Moving to the Cloud

This chapter will move from using our local virtual machine to using Ansible to launch instances with a public cloud provider.

For this chapter, we will be using Microsoft Azure, and we are targeting this provider as it allows us to launch virtual machines and interact with them without having too much configuration overhead.

We will then look at adapting our WordPress playbook to interact with the newly launched Microsoft Azure instance.

In this chapter, we will cover the following topics:

* An introduction to Microsoft Azure
* Launching instances in Microsoft Azure
* Bootstrapping WordPress

Technical requirements

In this chapter, we will launch instances in a public cloud, so if you are following along, you will need a Microsoft Azure account. As with other chapters, complete versions of the playbooks can be found in the repository in the chapters folder at https://github.com/PacktPublishing/Learn-Ansible-Second-Edition/tree/main/Chapter09.

An introduction to Microsoft Azure

In 2008, Microsoft took its first significant step into cloud computing by introducing Windows Azure, a cloud-based data center service. This launch marked a pivotal moment in what many people saw as a traditional software company’s history, signaling a strategic shift toward cloud computing.

Developed under the internal project *Project Red Dog*, Windows Azure represented Microsoft’s answer to the growing demand for scalable, accessible, and flexible computing resources.

Windows Azure was initially rolled out with five core components, each designed to offer distinct capabilities within the cloud computing spectrum:

* **Microsoft SQL Data Services**: This component offered a cloud version of Microsoft’s SQL database, simplifying the complexities associated with hosting and managing databases in a cloud environment.
* **Microsoft .NET Services**: As a **platform as a service** (**PaaS**) offering, it enabled developers to deploy their .NET-based applications within a Microsoft-managed runtime, streamlining the development process.
* **Microsoft SharePoint** and **Microsoft Dynamics**: These **software as a service** (**SaaS**) offerings provided cloud-based versions of the company’s renowned intranet and **customer relationship management** (**CRM**) products, enhancing collaboration and customer engagement.
* **Windows Azure (IaaS)**: An **infrastructure-as-a-service** (**IaaS**) solution, this allowed users to create and control virtual machines, storage, and networking services, addressing diverse compute workloads.

The preceding four definitions are from an older book I wrote, *Infrastructure as Code for Beginners*.

Central to Windows Azure’s architecture was the Red Dog operating system, a specially modified version of Windows NT. This system was engineered to include a cloud layer, ensuring the smooth delivery of data center services.

By 2014, reflecting its expanded range of services and a growing emphasis on Linux-based workloads, Microsoft rebranded the service as Microsoft Azure. This change underscored the platform’s evolution beyond Windows-centric solutions.

Fast forward to 2020, and it was evident that Microsoft Azure had embraced a more inclusive approach, with over half of its virtual machine cores and a significant number of Azure Marketplace images being Linux-based.

This shift demonstrated Microsoft’s broader adoption of Linux and open-source technologies, which remain integral to their current cloud service offerings at the time of writing.

Launching instances in Microsoft Azure

If you followed along in [*Chapter 7*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/7), *Ansible Windows Modules*, you will have already launched a virtual machine in Microsoft Azure using the Azure CLI.

Reminder

For instructions 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/. Remember, if you are following along on a Windows host, then make sure to install the Azure CLI within your Windows Subsystem for Linux installation alongside where you installed Ansible.

When talking through launching the Windows virtual machine, we did the following:

* We created a resource group to collect all the resources for our virtual machine workload.
* We then created a virtual network and subnet to attach to the machine’s network interface.
* We then created a network security group to secure our virtual machine.
* Once we had the basics, we launched a Windows virtual machine, attaching a public IP address directly to the network interface.
* Finally, we deployed a virtual machine extension that executed the PowerShell script on our Windows host to enable the WinRM protocol, allowing us to connect to and interact with the host using Ansible.

This chapter will repeat, tweak, and add to these steps using Ansible and the Azure collection of modules.

Preparing Ansible for Microsoft Azure

Before we dive into the Ansible role, which will launch our resources, we need to do a little preparation; first, let’s ensure that the Azure collection is installed by running the following:

$ ansible-galaxy collection install azure.azcollection

CopyExplain

Next, we must install the Python modules that allow the Azure collection to interact with the Azure APIs. To do this, we need to run the following command:

$ pip3 install -r ~/.ansible/collections/ansible\_collections/azure/azcollection/requirements-azure.txt

CopyExplain

With the necessary supporting Python modules installed, the next step is to ensure that you have signed into your Microsoft Azure account using the Azure CLI. To do this, run the following command and follow the onscreen prompts:

$ az login

CopyExplain

If you have access to more than one Azure subscription using your account, you should ensure that the subscription you intend to launch your resources in is selected.

To do this, you can list all the subscriptions and, if needed, switch to the right subscription by running the following commands:

$ az account list --output table

$ az account set --subscription <subscription\_id>

CopyExplain

Ensure you replace <subscription\_id> with the correct subscription ID from the az account list command.

Note

Using the az account set command only applies to your current session; if you close your terminal window and reopen a new session, you must ensure you have changed subscriptions again.

Reviewing the variables

There are several variables across the roles we will use to deploy our Azure resources and configure WordPress. The first ones we will look at can be found in the group\_vars/common.yml file.

To start with, we have some **feature flags**, the first of which, debug\_output, outputs the contents of the variables that are registered during the playbook run; setting this to true is helpful to pull back information on the Azure resources once they have launched during the development of the role.

The second feature flag is generate\_key; if this is set to true, then a private and public key pair will be created by Ansible if one does not exist at ~/.ssh/id\_rsa.

The playbook will use the key in this location when launching a virtual machine, so one must exist, as without it, Ansible cannot connect to the newly launched virtual machine.

These two variables look like the following:

debug\_output: false

genterate\_key: false

CopyExplain

Next, in group\_vars/common.yml, we define some information about our app workload; this contains a mixture of details about the application and some of the Azure details like the Azure region the workload will be launched in (location and location\_short) as well as the name that our WordPress site will be accessible on ( public\_dns\_name):

app:

  name: "learnansible-wordpress"

  shortname: "ansiblewp"

  location: "westeurope"

  location\_short: "euw"

  env: "prod"

  public\_dns\_name: "learnansible"

CopyExplain

The final set of variables, which are defined in the group\_vars/common.yml file, are for the tags that will be applied to each Azure resource that Ansible will launch:

common\_tags:

  "project": "{{ app.name }}"

  "environment": "{{ app.env }}"

  "deployed\_by": "ansible"

CopyExplain

The next set of variables we will be using can be found in roles/azure/defaults/main.yml, which are used to deploy our resources.

The first block of variables defines a quick dictionary of Azure service names for use when it comes to naming our resources:

dict:

  ansible\_warning: "Resource managed by Ansible"

  load\_balancer: "lb"

  network\_interface: "nic"

  nsg: "nsg"

  private\_endpoint: "pe"

  public\_ip: "pip"

  resource\_group: "rg"

  subnet: "snet"

  virtual\_machine: "vm"

  virtualnetwork: "vnet"

CopyExplain

Next, we define the resource names – as per [*Chapter 7*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/7), *Ansible Windows Modules*, I am naming the resources as close to the cloud adoption framework recommendations as possible:

load\_balancer\_name: "{{ dict.load\_balancer }}-{{ app.name }}-{{app.env}}-{{ app.location\_short }}"

load\_balancer\_public\_ip\_name: "{{ dict.public\_ip }}-{{ load\_balancer\_name }}"

nsg\_name: "{{ dict.nsg }}-{{ app.name }}-{{app.env}}-{{ app.location\_short }}"

resource\_group\_name: "{{ dict.resource\_group }}-{{ app.name }}-{{app.env}}-{{ app.location\_short }}"

virtual\_network\_name: "{{ dict.virtualnetwork }}-{{ app.name }}-{{app.env}}-{{ app.location\_short }}"

vm\_name: "{{ dict.virtual\_machine }}-admin-{{ app.name }}-{{app.env}}-{{ app.location\_short }}"

vnet\_name: "{{ dict.virtualnetwork }}-{{ app.name }}-{{app.env}}-{{ app.location\_short }}"

CopyExplain

Now that all the naming is out of the way, we can start defining the networking variables:

vnet\_config:

  cidr\_block: "10.0.0.0/24"

  subnets:

    - {

        name: "{{ dict.subnet }}-vms-{{ app.name }}-{{app.env}}-{{ app.location\_short }}",

        subnet: «10.0.0.0/27»,

        private: true,

        service\_endpoints: «Microsoft.Storage»,

      }

CopyExplain

Next, in networking, we have two lists of IPs – one is for fixed IPs, and the other is the IP address discovered when the playbook runs:

trusted\_ips:

  - ""

dynamic\_ips:

  - "{{ your\_public\_ip }}"

CopyExplain

The next block of variables takes the preceding lists of IP addresses and uses them when creating the two network security group rules:

nsg\_rules:

  - name: "allowHTTP"

    description: "{{ dict.ansible\_warning }}"

    protocol: «Tcp»

    destination\_port\_range: «80»

    source\_address\_prefix: "\*"

    access: "Allow"

    priority: "100"

    direction: "Inbound"

  - name: "allowSSH"

    description: "{{ dict.ansible\_warning }}"

    protocol: "Tcp"

    destination\_port\_range: «{{ load\_balancer.ssh\_port }}»

    source\_address\_prefix: "{{ trusted\_ips|select() + dynamic\_ips | unique }}"

    access: "Allow"

    priority: "150"

    direction: "Inbound"

CopyExplain

As you can see, the first rule, allowHTTP, opens port 80 to the world; but allowSSH locks down the SSH port to the IP addresses in our two lists. To do this, we take the list of IP addresses in the trusted\_ips variable, append the content of dynamic\_ips, and then finally only display the unique items in the list so any duplicate entries are removed.

The final networking block defines the basics needed to launch Azure Load Balancer:

load\_balancer:

  ssh\_port: "22"

  ssh\_port\_backend: "22"

  http\_port: "80"

  http\_port\_backend: "80"

CopyExplain

Now we have the virtual machine configuration:

vm\_config:

  admin\_username: "adminuser"

  ssh\_password\_enabled: false

  vm\_size: "Standard\_B1ms"

  image:

    publisher: "Canonical"

    offer: "0001-com-ubuntu-server-jammy"

    sku: "22\_04-LTS"

    version: "latest"

  disk:

    managed\_disk\_type: "Premium\_LRS"

    caching: "ReadWrite"

  key:

    path: "/home/adminuser/.ssh/authorized\_keys"

    data: "{{ lookup('file', '~/.ssh/id\_rsa.pub') }}"

CopyExplain

Then, finally, just two variables that define the location and the host group of our newly launched virtual machine will be placed:

location: "{{ app.location }}"

hosts\_group: "vmgroup"

CopyExplain

Now that we have covered all of the variables needed to launch our Azure resources, we can work through the tasks that do the actual work, all of which can be found in roles/azure/tasks/main.yml.

The resource group task

The first task we are going to look at is creating the resource group where all of the other Azure resources are going to be placed:

- name: "Create the resource group"

  azure.azcollection.azure\_rm\_resourcegroup:

    name: "{{ resource\_group\_name }}"

    location: "{{ location }}"

    tags: "{{ common\_tags }}"

  register: "resource\_group\_output"

CopyExplain

As you can see, there is not much to it; it takes the name, location, and tags variables we defined and creates the resource group using the azure. collection.azure\_rm\_resourcegroup module. The task output is then registered as a variable, allowing us to reuse the output in later tasks.

The next task prints the contents of the resource\_group\_output register variable on the screen if debug\_output is set to true; if it is false, then the task is skipped:

- name: "Debug - Resource Group result"

  ansible.builtin.debug:

    var: "resource\_group\_output"

  when: debug\_output

CopyExplain

This is a common pattern throughout the Azure role, so we will not cover this task again. Assume that if the task registers its output, there is a supporting debug task immediately after. Now that we have our resource group, we can make a start on configuring the networking.

The networking tasks

The first task launches the virtual network, placing it in the resource group we just created:

- name: "Create the virtual network"

  azure.azcollection.azure\_rm\_virtualnetwork:

    resource\_group: "{{ resource\_group\_output.state.name }}"

    name: "{{ virtual\_network\_name }}"

    address\_prefixes: "{{ vnet\_config.cidr\_block }}"

    tags: "{{ common\_tags }}"

  register: "virtual\_network\_output"

CopyExplain

As you can see, when referencing the resource group name, we use the registered output from the previous task by using {{ resource\_group\_output.state.name }}. Again, this is going to be a common thread throughout the remaining tasks.

Note, we are not defining the subnet as part of creating the virtual network; this is possible as we are only adding a single subnet, but it is considered best practice to use the azure.collection.azure\_rm\_subnet module to add subnets as this approach means that you can loop through adding subnets with a with\_items statement:

- name: "Add the subnets to the virtual network"

  azure.azcollection.azure\_rm\_subnet:

    resource\_group: "{{ resource\_group\_output.state.name }}"

    name: "{{ item.name }}"

    address\_prefix: "{{ item.subnet }}"

    virtual\_network: "{{ virtual\_network\_output.state.name }}"

    service\_endpoints:

      - service: "{{ item.service\_endpoints }}"

  with\_items: "{{ vnet\_config.subnets }}"

  register: "subnet\_output"

CopyExplain

With the virtual network now populated with subnets, we can move on to creating the network security group.

As you may remember, when we looked at the variables, we used a variable called your\_public\_ip, so our next task is to discover the external IP address of the host running Ansible using the community.general.ipify\_facts module:

- name: "Find out your current public IP address using https://ipify.org/"

  community.general.ipify\_facts:

  register: public\_ip\_output

CopyExplain

As you can see, there is not much to this, but we are not registering a variable called your\_public\_ip; this is done as a separate task that uses the ansible.builtin.set\_fact module:

- name: "Register your public ip as a fact"

  ansible.builtin.set\_fact:

    your\_public\_ip: "{{ public\_ip\_output.ansible\_facts.ipify\_public\_ip }}"

CopyExplain

Now we know the IP address, we can create the network security group:

- name: "Create the network security group"

  azure.azcollection.azure\_rm\_securitygroup:

    resource\_group: "{{ resource\_group\_output.state.name }}"

    name: "{{ nsg\_name }}"

    rules: "{{ nsg\_rules }}"

    tags: "{{ common\_tags }}"

  register: "nsg\_output"

CopyExplain

So far, so good; the next piece of networking configuration we need to do is to launch Azure Load Balancer. This is the first deviation from the resources we launched in [*Chapter 7*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/7), *Ansible Windows Modules*, so why is that?

While Microsoft allows you to directly assign a public IP address to a virtual machine’s network interface in Azure, it is generally frowned upon and not considered best practice – having a networking resource such as Azure Load Balancer to route and distribute your traffic to one or more hosts is deemed to be more secure as you are putting a layer between the virtual machine and the public internet.

Also, having traffic pass through a load balancer, even when running a single virtual machine like ours, allows you to perform a basic health check to see whether the port to which the load balancer sends traffic is healthy.

The first task we need to run when launching Azure Load Balancer is to create a public IP address resource, which will be attached to the load balancer when we launch it:

- name: "Create the public IP address needed for the load balancer"

  azure.azcollection.azure\_rm\_publicipaddress:

    resource\_group: "{{ resource\_group\_output.state.name }}"

    allocation\_method: "Static"

    name: "{{ load\_balancer\_public\_ip\_name }}"

    sku: "standard"

    domain\_name: "{{ app.public\_dns\_name }}"

    tags: "{{ common\_tags }}"

  register: "public\_ip\_output"

CopyExplain

Now that the public IP address is defined, we can move on to Azure Load Balancer itself.

As there is rather a lot to the task, I will break it up a little as we go along:

- name: "Create load balancer using the public IP we created"

  azure.azcollection.azure\_rm\_loadbalancer:

    resource\_group: "{{ resource\_group\_output.state.name }}"

    name: "{{ load\_balancer\_name }}"

    sku: "Standard"

CopyExplain

The next block in the task defines the front of the load balancer. This is where we attach the public IP address we just created:

    frontend\_ip\_configurations:

      - name: "{{ load\_balancer\_name }}-frontend-ip-config"

        public\_ip\_address: "{{ public\_ip\_output.state.name }}"

CopyExplain

Next up, we define the backend pool. This is the pool where our virtual machine will be placed to have traffic sent to it. If we had more than one virtual machine, all of them would be addressed to the pool:

    backend\_address\_pools:

      - name: "{{ load\_balancer\_name }}-backend-address-pool"

CopyExplain

Now we have the health probe, which probes the HTTP port on the backend pool to make sure that the virtual machines are ready to access traffic on port 80 by seeing if the port is open:

    probes:

      - name: "{{ load\_balancer\_name }}-http-probe"

        port: «{{ load\_balancer.http\_port\_backend }}»

        fail\_count: "3"

        protocol: "Tcp"

CopyExplain

For our WordPress workload, we want our HTTP port to be exposed. To do this, we will create a load-balancing rule that allows you to create a one-to-many relationship with one or more virtual machines in the backend pool. This rule exposes the HTTP port on the load balancer and sends the traffic to the HTTP port on the backend virtual machines. If we had more than one virtual machine, the traffic would be evenly distributed across all hosts in the backend on the HTTP port:

    load\_balancing\_rules:

      - name: "{{ load\_balancer\_name }}-rule-http"

        frontend\_ip\_configuration: "{{ load\_balancer\_name }}-frontend-ip-config"

        backend\_address\_pool: "{{ load\_balancer\_name }}-backend-address-pool"

        frontend\_port: «{{ load\_balancer.http\_port }}»

        backend\_port: "{{ load\_balancer.http\_port\_backend }}"

        probe: "{{ load\_balancer\_name }}-http-probe"

CopyExplain

While a load balancing rule takes traffic from a single port on the frontend and distributes it across multiple virtual machines in the backend pool, an inbound **NAT** (which stands for **Network Address Translation**) rule distributes traffic on a one-to-one basis, which makes it perfect for services such as SSH that are not meant to be distributed across multiple hosts:

    inbound\_nat\_rules:

      - name: "{{ load\_balancer\_name }}-nat-ssh"

        frontend\_ip\_configuration: "{{ load\_balancer\_name }}-frontend-ip-config"

        backend\_port: "{{ load\_balancer.ssh\_port }}"

        frontend\_port: "{{ load\_balancer.ssh\_port }}"

        protocol: "Tcp"

CopyExplain

If we were to have more than one machine, we would add more rules that take different ports and map them to port 22 on the backend virtual machines. Typically, I would use high ports such as 2220 > 2229 so I don’t clash with over services – 2220 would send traffic to port 22 on the first machine and 2221 would do the same for the second machine, and so on.

However, in this example, we just have a single host, so I am mapping port 22 to port 22.

Lastly, we will tag the resource and register the output:

    tags: "{{ common\_tags }}"

  register: "load\_balancer\_output"

CopyExplain

Now we have the load balancer, we need to create a network interface, which will be placed in the backend pool and attached to our virtual machine.

For those of you who have already looked at the Ansible Azure collection, you may have noticed a module called azure.azcollection.azure\_rm\_networkinterface, which is used to manage network interfaces. Hence, you’d assume that the task we will be looking at uses that. Well, you would be wrong.

While the pre-written module has pretty good feature parity with the API endpoint it interacts with, it is missing one key piece of functionality we require for our deployment: the ability to assign the network interface to a NAT rule.

However, all is not lost, and there is a workaround.

There is an Azure module whose only purpose is to interact with the Azure Resource Manager API directly, called azure.collection.azure\_rm\_resource, and by using this module, we can make an API call directly to the Microsoft.Network/networkInterfaces endpoint from within Ansible.

Having the ability to do this for any of the Azure Resource Manager APIs is quite powerful as it opens new features as soon as Microsoft releases them, and it means you don’t have to wait for the Ansible Azure collection developers to write, test, and release the module.

It does come with one downside, though: using this method does add an additional layer of complexity to your playbook.

The following URL is the link to the REST API documentation, which covers the creation of a network interface: https://learn.microsoft.com/en-us/rest/api/virtualnetwork/network-interfaces/create-or-update?view=rest-virtualnetwork-2023-05-01&tabs=HTTP.

As we will see from working through the task, the general gist of what we are doing is constructing the URL of the API we would like to target and then constructing the request body detailed in the REST documentation.

To start with, let’s look at the part of the task that generates the URL:

- name: "Create the network interface for the wordpress vm"

  azure.azcollection.azure\_rm\_resource:

    api\_version: "2023-05-01"

    resource\_group: "{{ resource\_group\_output.state.name }}"

    provider: "network"

    resource\_type: "networkinterfaces"

    resource\_name: "{{ dict.network\_interface }}-{{ vm\_name }}"

    idempotency: true

CopyExplain

The preceding information constructs the URL given in the documentation, which is the following:

PUT https://management.azure.com/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}/providers/Microsoft.Network/networkInterfaces/{networkInterfaceName}?api-version=2023-05-01

CopyExplain

Let us look at how this is generated:

* {subscriptionId} is automatically generated by the module, and we do not need to provide this information.
* {resourceGroupName} is added by providing the resource\_group key, and as per the rest of the tasks, we are using the resource group name, which is the output of our registering the variable in the resource group task.
* The providers are provided by us by filling in the provider and resource\_type keys. Don’t worry – the URL is not case-sensitive, and the module adds the Microsoft. part for us.
* {networkInterfaceName} is the resource\_name key.
* Finally, the API version is provided by filling in the api\_version key.

The last part of the “header” does not form part of the URL, but instead, it instructs Ansible to perform a GET request and then compares the body of what will be posted to what is returned by the GET request, and if there are any problems, it will error before the body is posted.

Now that we have the URL of the Azure Resource Manager API endpoint to which we will send our request, we need to populate the body of the request.

For our case, this looks like the following code:

    body:

      location: "{{ location }}"

      properties:

        enableAcceleratedNetworking: false

        primary: true

        networksecuritygroup:

          id: "{{ nsg\_output.state.id }}"

        configurations:

          - name: "{{ vm\_name }}-ipcfg"

            properties:

              subnet:

                id: "{{ subnet\_output.results[0].state.id }}"

              loadBalancerBackendAddressPools:

                - id: "{{ load\_balancer\_output.state.backend\_address\_pools[0].id }}"

              loadBalancerInboundNatRules:

                - id: "{{ load\_balancer\_output.state.inbound\_nat\_rules[0].id }}"

      tags: "{{ common\_tags }}"

CopyExplain

When the module runs, properties will be rendered as JSON and posted alongside location and tags in the request’s body, leaving the final part of the task to register the output:

  register: "network\_interface\_output"

CopyExplain

We now have all the base Azure configuration and resources in place; we can launch our virtual machine. As we will be using SSH to connect to the virtual machine and bootstrap our WordPress installation, we need to ensure we have a valid SSH key generated.

As we will connect to a remote virtual machine, we want to ship a test key as we have been doing on our locally deployed hosts. So, if there is not a key at ~/.ssh/id\_rsa on your local machine, then set the genterate\_key variable in the group\_vars/common.yml file to true (it is false by default), then Ansible will generate the key for you.

Do not worry if a key already exists at that location; Ansible will only create a key if one does not exist:

- name: "Check user has a key, if not create one for {{ ansible\_user\_id }}"

  ansible.builtin.user:

    name: "{{ ansible\_user\_id }}"

    generate\_ssh\_key: true

    ssh\_key\_file: "~/.ssh/id\_rsa"

  when: genterate\_key

CopyExplain

Next, we have the task that launches the virtual machine itself. It uses all of the resources we have already deployed and configured so I will not go into too much detail:

- name: Create the admin virtual machine

  azure.azcollection.azure\_rm\_virtualmachine:

    resource\_group: "{{ resource\_group\_output.state.name }}"

    name: "{{ vm\_name }}"

    admin\_username: "{{ vm\_config.admin\_username }}"

    ssh\_public\_keys:

      - path: "{{ vm\_config.key.path }}"

        key\_data: "{{ vm\_config.key.data }}"

    ssh\_password\_enabled: "{{ vm\_config.ssh\_password\_enabled }}"

    vm\_size: "{{ vm\_config.vm\_size }}"

    managed\_disk\_type: "{{ vm\_config.disk.managed\_disk\_type }}"

    network\_interfaces: "{{ network\_interface\_output.response.name }}"

    image:

      offer: "{{ vm\_config.image.offer }}"

      publisher: "{{ vm\_config.image.publisher }}"

      sku: "{{ vm\_config.image.sku }}"

      version: «{{ vm\_config.image.version }}»

    tags: «{{ common\_tags }}»

  register: «vm\_output»

CopyExplain

As with most tasks we have run in this role, immediately after there is a debug task.

You may think, *“That’s the end of the role, right?”* but we have two tasks to cover.

The first of these final two tasks takes information about the hosts, such as the public IP address and SSH port, and then adds it to the host group defined as the hosts\_group variable.

This means that there is no hardcoding of IP addresses or connections in our host’s inventory file. The task to register the host looks like the following:

- name: Add the Virtual Machine to the host group

  ansible.builtin.add\_host:

    groups: "{{ hosts\_group }}"

    hostname: "{{ public\_ip\_output.state.ip\_address }}-{{ load\_balancer.ssh\_port }}"

    ansible\_host: "{{ public\_ip\_output.state.ip\_address }}"

    ansible\_port: «{{ load\_balancer.ssh\_port }}»

CopyExplain

So, what could this task be? We have the networking in place, our virtual machine has been launched, and we have registered our host, so we must be ready to start bootstrapping WordPress.

That’s the problem; we might be ready, but the host we just launched might not be as it can sometimes take a minute or two for the virtual machine to finish booting up. If we were to immediately try and SSH into the host before it has finished booting, then our playbook would error and halt running.

Luckily, an Ansible module was developed for use in this scenario, ansible.builtin.wait\_for:

- name: "Wait for the virtual machine to be ready"

  ansible.builtin.wait\_for:

    host: "{{ public\_ip\_output.state.ip\_address }}"

    port: «{{ load\_balancer.ssh\_port }}»

    delay: 10

    timeout: 300

CopyExplain

This will wait for 10 seconds and then attempt to SSH to the host for up to 5 minutes (300 seconds); when SSH is accessible, the Ansible playbook will then progress to the next set of roles, which, in our case, bootstrap WordPress.

Bootstrapping WordPress

It won’t be of any surprise to you that the bulk of the WordPress roles remain intact from our previous chapters so we will not cover those parts there and will instead review some of the small changes.

The site and host environment files

The site.yml is now split into two sections; the first runs locally and interacts with the Azure Resource Manager API to launch and configure the Azure resources:

- name: "Deploy and configure the Azure Environment"

  hosts: localhost

  connection: local

  gather\_facts: true

  vars\_files:

    - group\_vars/common.yml

  roles:

    - "azure"

CopyExplain

The second section targets the vmgroup host group and looks more like what we have been working with so far in the previous chapters:

- name: "Install and configure Wordpress"

  hosts: vmgroup

  gather\_facts: true

  become: true

  become\_method: "ansible.builtin.sudo"

  vars\_files:

    - group\_vars/common.yml

  roles:

    - "secrets"

    - "stack\_install"

    - "stack\_config"

    - "wordpress"

CopyExplain

The hosts file looks like the hosts files we have been using throughout the previous chapters; it is just missing the lines where we explicitly define the target hosts and instead is just made up of the host groups’ definitions.

You may have noticed that we are adding a new role, and the remaining ones are mostly the same; the role is called secrets, so let’s see what it does.

The secrets role

The sole purpose of this role is to generate secure passwords for WordPress and the database. Its tasks are delegated to the local machine as it creates a variables file at group\_vars/secrets.yml and loads them into the playbook run.

First, it checks if group\_vars/secrets.yml already exists and if it does, we don’t want to change the contents of the file:

- name: "Check if the file secrets.yml exists"

  ansible.builtin.stat:

    path: "group\_vars/secrets.yml"

  register: secrets\_file

  delegate\_to: "localhost"

  become: false

CopyExplain

If there is no file, then it and its contents are generated from a template file:

- name: "Generate the secrets.yml file using a template file if not exists"

  ansible.builtin.template:

    src: "secrets.yml.j2"

    dest: "group\_vars/secrets.yml"

  when: secrets\_file.stat.exists == false

  delegate\_to: "localhost"

  become: false

CopyExplain

The template file at roles/secrets/templates/secrets.yml.j2 looks like the following:

db\_password: "{{ lookup('community.general.random\_string', length=20, upper=true, special=false, numbers=true) }}" wp\_password: "{{ lookup('community.general.random\_string', length=20, upper=true, special=true, override\_special="@-&\*", min\_special=2, numbers=true) }}"

CopyExplain

As you can see, it uses the community.general.random\_string module to generate a random string with some sensible rules, which we will use as passwords.

Other changes

Most of the changes to the roles are to the variables; for example, in roles/wordpress/defaults/main.yml we have the following:

wordpress:

  domain: "http://{{ app.public\_dns\_name }}.{{ app.location }}.cloudapp.azure.com/"

  password: "{{ wp\_password }}"

CopyExplain

This uses the public URL we are configuring on the Azure Load Balancer public IP address and the password variable from the secrets role that just ran.

Everything else in the roles remains as we left it in [*Chapter 5*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/5), *Deploying WordPress*.

Running the playbook

Running the playbook uses the same command we have been running throughout the book:

$ ansible-playbook -i hosts site.yml

CopyExplain

The playbook will execute and by the end of it you should see something like the output on the following screen:

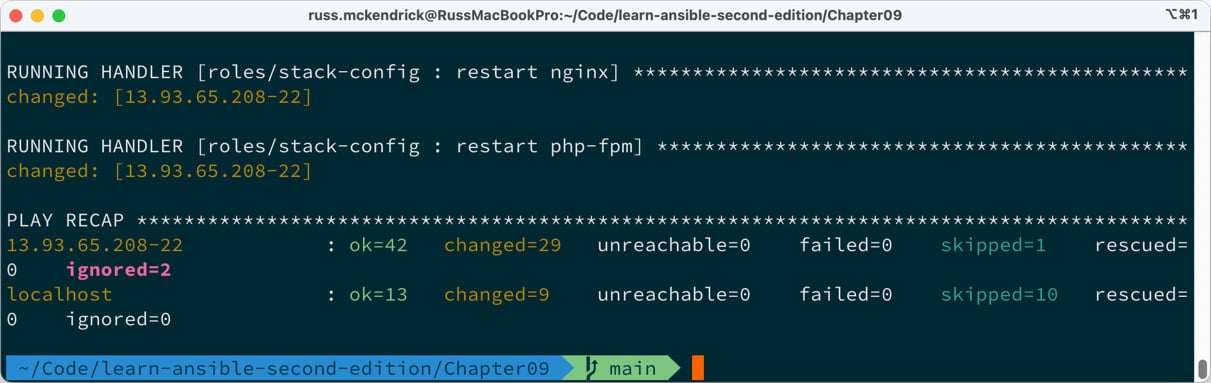


Figure 9.1 – Running the playbook in a terminal

Visiting the Azure portal at https://portal.azure.com/ and viewing the resource group that Ansible created should show you something like the following:

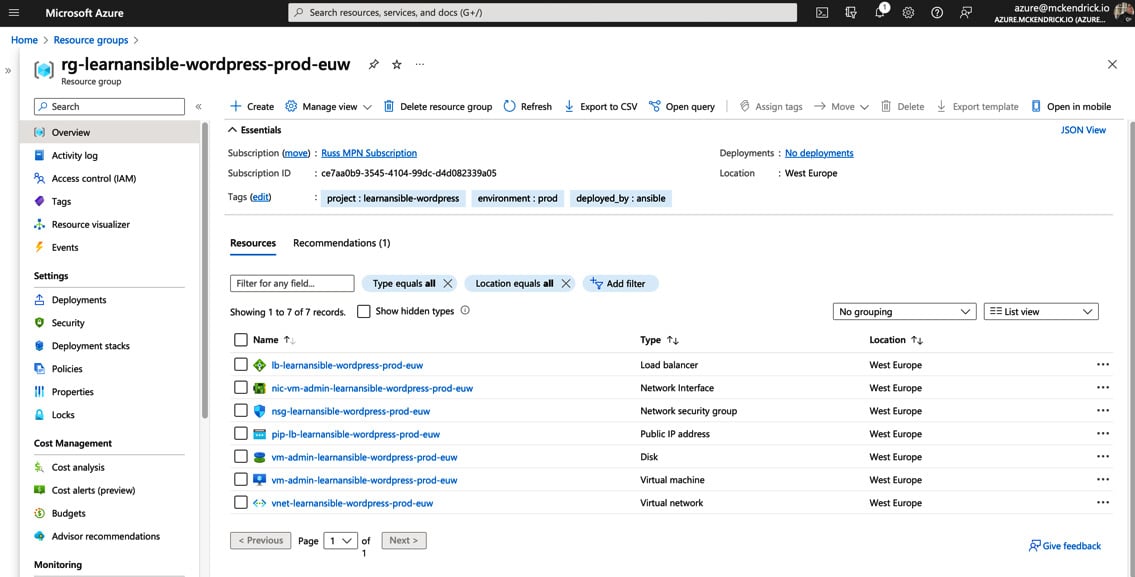


Figure 9.2 – Viewing the resources in the Azure portal

From here, you should be able to enter the DNS name assigned on the public IP address; for example, in my instance, it was http://learnansible.westeurope.cloudapp.azure.com/. This may be different in your case and you should see your newly bootstrapped WordPress site.

Just like when we launched Azure resources in [*Chapter 7*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/7), *Ansible Windows Modules*, to terminate the resources, we need to remove the resource group, which will remove all the resources contained there.

To do this using Ansible, there is a small, self-contained playbook called destroy.yml, which can be executed by running the following:

$ ansible-playbook -i hosts destory.yml

CopyExplain

This will take a few minutes to run, but it will remove all resources deployed in the *site.yml* playbook, including the ones in Azure and the group\_vars/secrets.yml file, leaving you with a nice clean slate for when you next run the main site.yml playbook.

Summary

In this chapter, we launched our first instances in a public cloud using the Azure Ansible modules; as you have seen, the process was relatively straightforward, and we managed to securely launch the network and compute resource in Microsoft Azure, ready for us to then install WordPress on it without making any significant changes to the roles we covered in [*Chapter 5*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/5), *Deploying WordPress*.

In the next chapter, we will expand on some of the techniques we have covered in this chapter and return to networking, but unlike the previous chapter, where we covered networking devices, we will be looking at networking in public clouds.