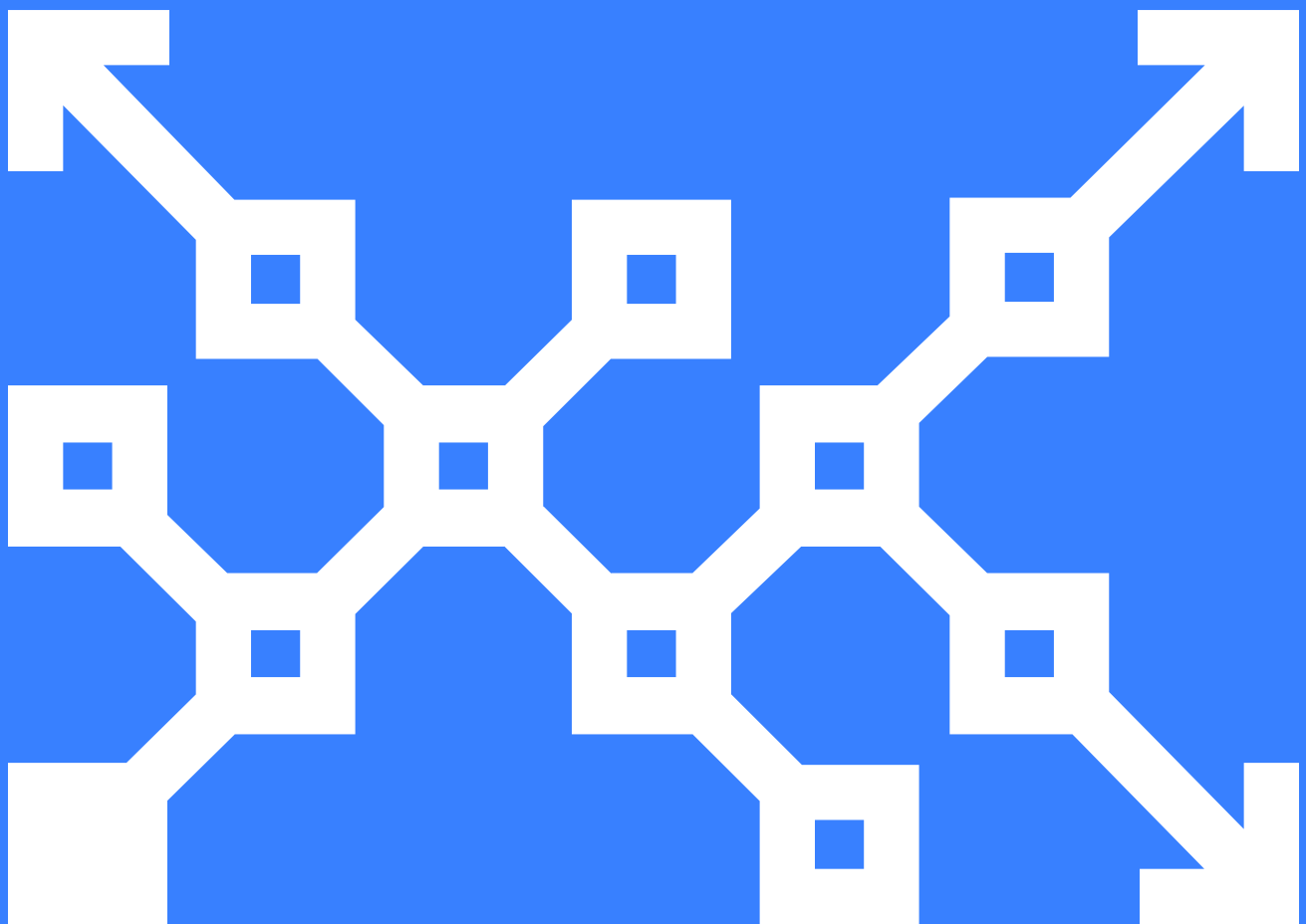


# Managing Windows with Puppet Enterprise



# Contents

<b>3</b>	<b>Basics first</b>
<b>5</b>	<b>Get up and running with the essential Windows modules</b>
5	ACL
5	Chocolatey
6	DSC
6	PowerShell
6	Reboot
7	Registry
7	WSUS client
7	Download_file
8	IIS
8	Windows environment
8	Windowsfeature
<b>9</b>	<b>Managing Powershell DSC with Puppet</b>
9	Use the skills you have to do more
11	DSC and Puppet: the best of both worlds
13	Advanced scenario
<b>17</b>	<b>Reporting with Puppet and DSC</b>
<b>19</b>	<b>Setting up Windows for system and application monitoring</b>
19	Monitoring with Puppet and Nagios
<b>23</b>	<b>Patch management on Windows with Puppet</b>
23	Managing patches with Puppet
24	Chocolatey and Puppet
<b>25</b>	<b>Managing software on Windows with Chocolatey</b>
26	Creating packages
<b>27</b>	<b>Deploying IIS and ASP.NET with Puppet</b>
<b>31</b>	<b>Active Directory management</b>
31	Installing Active Directory with Puppet
32	Managing Windows users and groups with Puppet
34	Managing Active Directory with Puppet
<b>35</b>	<b>Puppet and Microsoft Azure</b>
<b>37</b>	<b>Conclusion</b>

Puppet Enterprise makes configuring and maintaining a large Windows-based infrastructure simple and straightforward. With Puppet modules, you can easily deploy Windows servers, install Windows software across multiple machines, build and deploy ASP.NET websites, manage software patches, run PowerShell scripts, and even deploy Windows Azure machines.

We've ordered the following chapters so you can get your Windows infrastructure puppetized in a clear and logical way.

## Basics first

Puppet helps you manage basic configurations for any server, including services, administrator accounts, and scheduled tasks. If you haven't installed Puppet Enterprise yet, [this section of the Puppet Docs site](#) will help you do that. There's also a good section in the Puppet Docs site about [managing Windows configurations with Puppet Enterprise](#), but for now, we'll show you some simple examples.

Let's start by managing the Windows Time service. The following code ensures that this service is up and running on a server:

```
service { 'w32time':  
  ensure => 'running'  
}
```

In this manifest, we identified that we wanted to manage a `service` called `w32time`, and that its state should be `running`. In general, to ensure that a service is running, you can follow this pattern:

```
service { '<service name>':  
  ensure => 'running'  
}
```

Another basic task that you may want to automate is a scheduled task. Regularly scheduled tasks are often necessary for Windows machines to perform routine system maintenance. For instance, if you need to clean out the `C:\Windows\Temp` directory of your server each day at 8:00 a.m., you could manage that task in Puppet as follows:

```

scheduled_task { 'Purge global temp files':
  ensure    => present,
  enabled   => true,
  command   => 'c:\\windows\\system32\\cmd.exe',
  arguments => '/c "del c:\\windows\\temp\\*. * /F /S /Q"',
  trigger   => {
    schedule => daily,
    start_time => '08:00',
  }
}

```

In this example, we have a resource called `scheduled_task` that has a number of parameters dictating whether the task is enabled or disabled, which arguments to pass to the command being run, and even which trigger will cause the scheduled task to run.

Finally, you may want to manage the group of administrators that have elevated permissions on a given server. How do we manage that? Say you have a domain user who is not currently present in the Domain Administrators group, and you want to add them to the Local Administrators group on a server. You can use this Puppet code:

```

group { 'Administrators':
  ensure => 'present',
  members => ['DOMAIN\\User'],
  auth_membership => false
}

```

In this case, `auth_membership` is set to `false` to ensure that `DOMAIN\\User` is present in the Administrators group, and to ensure that other accounts that might be present in Administrators are *not* removed. With a couple of basic configurations, Puppet ensures that your Windows server is configured the way you want.

Now let's move on to some of the most common things Windows administrators do with Puppet Enterprise.

## Get up and running with the essential Windows modules

The Puppet Enterprise Windows module pack is a collection of the most essential Windows modules available on the Puppet Forge. It has everything you need to get started using Puppet on Windows, including:

- **ACL**
- **Chocolatey**
- **DSC**
- **PowerShell**
- **Reboot**
- **Registry**
- **WSUS client**
- **Windows environment**
- **Download\_file**
- **IIS**
- **Windowsfeature**

### ACL

The **Puppet ACL (or Access Control List)** module controls permissions for files in Windows environments. This example controls permissions on a particular directory:

```
acl {'c:/temp':  
  permissions => [  
    { identity => 'Administrators', rights => ['full'] }  
  ],  
  purge => true,  
  inherit_parent_permissions => false,  
}
```

There's no need to purge or turn off inheritance. The Puppet ACL module works with files, and could potentially have support for services and registries.

### Chocolatey

Chocolatey is a package manager for Windows that can manage and configure software installations across your entire Windows infrastructure. In this simple example, we're using Puppet to install Chocolatey itself and make sure it stays up to date.

```
include chocolatey  
  
package {'git':  
  ensure => latest,  
}
```

You can find the Puppet Supported Chocolatey module [here](#). Chocolatey is also covered more in the section [Managing software on Windows with Chocolatey](#).

## DSC

Puppet Enterprise is integrated with Microsoft PowerShell desired state configuration, or DSC. In this example of using the **DSC module**, a Puppet DSC resource disables a firewall port. It's a common task for Windows admins, and it's easy to roll out across your entire infrastructure with Puppet.

```
dsc_xFirewall {'inbound-2222':  
  dsc_ensure      => 'present',  
  dsc_name        => 'inbound2222',  
  dsc_displayname => 'Inbound DSC 2222',  
  dsc_displaygroup => 'A Puppet + DSC Test',  
  dsc_action      => 'Allow',  
  dsc_enabled     => 'false',  
  dsc_direction  => 'Inbound',  
}
```

For more information about DSC, see the section **Managing Powershell DSC with Puppet**.

## PowerShell

Puppet Enterprise has **full PowerShell support**. In this example, Puppet first checks to see if the Windows power management scheme is set to performance, before ensuring that it is.

```
# performance power scheme GUID  
$guid = '8c5e7fda-e8bf-4a96-9a85-a6e23a8c635c'  
  
exec { 'set performance power scheme':  
  command => "PowerCfg -SetActive ${guid}",  
  path    => 'C:\Windows\System32;C:\Windows\System32\  
WindowsPowerShell\v1.0',  
  unless  => "if((Powercfg -GetActiveScheme).Split()[3] -ne '${guid}')  
{ exit 1 }",  
  provider => powershell,  
  logoutput => true,  
}
```

## Reboot

Puppet can **reboot Windows machines** across an infrastructure. There are two ways to reboot Windows machines with Puppet. In this example, the system detects there's a pending reboot. You can also set a mode to **refresh only**.

```
reboot {'reboot_pending':  
  when      => pending,  
  timeout   => 15,  
}
```

The command can also be used in conjunction with configuration changes or software installations.

## Registry

The **Puppet Registry module** can create and manage Registry keys and values directly. In this example, Puppet is editing the Registry to turn off auto login ability.

```
registry_value {'HKLM\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\WinLogon\AutoAdminLogon':
  ensure => present,
  type   => dword,
  data   => 0,
}
```

## WSUS client

The **Windows Server update services (WSUS) module** can manage Windows updates internally instead of reaching out to Microsoft's servers. In this example, the WSUS module is scheduled to make updates from a internal server every Tuesday at 2:00 a.m.

```
class {'wsus_client':
  server_url           => 'https://internal_server:8530',
  auto_update_option   => "Scheduled",
  scheduled_install_day => "Tuesday",
  scheduled_install_hour => 2,
}
```

For more information about WSUS, see the section **Patch management on Windows with Puppet**.

## Download\_file

**Download\_file** is a simple module that downloads a file or files to a Windows machine. In this example, it's downloading the .NET framework from Microsoft to a directory on a machine.

```
download_file { '.NET Framework 4.0':
  url           => 'https://download.microsoft.com/download/9/5/A/95A9616B-
7A37-4AF6-BC36-D6EA96C8DAAE/dotNetFx40_Full_x86_x64.exe',
  destination_directory => 'C:\temp'
}
```

## IIS

The **Puppet IIS module** can create sites, manage application pools and more. In this example, Puppet is setting up a pool and a site called `The Server`. Paths, ports and much more can be designated in the parameters.

```
iis::manage_app_pool { 'somepool':  
  enable_32_bit      => true,  
  managed_runtime_version => 'v4.0',  
} ->  
iis::manage_site { 'TheServer':  
  site_path  => 'c:\sites\server',  
  port       => '8080',  
  ip_address => '*',  
  app_pool   => 'somepool',  
}
```

## Windows environment

Managing environmental variables is very important in Windows. In this example, Puppet ensures a particular set of variables are installed in the `c:\tools\bin` path, and sets them to `merge mode`. Also note that the module can manage environmental factors per user.

```
windows_env { 'ValueOnPat':  
  ensure  => present,  
  variable => 'PATH',  
  value    => 'c:\tools\bin',  
  mergemode => insert,  
}
```

## Windowsfeature

**Windowsfeature** can turn Windows features on or off. With just a few lines of code, it can ensure that IIS is installed and that ASP.NET is configured to be used for Windows Server 2012:

```
windowsfeature { 'Web-WebServer':  
} ->  
windowsfeature { 'Web-Asp-Net45': }
```

These are just some of the modules that are included in the Windows module pack. You'll find more on the **Puppet Forge**.



## Managing Powershell DSC with Puppet

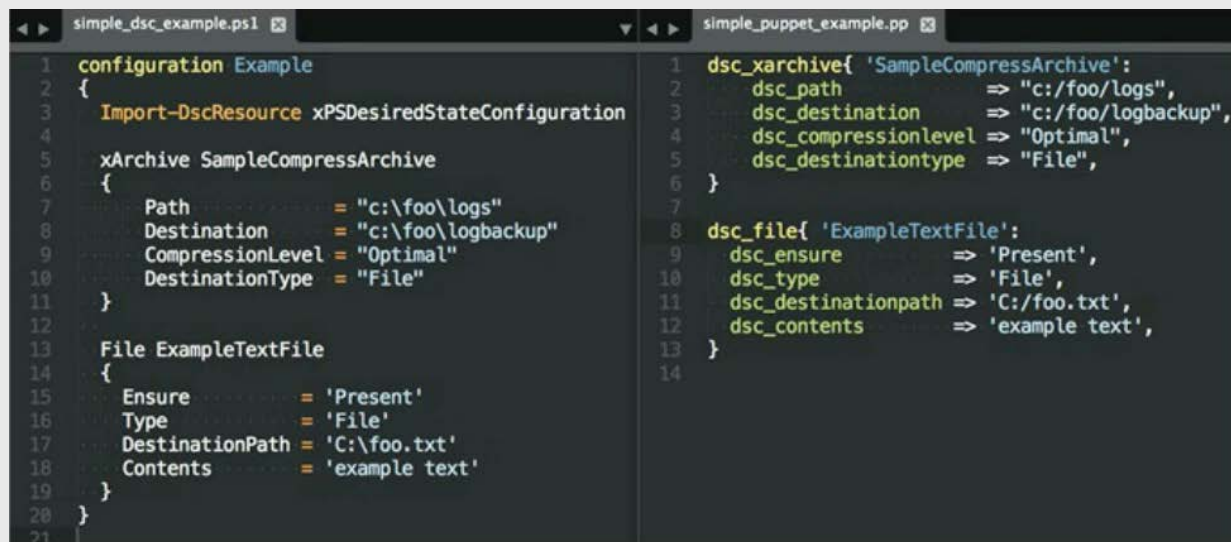
Puppet and PowerShell DSC can work together to make a powerful management tool. The **Puppet DSL (domain-specific language)** is very similar to PowerShell DSC syntax, making it easy to transition between the two. It's possible to migrate a PowerShell DSC configuration script to a Puppet manifest with just a few text edits, and you end up with less code. And you don't have to go it alone when managing your systems — there are thousands of resources from Puppet, Microsoft, and the community that can be used right away to work with multiple platforms across diverse environments.

Puppet provides rich management capabilities that can layer on top of DSC, including node classification, status of nodes, global-based access control, and more. Tracking and reporting change are easier with Puppet — and reporting is a must for any company that has to do any kind of compliance auditing. To learn more, see the chapter **Reporting with Puppet and DSC**.

### Use the skills you have to do more

If you have PowerShell experience, you'll find that the Puppet DSC module is easy to use. DSC works like Puppet in many ways. It's declarative, uses a similar syntax and similar terminology. With Puppet, you can use your existing PowerShell DSC skills and knowledge to fix problems, without having to learn a new system up front. Oftentimes, a Puppet manifest will require you to write less code than a PowerShell DSC configuration script.

Let's take a look at PowerShell DSC configuration script on the left and a Puppet manifest on the right. Both compress a log directory and ensure a file is present on the system with specific text inside it. You can see in the next image that you can copy and paste your existing DSC code into a Puppet manifest, and be up and running with just a few text edits.



```
1 configuration Example
2 {
3   Import-DscResource xPSDesiredStateConfiguration
4
5   xArchive SampleCompressArchive
6   {
7     Path           = "c:\foo\logs"
7     Destination    = "c:\foo\logbackup"
7     CompressionLevel = "Optimal"
7     DestinationType = "File"
8   }
9
10  File ExampleTextFile
11  {
12    Ensure          = 'Present'
12    Type             = 'File'
12    DestinationPath = 'C:\foo.txt'
12    Contents         = 'example text'
13  }
14 }
15
16
17
18
19
20
21
```

```
1 dsc_xarchive{ 'SampleCompressArchive':
2   dsc_path           => "c:/foo/logs",
3   dsc_destination    => "c:/foo/logbackup",
4   dsc_compressionlevel => "Optimal",
5   dsc_destinationtype => "File",
6 }
7
8 dsc_file{ 'ExampleTextFile':
9   dsc_ensure          => 'Present',
10  dsc_type             => 'File',
11  dsc_destinationpath => 'C:/foo.txt',
12  dsc_contents         => 'example text',
13 }
14
```

There are only a few syntactical differences to account for when you're migrating to a Puppet manifest:

- Add the prefix `dsc_` to all resource declarations and to all parameters.
- Change equal signs into hashrockets.
- Add commas to the end of each line.

These are simple, short changes that make sense. The syntax is similar, so there's no cognitive dissonance switching from a PowerShell DSC resource to a Puppet module declaration. You'll notice there's less to write using Puppet — no configuration blocks to add, no DSC Resource module import declarations to keep track of. Just the individual components you need to get the job done. All PowerShell DSC types are supported in Puppet, including base types like integers, complex types like PS Credentials, and even custom bindings.

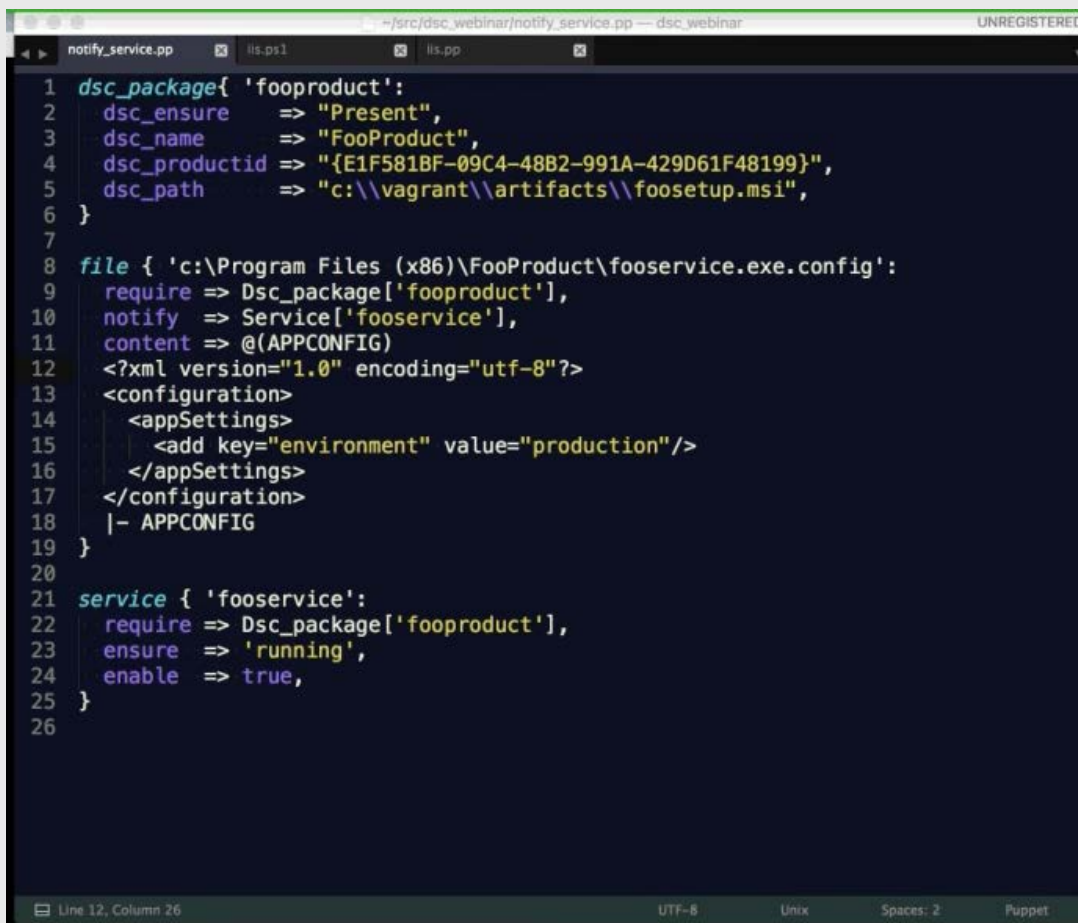
Moving your PowerShell DSC to Puppet is just the beginning. You can do much more when you use PowerShell and Puppet together.

## DSC and Puppet: the best of both worlds

Every developer or system architect reuses existing code. That's why the Puppet DSC module uses 200-plus PowerShell DSC Resources that have already been released and tested by the PowerShell community. Some of these resources also cover scenarios not yet addressed by Puppet. There are also more than 4,500 Puppet modules on the [Puppet Forge](#) to support almost all operating systems, platforms and resources across the data center — Windows, Linux, network devices, storage arrays, containers, cloud infrastructures and more.

This means that whatever situation you're trying to address, there's likely a Puppet module or DSC Resource that can help you. Getting things done fast and correctly is key, and using existing Puppet modules or DSC Resources streamlines your work by plugging in already-tested code that solves the problems you're faced. You don't have to write code yourself — you can use the work from Puppet, Microsoft, or the community to get the job done. You'll also likely run into a situation where either Puppet or DSC doesn't cover the entirety of the problem all by itself. Instead of trying to make one solution fit all, you can mix and match Puppet and DSC to address the problem.

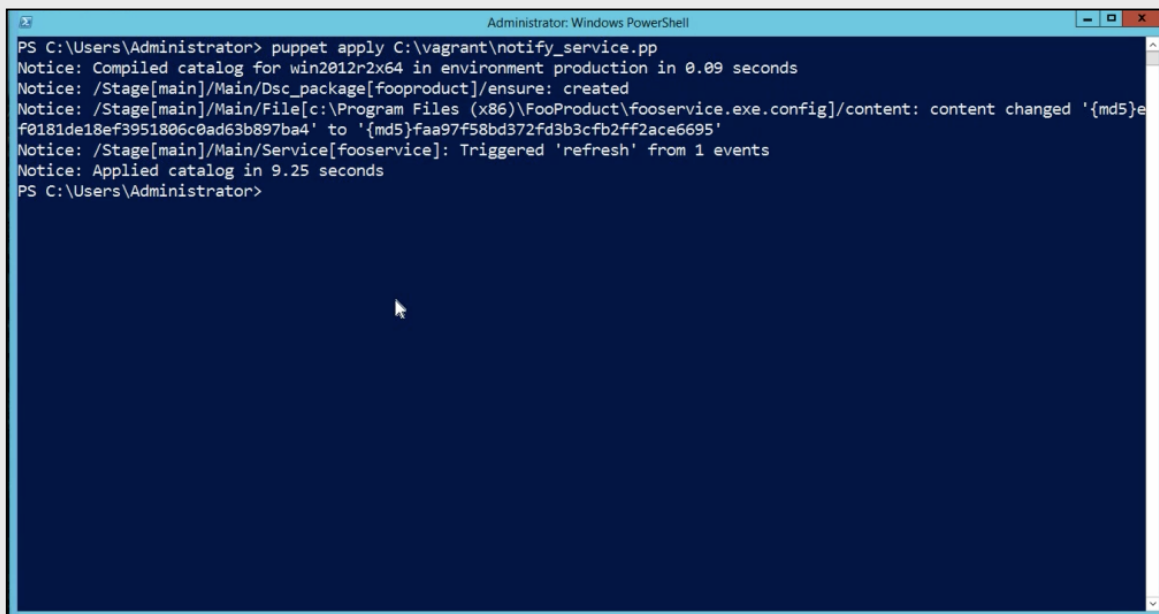
For example, one common task is updating a configuration file and restarting a system after a product is installed. We can model this by first declaring a PowerShell DSC resource that will handle installing MSI. Then we'll use the Puppet file module to ensure that the file is present on the target node, and continues to use the text we want. Lastly, we'll set the Puppet service module to ensure that the service is running.



```
1 dsc_package { 'fooproduct':
2   dsc_ensure    => "Present",
3   dsc_name      => "FooProduct",
4   dsc_productid => "{E1F581BF-09C4-48B2-991A-429D61F48199}",
5   dsc_path      => "c:\\vagrant\\artifacts\\foosetup.msi",
6 }
7
8 file { 'c:\\Program Files (x86)\\FooProduct\\fooservice.exe.config':
9   require => Dsc_package['fooproduct'],
10  notify  => Service['fooservice'],
11  content => @(APPCONFIG)
12  <?xml version="1.0" encoding="utf-8"?>
13    <configuration>
14      <appSettings>
15        <add key="environment" value="production"/>
16      </appSettings>
17    </configuration>
18  |- APPCONFIG
19 }
20
21 service { 'fooservice':
22   require => Dsc_package['fooproduct'],
23   ensure  => 'running',
24   enable  => true,
25 }
26
```

Notice that the line in the file declaration says the service module will be notified. That means if the file module changes anything, it will send a notification to Puppet to tell the service module to perform a refresh. The service module doesn't know anything about the config file or when it changes. It just receives the notification from Puppet, and initiates a refresh or restart. These change notifications make Puppet a powerful way to model and control change in your environment.

In this example, Puppet will start up, run, process the file, create the catalog, make the dependencies and examine the system to see if it needs to perform a certain set of operations.



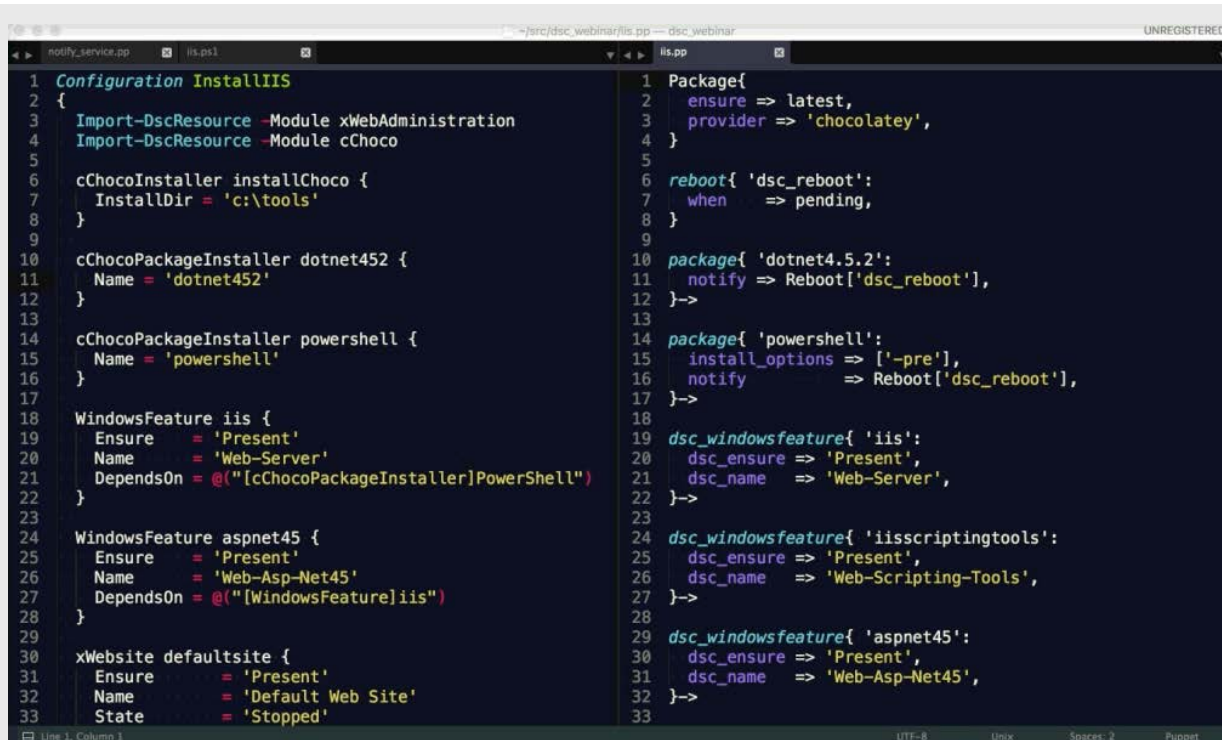
```
Administrator: Windows PowerShell
PS C:\Users\Administrator> puppet apply C:\vagrant\notify_service.pp
Notice: Compiled catalog for win2012r2x64 in environment production in 0.09 seconds
Notice: /Stage[main]/Main/Dsc_package[fooproduct]/ensure: created
Notice: /Stage[main]/Main/File[c:\Program Files (x86)\FooProduct\fooservice.exe.config]/content: content changed '{md5}ef0181de18ef3951806c0ad63b897ba4' to '{md5}faa97f58bd372fd3b3cfb2ff2ace6695'
Notice: /Stage[main]/Main/Service[fooservice]: Triggered 'refresh' from 1 events
Notice: Applied catalog in 9.25 seconds
PS C:\Users\Administrator>
```

Puppet will install the product, start the service, set the configuration file and then restart the service.

## Advanced scenario

Now you've seen a simple example of how Puppet and DSC can work together, let's move on to a more complicated example: how to deploy an ASP.NET website.

This example is more complicated, because there are several software modules needed for the ASP.NET website to run, as well as configuration settings that need to be set. We're using DSC along with Puppet because the IIS DSC resource has some features that we don't have in Puppet yet. And it will let us get the website up and running quickly.



```
1 Configuration InstallIIS
2 {
3   Import-DscResource -Module xWebAdministration
4   Import-DscResource -Module cChoco
5
6   cChocoInstaller installChoco {
7     InstallDir = 'c:\tools'
8   }
9
10  cChocoPackageInstaller dotnet452 {
11    Name = 'dotnet452'
12  }
13
14  cChocoPackageInstaller powershell {
15    Name = 'powershell'
16  }
17
18  WindowsFeature iis {
19    Ensure = 'Present'
20    Name = 'Web-Server'
21    DependsOn = @(["cChocoPackageInstaller"]PowerShell")
22  }
23
24  WindowsFeature aspnet45 {
25    Ensure = 'Present'
26    Name = 'Web-Asp-Net45'
27    DependsOn = @(["WindowsFeature"]iis)
28  }
29
30  xWebsite defaultsite {
31    Ensure = 'Present'
32    Name = 'Default Web Site'
33    State = 'Stopped'
34  }
35}
```

```
1 Package{
2   ensure => latest,
3   provider => 'chocolatey',
4 }
5
6 reboot{ 'dsc_reboot':
7   when => pending,
8 }
9
10 package{ 'dotnet4.5.2':
11   notify => Reboot['dsc_reboot'],
12 }->
13
14 package{ 'powershell':
15   install_options => ['-pre'],
16   notify => Reboot['dsc_reboot'],
17 }->
18
19 dsc_windowsfeature{ 'iis':
20   dsc_ensure => 'Present',
21   dsc_name => 'Web-Server',
22 }->
23
24 dsc_windowsfeature{ 'iiscriptingtools':
25   dsc_ensure => 'Present',
26   dsc_name => 'Web-Scripting-Tools',
27 }->
28
29 dsc_windowsfeature{ 'aspnet45':
30   dsc_ensure => 'Present',
31   dsc_name => 'Web-Asp-Net45',
32 }->
33
```

Above, you see two script files. On the left is our existing PowerShell DSC configuration file; on the right is the Puppet manifest that we ported from the DSC configuration file. When we move to the Puppet manifest, we immediately see a reduction in code ceremony that results in less code written, and less to think about when writing it.

We immediately see several benefits after writing the Puppet manifest. We don't have to worry about listing the DSC Resources module imports. We don't have to worry about having a correct configuration block declaration.

We can describe the dependencies between the resources using the simple dependency symbol, which results in less to write, as well as making it easier to move resources without having to update the syntax. In the Puppet manifest, global parameter defaults reduce the amount of repetitive code you have to write.



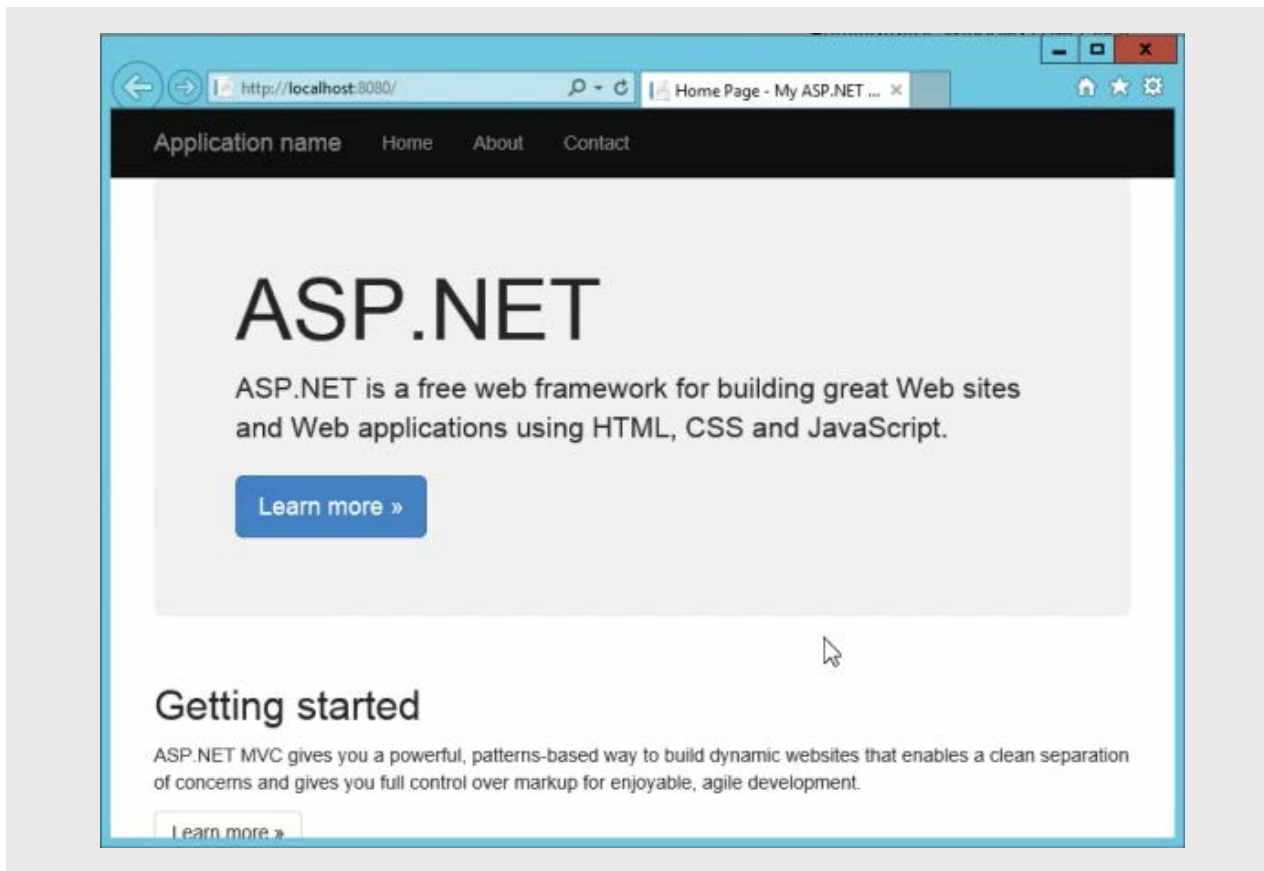
```
50 ManagedRuntimeVersion = 'v4.0'
51 LogEventOnRecycle      = 'Memory'
52 RestartMemoryLimit     = '1000'
53 RestartPrivateMemoryLimit = '1000'
54 IdentityType           = 'ApplicationPoolIdentity'
55 State                  = 'Started'
56 DependsOn              = @("["WindowsFeature]iis",
57                             ["WindowsFeature]aspnet45",
58                             ["ChocoPackageInstaller]dotnet452",
59                             ["File]websitefolder",
60                             ["XWebAppPool]newwebapppool")
61 }
62 xWebsite newwebsite {
63   Ensure = 'Present'
64   Name    = 'PuppetCodez'
65   State   = 'Started'
66   PhysicalPath = 'c:\inetpub\foo'
67   ApplicationPool = 'PuppetCodezAppPool'
68   BindingInfo = MSFT_xWebBindingInformation{
69     Protocol = 'HTTP'
70     Port     = 80
71   }
72   DependsOn = @("["WindowsFeature]iis",
73                 ["WindowsFeature]aspnet45",
74                 ["ChocoPackageInstaller]dotnet452",
75                 ["File]websitefolder",
76                 ["XWebAppPool]newwebapppool")
77 }
78 }
79 InstallIIS
80 Start-DscConfiguration -Path .\InstallIIS -Verbose -Wait
81
1 Package{
2   ensure => latest,
3   provider => 'chocolatey',
4 }
5
6 reboot{ 'dsc_reboot':
7   when => pending,
8 }
9
10 package{ 'dotnet4.5.2':
11   notify => Reboot['dsc_reboot'],
12 }->
13
14 package{ 'powershell':
15   install_options => ['-pre'],
16   notify          => Reboot['dsc_reboot'],
17 }->
18
19 dsc_windowsfeature{ 'iis':
20   dsc_ensure => 'Present',
21   dsc_name   => 'Web-Server',
22 }->
23
24 dsc_windowsfeature{ 'iis scripting tools':
25   dsc_ensure => 'Present',
26   dsc_name   => 'Web-Scripting-Tools',
27 }->
28
29 dsc_windowsfeature{ 'aspnet45':
30   dsc_ensure => 'Present',
31   dsc_name   => 'Web-Asp-Net45',
32 }->
33
```

Even in a complicated example like the one above, the syntax is nearly one-to-one. Someone looking at a PowerShell DSC configuration can look at the Puppet manifest file and immediately understand what's going on.

When we run the Puppet manifest, we see the output below.

```
Administrator: Windows PowerShell
PS C:\Users\Administrator> puppet apply C:\vagrant\notify_service.pp
Notice: Compiled catalog for win2012r2x64 in environment production in 0.09 seconds
Notice: /Stage[main]/Main/Dsc_package[fooproduct]/ensure: created
Notice: /Stage[main]/Main/File[c:\Program Files (x86)\FooProduct\fooservice.exe.config]/content: content changed '{md5}e
f0181de18ef3951806c0ad63b897ba4' to '{md5}faa97f58bd372fd3b3cfb2ff2ace6695'
Notice: /Stage[main]/Main/Service[fooservice]: Triggered 'refresh' from 1 events
Notice: Applied catalog in 9.25 seconds
PS C:\Users\Administrator> puppet apply C:\vagrant\iis.pp
Notice: Compiled catalog for win2012r2x64 in environment production in 0.25 seconds
Notice: /Stage[main]/Main/Dsc_file[websitefolder]/ensure: created
Notice: /Stage[main]/Main/Dsc_xwebapppool[newwebapppool]/ensure: created
Notice: /Stage[main]/Main/Dsc_xwebsite[newwebsite]/ensure: created
Notice: /Stage[main]/Main/Reboot[dsc_reboot]: Triggered 'refresh' from 3 events
Notice: Applied catalog in 18.64 seconds
PS C:\Users\Administrator>
```

You can see in the output above that the Puppet run has finished — and in just over 18 seconds. Here's the fully functioning ASP.NET website that was installed:





## Reporting with Puppet and DSC

Change reporting is a must for any company that has to comply with auditing. Being able to pinpoint when something has been successfully changed is a huge step toward gaining true control over your environment.

Seeing changes across your environment in an easily accessible manner reduces the amount of time spent verifying the change. You don't have to manually check the state, because the data is already collected for you. Historical reporting is equally important, because knowing *when* something broke is just as important as knowing *what* broke. If you can find out when the change happened, correlation to other events becomes easier, and reduces the amount of time you spend troubleshooting.

PowerShell DSC does not keep historical data on changes performed, but it does provide some ways for you to manually find out what changed. PowerShell shows the status of the whole operation, not the status on a resource-by-resource basis. This provides a good indication of the state of the last run, but you can't find out what changed over time. In PowerShell 4, this is accomplished by using the event logs and searching for the last run result. In PowerShell 5, you can use the `Get-DscConfiguration` status command to get the same information.

PowerShell results are available only for the target node, unless a DSC pull server is set up. The DSC pull server can store the last result for all the target nodes that have been configured to point to it. But it still requires a manual call to get to the information. You can script these calls to generate reports, but it's not built in.

Puppet can extend DSC by providing historical change tracking and reporting. Configuration results and history are available on the target nodes, as well as in the Puppet server Web UI. The Web UI provides a single interface that shows the status of your environment, down to the individual servers. It even shows resource-by-resource change. It shows the result of each DSC resource execution, and a log of the execution as well. Puppet keeps all of these reports. This lets you investigate over time how things change in your environment.

But how do you find errors that occur during a DSC configuration run? You can easily search the event log. It can be done manually with the event viewer, or by using the DSC diagnostics module commands `Get-DscOperation` and `Trace-DscOperation`. They can be run either locally or remotely.

`Get-DSC Operation` lists the statuses for the last few DSC operation runs, and returns an object that has information about the time it was created, whether the run was successful, and all the events generated by that run. You can use this command to find the specific DSC job that created the condition you want to investigate.

`Trace-DscOperation` takes the job ID or sequence ID from the `Get-DscOperation` command as parameters, and delivers a readable list of events that were generated by the respective operation. By default, `Trace-DscOperation` will list all the events generated. This command returns an object that contains the properties, like the event type, event message, and event creation. The results of this command are what you use to figure out which part of the DSC configuration is failing, and what the root cause is.

In Puppet, it's a different experience. With Puppet, errors are put up front. And they're easy to diagnose, because they're in the same UI used to find errors in DSC. These are the same messages you would see in the DSC, in the event log, or in the PowerShell console. But instead, they're viewable inside the Puppet Server UI. That means they can be exported in reports, investigated at later times, or examined among servers in your environment.

Puppet automatically logs the results of all of the actions taken either by Puppet or DSC. So you can review exactly what happened without having to go to the target node or run commands manually yourself. This is a huge time saver when you're on the line to figure out why something went wrong. It's just a few clicks, compared to several minutes running commands and searching through events.

## Setting up Windows for system and application monitoring

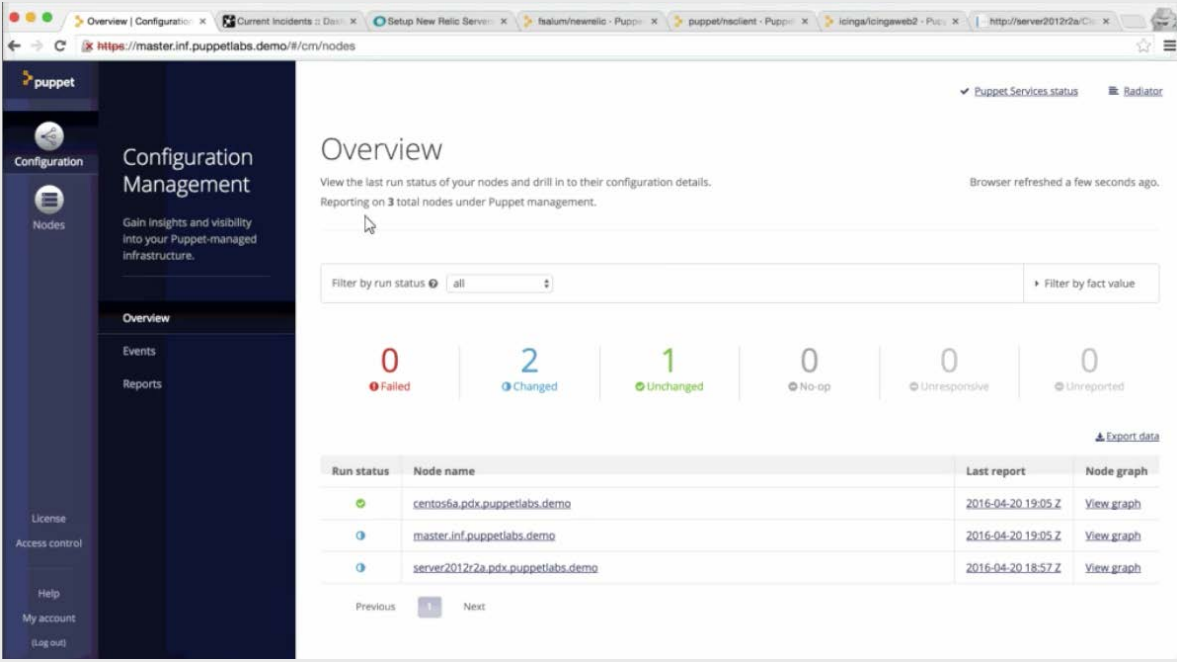
Consistent monitoring is crucial for any IT shop. When an incident happens — a system goes down, experiences unusually heavy load, or an application isn't performing properly — you'll want to be able to diagnose it quickly and accurately.

Puppet can deploy web applications with robust measuring tools in an automated way, whether you have 100 nodes or 1,000. And Puppet can be used with a wide range of monitoring solutions, including Windows monitoring tools.

Let's look at how monitoring works with Puppet and a couple of popular monitoring tools. Nagios is a very popular monitoring tool that traditionally has been used with Linux, but it can also be used with Windows systems. The Nagios and NSClient can deliver metrics about server configuration, load, and more. There are also plugins to monitor IIS servers, traffic and application status. New Relic can also be used with Windows to gather even more information about servers.

### Monitoring with Puppet and Nagios

Puppet can easily coordinate the relationship between Nagios servers and agents. In this example, Puppet will be used to run Icinga (a version of Nagios) on a CentOS machine. There will be two Windows machines, master and server 2012 boxes. (In this example, the Puppet agent has been installed on all machines.)



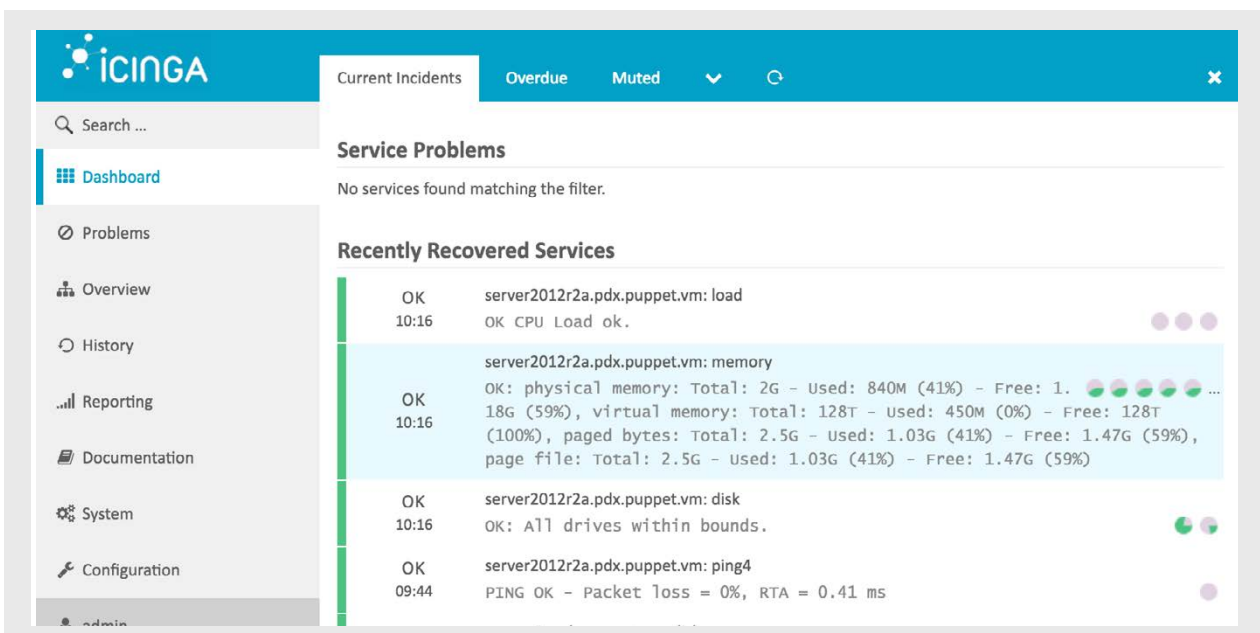
The screenshot displays the Puppet Configuration Management web interface. The left sidebar contains navigation links for Configuration, Nodes, Overview, Events, Reports, License, Access control, Help, My account, and Log out. The main content area is titled 'Overview' and shows the last run status of nodes. It reports on 3 total nodes under Puppet management. A filter by run status is set to 'all'. The status summary shows 0 Failed, 2 Changed, 1 Unchanged, 0 No-op, 0 Unresponsive, and 0 Unreported. Below this is a table with columns for Run status, Node name, Last report, and Node graph.

Run status	Node name	Last report	Node graph
Unchanged	centos6a.pdx.puppetlabs.demo	2016-04-20 19:05 Z	<a href="#">View graph</a>
Changed	master.inf.puppetlabs.demo	2016-04-20 19:05 Z	<a href="#">View graph</a>
Changed	server2012r2a.pdx.puppetlabs.demo	2016-04-20 18:57 Z	<a href="#">View graph</a>

With Puppet, adding the necessary components to Windows machines for Icinga monitoring is as simple as adding a class to the server profiles in Puppet. The class `profile::icinga_win_node` uses the Windows installer Chocolatey to install the components needed to link Windows servers to Icinga on the CentOS machine.

After adding the Icinga class, trigger a Puppet run from the dashboard or from the Puppet console. Puppet will automatically install and configure all the components, then link the Windows machines back to Icinga. NSClient++ will run locally on the Windows machine and report metrics back to your Icinga server — like CPU load, whether updates need to be installed, all the information around memory and capacity that will be related to that system.

The Icinga dashboard shows metrics about the Windows machines' performance:



HostServiceServicesHistory

UP

for 5m 21s

OK

since 10:16

server2012r2a.pdx.puppet.vm

10.20.1.163

Service: memory

Plugin Output

OK: physical memory: Total: 2G - Used: 840M (41%) - Free: 1.18G (59%), virtual memory: Total: 128T - Used: 450M (0%) - Free: 128T (100%), paged bytes: Total: 2.5G - Used: 1.03G (41%) - Free: 1.47G (59%), page file: Total: 2.5G - Used: 1.03G (41%) - Free: 1.47G (59%)

Problem handling

Comments

Add comment

Downtimes

Schedule downtime

Performance data

Label	Value	Max	Warning	Critical
paged bytes	1.03	2.50	2.00	2.25
page file	1.03	2.50	2.00	2.25
physical memory	839.73	2,047.55	1,638.04	1,842.80
paged bytes %	41%	-	80%	90%
physical memory %	41%	-	80%	90%
page file %	41%	-	80%	90%
virtual memory	449.75	134,217,727.88	107,374,182.30	120,795,955.09
virtual memory %	0%	-	80%	90%

New Relic monitoring can also be used with Puppet. Add the `profile::newrelic_server` class. Then define the source of the New Relic component and the license information in the Puppet code for that class:

```
node default {
  class {'newrelic::server::windows':
    newrelic_license_key => 'your license key here',
  }
  class {'newrelic::agent::dotnet':
    newrelic_license_key => 'your license key here',
  }
}
```

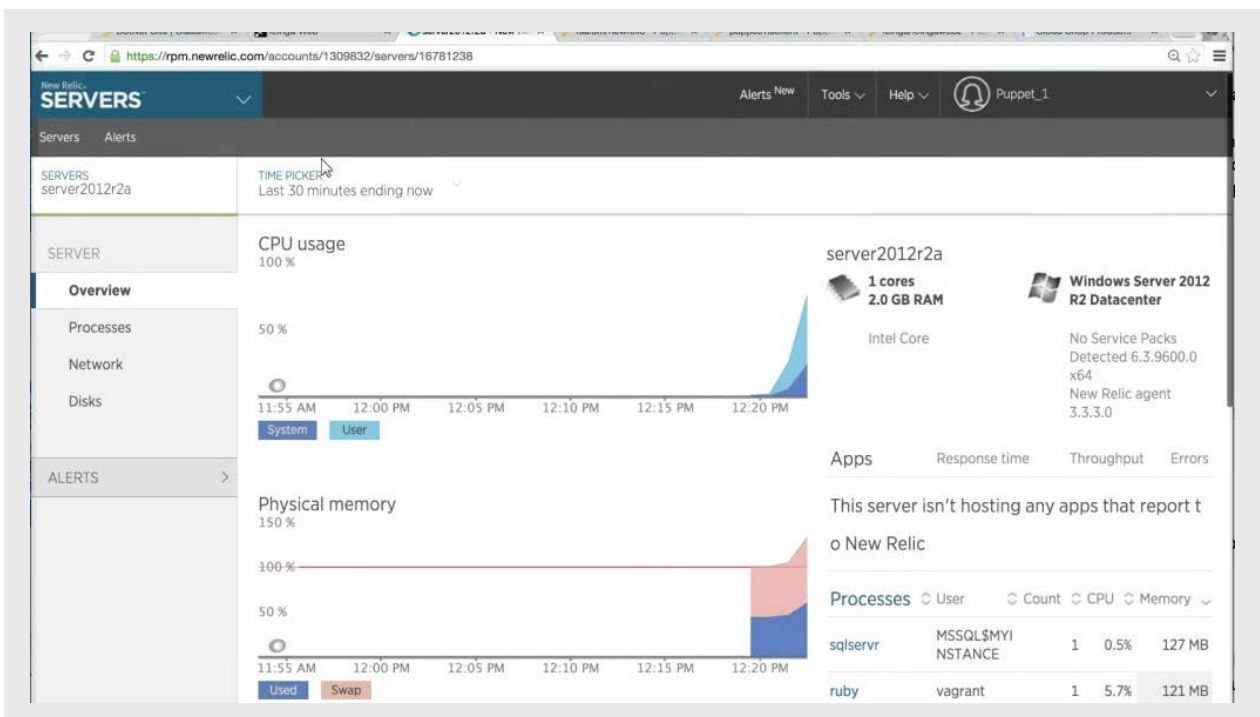
After setting license keys in the Puppet code, the New Relic class can be used to install and configure a New Relic agent on every Windows server in your infrastructure. With Puppet, New Relic can be further customized to monitor individual apps and Windows servers.

You can use Puppet modules to build a fully functional .NET website. The next example will show how to use Puppet not only to install a .NET site, but also to monitor it.

From the Puppet console, add the corresponding classes to automatically set up applications, SQL server components and database components. In this example, the Windows server has already been configured with these components.

After setting up a .NET site, add a New Relic class that will collect information about the application. In this case, it's the `profile::newrelic_win_node` class, which manages the Windows feature for installing IIS and the additional New Relic components.

After Puppet installs the components, we can see that the server is now reporting CPU usage and processes, including information about the MySQL instance. In this example, the app is vCloudShop, a simple shopping cart.



As the server and application runs, New Relic will deliver more detailed graphs and metrics around each one of the app components.

## Patch management on Windows with Puppet

Windows has a number of technologies that enable effective patch management on the operating system layer. Operating-system patches can be stored in a central repository, which can live on the internet, through Windows Update, or you can host the repo internally.

Windows Server Update Services (WSUS) and System Center Configuration Manager (SCCM) are both used to distribute patches in a Windows infrastructure. WSUS distributes updates through Microsoft Update, and pulls them into your organization through a WSUS server. Rather than going directly to Windows Update, updates are downloaded to one or more target WSUS servers. Those servers then stage the updates, and clients pull them from those servers to their systems.

SCCM integrates with and extends the functionality of WSUS, providing enhanced capabilities that help automate patch management workflows in a way that wasn't possible with WSUS alone. SCCM lets you define targets or deployment of patches, and define schedules for downloading updates and making them available to servers. SCCM can also indirectly manage WSUS clients behind the scenes by pushing an updated schedule for patch delivery to target systems.

These technologies are generally used by sysadmins to keep the operating system up to date. Application updates are generally managed by other teams through other means. This reinforces the traditional notion of the operating system as a platform.

### Managing patches with Puppet

With Puppet, OS patches and application updates are treated as interlocking pieces. Patches and application updates can be tested and deployed together. With package management, updates can be delivered from curated repositories. Packages are atomic bundles that support versioning and metadata. When software is distributed with packages, sysadmins can check to make sure the latest version of the application is installed, and check for other software it may depend on. Finally, packages can also allow scripts to be run. These scripts can be used to remove temporary files after package installation.

Packages can be hosted on a centralized repository, which allows them to be searched and maintained more easily. This notion of centralized package management is fundamental to the way that Puppet manages application state. Although Linux has used package management for quite some time, this isn't something that has always been a part of the Windows application management methodology. However, in the last several years, more Windows package management tools have emerged.

## Chocolatey and Puppet

Chocolatey, described in the chapter [Managing software on Windows with Chocolatey](#), is a Windows package manager developed by Puppet software engineer Rob Reynolds. Chocolatey can be used to manage third-party and internal software. It has the notion of dependencies and versions, and allows you full access to PowerShell for the automation scripts with built-in functions that can reduce complex tasks down to simple function calls.

Chocolatey can be used to install or uninstall any application and its dependencies. This allows patches to be applied much more naturally at the application level. It is also possible to associate a patch level of the operating system with a patch level of all the other applications running on a system.

With Puppet and Chocolatey, we can now completely manage the state of a package on Windows automatically. For example, MySQL can be installed and configured on a Windows system using Puppet and Chocolatey:

```
file { 'c:/mysql/my.ini':  
  ensure => 'file',  
  mode   => '0660',  
  owner  => 'mysql',  
  group  => 'Administrators',  
  source => 'N:/software/mysql/my.ini',  
}
```

Puppet can also manage the state of the installed packages on the system:

```
package{ 'mysql':  
  ensure => latest,  
}
```

Microsoft's Desired State Configuration (DSC) works in a very similar way to Puppet. DSC manages Windows-native resources with code. DSC is fully compatible with Puppet, which means Puppet code can be used to define DSC. Patches can be treated as things that apply to the entire system.

Here's a quick look at how to manage WSUS clients on a Windows system. This Puppet code specifies a server URL and the patch schedule. The code can be applied to multiple systems, and works with virtually any OS — Windows, Linux, Solaris, etc.

```
Class { '::wsus_client':  
  server_url           => 'http://server2012r2a.puppet.demo:8530',  
  auto_update_option   => 'Scheduled',  
  scheduled_install_day => 'Tuesday',  
  scheduled_install_hour => 2,  
}
```



Chocolatey and Puppet modules deliver full control over third-party patching. Because Puppet works with virtually any operating system, using Puppet makes it easy to configure complex software management across any heterogeneous infrastructure.

## Managing software on Windows with Chocolatey

Chocolatey is a package manager for Windows, like Yum. It uses the NuGet packaging framework and PowerShell for automation scripts. Chocolatey can use non-centralized and private repositories' custom packages. Chocolatey is a Microsoft-validated tool, and any installer, zip, or binary can be packaged with Chocolatey.

Chocolatey can manage all aspects of Windows software: installation, configuration, upgrade and uninstalling. It works with runtime binaries, zips and all existing installer technologies, including MSI, NSIS, and InnoSetup. Chocolatey takes advantage of PowerShell to turn complex tasks into simple function calls.

The Chocolatey community repository has thousands of packages that anyone can use, but it is recommended that organizations build and host their own packages on an internal package repository server (rather than using the packages on Chocolatey.org) to allow for a completely reliable, repeatable process, and to maintain trust and control.

The Chocolatey Puppet provider is a Puppet Supported module that can manage the installation and configuration of Chocolatey, and manage packages (a.k.a. software management). There are three configuration resources that allow for managing config settings, features, and repository locations (sources). Here are examples of ensuring Chocolatey is installed:

```
include chocolatey
# OR
class {'chocolatey':
  chocolatey_download_url => 'http://url/to/chocolatey.nupkg',
}
```

The first option ensures Chocolatey is installed based on its default install location at Chocolatey.org. Most organizations will want to fully control this, so they will opt for the second option and host the Chocolatey package and installer internally. More options can be seen at

<https://forge.puppet.com/puppetlabs/chocolatey#class-chocolatey>.

```
package {'name_of_package':
  provider      => chocolatey,
  ensure        => absent, installed, latest, '1.0.0', held,
  source        => 'http://some_odata_feed/;c: \\local;\\some\\network\\
share',
  install_options => ['-installArgs', '', 'addtl', 'args', ''],
  uninstall_options => ['-uninstallargs', '', 'addtl', 'args', ''],
}
```

The previous image shows the anatomy of a package resource. Chocolatey is the provider, and can ensure software is removed, installed, stays up to date, is a particular version, or hold a package on a version.

Default sources can be specified in Chocolatey itself, through Puppet code. Packages can be pulled from one source or more (separate sources with semicolons). Sources can be a folder share or an HTTP NuGet OData feed. It's easy to get started by using a file/folder share for your package source, and you can easily move into a simple OData server and/or a gallery server later. See <https://chocolatey.org/docs/how-to-host-feed> for more details and options.

Why Chocolatey? It's a unified interface to all different types of installers. Here's the difference between the built-in Windows package provider and Chocolatey:

```
#Built-in provider
package { "Git version 2.6.1":
  ensure      => installed,
  source      => 'C:\temp\Git-2.6.1-32bit.exe',
  install_options => ['/VERYSILENT']
}

#Chocolatey provider
package { 'git':
  ensure      => latest,
}
```

Note the differences. For the built-in provider, when we need to upgrade, we'll run into maintenance issues. We'll need to change the manifest over time. Instead, if we take a look at the Chocolatey provider, we can see it looks just like the package provider for other operating systems. The packaging is platform-agnostic. The exact same packaging can be used to ensure the latest version of Git is installed across *all* of your platforms and operating systems.

## Creating packages

When you are creating packages, we suggest you use the `choco.exe` client tool and/or Chocolatey's **Package Builder** to do that. To get started, run `choco new nameofpackage`, and inspect the output. You can also see <https://chocolatey.org/docs/create-packages> for detailed information on creating packages.

When you run `choco new`, there are a few files in the default Chocolatey package template, as it is meant to provide a generalist approach to account for many of the different use cases and packaging types. Chocolatey will by default generate a README file, install and uninstall PowerShell scripts, and create a **nuspec** — a way of describing the packaging format metadata. The nuspec contains information about the version of the software, any dependencies and more.

You can create **custom templates** as you start to get familiar with the different concepts of packaging types. For instance, you could create a template strictly for MSI packages.

## Deploying IIS and ASP.NET with Puppet

Two of the most common tasks for Windows admins are deploying ASP.NET and deploying IIS. These are both much easier to do with Puppet.

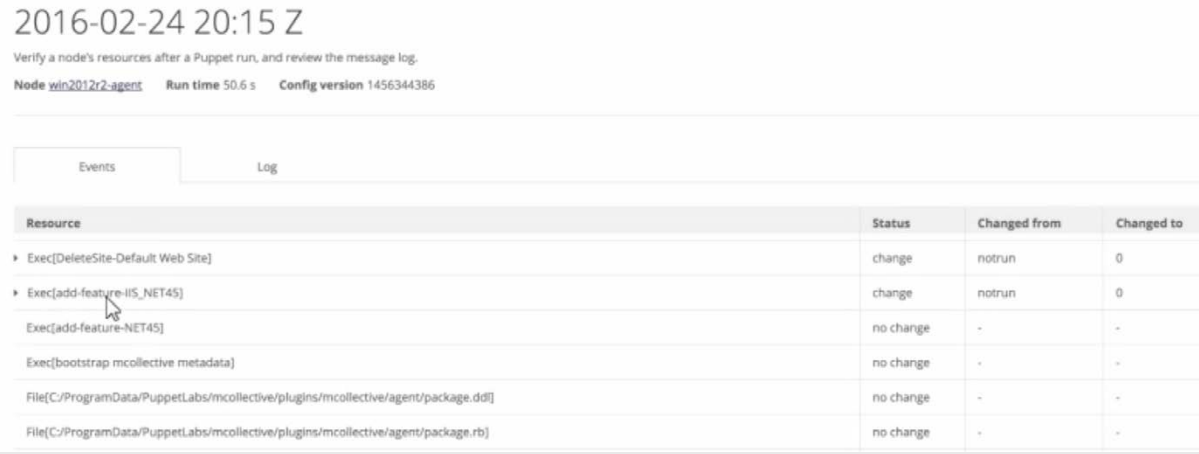
First, write a basic manifest that defines what you want, and then apply it to your Windows servers.

This is what the code looks like (note this is for Windows Server 2012, and some of the specifics may be different for other versions of Windows):

```
class widemo::iis_enable {
  Include widemo::dotnet_enable
  Windowsfeature{'IIS_NET45':
    feature_name => [
      'Web-WebServer',
      'Web-Http-Errors',
      'Web-Http-Logging',
      'Web-Asp-Net45',
      'NET-Framework-45-ASPNET',
    ],
    installmanagementtools => true,
  } ~>
  # Remove default binding by removing default website
  # (so it can be used by something else)
  Iis::manage_site {'Default Web Site':
    ensure => absent,
    site_path => 'any',
    app_pool => 'DefaultAppPool',
  }
}
```

Use the Windows Feature Module to enable .NET Framework 4.5. Also, enable IIS and turn on HTTP errors and logging. Finally, use the community-provided **Vox Pupuli IIS module** to remove the default binding.

Puppet runs and generates a report. In this case, the report shows that a couple of things were changed:



The screenshot shows a web interface for a Puppet report. At the top, it displays the date and time '2016-02-24 20:15 Z' and a message: 'Verify a node's resources after a Puppet run, and review the message log.' Below this, it shows 'Node win2012r2-agent', 'Run time 50.6 s', and 'Config version 1456344386'. There are two tabs: 'Events' and 'Log'. The 'Events' tab is active, showing a table of resource changes.

Resource	Status	Changed from	Changed to
Exec[DeleteSite-Default Web Site]	change	notrun	0
Exec[add-feature-IIS_NET45]	change	notrun	0
Exec[add-feature-NET45]	no change	-	-
Exec[bootstrap mcollective metadata]	no change	-	-
File[C:/ProgramData/PuppetLabs/mcollective/plugins/mcollective/agent/package.ddf]	no change	-	-
File[C:/ProgramData/PuppetLabs/mcollective/plugins/mcollective/agent/package.rb]	no change	-	-

Puppet executed the commands necessary to set up a basic web application. In addition, the IIS management tools were installed, and the Puppet module executed PowerShell code to remove the default website.

With the following code, Puppet can also install SQL Server Compact Edition, which our demo ASP.NET application requires.

```
class windemo::sqlce {
  $installer = 'SSCERuntime_x64-ENU.exe'
  package { 'Microsoft SQL Server Compact 4.0 SP1 x64 ENU':
    ensure => '4.0.8876.1',
    provider => 'windows',
    # NOTE: would like to use this Puppet style, but must have file
    # source => "puppet:///modules/widemo/${installer}",
    source => "C:/vagrant/modules/windemo/files/${installer}",
    Install_options => [ '/1', '/passive' ] # [ '/qn' ] #/l*v install
  }
}
```

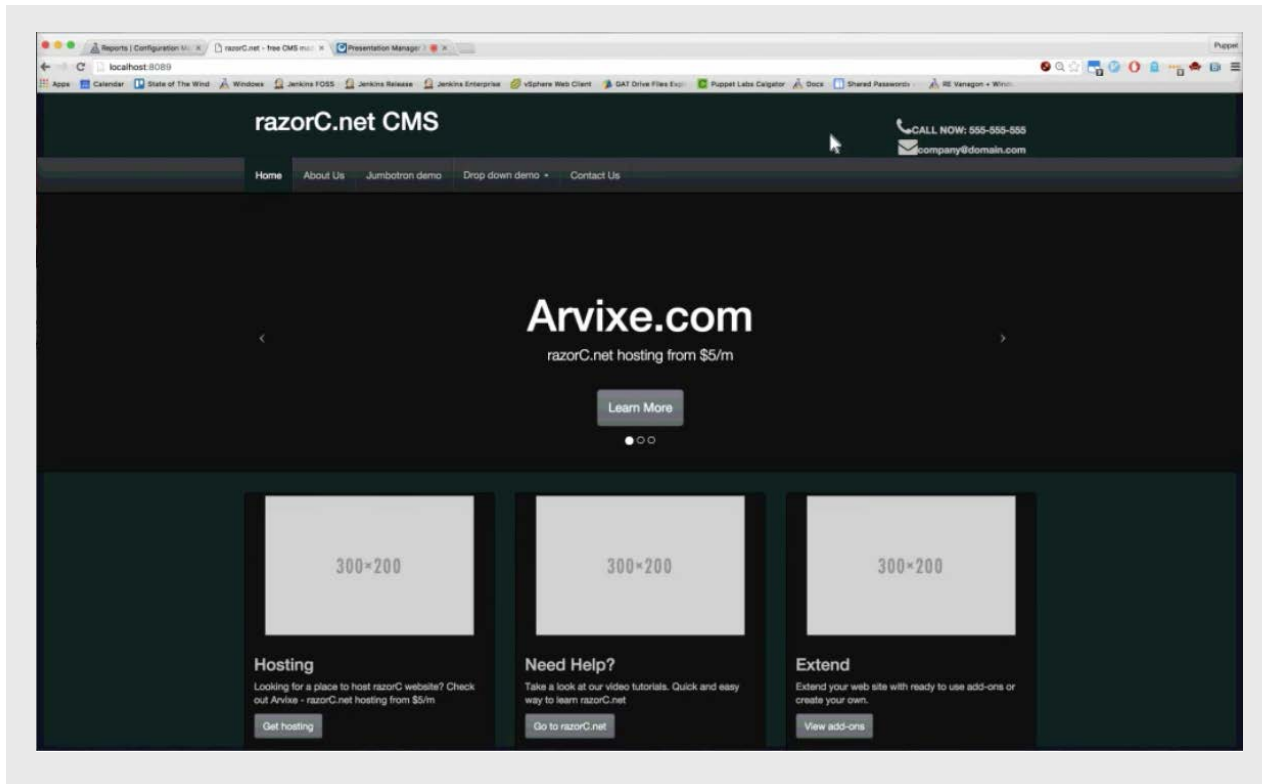
Now Puppet can install a bare-bones application that uses the SQL Server installed in the previous step. In this case, it's Razor C, which is an ASP.NET-based CMS. Here's what the Puppet code looks like for installing Razor C:

```
# == Class: mvcapp
#
# This class installs the razorC MVC application
#
class windemo::mvcapp {
  $app_zip = 'razorC_v1.1.1.zip'
  $app_zip_path = "C: \\Windows\\Temp\\${app_zip}"
  $app_pool = 'mvc'
  $app_location = 'C:\\inetpub\\wwwroot\\razorC'
  file { "${app_zip_path}":
    ensure => file,
    source => "puppet:///modules/windemo/${app_zip}",
    source_permissions => ignore,
  } ~>
  iis::manage_app_pool {"$app_pool":
    ensure => present,
    enable_32_bit => true,
    managed_runtime_version => 'v4.0',
    managed_pipeline_mode => 'Integrated',
  } ~>
  #NOTE: IIS is very touchy around extra slashes
  Iis::manage_site {'razorC':
    ensure => present,
    site_path => "${app_location}",
    port => '80',
    Ip_address => '*',
  }
```

Puppet has been pointed to a zip file containing the full distribution of the application. It will copy the zip file from the module, stage it in a temporary directory, and extract it to the default location for ASP.NET applications on disk at `c:\\inetpub\\wwwroot`. There is also an application pool and a site mapped to Port 80, pointing to the location on disk where the Razor C application has been extracted.

With just a short bit of manifest code, you can configure Windows to run IIS and install a CMS application.

The deployed application can be viewed in a browser:



## Active Directory management

Managing users and groups in a Windows environment is a big job for any sysadmin. Creating new users and setting permissions can be arduous. Security audits can also slow things to a crawl. With Puppet, you can manage all your users and groups from a single point of truth and automatically ensure they remain in compliance with security standards.

### Installing Active Directory with Puppet

In this example, we'll be using Puppet to install and manage Active Directory, then manage some domain users and groups.

The first step is defining the server in our infrastructure that'll be used as a domain controller and host Active Directory. In Puppet, we can put machines in groups defined by their attributes or facts. Once you create a group, any new machines you add to your infrastructure will be added to that group. In this case, we have an environment with a few Windows domain controllers, so we can create a group that looks for them. It's easy to define a group based on operating system type using the `osfamily` fact. For this example, we've created a group that only includes our single Windows domain controller. Once we have a group, we can use Puppet to install Active Directory.

First, we'll go to the Puppet Forge and nab some code to get us started. There are dozens of Windows modules available, but for this we'll use the `windows_ad` module.

#### Usage

Class: windows\_ad

```
1  Example - Create a new forest
2  class {'windows_ad':
3      install           => present,
4      installmanagementtools => true,
5      restart           => true,
6      installflg        => true,
7      configure         => present,
8      configureflg      => true,
9      domain            => 'forest',
10     domainname         => 'jre.local',
11     netbiosdomainname  => 'jre',
12     domainlevel        => '6',
13     forestlevel        => '6',
14     databasepath       => 'c:\\windows\\ntds',
15     logpath            => 'c:\\windows\\ntds',
16     sysvolpath         => 'c:\\windows\\sysvol',
17     installtype        => 'domain',
18     dsrmppassword      => 'password',
19     installdns         => 'yes',
20     localadminpassword => 'password',
21 }
```

Using this module, we can quickly get Active Directory installed on our server. Just change the parameters to meet your needs — domain names, data base path, log path, etc. We install the `windows_ad` module, then apply the class to the server we defined earlier. When we trigger a Puppet run, Active Directory will be automatically installed and the system will be restarted.

Now we can move on to managing users and groups with Puppet.

## Managing Windows users and groups with Puppet

First, we'll define our groups in the same way we defined our servers. In this example, we'll write code that defines our user groups. In this case, we can define users for both RedHat and Windows with a few simple lines of code:

23 lines (19 sloc) | 348 Bytes

```
1  class admin_users (
2    $userlist = [ "temp" ]
3  ){
4
5    case $::osfamily {
6      'RedHat': {
7        user { $userlist:
8          ensure => present,
9          groups  => ['wheel'],
10         managehome => true,
11       }
12     }
13
14     'windows': {
15       user { $userlist:
16         ensure => present,
17         groups  => ['Administrators'],
18       }
19     }
20   }
21 }
22 }
```

In this class, we're defining admin users. If the OS is RedHat, users will be placed in the `wheel` group. If it's Windows, they'll be placed in the `Administrators` group. We can also add a parameter that defines the users themselves. We can do this in Puppet code, or simply define them through the Puppet console:



# Local Users

Parent [All Nodes](#)

Environment [production](#)

[Edit node group metadata](#) [Remove node group](#)

[Rules \(3 changes\)](#)

[Matching nodes](#)

[Classes \(1 change\)](#)

[Variables](#)

[Activity](#)

Add or modify classes that apply to this group and its child groups.

Class definitions updated 13 minutes ago [Refresh](#)

Add new class

Class name

[+ Add class](#)

Class: `profile::admin_users`

Parameter		Value	
<input type="text" value="userlist"/>	=	<input type="text" value="['grace','bryan','joe']"/>	<a href="#">Add parameter</a>

[Discard this class](#)

[Discard changes](#)

[Commit 4 changes](#)

When we save this code, the next time Puppet runs it will create the admin users we defined on our Linux and Windows machines. We can define more about our users in the `windows_ad` module:

For adding a simple User :

```
1  windows_ad::user{'Add_user':
2    ensure                               => present,
3    domainname                           => 'jre.local',
4    path                                 => 'OU=PLOP,DC=JRE,DC=LOCAL',
5    accountname                          => 'test',
6    lastname                             => 'test',
7    firstname                            => 'test',
8    passwordneverexpires                 => true,
9    passwordlength                       => 15,
10   password                             => 'M1Gr3atP@ssw0rd',
11   xmlpath                              => 'C:\\\\users.xml',
12   writetoxmlflag                       => true,
13   emailaddress                         => 'test@jre.local',
14 }
```

Here we can define many parameters, including account name, first and last names, email address, passwords and more. You can use this module to create multiple users across your infrastructure in minutes.

## Managing Active Directory with Puppet

Once you've used Puppet to set up an Active Directory server, you can manage its users and groups using the `domain_membership` module:

### domain\_membership

Manage Active Directory domain membership with this module.

#### Parameters

- `domain` - AD domain which the node should be a member of.
- `username` - User with ability to join machines to a Domain.
- `password` - Password for domain joining user.
- `machine_ou` - [Optional] OU in the directory for the machine account to be created in.
- `resetpw` - [Optional] Whether or not to force machine password reset if it becomes out of sync with the domain.
- `reboot` - [Optional] Whether or not to reboot when the machine joins the domain. (reboot by default)
- `join_options` - [Optional] A bit field for options to use when joining the domain. See [http://msdn.microsoft.com/en-us/library/aa392154\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/aa392154(v=vs.85).aspx) Defaults to '1' (default domain join).
- `user_domain` - [Optional] Domain of user account used to join machine, if different from domain machine will be joined to. If not specified, `domain` will be used.

#### Usage

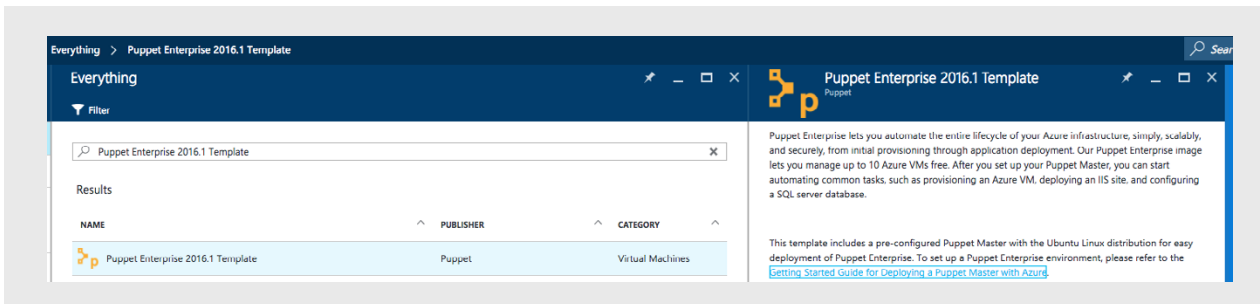
```
1 class { 'domain_membership':  
2   domain      => 'puppet.example',  
3   username    => 'joinmember',  
4   password    => 's!p3r_s3cR3t!',  
5   join_options => '3',  
6 }
```

The code for this module is incredibly simple. With the class `domain_membership`, we can add a user and password quickly and easily to your Active Directory server. In fact, managing users and groups with Puppet on any OS or system is quick and easy.

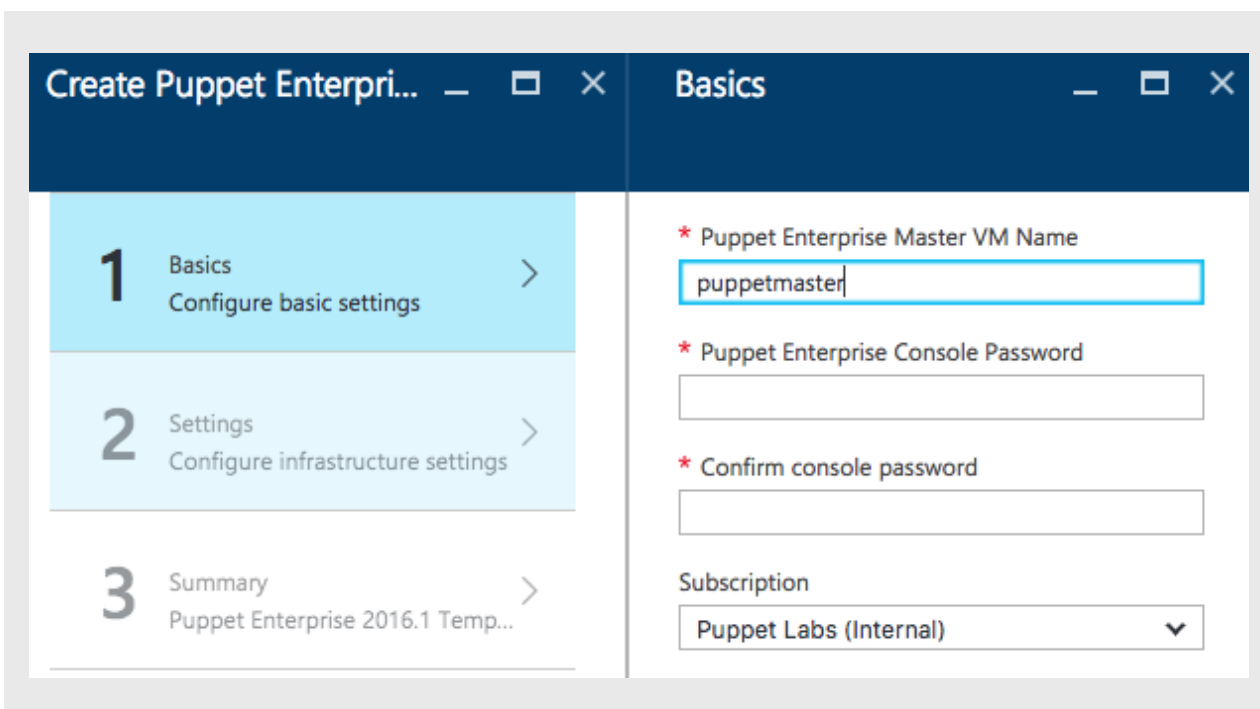
## Puppet and Microsoft Azure

Microsoft has recently been working with open source communities to support a wide variety of platforms. Azure is now an open cloud platform. Puppet is available through the Azure Marketplace, making it easy to deploy and manage a virtual infrastructure using Puppet modules.

Deploying Puppet-managed virtual machines is now as easy as deploying any VM in Azure. And you can enjoy the advantage of managing your Azure VMs with the same platform you use for your physical infrastructure. Simply search for **Puppet Enterprise 2016.1 Template** in the Azure Marketplace to get started.



As shown in the image below, you click on the Puppet Enterprise module to get started. Create and configure Puppet virtual machines through the Azure dashboard. Set username/password, then select machine size. Standard D2 V2 Azure machine configurations are highly recommended — they're fast, inexpensive and run Puppet really well.

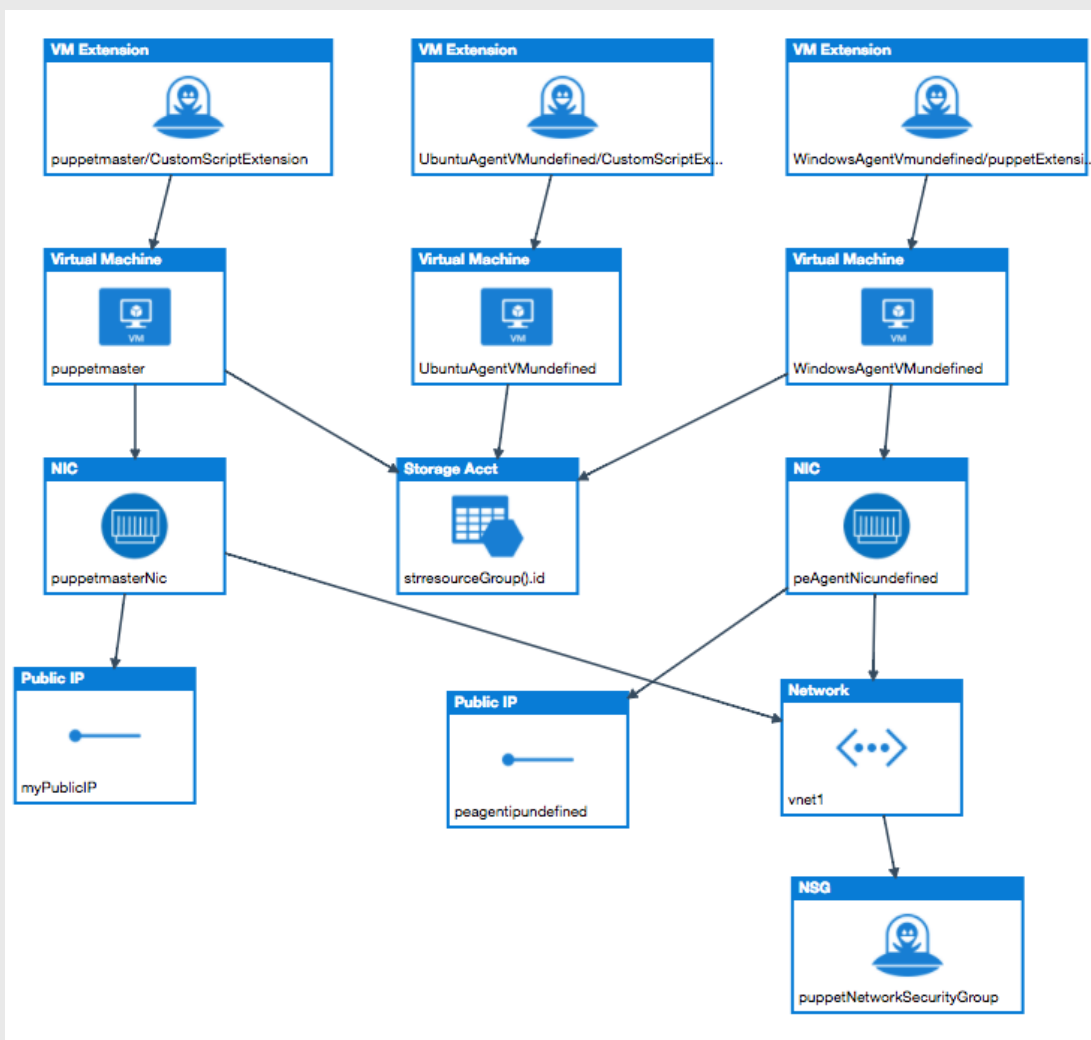


A preconfigured system will be created to run and test Puppet. Storage, security ports and more will be prefilled with common settings, but can be changed to meet specific needs.

Creating a virtual Linux machine (using Ubuntu 14.04), takes about five minutes. Once the machine has been created, Puppet will run a set of install scripts for approximately 10 minutes.

Next, create a Windows Server 2012 R2 data center image through the Azure resource manager. Install a Puppet extension to configure the Puppet master for each virtual machine. It's possible to connect hundreds of agents to a single Puppet master. For examples of this, please check out our sample [template](#).

Once your Puppet network is set up, it's easy to visualize the network using the [Azure Resource Manager Template Visualizer](#):



Once you've configured and deployed your Azure network, you can manage it like any other Puppet cluster. For more detailed instructions, check out our white paper [here](#).

## Conclusion

This ebook was created to help you automate Windows with the help of Puppet, a solution that you can use to automate *all* your infrastructure, no matter which operating systems you're running.

Puppet has a large, active and friendly community of people who enjoy helping each other. We invite you to reach out to our company and the community in a number of ways:

- [Puppet Google Group and mailing list](#)
- [Puppet Q&A site](#)
- [Puppet User Groups \(PUGs\)](#)

**Want to know more about how Puppet can help your organization?**

Please contact [sales@puppet.com](mailto:sales@puppet.com).