Ansible Windows Modules

In this chapter, we will look at the ever-growing collection of built-in Ansible modules that support and interact with Windows-based servers; coming from an almost exclusively macOS and Linux background, it seemed odd to be using a tool not natively supported on Windows to manage Windows.

By the end of our time in this chapter, I am sure you will agree that looking at the options available, the Ansible developers have made managing Windows Server workloads with a playbook as seamless and familiar as possible.

In this chapter, we will learn how to do the following:

* Launch a Windows server instance in Microsoft Azure
* Enable features in Windows
* Create users
* Install third-party packages using Chocolatey

The chapter covers the following topics:

* Launching a Windows server in Azure
* Ansible preparation
* The Windows Playbook roles
* Running the Playbook

Technical requirements

Rather than trying to run a full Windows Server 2022 locally in a **virtual machine** (**VM**), in this chapter, we will cover securely launching and configuring a Windows Server 2022 VM hosted in Microsoft Azure. If you are following along, you must have an active Microsoft Azure subscription and the Azure **command-line interface** (**CLI**) installed.

For details on how to install and configure the Azure CLI, please see the documentation at https://learn.microsoft.com/en-us/cli/azure/install-azure-cli/. If you are following along on a Windows host, I recommend installing the Azure CLI within your Windows Subsystem for Linux installation alongside where you installed Ansible.

Launching a Windows server in Azure

We will not use Ansible to deploy the Azure resources as we will do in [*Chapter 9*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/9), *Moving to the Cloud*; instead, we will use the Azure CLI to launch our VM.

Note

As some of the commands in this chapter will be pretty long, I will break them up with a backslash. In Linux command lines, the backslash (\) followed by a newline is a line continuation character. It lets you split a single command over multiple lines for better readability.

Start by changing to the Chapter07 folder within your checked-out copy of the repository that accompanies this title and run the following commands:

$ MYIP=$(curl https://api.ipify.org 2>/dev/null)

$ VMPASSWORD=$(openssl rand -base64 24)

$ echo $VMPASSWORD > VMPASSWORD

CopyExplain

The first two commands set two variables on your command line; the first uses the **ipify** service (https://www.ipify.org/) to populate the $MYIP variable with the public IP address of your current network session.

The second generates a random password using the openssl command and assigns it to the variable called $VMPASSWORD.

The third command copies the content of $VMPASSWORD to a file called VMPASSWORD; this command must be executed in the same folder as the host inventory file, as it will be called in our host inventory file, which we will discuss later in the chapter.

Note

I will follow the Azure Cloud Adoption Framework recommendations around resource naming and launching the resources in the UK South region.

Now that we know our IP address and have a password, we can start using the Azure CLI to launch resources. The first thing we need to do is make sure that we are logged in by running the following:

$ az login

CopyExplain

Once logged in, we can then create an **Azure Resource Group** by executing the following:

$ az group create \

    --name rg-ansible-windows-server-uks \

    --location uksouth

CopyExplain

The Azure Resource Group is the logic container we will be deploying our Azure resources to, the first of which will be an **Azure Virtual Network**.

The following command will create the Azure Virtual Network with an address space of 10.0.0.0/24 and a single subnet using 10.0.0.0/27; this is where we will launch our Windows Server:

$ az network vnet create \

    --resource-group rg-ansible-windows-server-uks \

    --name vnet-ansible-windows-server-uks \

    --address-prefix 10.0.0.0/24 \

    --subnet-name sub-vms \

    --subnet-prefix 10.0.0.0/27

CopyExplain

Now, we need to create a **Network Security Group** to assign to the network interface of our VM once it has been launched.

We need this as we will assign a public IP address to the VM, and we don’t want to expose our three management ports directly to the internet; instead, we want to lock them down to just us:

$ az network nsg create \

    --resource-group rg-ansible-windows-server-uks \

    --name nsg-ansible-windows-server-uks

CopyExplain

We now have an empty Network Security Group created. Let’s add some rules, starting with the rule that opens port 80 to everyone to allow HTTP traffic:

$ az network nsg rule create \

   --resource-group rg-ansible-windows-server-uks \

   --nsg-name nsg-ansible-windows-server-uks \

   --name allowHTTP \

   --protocol tcp \

   --priority 100 \

   --destination-port-range 80 \

   --access allow

CopyExplain

Next, we have the rule that opens port 3389, which **Remote Desktop** uses to allow you to create a session to the host; we only want this open to us, so the command here would be as follows:

$ az network nsg rule create \

    --resource-group rg-ansible-windows-server-uks \

    --nsg-name nsg-ansible-windows-server-uks \

    --name allowRDP \

    --protocol tcp \

    --priority 1000 \

    --destination-port-range 3389 \

    --source-address-prefix $MYIP/32 \

    --access allow

CopyExplain

Note that we are passing in the $MYIP variable we registered when launching the resources. This will pass your IP address, and as you can see, we are then appending /32 to the end; this is the **Classless Inter-Domain Routing** (**CIDR**) notation for a single IP address.

Now that we have the rule for Remote Desktop in place, which is how we, as end users, will connect to the VM, we need to open the **Windows Remote Management** (**WinRM**) port, which is how Ansible will be connecting to the machine:

$ az network nsg rule create \

    --resource-group rg-ansible-windows-server-uks \

    --nsg-name nsg-ansible-windows-server-uks \

    --name allowWinRM \

    --protocol tcp \

    --priority 1050 \

    --destination-port-range 5985-5986 \

    --source-address-prefix $MYIP/32 \

    --access allow

CopyExplain

The next of the commands we need to run is the one that launches the VM itself and configures it to use the core networking components we have just launched:

$ az vm create \

     --resource-group rg-ansible-windows-server-uks \

     --name vm-ansible-windows-server-uks \

     --computer-name ansibleWindows \

     --image Win2022Datacenter \

     --admin-username azureuser \

     --admin-password $VMPASSWORD \

     --vnet-name vnet-ansible-windows-server-uks \

     --subnet sub-vms \

     --nsg nsg-ansible-windows-server-uks \

     --public-ip-sku Standard \

     --public-ip-address-allocation static

CopyExplain

As you can see, we are instructing the Azure CLI to launch a VM that uses the **Win2022Datacenter** VM image as its base; the VM is being deployed into the rg-ansible-windows-server-uks resource group and using all of the network resources we launched using the previous commands.

You might be thinking, great, let’s get back to looking at Ansible. However, there is one more command we need to run before we can connect to the VM using Ansible – and the reason is that while we have a Windows 2022 server up and running, the WinRM protocol is not enabled by default.

The command to enable this functionality is as follows:

$ az vm extension set \

    --resource-group rg-ansible-windows-server-uks \

    --vm-name vm-ansible-windows-server-uks \

    --name CustomScriptExtension \

    --publisher Microsoft.Compute \

    --version 1.10 \

    --settings "{'fileUrls': ['https://raw.githubusercontent.com/PacktPublishing/Learn-Ansible-Second-Edition/main/Scripts/ConfigureRemotingForAnsible.ps1'],'commandToExecute': 'powershell -ExecutionPolicy Unrestricted -File ConfigureRemotingForAnsible.ps1'}"

CopyExplain

This enables a VM Extension on the Azure VM we have just deployed. There are many different types of Virtual Machine Extensions; the one we are using is **Custom Script Extension**. This extension downloads a script from a URL passed to it and then executes a command; in our case, we are downloading the script configuring WinRM from the GitHub repository accompanying this title.

You can see the script that will be downloaded at the following URL: https://raw.githubusercontent.com/PacktPublishing/Learn-Ansible-Second-Edition/main/Scripts/ConfigureRemotingForAnsible.ps1

The command that runs once the script is downloaded is as follows:

$ powershell -ExecutionPolicy Unrestricted -File ConfigureRemotingForAnsible.ps1

CopyExplain

The Virtual Machine Extension executes the preceding command, so we do not have to run it directly.

The Resource Visualizer in the Azure portal for the resource group should show you something that looks like the following overview:

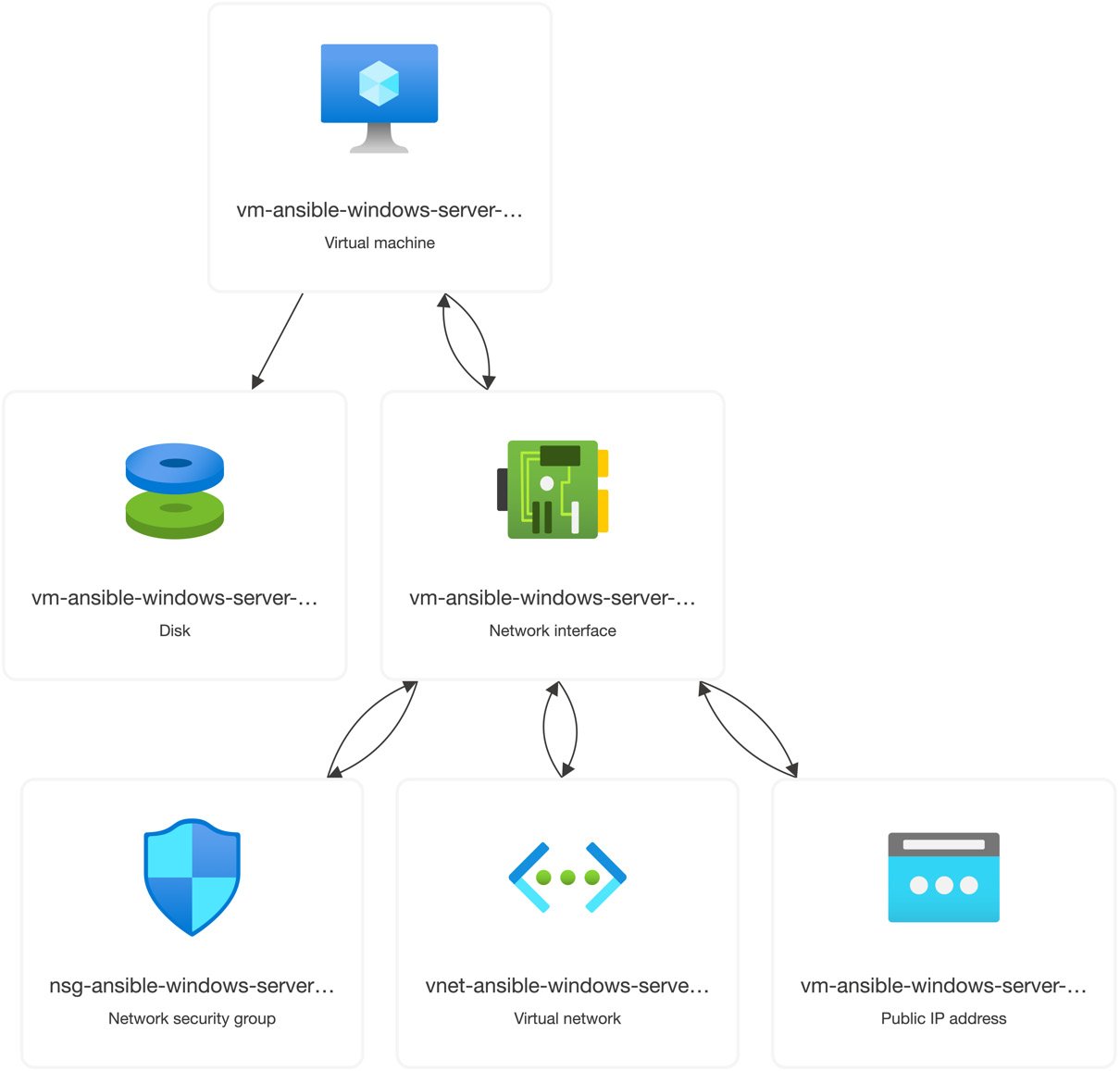


Figure 7.1 – Reviewing our resources in the Azure Resource Visualizer

Once completed, our Windows Server VM is ready to have our Ansible run against it.

Ansible preparation

As mentioned in the previous section, Ansible will use WinRM to interact with our Windows host.

Information

WinRM provides access to a **Simple Object Access Protocol** (**SOAP**)-like protocol called **WS-Management**. Unlike **Secure Shell** (**SSH**), which provides the user with an interactive shell to manage the host, WinRM accepts executed scripts, and the results are passed back to you.

Ansible requires us to install a few Python modules to enable it to use the protocol; these modules need to be installed as they are not typically installed alongside Ansible.

To install the module, if you are running on Ubuntu, run the following command:

$ sudo -H pip install pywinrm

CopyExplain

On macOS, run the following:

$ pip install pywinrm

CopyExplain

Once installed, we need to update our environment file to instruct Ansible to use the WinRM protocol rather than SSH.

Our updated hosts file looks like the following file, which is a copy of the hosts.example file from the Chapter07 folder in the accompanying repository. If you are following along with the exercise, you will need to update yours to update the IP address to match that of your Azure Virtual Machine once it has been launched:

WindowsServer ansible\_host=123.123.123.123.nip.io

[ansible\_hosts]

WindowsServer

[ansible\_hosts:vars]

ansible\_connection=winrm

ansible\_user="azureuser"

ansible\_password="{{ lookup('ansible.builtin.file', 'VMPASSWORD') }}"

ansible\_winrm\_server\_cert\_validation=ignore

CopyExplain

The start of the file mirrors what we have been used to so far in that it contains a name for a host and the resolvable hostname of the VM, again using the **Nip.io** service (https://nip.io/).

Next, we take the named host and put it in the ansible\_hosts group before defining a bunch of settings for the group.

The first of these settings instructs Ansible to use winrm by setting it as the value for the ansible\_connection key.

Next, we set the ansible\_user key; the value is azureuser, which we defined when we launched the Azure Virtual Machine; and also the ansible\_password key.

If you recall, at the start of the chapter, we ran the following command:

$ echo $VMPASSWORD > VMPASSWORD

CopyExplain

This took the random password we generated, that is, $VMPASSWORD, and placed it inside a file named VMPASSWORD; this means that when we define the ansible\_password key, we can use a lookup value, using {{ lookup('ansible.builtin.file', 'VMPASSWORD') }}, to read the contents of the VMPASSWORD file and use that rather than us having to hardcode the password into our environment file.

Finally, we tell Ansible to ignore any certificate errors by setting the ansible\_winrm\_server\_cert\_validation key to false; we need to do this because WinRM has been configured to use a self-signed certificate, which will cause a certificate error as our local machine does not know to trust the certificate.

Now that we have Windows up and running and Ansible configured, we can start interacting with it.

The ping module

Not all Ansible modules work with Windows hosts, and ansible.builtin.ping is one of them.

If you were to run the following command:

$ ansible WindowsServer -i hosts -m ansible.builtin.ping

CopyExplain

You would then get quite a verbose error with the following warning:

[WARNING]: No python interpreters found for host WindowsServer (tried ['python3.11', 'python3.10',

'python3.9', 'python3.8', 'python3.7', 'python3.6', 'python3.5', '/usr/bin/python3',

'/usr/libexec/platform-python', 'python2.7', '/usr/bin/python', 'python'])

CopyExplain

Luckily, there is a module provided for Windows called ansible.windows.win\_ping, so let’s update our command to run that instead:

$ ansible WindowsServer -i hosts -m ansible.windows.win\_ping

CopyExplain

This returns the result you would expect to receive if you sent a ping:

WindowsServer | SUCCESS => {

    "changed": false,

    "ping": "pong"

}

CopyExplain

The next module we will look at doesn’t require any changes from how we ran it against a Linux host.

The setup module

As before, we need to run the module and target our host, so the command is as follows:

$ ansible WindowsServer -i hosts -m ansible.builtin.setup

CopyExplain

This will return information on the host as it did when executing the same module against our Linux host, a snippet of which can be seen in the following screenshot:

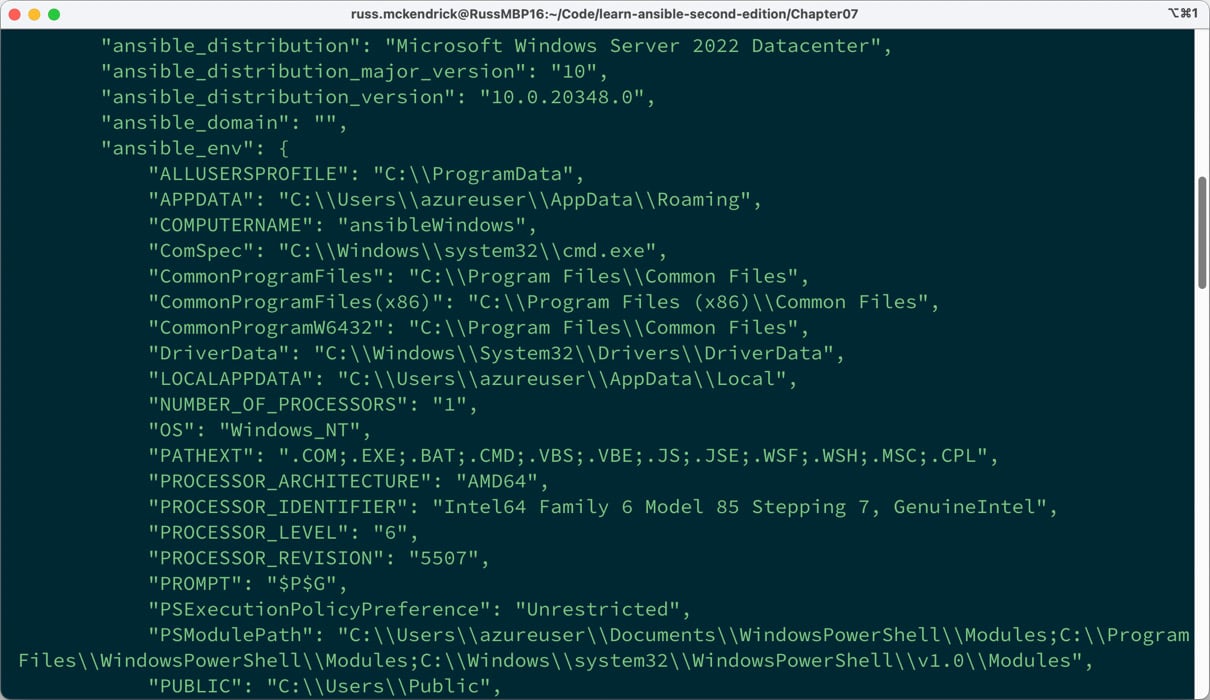


Figure 7.2 – Some of the output from the setup module

This is one of the only modules that will work on Windows and Linux hosts.

Now that we have confirmed that our host is accessible, let’s look at the changes we need to make to the playbooks.

The Windows Playbook roles

The entire playbook can be found in the Chapter 07 folder in the repository that accompanies the title, so I will not cover how to create roles in this chapter as we have covered it at length in the previous chapters.

Enabling Windows features

Two roles cover how to enable features; the first role, called iis, enables the **Internet Information Services** (**IIS**) on our Windows Server.

Information

IIS is the default web server that ships with Windows Server, and it supports the following protocols: HTTP, HTTPS, and HTTP/2, as well as FTP, FTPS, SMTP, and NNTP. It was first released in 1995 as part of Windows NT.

There are some default variables in roles/iis/defaults/main.yml; these define where things need to be copied to on the server and also include the contents on an HTML file we will copy to the host:

document\_root: 'C:\inetpub\wwwroot\'

html\_file: "ansible.html"

html\_heading: "Success !!!"

html\_body: |

  This HTML page has been deployed using Ansible to a <b>{{ ansible\_distribution }}</b> host.<br><br>

  The weboot is <b>{{ document\_root }}</b> this file is called <b>{{ html\_file }}</b>.<br>

CopyExplain

There are then two tasks in roles/iis/tasks/main.yml. The first task is *where the magic happens*:

- name: "Enable IIS"

  ansible.windows.win\_feature:

    name:

      - "Web-Server"

      - "Web-Common-Http"

    state: "present"

CopyExplain

I say *where the magic happens* because I don’t get to touch Windows hosts very often as a Linux system administrator by trade.

Still, as you can see from the preceding task, Ansible is giving us a Linux-like experience, meaning that I don’t have to roll up my sleeves and get under the hood of Windows too much.

The task uses the ansible.windows.win\_feature module to enable the Web-Server and Web-Common-Http features; as we are sticking with the default out-of-the-box settings, there isn’t any more configuration we need to do other than to copy an HTML file across to the document root:

- name: "Create an html file from a template"

  ansible.windows.win\_template:

    src: "index.html.j2"

    dest: "{{ document\_root }}{{ html\_file }}"

CopyExplain

As you can see, we are using a Jinja2 template file, an abridged version of which looks like the following:

<!--{{ ansible\_managed }}-->

<title>{{ html\_heading }}</title>

<article>

    <h1>{{ html\_heading }}</h1>

    <div>

        <p>{{ html\_body }}</p>

    </div>

</article>

CopyExplain

But rather than ansible.builtin.template, we are using ansible.windows.win\_template, which is the Windows module version, as I am sure you will have already guessed.

Suppose we were to use the ansible.builtin.template version; we would get the same error as when we ran the ansible.builtin.ping module, and complaints about Python not being installed.

The next role expands on the iis file and enables .Net; the role is called dotnet.

Again, there are some default variables in roles/dotnet/defaults/main.yml:

aspx\_document\_root: 'C:\inetpub\wwwroot\ansible\'

aspx\_file: "default.aspx"

aspx\_heading: "Success !!!"

aspx\_body: |

  This HTML page has been deployed using Ansible to a <b>{{ ansible\_distribution }}</b> host.<br><br>

  The weboot is <b>{{ aspx\_document\_root }}</b> this file is called <b>{{ aspx\_file }}</b>.<br><br>

  The output below is from ASP.NET<br><br>

  Hello from <%= Environment.MachineName %> at <%= DateTime.UtcNow %><br><br>

CopyExplain

As you can see, this time, the body contains some inline code.

However, you may have yet to spot a subtle difference in how we define the paths in the variables. For both tasks for our Windows workload, the path variables have been defined as follows:

document\_root: 'C:\inetpub\wwwroot\'

aspx\_document\_root: 'C:\inetpub\wwwroot\ansible\'

CopyExplain

But, if we look at how we defined the path in [*Chapter 5*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/5), *Deploying WordPress*, there is quite a crucial difference:

wordpress\_system:

    home: "/var/www/wordpress"

CopyExplain

The difference is not that we have used wordpress\_system.home as the variable; it is more subtle than that.

If you have noticed that the Windows workload paths are using single quotes and the Linux one is using double quotes, give yourself a pat on the back.

In Ansible, single quotes (') that enclose strings are treated as literals, ensuring special characters aren’t interpreted or expanded, making them ideal for Windows paths.

Double quotes (") allow for string interpolation, meaning embedded Jinja2 template expressions or special characters will be expanded. They also support escape sequences, such as \n for new lines, because many escape sequences such as \, which is in our path, could cause problems.

If we needed to use double quotes because we needed to pass in something that needed to be expanded, then you could have a double slash (\\) like this:

document\_root: "C:\\inetpub\\wwwroot\\"

aspx\_document\_root: "C:\\inetpub\\wwwroot\\ansible\\"

CopyExplain

However, it can confuse reading the paths, so I used single quotes in our examples – back to the role now.

The first of four tasks in roles/dotnet/tasks/main.yml enables .Net:

- name: "Enable .NET"

  ansible.windows.win\_feature:

    name:

      - "Net-Framework-Features"

      - "Web-Asp-Net45"

      - "Web-Net-Ext45"

    state: "present"

  notify: "Restart IIS"

CopyExplain

We are also triggering a restart of IIS via a handler if any changes are detected; this uses ansible.windows.win\_service:

- name: "Restart IIS"

  ansible.windows.win\_service:

    name: "w3svc"

    state: "restarted"

CopyExplain

The next task creates a folder if one doesn’t exist:

- name: "Create the folder for our asp.net app"

  ansible.windows.win\_file:

    path: "{{ aspx\_document\_root }}"

    state: "directory"

CopyExplain

Again, a Windows version of an existing module we have used is called, this time ansible.windows.win\_file. Next, we copy the file to the folder we just created:

- name: "Create an aspx file from a template"

  ansible.windows.win\_template:

    src: "default.aspx.j2"

    dest: "{{ aspx\_document\_root }}{{ aspx\_file }}"

CopyExplain

The final task in the role configures IIS to consider we are now running an application:

- name: "Ensure the default web application exists"

  community.windows.win\_iis\_webapplication:

    name: "Default"

    state: "present"

    physical\_path: "{{ aspx\_document\_root }}"

    application\_pool: "DefaultAppPool"

    site: "Default Web Site"

CopyExplain

There are a few more roles to cover before we run the playbook; let’s look at the next one.

Creating a user

This role creates a user for us to connect to our instance with. The defaults that can be found in roles/user/defaults/main.yml are as follows:

ansible:

  username: "ansible"

  password: "{{ lookup('password', 'group\_vars/generated\_password chars=ascii\_letters,digits length=30') }}"

  groups:

    - "Users"

    - "Administrators"

CopyExplain

As you can see, here, we are defining a user called ansible that has a 30-character random password, which Ansible will create using a lookup plugin if one doesn’t exist. The ansible user will be a member of the Users and Administrators groups.

There is a single task in roles/user/tasks/main.yml using the ansible.windows.win\_user module, which looks like the following:

- name: "Ensure that the ansible created users are present"

  ansible.windows.win\_user:

    name: "{{ ansible.username }}"

    fullname: "{{ ansible.username | capitalize }}"

    password: "{{ ansible.password }}"

    state: "present"

    groups: "{{ ansible.groups }}"

CopyExplain

Like all Windows modules, the syntax is similar to the Linux equivalent, so you should know what each key means. As you can see from the previous task, we are using a Jinja2 transformation to capitalize the first letter of the ansible.username variable.

Installing applications using Chocolatey

The next role, called choco, uses **Chocolatey** to install a few bits of software on the machine.

Information

Chocolatey is Windows’ answer to macOS’s Homebrew – a package manager streamlining software installations. Like we used Homebrew earlier, Chocolatey wraps typical Windows installations into neat PowerShell commands, making it a perfect match for orchestration tools such as Ansible.

In roles/choco/defaults/main.yml, we have a single variable that contains a list of the packages we want to install:

apps:

  - "notepadplusplus.install"

  - "putty.install"

  - "googlechrome"

CopyExplain

As you may have already guessed, this is the task that installs the applications:

- name: "Install software using chocolatey"

  chocolatey.chocolatey.win\_chocolatey:

    name: "{{ apps }}"

    state: "present"

CopyExplain

Again, the module takes a similar input to the previous package manager modules, ansible.builtin.apt and ansible.builtin.dnf, that we used. This means that the logic Ansible uses across the modules that do similar tasks is consistent across multiple operating systems and not just different Linux distributions.

Information role

The final role is called info; its only purpose is to output once the playbook has finished running. The role has a single task defined in roles/info/tasks/main.yml:

- name: "Print out information on the host"

  ansible.builtin.debug:

    msg: "You can connect to '{{ ansible\_host }}' using the username of '{{ ansible.username }}' with a password of '{{ ansible.password }}'."

CopyExplain

As you can see, this will provide us with the hostname to create a Remote Desktop session, along with confirming the username and password we should use.

That concludes our look at the roles that will be called when we run the playbook, which we are now ready to do.

Running the Playbook

The site.yml is missing some of the settings at the top because we are targeting a Windows host:

---

- name: "Install IIS, .NET, create user, install chocolatey and display info"

  hosts: "ansible\_hosts"

  gather\_facts: true

  vars\_files:

    - "group\_vars/common.yml"

  roles:

    - "iis"

    - "dotnet"

    - "user"

    - "choco"

    - "info"

CopyExplain

As you can see, there is no need for the become or become\_method keys to be set, as we do not need to change users once connected to the host.

Outside of that, the rest of the file is as expected, as is the way we run the playbook:

$ ansible-playbook -i hosts site.yml

CopyExplain

It will take a little while to run as a lot is going on in the background, as you will see from the output when the playbook runs for the first time:

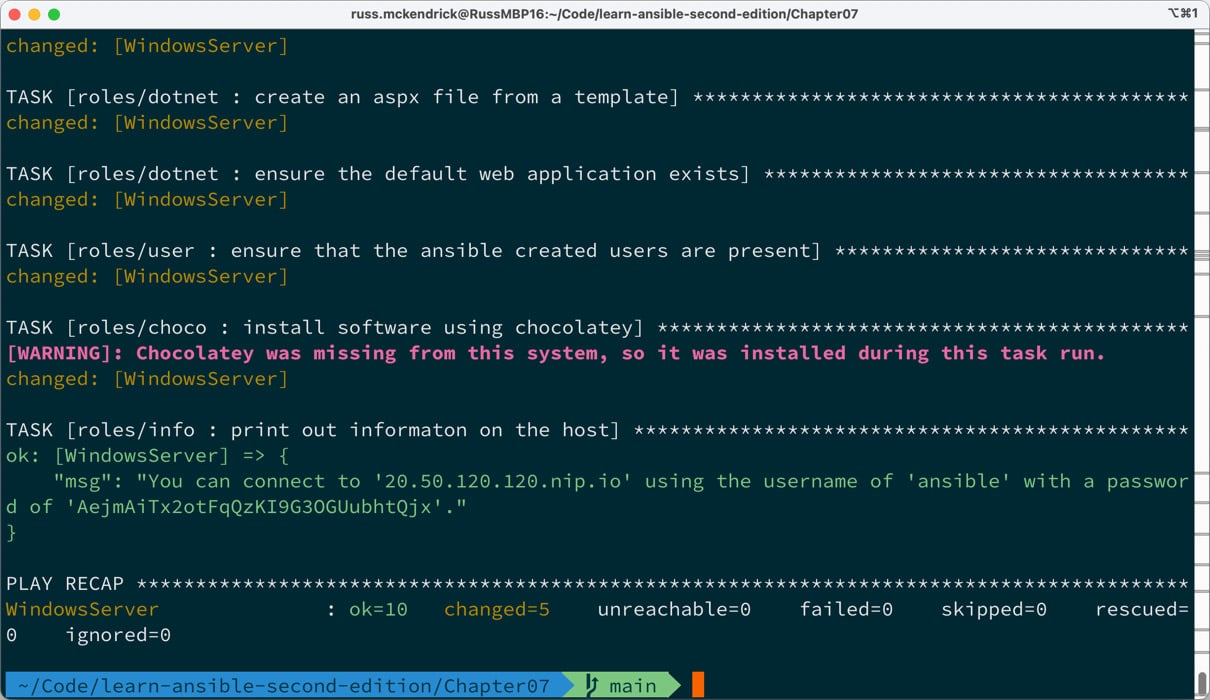


Figure 7.3 – Reviewing the playbook output

As you can see from the preceding output, the host I was given was 20.50.120.120.nip.io (this host has long since been terminated, but if you are following along, you can replace the preceding host with your own).

To view the static HTML and .Net pages we uploaded, you can visit http://20.50.120.120.nip.io/ansible.html or http://20.50.120.120.nip.io/ansible/default.aspx, making sure to update the host to reflect your own.

You can also open a remote desktop session to the host using the credentials given in the output; the following screenshot shows a session using the user we created and opening the side using **Google Chrome** with notes in **Notepad++**, both of which are applications we installed with the Playbook:

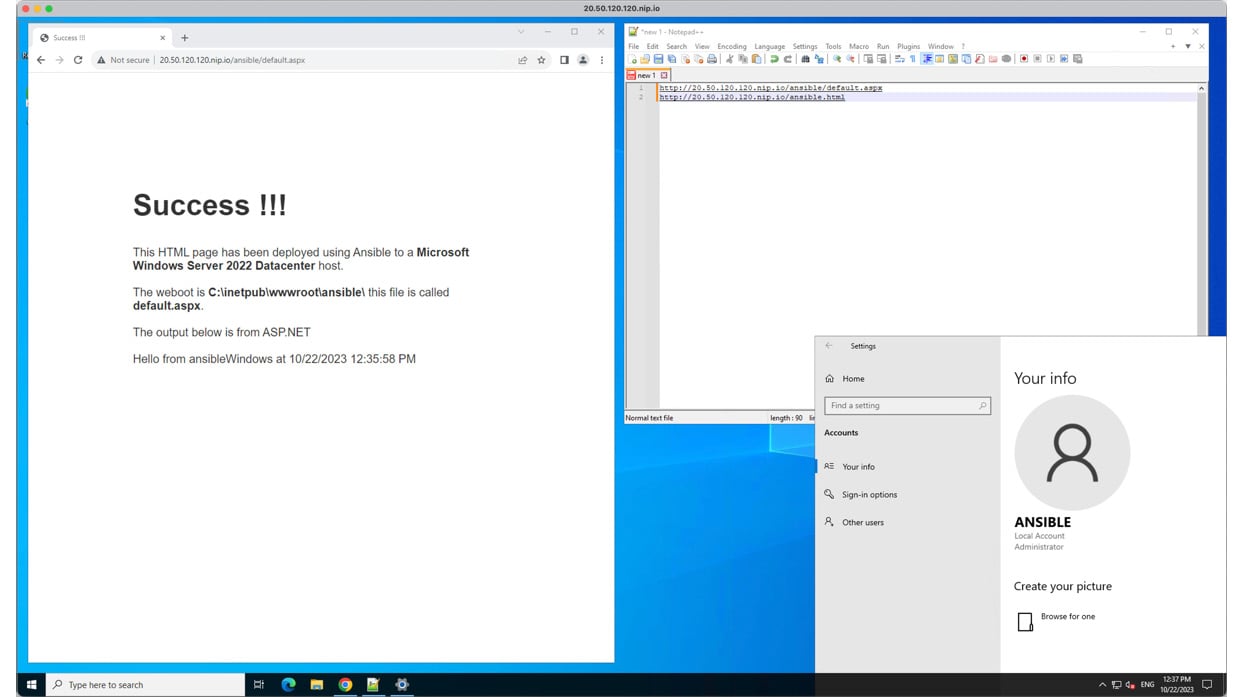


Figure 7.4 – A remote desktop session

Once you have finished with the host, you can run the following Azure CLI command to terminate all the resources we created:

$ az group delete \

    --name rg-ansible-windows-server-uks

CopyExplain

Double-check that everything has been removed as expected to ensure you do not get any unexpected bills.

Summary

As mentioned at the start of the chapter, using what we would consider a traditional Linux tool such as Ansible on Windows always feels a little strange. However, I am sure you will agree that the experience is as Linux-like as possible.

When I first started experimenting with the Windows modules, I was surprised that I managed to launch a Windows Server in Azure and deploy a simple web application without having to remote desktop into the target instance.

With each new release, Ansible gets more and more support for Windows-based hosts, making it easy to manage mixed workloads from your playbooks.

In the next chapter, we will examine the networking modules available in Ansible.