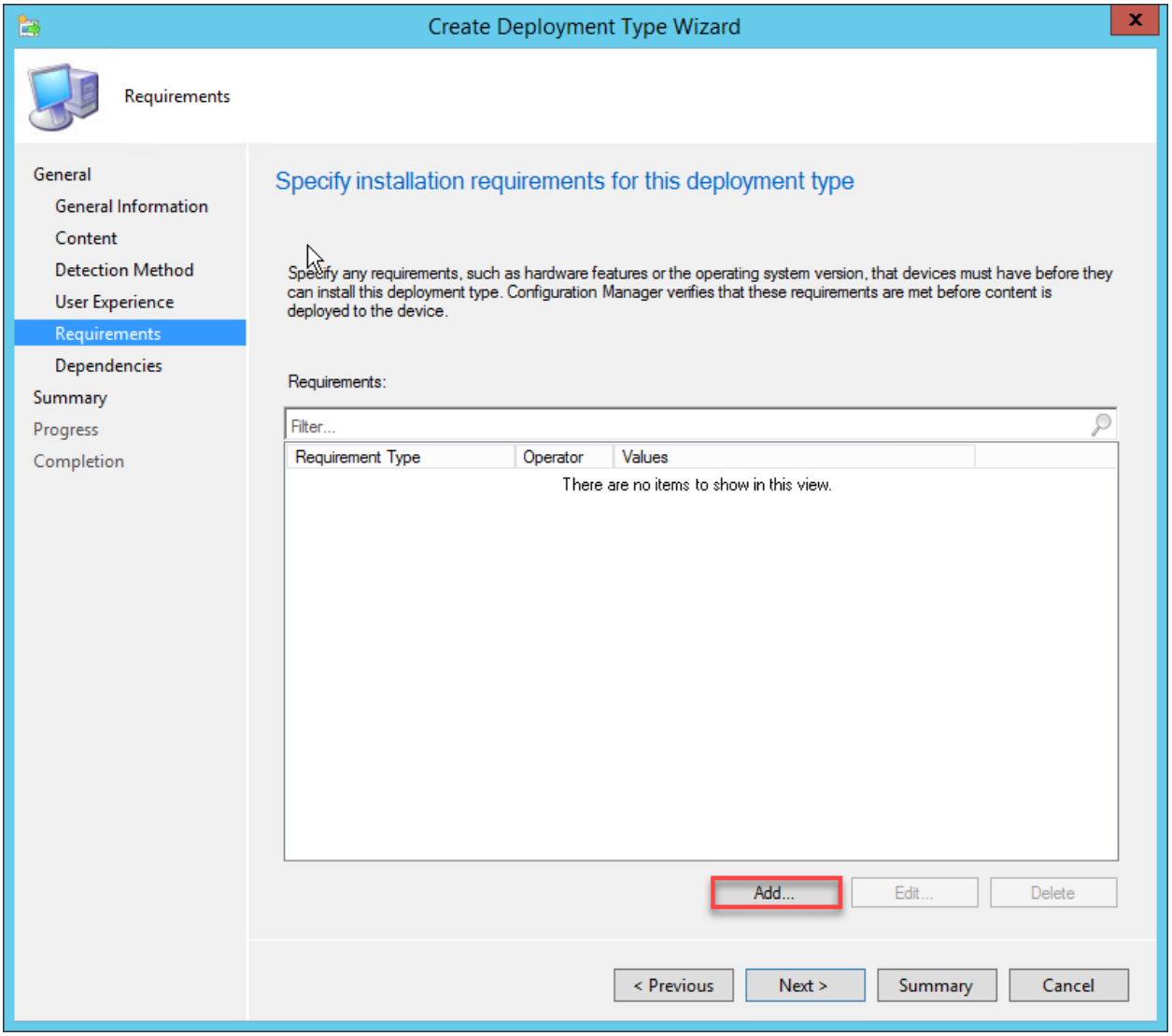
The detection rule uses the PowerShell cmdlet 'Get-WindowsCapability' and this needs to be run elevated. This means you *must* only deploy this application to a computer collection. (See Detection Rule Context) for a refresher as to why.

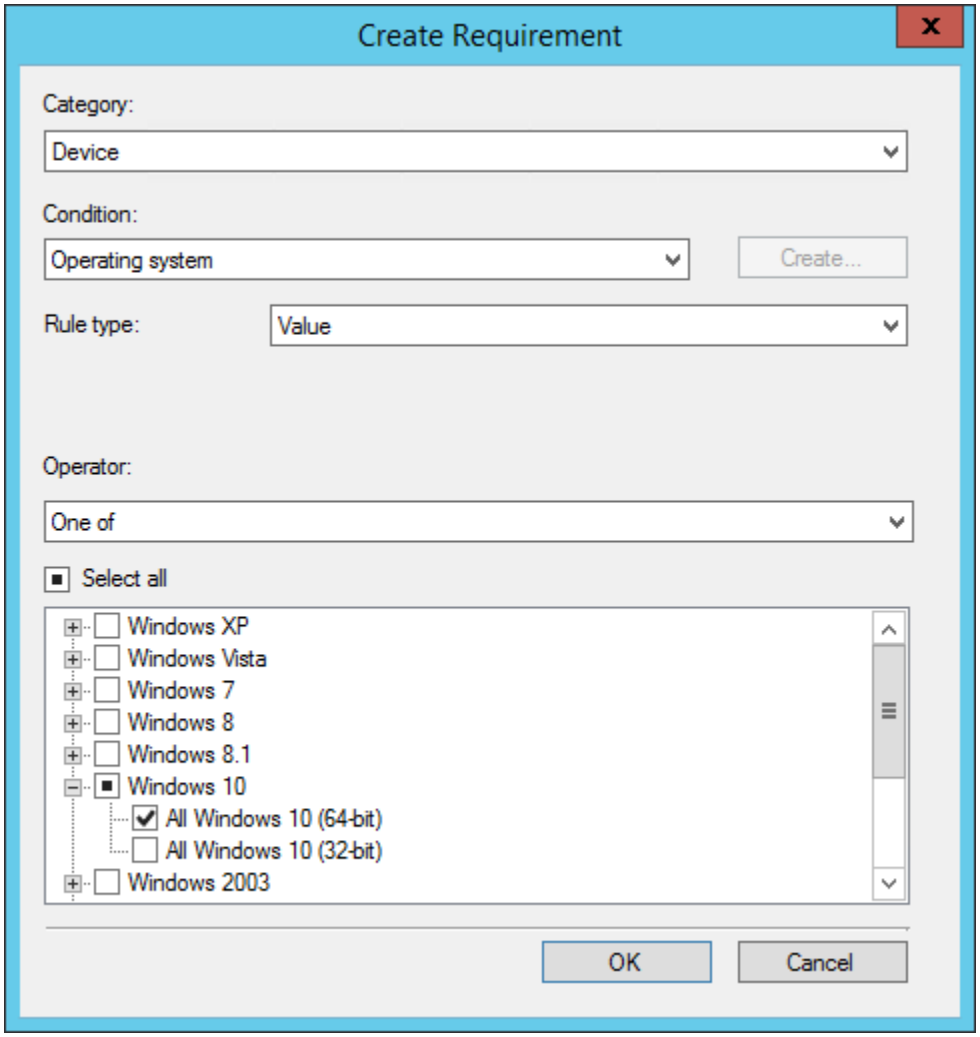
You will need to adjust this detection rule if you have modified the `$Components` variable. Just copy \ paste your components variable instead of mine so that it reflects the capability names you are installing. If you are **uninstalling** capabilities, you will need to change the line `If ($InstallState -eq 'Installed') {` to `If ($InstallState -eq 'NotPresent') {`

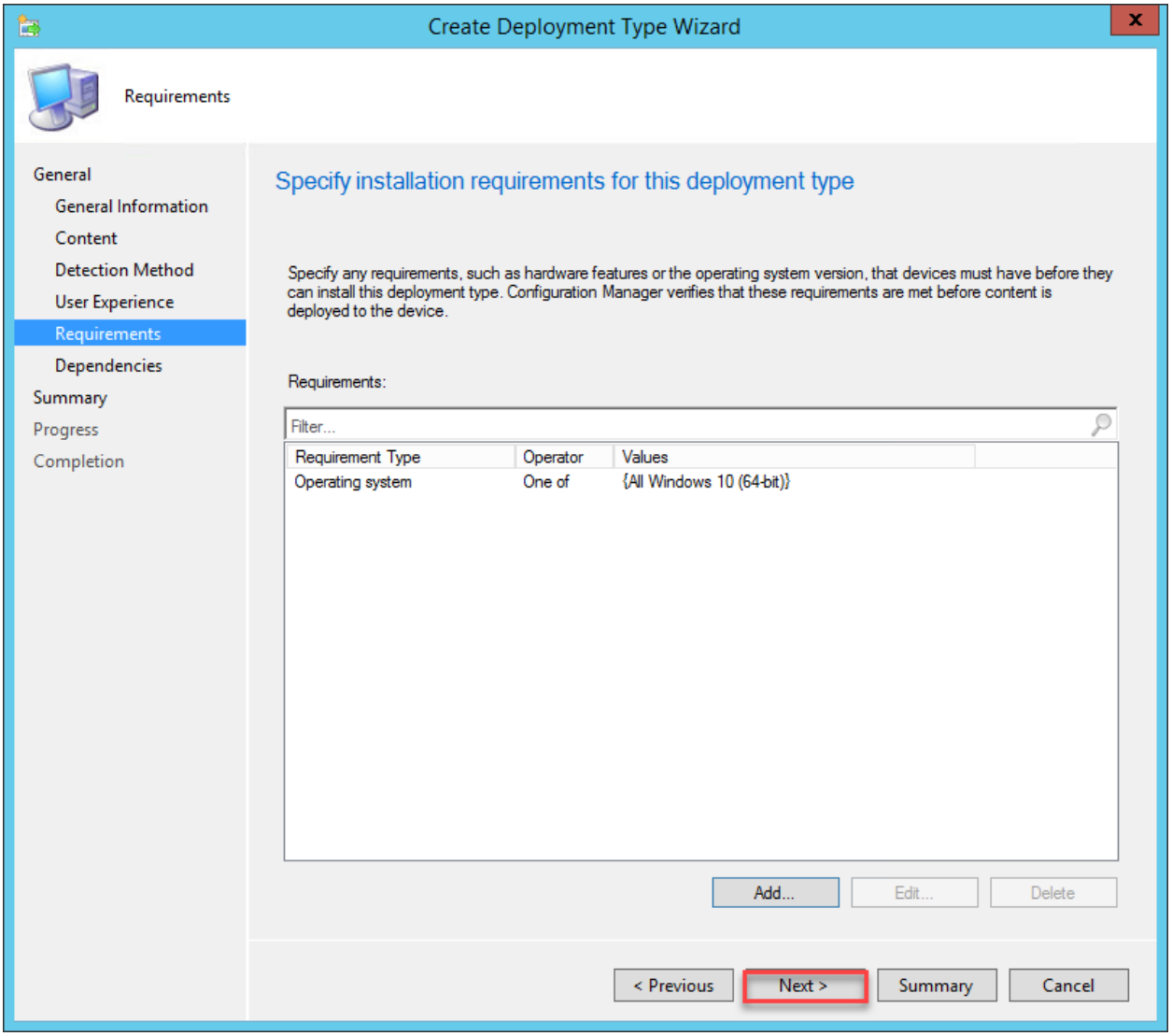

Script Editor
Specify the script that will be used to detect this application
To view the contents of an existing script as a starting point, click the Open button. Opening a script will clear any previously entered script data.
Script type:
PowerShell
Open...
Clear
Run script as 32-bit process on 64-bit clients
Script contents:
$Components = @
('Rsat.Dns.Tools~~~~0.0.1.0','Rsat.GroupPolicy.Management.Tools~~~~0.0.1.0','Rsat.ActiveDirectory.DS-LDS.Tools~~~~0.0.1.0',
'Rsat.DHCP.Tools~~~~0.0.1.0','Rsat.FileServices.Tools~~~~0.0.1.0','Rsat.IPAM.Client.Tools~~~~0.0.1.0',
'Rsat.VolumeActivation.Tools~~~~0.0.1.0','Rsat.CertificateServices.Tools~~~~0.0.1.0')
foreach ($Component in $Components) {
$InstallState = (Get-WindowsCapability -Name $Component -Online -ErrorAction SilentlyContinue).state
if ($InstallState -eq "Installed") {
$Result = $true
} else {
$result = $false
break
}
}
if ($result) {
write-host "Installed!"
}
OK
Cancel

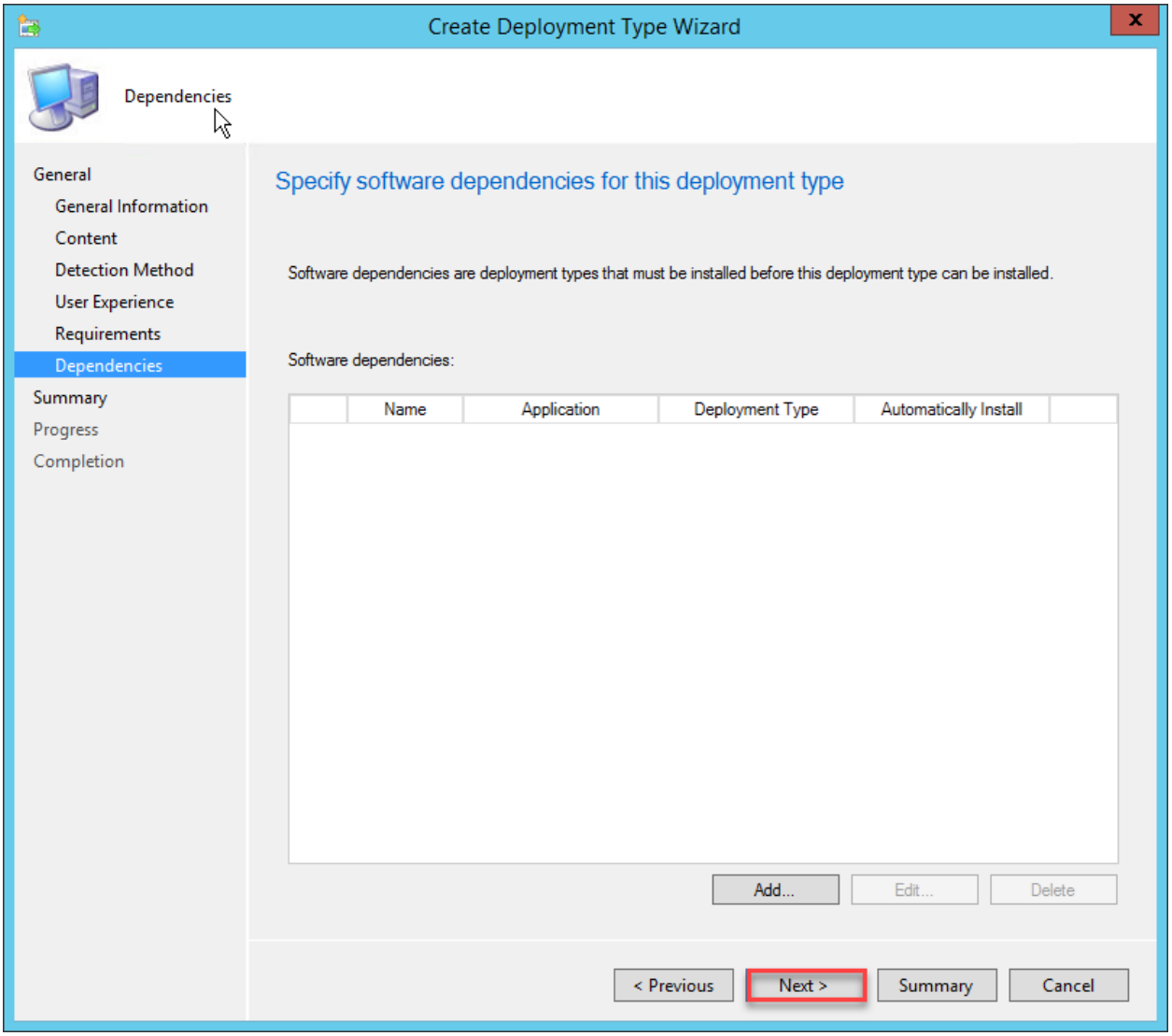

Create Deployment Type Wizard
Detection Method
General
General Information
Content
Detection Method
User Experience
Requirements
Dependencies
Summary
Progress
Completion
Specify how this deployment type is detected
Specify how Configuration Manager determines whether this deployment type is already present on a device. This detection occurs before the content is installed or when software inventory data is collected.
Configure rules to detect the presence of this deployment type:
Connector
(
Clause
)
Add Clause...
Edit Clause...
Delete Clause
Group
Ungroup
Use a custom script to detect the presence of this deployment type:
Script type:
PowerShell
Edit...
Script length:
15 lines
< Previous
Next >
Summary
Cancel

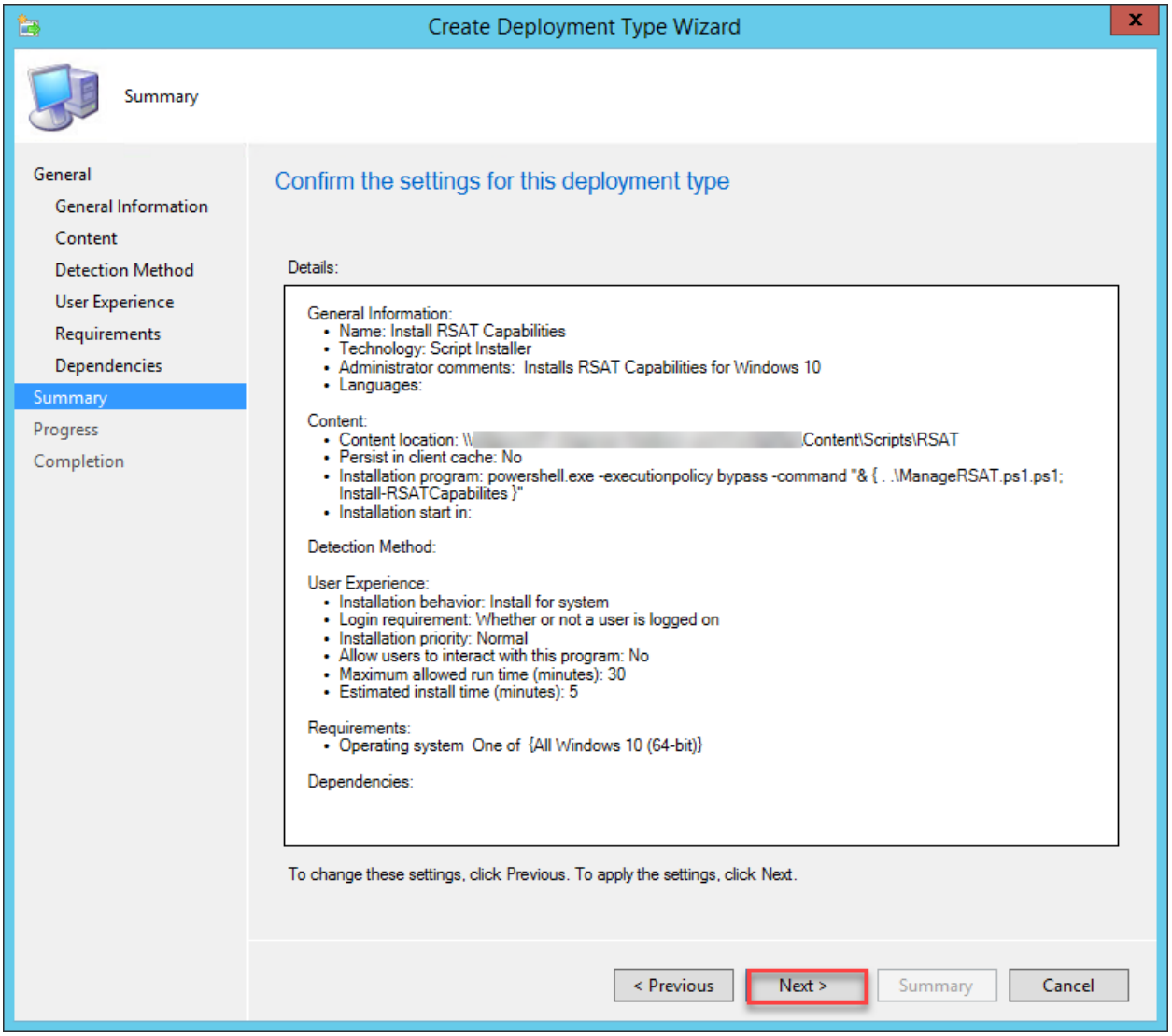

Create Deployment Type Wizard
User Experience
General
General Information
Content
Detection Method
User Experience
Requirements
Dependencies
Summary
Progress
Completion
Specify user experience settings for the application
Installation behavior:
Install for system
Logon requirement:
Whether or not a user is logged on
Installation program visibility:
Normal
Allow users to view and interact with the program installation
Specify the maximum run time and estimated installation time of the deployment program for this application. The estimated installation time displays to the user when the application installs.
Maximum allowed run time (minutes):
30
Estimated installation time (minutes):
5
< Previous
Next >
Summary
Cancel

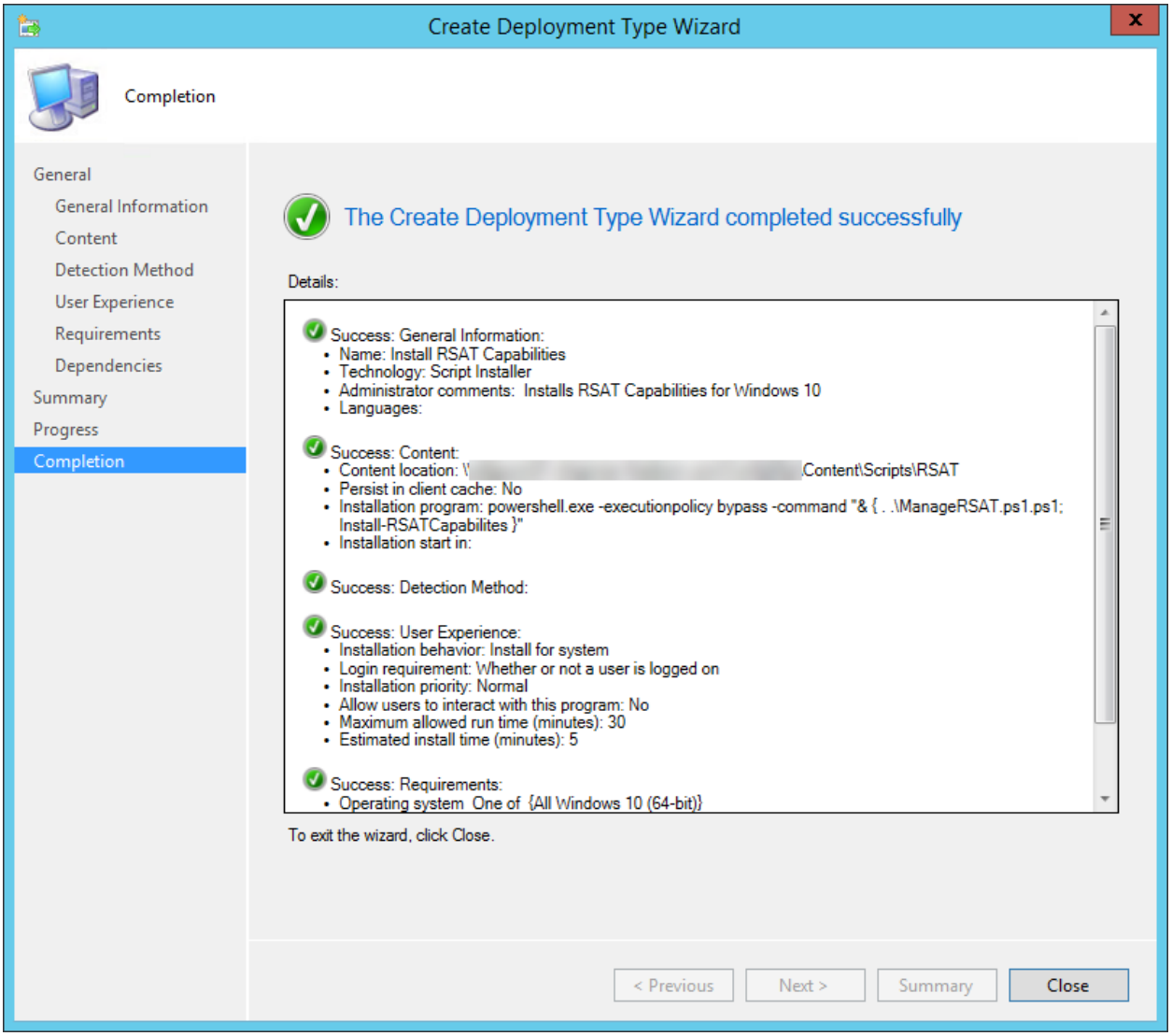

Create Deployment Type Wizard
Requirements
General
General Information
Content
Detection Method
User Experience
Requirements
Dependencies
Summary
Progress
Completion
Specify installation requirements for this deployment type
Specify any requirements, such as hardware features or the operating system version, that devices must have before they can install this deployment type. Configuration Manager verifies that these requirements are met before content is deployed to the device.
Requirements:
Filter...
Requirement Type
Operator
Values
There are no items to show in this view.
Add...
Edit...
Delete
< Previous
Next >
Summary
Cancel

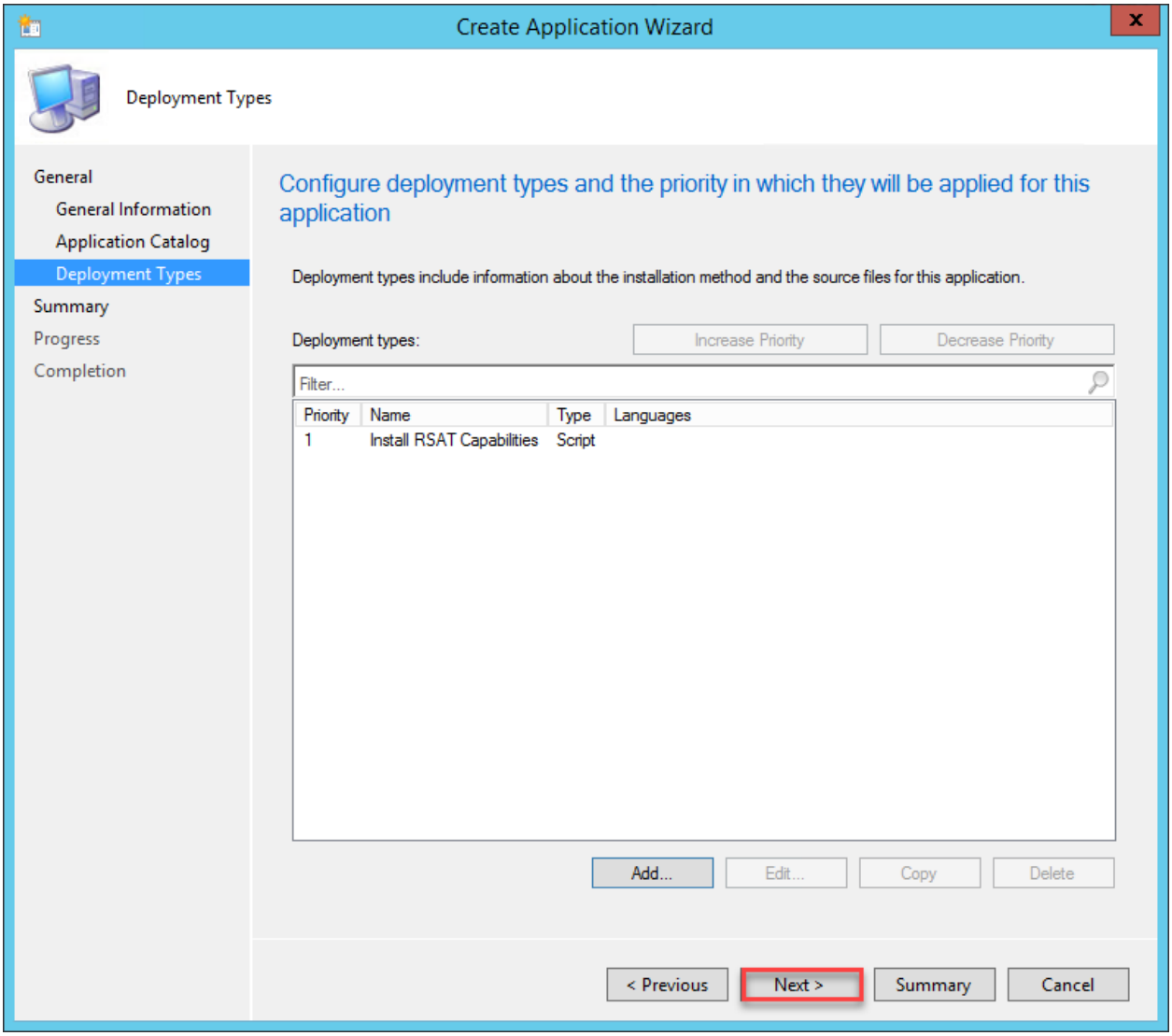

Create Requirement
Category:
Device
Condition:
Operating system
Create...
Rule type:
Value
Operator:
One of
Select all
Windows XP
Windows Vista
Windows 7
Windows 8
Windows 8.1
Windows 10
All Windows 10 (64-bit)
All Windows 10 (32-bit)
Windows 2003
OK
Cancel

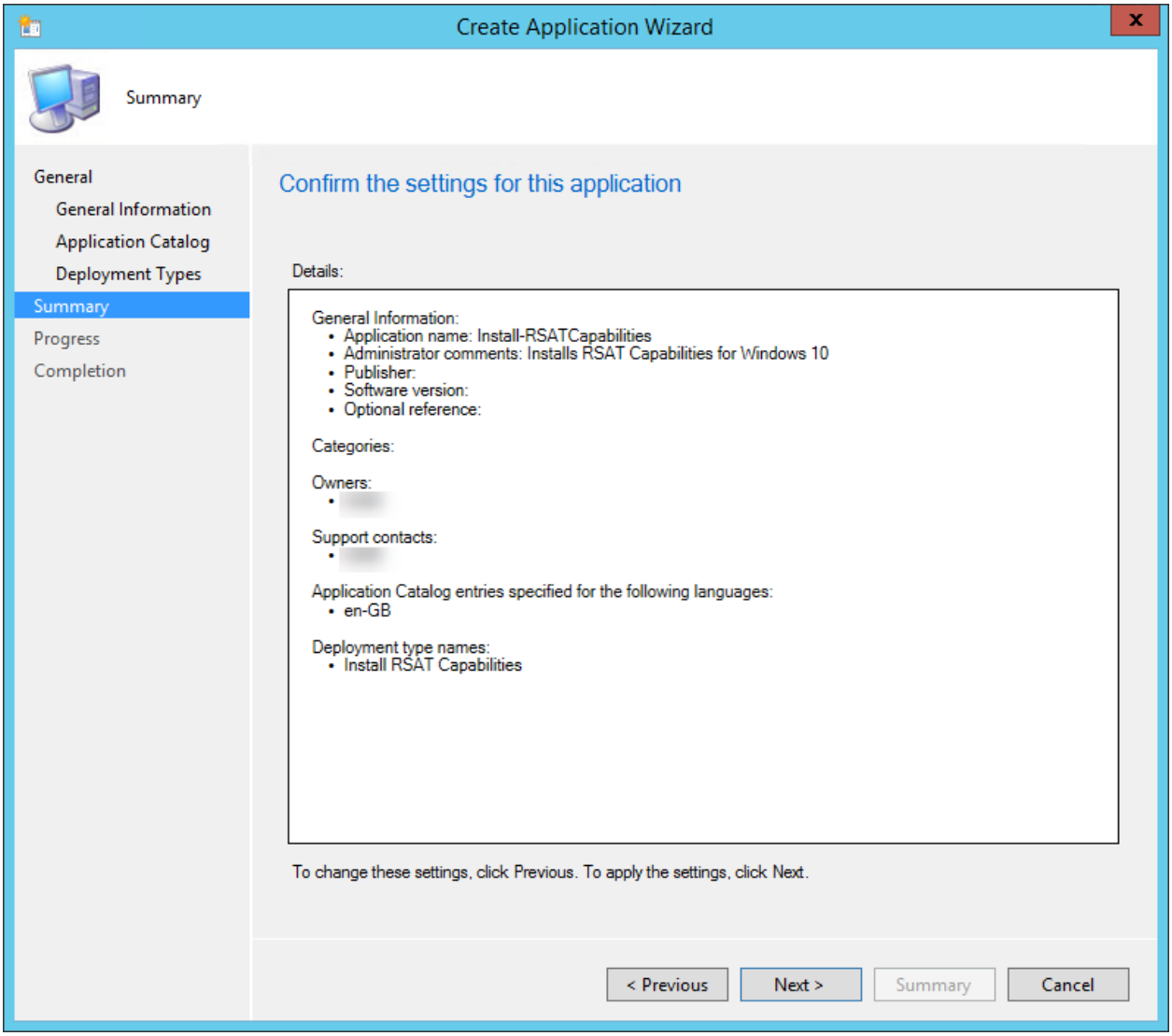

Create Deployment Type Wizard
Requirements
General
General Information
Content
Detection Method
User Experience
Requirements
Dependencies
Summary
Progress
Completion
Specify installation requirements for this deployment type
Specify any requirements, such as hardware features or the operating system version, that devices must have before they can install this deployment type. Configuration Manager verifies that these requirements are met before content is deployed to the device.
Requirements:
Filter...
Requirement Type
Operator
Values
Operating system
One of
{All Windows 10 (64-bit)}
Add...
Edit...
Delete
< Previous
Next >
Summary
Cancel

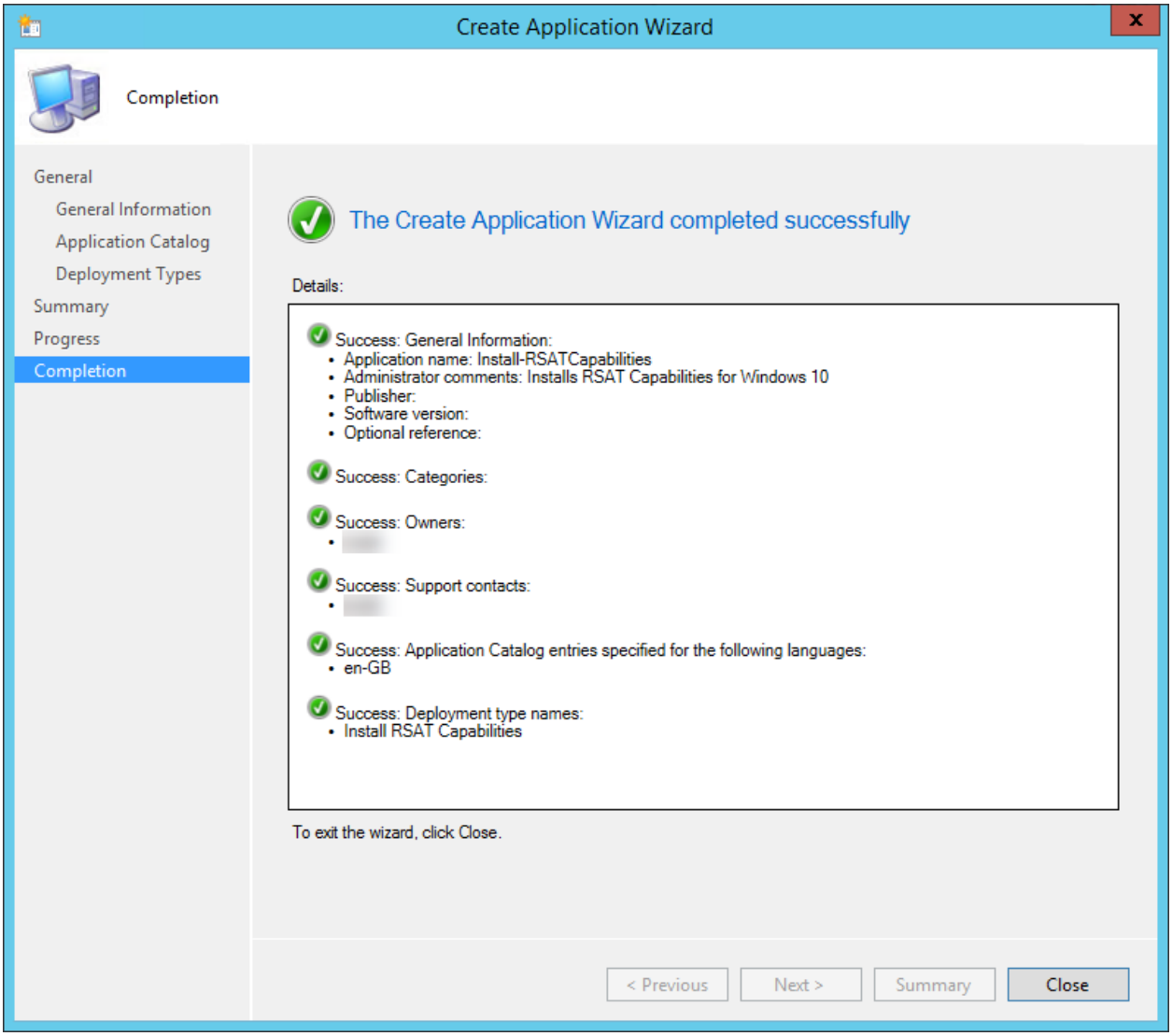

Create Deployment Type Wizard
Dependencies
General
General Information
Content
Detection Method
User Experience
Requirements
Dependencies
Summary
Progress
Completion
Specify software dependencies for this deployment type
Software dependencies are deployment types that must be installed before this deployment type can be installed.
Software dependencies:
Name
Application
Deployment Type
Automatically Install
Add...
Edit...
Delete
< Previous
Next >
Summary
Cancel


Create Deployment Type Wizard
Summary
General
General Information
Content
Detection Method
User Experience
Requirements
Dependencies
Summary
Progress
Completion
Confirm the settings for this deployment type
Details:
General Information:
Name: Install RSAT Capabilities
Technology: Script Installer
Administrator comments: Installs RSAT Capabilities for Windows 10
Languages:
Content:
Content location: \\ \Content\Scripts\RSAT
Persist in client cache: No
Installation program: powershell.exe -executionpolicy bypass -command "& { . .\ManageRSAT.ps1.ps1; Install-RSATCapabilites }"
Installation start in:
Detection Method:
User Experience:
Installation behavior: Install for system
Login requirement: Whether or not a user is logged on
Installation priority: Normal
Allow users to interact with this program: No
Maximum allowed run time (minutes): 30
Estimated install time (minutes): 5
Requirements:
Operating system One of {All Windows 10 (64-bit)}
Dependencies:
To change these settings, click Previous. To apply the settings, click Next.
< Previous
Next >
Summary
Cancel


Create Deployment Type Wizard
Completion
General
General Information
Content
Detection Method
User Experience
Requirements
Dependencies
Summary
Progress
Completion
The Create Deployment Type Wizard completed successfully
Details:
Success: General Information:
Name: Install RSAT Capabilities
Technology: Script Installer
Administrator comments: Installs RSAT Capabilities for Windows 10
Languages:
Success: Content:
Content location: \ Content\Scripts\RSAT
Persist in client cache: No
Installation program: powershell.exe -executionpolicy bypass -command "& { . .\ManageRSAT.ps1.ps1; Install-RSATCapabilites }"
Installation start in:
Success: Detection Method:
Success: User Experience:
Installation behavior: Install for system
Login requirement: Whether or not a user is logged on
Installation priority: Normal
Allow users to interact with this program: No
Maximum allowed run time (minutes): 30
Estimated install time (minutes): 5
Success: Requirements:
Operating system One of {All Windows 10 (64-bit)}
To exit the wizard, click Close.
< Previous
Next >
Summary
Close


Create Application Wizard
Deployment Types
General
General Information
Application Catalog
Deployment Types
Summary
Progress
Completion
Configure deployment types and the priority in which they will be applied for this application
Deployment types include information about the installation method and the source files for this application.
Deployment types:
Increase Priority
Decrease Priority
Filter...
Priority
Name
Type
Languages
1
Install RSAT Capabilities
Script
Add...
Edit...
Copy
Delete
< Previous
Next >
Summary
Cancel


Create Application Wizard
Summary
General
General Information
Application Catalog
Deployment Types
Summary
Progress
Completion
Confirm the settings for this application
Details:
General Information:
Application name: Install-RSATCapabilities
Administrator comments: Installs RSAT Capabilities for Windows 10
Publisher:
Software version:
Optional reference:
Categories:
Owners:
Support contacts:
Application Catalog entries specified for the following languages:
en-GB
Deployment type names:
Install RSAT Capabilities
To change these settings, click Previous. To apply the settings, click Next.
< Previous
Next >
Summary
Cancel

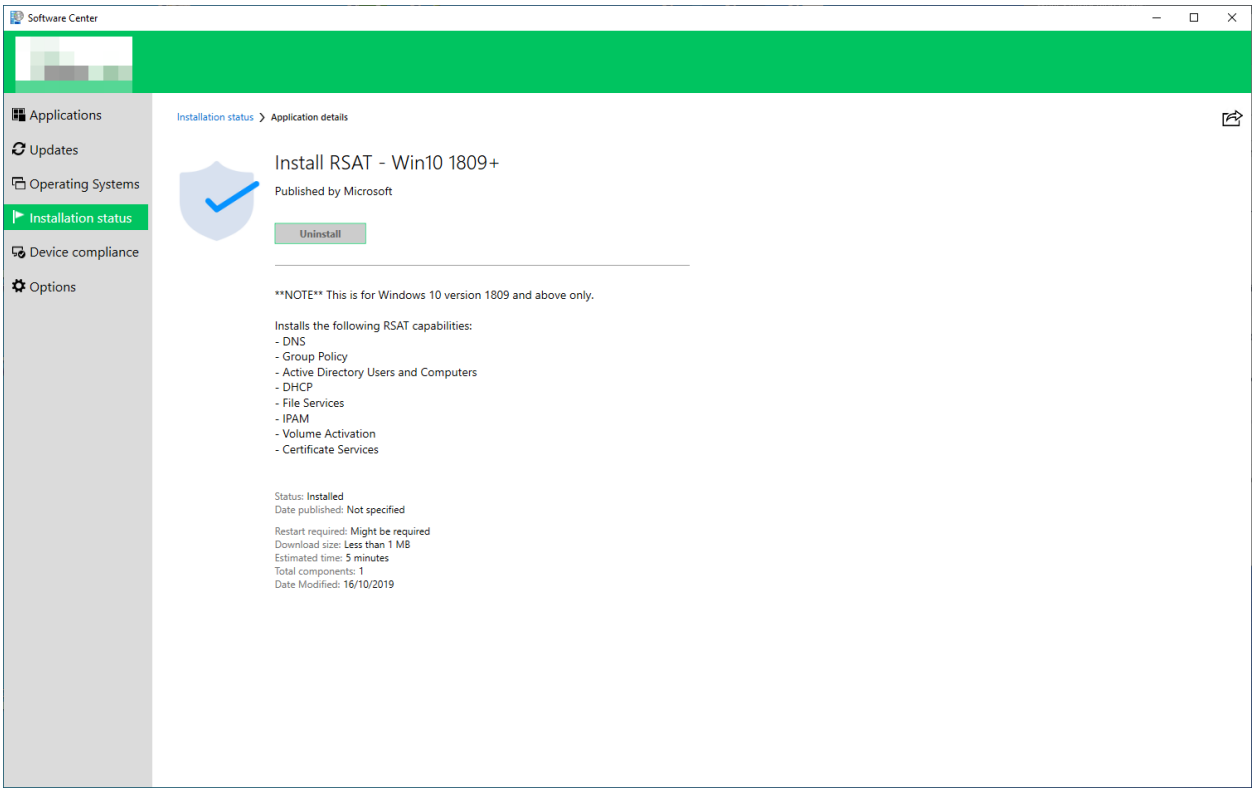


Distribute and Deploy

That's it! Now you just need to distribute the application and deploy it! (Remember to deploy to a computer collection, not a user collection)

The Result

Here you can see the result of my installation:



Afterword

It's not easy learning something new: A few years ago, I was teaching my seven-year-old daughter to ride her bike. She kept falling off and at one point she threw her bike to the ground and stormed off in a temper shouting about how she will never be able to do it. But she persevered and over the next two or three weeks it just 'clicked' and that was it…she was off. Now there's no stopping her! I sometimes remind her of this same story when she is struggling with other things that she is finding difficult to achieve. Like my daughter, you too may make mistakes and feel like quitting. You may struggle and think it's just too hard or not worth the bother. But believe me, keep at it and I can assure you that when the day comes that it all finally 'clicks' into place, you will never look back.

Don’t Be a Stranger!

Hey, now we’ve made it this far on our journey together I feel we’ve become friends! So don’t be a stranger.
Feel free to contact me about this book whether its typo’s or clarification required or anything else.

You can reach out to me on Twitter: *@ozthe2*

Feel free to peruse my website too: https://fearthepanda.com

And last but not least, my GitHub: https://www.github.com/ozthe2

And did I mention I have a (not-so) regular podcast? It’s called: **The Secret Diary of a Network Administrator**. Search for it using any of your podcast apps \ outlets.

Suggested Reading

You can't go wrong if you read the '..in a Month of Lunches' series of books.

I recommend reading them in the order shown below:

- Learn Windows PowerShell in a Month of Lunches by Donald W. Jones and Jeffery D. Hicks
- Learn PowerShell Toolmaking in a Month of Lunches by Don Jones and Jeffery Hicks
- Learn PowerShell Scripting in a Month of Lunches by Don Jones and Jeffery Hicks