Suppose you want to create three resource groups, `terraform-exercise-dev`, `terraform-exercise-test`, and `terraform-exercise-prod`. Each resource group will contain a similar set of infrastructure with similar properties. For example, each resource group includes an Ubuntu **Virtual Machine** (**VM**).

A simple method to approach the problem is by creating a structure like the following:

```
├── dev
│   ├── backend.tf
│   ├── main.tf
│   ├── terraform.tfvars
│   └── vars.tf
├── prod
│   ├── backend.tf
│   ├── main.tf
│   ├── terraform.tfvars
│   └── vars.tf
└── test
    ├── backend.tf
    ├── main.tf
    ├── terraform.tfvars
    └── vars.tf
```

Can you see the duplication? The same files occur multiple times, all containing the same configuration. The only thing that might change is the `terraform.tfvars` file for each environment.

So, this does not sound like a great way to approach this problem, and that's why Terraform provides workspaces for it.

Terraform workspaces are nothing but independent state files. So, you have a single configuration and multiple state files for each environment. Sounds simple, right? Let's have a look.

Another way to represent the same configuration by using Terraform workspaces is the following:

```
├── backend.tf
├── main.tf
├── terraform.tfvars
└── vars.tf
```

Now, this looks simple. It just contains a single set of files. Let's have a look at each of them to understand them better.

To access the resources for this section, `cd` into the following:

```
$ cd ~/modern-devops/ch8/terraform-workspaces/
```

The `main.tf` file contains a `resource_group` resource with a name that includes an environment suffix, along with other resources that we need to create within the resource group, such as the VNet, subnet, and VM, something like the following:

```
...
resource "azurerm_resource_group" "main" {
  name     = "${var.rg_prefix}-${terraform.workspace}"
  location = var.rg_location
}
resource "azurerm_virtual_network" "main" {
  ...
}
resource "azurerm_subnet" "internal" {
  ...
}
resource "azurerm_network_interface" "main" {
...
}
resource "azurerm_virtual_machine" "main" {
  ...
}
...
```

To access the name of the workspace, Terraform provides the `terraform.workspace` variable, which we have used to define the `resource_group` name. So, the template is now ready to take configuration for any environment, and we will have a separate resource group for each environment.

Also, update the `backend.tf` file with the `tfstate` container name we created in the last section and initialize the Terraform workspace by using the following command:

```
$ terraform init
```

Now, once Terraform has initialized, let's create a `dev` workspace by using the following command:

```
$ terraform workspace new dev
Created and switched to workspace "dev"!
```

You're now in a new, empty workspace. Workspaces isolate their state, so if you run `terraform plan`, Terraform will not see any existing state for this configuration.

So, as we're in a new, empty workspace called `dev`, let's run a plan.

Use the following command to run a plan on the `dev` environment:

```
$ terraform plan -out dev.tfplan
Acquiring state lock. This may take a few moments...
Terraform will perform the following actions:
  + resource "azurerm_network_interface" "main" {
      ...
    }
  + resource "azurerm_resource_group" "main" {
```

```
      + id       = (known after apply)
      + location = "westeurope"
      + name     = "terraform-ws-dev"
    }
  + resource "azurerm_subnet" "internal" {
      ...
    }
  + resource "azurerm_virtual_machine" "main" {
    ...
    }
  + resource "azurerm_virtual_network" "main" {
    ...
    }
Plan: 5 to add, 0 to change, 0 to destroy.
```

Now, let's go ahead and apply the plan using the following command:

```
$ terraform apply "dev.tfplan"
Acquiring state lock. This may take a few moments...
azurerm_resource_group.main: Creating...
azurerm_virtual_network.main: Creating...
azurerm_subnet.internal: Creating...
azurerm_network_interface.main: Creating...
azurerm_virtual_machine.main: Creating...
Apply complete! Resources: 5 added, 0 changed, 0 destroyed.
Releasing state lock. This may take a few moments...
```

As the `dev` plan has been applied and the resources are created in the `dev` resource group, let's create a workspace for testing:

```
$ terraform workspace new test
```

As the new workspace is created, let's run a plan on the test workspace using the following command and save it to the `test.tfplan` file:

```
$ terraform plan -out test.tfplan
...
  + resource "azurerm_resource_group" "main" {
      + id       = (known after apply)
      + location = "westeurope"
      + name     = "terraform-ws-test"
    }
...
```

As we can see, the resources will be created in the `terraform-ws-test` resource group. So, let's go ahead and apply the plan using the following command:

```
$ terraform apply test.tfplan
```

The `test` plan has been applied as well. Now let's go ahead and inspect the created resources.

> **Tip**
>
> Terraform workspaces are ideal for maintaining separate infrastructure configurations for different environments, such as development, staging, and production. This helps prevent accidental configuration changes and ensures consistent setups.

Inspecting resources

Let's use the `az` command to list the resource groups. As we know, our resource groups have a resource group prefix of `terraform-ws`. Therefore, use the following command to list all resource groups containing the prefix:

```
$ az group list | grep name | grep terraform-ws
    "name": "terraform-ws-dev",
    "name": "terraform-ws-test",
```

As we can see, we have two resource groups, `terraform-ws-dev` and `terraform-ws-test`. So, two resource groups have been created successfully.

You can also verify this in the Azure portal, as shown in the following screenshot:

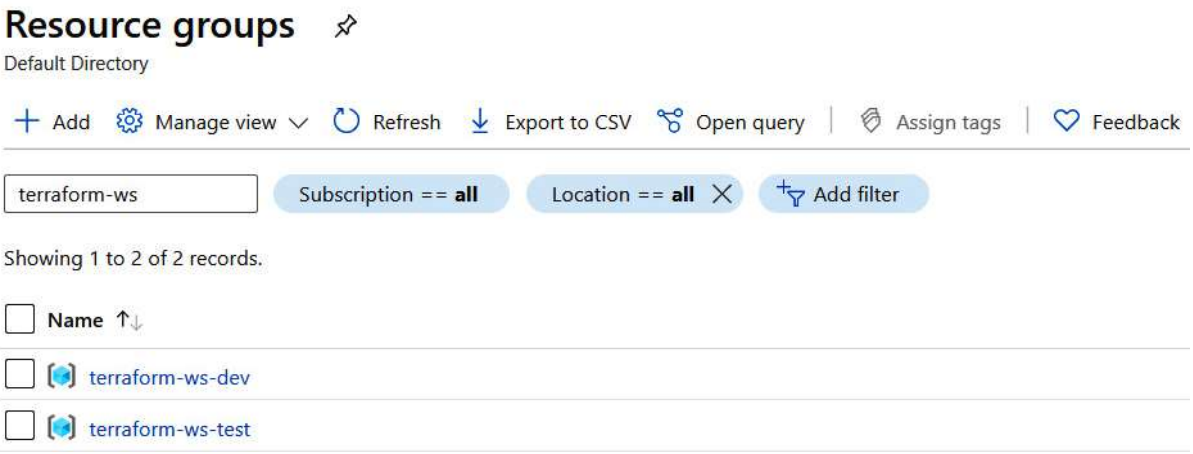

Figure 8.3 – Resource groups

Now, let's go ahead and inspect the resources on the `terraform-ws-dev` resource group using the Azure portal by clicking on `terraform-ws-dev`:

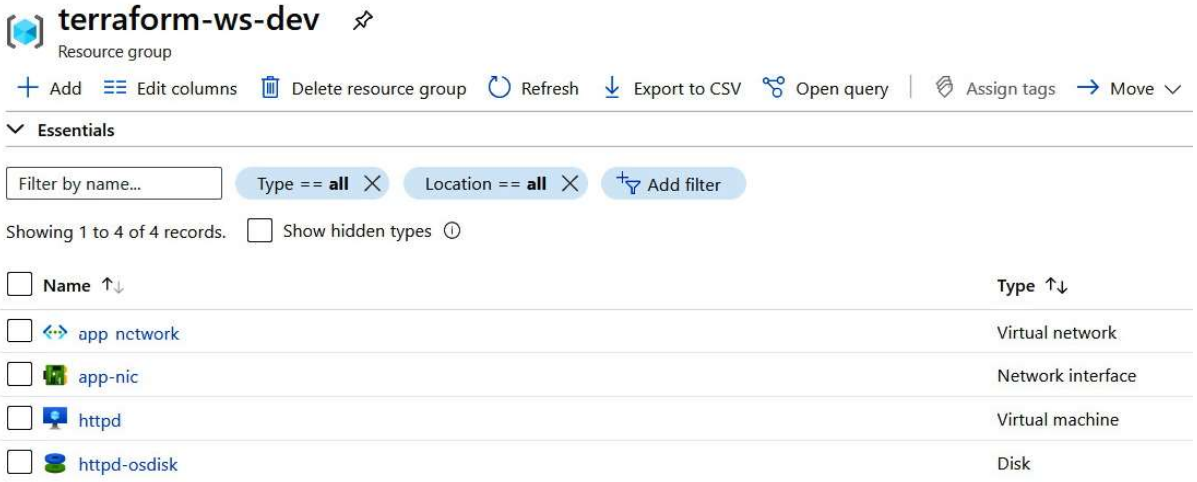

Figure 8.4 – Terraform dev resource group

We have a virtual network, a network interface, an OS disk, and a VM within the resource group. We should expect similar resources with the same names in the `terraform-ws-test` resource group. Let's go ahead and have a look:

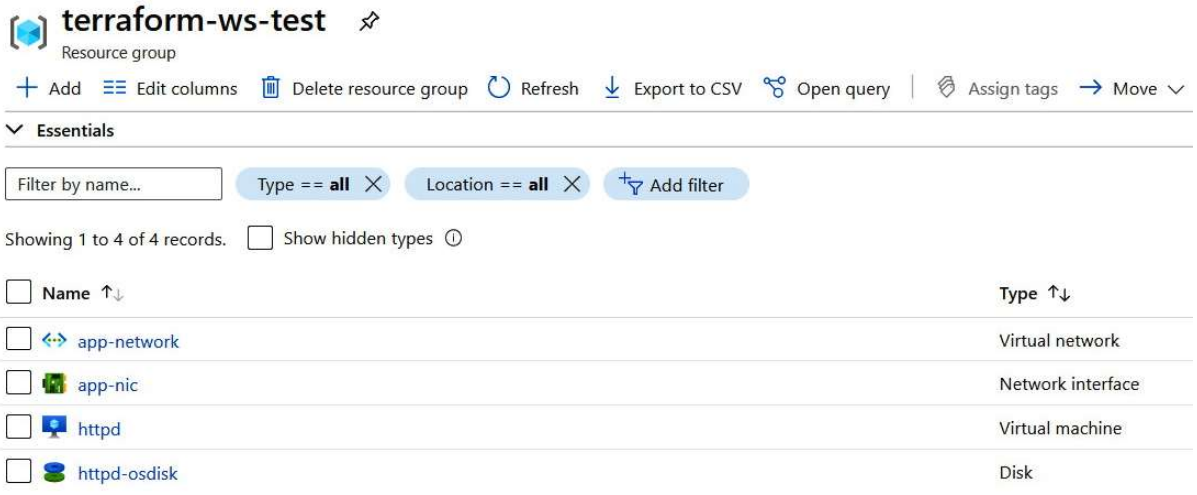

Figure 8.5 – Terraform test resource group

As we can see, we also have similar resources in the `terraform-ws-test` resource group.

We did all this using a single configuration, but there should be two state files for each workspace since they are two sets of resources. Let's have a look.

Inspecting state files

If we had used the local backend for the state files, we would get the following structure:

```
|-- terraform.tfstate.d
    |-- dev
    |   `-- terraform.tfstate
    `-- test
        `-- terraform.tfstate
```

So, Terraform creates a directory called `terrafom.tfstate.d`; within that, it creates directories for each workspace. Within the directories, it stores the state file for each workspace as `terraform.tfstate`.

But since we are using a remote backend and using Azure Blob Storage for it, let's inspect the files within it using the Azure console:

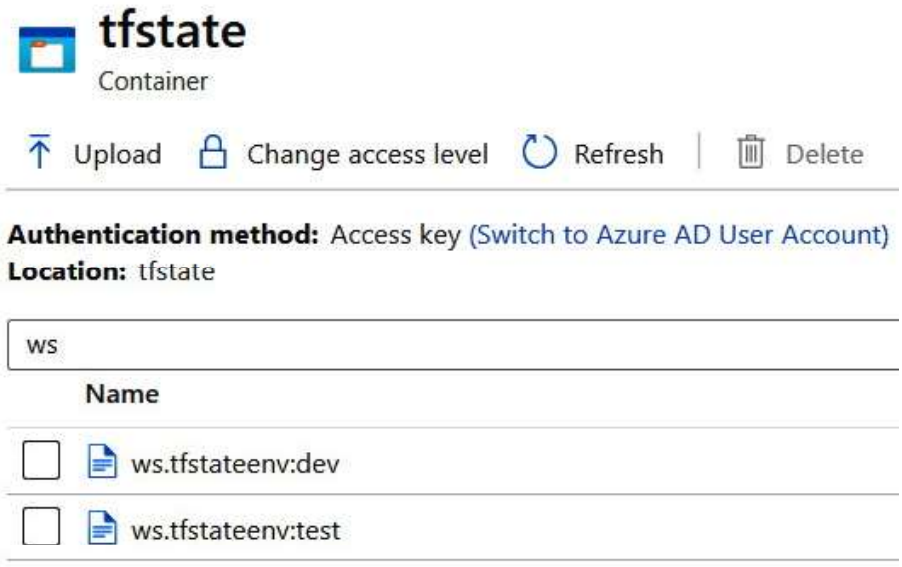

Figure 8.6 – Terraform workspace state

As we see, there are two state files, one for each environment. Therefore, the state files are suffixed with an `env:dev` or `env:test` string. That is how workspaces are managed in Azure Blob Storage. The remote backend's structure for maintaining state files depends on the provider plugins, and therefore, there might be different ways of managing multiple states for various backends. However, the Terraform CLI will interpret workspaces the same way, irrespective of the backends, so nothing changes for the end user from a CLI perspective.

Cleaning up

Now, let's go ahead and clean up both resource groups to avoid unnecessary charges.

As we're already within the test workspace, let's run the following command to destroy resources within the test workspace:

```
$ terraform destroy --auto-approve
```

Now, let's switch to the `dev` workspace using the following command:

```
$ terraform workspace select dev
Switched to workspace "dev".
```

As we're within the `dev` workspace, use the following command to destroy all resources within the `dev` workspace:

```
$ terraform destroy --auto-approve
```

In a while, we should see that both resource groups are gone. Now, let's look at some of the advanced concepts of Terraform in the next section.

Terraform output, state, console, and graphs

While we understand that Terraform uses state files to manage resources, let's look at some advanced commands to help us appreciate and make more sense of the Terraform state concept.

To access the resources for this section, `cd` into the following:

```
$ cd ~/modern-devops/ch8/terraform-workspaces/
```

Now, let's go ahead and look at our first command – `terraform output`.

terraform output

So far, we've looked at variables but haven't yet discussed outputs. Terraform outputs are return values of a Terraform configuration that allow users to export configuration to users or any modules that might use the current module.

Let's go with the last example and add an output variable that exports the private IP of the network interface attached to the VM in the `outputs.tf` file:

```
output "vm_ip_addr" {
  value = azurerm_network_interface.main.private_ip_address
}
```

Now, let's go ahead and apply the configuration:

```
$ terraform apply --auto-approve
...
Outputs:
vm_ip_addr = "10.0.2.4"
```

After Terraform has applied the configuration, it shows the outputs at the end of the console result. You can run the following to inspect the output anytime later:

```
$ terraform output
vm_ip_addr = "10.0.2.4"
```

Outputs are stored in the state file like everything else, so let's look at how we can manage Terraform state using the CLI.

Managing Terraform state

Terraform stores the configuration it manages in state files and therefore provides a command for advanced state management. The `terraform state` command helps you manage the state of the current configuration. While the state file is plaintext and you can manually modify it, using the `terraform state` command is recommended.

But before we get into details, we must understand why we want to do that. Things might not always go according to plan, so the state file may have corrupt data. You also might want to see specific attributes of a particular resource after you've applied it. The state file might need to be investigated for the root cause analysis of a specific infrastructure provisioning problem. Let's have a look at the most common use cases.

Viewing the current state

To view the current state, we can run the following command:

```
$ terraform show
```

That will output all resources that Terraform has created and manages, including outputs. Of course, this can be overwhelming for some, and we may want to view the list of resources Terraform manages.

Listing resources in the current state

To list the resources in the Terraform state file, run the following command:

```
$ terraform state list
azurerm_network_interface.main
azurerm_resource_group.main
azurerm_subnet.internal
```

```
azurerm_virtual_machine.main
azurerm_virtual_network.main
```

And as we see, there are five resources managed by Terraform. You might want to remove a resource from the Terraform state. It might be possible that someone has removed a resource manually as it is no longer required, but it isn't removed from the Terraform configuration.

Removing a resource from the state

To remove a state manually from the Terraform state file, you must use the `terraform state rm <resource>` command. For example, to remove the Azure VM resource from the Terraform state, run the following command:

```
$ terraform state rm azurerm_virtual_machine.main
Acquiring state lock. This may take a few moments...
Removed azurerm_virtual_machine.main
Successfully removed 1 resource instance(s).
Releasing state lock. This may take a few moments...
```

Bear in mind that this has merely removed the resource from the state file and has not touched the actual resource sitting on Azure.

There might be instances where someone spun up a VM manually within Azure, and we now want Terraform to manage it. This kind of situation happens mostly in brownfield projects. In that case, we must declare the same configuration within Terraform and then import existing resources in the Terraform state. To do so, we can use the `terraform import` command.

Importing existing resources into Terraform state

You can use the `terraform import` command to import existing resources into Terraform state. The `terraform import` command is structured as follows:

```
terraform import <resource> <resource_id>
```

For example, to reimport the `httpd` VM into the state, run the following command:

```
$ terraform import azurerm_virtual_machine.main \
"/subscriptions/<SUBSCRIPTION_ID>/resourceGroups\
/terraform-ws-dev/providers/Microsoft.Compute/virtualMachines/httpd"
Acquiring state lock. This may take a few moments...
azurerm_virtual_machine.main: Importing from ID "/subscriptions/id/resourceGroups/
terraform-ws-dev/providers/Microsoft.Compute/virtualMachines/httpd"...
azurerm_virtual_machine.main: Import prepared!
  Prepared azurerm_virtual_machine for import
azurerm_virtual_machine.main: Refreshing state... [id=/subscriptions/1de491b5-f572-
459b-a568-c4a35d5ac7a9/resourceGroups/terraform-ws-dev/providers/Microsoft.Compute/
virtualMachines/httpd]
Import successful!
```

To check whether the resource is imported to the state, we can list the resources again using the following command:

```
$ terraform state list | grep azurerm_virtual_machine
azurerm_virtual_machine.main
```

As we see, we have the VM within the state file. If we want to dig further into the resources, we can use `terraform console`.

terraform console

The `terraform console` command provides an interactive console to investigate state files, dynamically build paths, and evaluate expressions even before using them in resources. It is a potent tool that most advanced Terraform users use. For example, let's launch the console and look through the configuration of the VM resource we just imported.

Use the following commands to launch the console and get the resource group of the VM and the `id` value:

```
$ terraform console
Acquiring state lock. This may take a few moments...
> azurerm_virtual_machine.main.resource_group_name
"terraform-ws-dev"
> azurerm_virtual_machine.main.id
"/subscriptions/id/resourceGroups/terraform-ws-dev/providers/Microsoft.Compute/
virtualMachines/httpd"
> exit
Releasing state lock. This may take a few moments...
```

As we can see, the VM is in the correct resource group, and we're satisfied that the import was correct.

Terraform dependencies and graphs

Terraform uses a dependency model to manage in what order resources are created and destroyed. There are two kinds of dependencies – *implicit* and *explicit*. We've been using implicit dependencies until now, where the VM depended upon the network interface, and the network interface depended upon the subnet. The subnet depended upon the virtual network, and all of these resources depended on the resource group. These dependencies naturally occur when we use one resource's output as another's input.

However, sometimes, we want to define an explicit dependency on a resource, especially when there is no way to define an implicit dependency on it. You can use the `depends_on` attribute for that kind of operation.

> **Tip**
> Avoid explicit dependencies unless needed, as Terraform uses parallelism to manage resources. If explicit dependencies are not required, it will slow down Terraform runs because it can process multiple parallel resources.

To visualize the dependencies between resources, we can export a graph from the state file and convert that into a PNG file using a tool such as **Graphviz**.

Run the following command to export the dependency graph:

```
$ terraform graph > vm.dot
```

We can then process the graph file using the Graphviz tool. To install the tool on Ubuntu, run the following command:

```
$ sudo apt install graphviz -y
```

Now run the following command to convert the graph file into a PNG file:

```
$ cat vm.dot | dot -T png -o vm.png
```

The graph is available at `https://github.com/PacktPublishing/Modern-DevOps-Practices-2e/blob/main/ch8/terraform-graph.png`. Now, let's go ahead and see how we can clean up our resources.

Cleaning up resources

As we already know, we run the following command to clean up the resources:

```
$ terraform destroy --auto-approve
```

It will clear resources from the resource group and delete the resource group after that.

While using `terraform destroy` can be an easy way to eliminate resources you don't need, it is best if you stick to this only in the `dev` environment and never use it in production. Instead, you can remove resources you don't need from the configuration and then run `terraform apply`.

Summary

In this chapter, we've discussed Terraform's core and understood some of the most common commands and functionalities from a hands-on perspective. We started with understanding IaC, introduced Terraform as an IaC tool, installed Terraform, understood Terraform providers, and used the Azure Terraform provider to manage infrastructure in Azure.

We then looked at Terraform variables and multiple ways of supplying values to the variables. We discussed the core Terraform workflow and several commands you would use to manage infrastructure using Terraform. We then looked at Terraform modules and then at Terraform state as an essential component that helps Terraform keep track of the infrastructure it is managing.

We looked at local and remote state storage and used Azure Blob Storage as the remote state backend. We then discussed Terraform workspaces and how they enable us to use the same Terraform configuration to build multiple environments with hands-on exercises.

We then looked at some advanced operations with Terraform state using the `outputs`, `state`, and `console` commands. We finally looked at how Terraform manages dependencies and viewed a dependency graph using the `graph` command.

In the next chapter, we will delve into configuration management using Ansible.

Questions

1. Why should we constrain the provider version?
2. You should always use the `fmt` and `validate` functions before a Terraform plan. (True/False)
3. What does the Terraform `plan` command do? (Choose two)

   A. Refreshes the current state with the existing infrastructure state

   B. Gets the delta between the current configuration and the expected configuration

   C. Applies the configuration to the cloud

   D. Destroys the configuration in the cloud
4. What does the `terraform apply` command do? (Choose three)

   A. Refreshes the current state with the existing infrastructure

   B. Gets the delta between the current configuration and the expected configuration

   C. Applies the configuration to the cloud

   D. Destroys the configuration in the cloud

5. Why should you never store state files in source control? (Choose two)

   A. State files are plaintext, and therefore you expose sensitive information to unprivileged users.

   B. Source control does not support state locking, and therefore it might result in potential conflicts between users.

   C. Multiple admins cannot work on the same configuration.

6. Which of the following are valid Terraform remote backends? (Choose five)

   A. S3

   B. Azure Blob Storage

   C. Artifactory

   D. Git

   E. HTTP

   F. Terraform Enterprise

7. Which command will mark a resource for recreation in the next `apply`?
8. Where are state files stored in the local backend if you use workspaces?
9. What command should we use to remove a Terraform resource from the state?
10. What command should we use to import an existing cloud resource within the state?

Answers

1. Because Terraform providers are released separately to the Terraform CLI, different versions might break the existing configuration
2. True
3. A, B
4. A, B, C
5. A, B
6. A, B, C, E, F
7. The `taint` command
8. `terraform.tfstate.d`
9. `terraform state rm <resource>`
10. `terraform import <resource> <id>`

9

Configuration Management with Ansible

In the last chapter, we looked at **Infrastructure as Code** (**IaC**) with Terraform, its core concepts, IaC workflow, state, and debugging techniques. We will now delve into **configuration management** (**CM**) and **Configuration as Code** (**CaC**) with Ansible. Ansible is a CM tool that helps you to define configuration as idempotent chunks of code.

In this chapter, we're going to cover the following main topics:

- Introduction to configuration management
- Setting up Ansible
- Introduction to Ansible playbooks
- Ansible playbooks in action
- Designing for reusability

Technical requirements

You will need an active Azure subscription to follow the exercises for this chapter. Currently, Azure is offering a free trial for 30 days with $200 worth of free credits, and you can sign up at `https://azure.microsoft.com/en-in/free`.

You will also need to clone the following GitHub repository for some of the exercises:

`https://github.com/PacktPublishing/Modern-DevOps-Practices-2e`

Run the following command to clone the repository into your home directory, and `cd` into the `ch9` directory to access the required resources:

```
$ git clone https://github.com/PacktPublishing/Modern-DevOps-Practices-2e.git \
modern-devops
$ cd modern-devops/ch9
```

You also need to install Terraform on your system. Refer to *Chapter 8*, *Infrastructure as Code (IaC) with Terraform*, for more details on installing and setting up Terraform.

Introduction to configuration management

CM, in the realm of technology and systems administration, can be compared to the role of a conductor leading an orchestra. Imagine yourself guiding a group of musicians, each playing a unique instrument. Your responsibility is to ensure that everyone is harmoniously in sync, adhering to the correct musical score, and executing their parts at precisely the right moments.

In the context of technology and systems administration, CM is the practice of skilfully orchestrating and overseeing the creation, updates, and maintenance of computer systems and software, much akin to how a conductor directs musicians to produce splendid music.

Here's a breakdown of how it functions:

- **Standardization**: As with musicians employing the same musical notes and scales, CM guarantees that all computers and software within an organization adhere to standardized configurations. This uniformity mitigates errors and bolsters system reliability.
- **Automation**: In an orchestra, musicians don't manually fine-tune their instruments during a performance. Likewise, CM tools automate the configuration and upkeep of computer systems, consistently applying configurations without the need for manual intervention.
- **Version control**: Musicians follow specific sheet music, and if alterations occur, everyone receives updated sheet music. CM maintains a version history of system configurations, simplifying the tracking of changes, reverting to previous versions, and ensuring alignment across the board.
- **Efficiency**: Just as a conductor synchronizes the timing of each instrument, CM optimizes system performance and resource allocation. It guarantees that software and systems operate efficiently and can scale as required.
- **Compliance and security**: Analogous to a conductor enforcing performance guidelines, CM enforces adherence to security policies and best practices. It plays a crucial role in upholding a secure and compliant IT environment.
- **Troubleshooting**: When issues arise during a performance, the conductor swiftly identifies and addresses them. CM tools assist in troubleshooting and rectifying configuration-related problems in IT systems.

To understand CM better, let's first look at the traditional way of hosting and managing applications. We first create a **virtual machine** (**VM**) from physical infrastructure and then log in manually to VMs. We can then run a set of scripts or do the setup manually. At least, that's what we've been doing till now, even in this book.

There are several problems with this approach. Let's look at some of them:

- If we set up the server manually, the process is not repeatable. For example, if we need to build another server with a similar configuration, we must repeat the entire process to build another server.
- Even if we use scripts, the scripts themselves are not idempotent. This means they cannot identify and apply only the delta configuration if needed.
- Typical production environments consist of many servers; therefore, setting everything up manually is a labor-intensive task and adds to the toil. Software engineers should focus on novel ways of automating processes that cause toil.
- While you can store scripts within source control, they are *imperative*. We always encourage a *declarative* way of managing things.

Modern CM tools such as Ansible solve all these problems by providing the following benefits:

- They manage configuration through a set of declarative code pieces
- You can store code in version control
- You can apply code to multiple servers from a single control node
- As they are idempotent, they only apply the delta configuration
- It is a repeatable process; you can use variables and templates to apply the same configuration to multiple environments
- They provide deployment orchestration and are mostly used within CI/CD pipelines

Although many tools available on the market provide CM, such as **Ansible**, **Puppet**, **Chef**, and **SaltStack**, Ansible is the most popular and straightforward tool used for this. It is more efficient, and its simplicity makes it less time-consuming than others.

It is an open source CM tool built using Python and is owned by **Red Hat**. It provides the following features:

- It helps you to automate routine tasks such as OS upgrades, patches, and backups while also creating all OS-level configurations, such as users, groups, permissions, and others
- The configuration is written using simple YAML syntax
- It uses **Secure Shell** (**SSH**) to communicate with managed nodes and sends commands
- The commands are executed sequentially within each node in an idempotent manner
- It connects to nodes parallelly to save time

Let's delve into the reasons why using Ansible is a great choice for CM and automation. Here are some compelling factors:

- **Simplicity and user-friendliness**: Ansible boasts an uncomplicated, human-readable YAML syntax that's easy to grasp and employ, even for those with limited coding experience.
- **Agentless approach**: Ansible communicates through SSH or WinRM, eliminating the need to install agents on managed nodes. This reduces overhead and security concerns, a topic we'll explore further when we discuss Ansible architecture.
- **Idempotent operations**: Ansible ensures the desired system state is achieved, even if configurations are applied repeatedly. This minimizes the risk of unintended changes.
- **Broad adoption**: With a thriving and active user community, Ansible offers extensive documentation, modules, and playbooks for various use cases.
- **Cross-platform compatibility**: Ansible can handle diverse environments, managing various operating systems, cloud providers, network devices, and infrastructure components with a single tool.
- **Seamless integration**: Ansible seamlessly integrates with other tools, including **version control systems** (**VCSs**), monitoring solutions, and CI/CD pipelines.
- **Scalability**: Ansible scales effortlessly to handle both small and large environments, catering to both enterprises and start-ups.
- **Version control**: Infrastructure configurations are stored in plain text files, simplifying change management, history tracking, and collaboration through Git or similar VCSs.
- **Automation of routine tasks**: Ansible automates repetitive chores such as software installations, configuration updates, and patch management, freeing up time for strategic tasks.
- **Security and compliance**: Implement security policies and compliance standards consistently across your infrastructure using Ansible's **Role-Based Access Control** (**RBAC**) and integrated security modules.
- **Rollback and recovery**: Ansible enables easy rollback to prior configurations in the case of issues, reducing downtime and minimizing the impact of changes.
- **Modularity and reusability**: Ansible encourages the creation of modular, reusable playbooks and roles, fostering an organized and efficient automation approach.
- **Supportive community**: Benefit from a robust Ansible community that offers support, documentation, and a repository of contributed roles and modules.
- **Cost-effective**: Ansible is open source and free to use, cutting down on licensing expenses compared to other automation tools.
- **Orchestration and workflow automation**: Beyond CM, Ansible can orchestrate intricate workflows, including application deployment and infrastructure provisioning.

- **Immutable infrastructure**: Ansible supports the concept of immutable infrastructure, where changes involve recreating components rather than modifying them in place. This leads to more predictable and dependable deployments.
- **Real-time feedback**: Ansible provides real-time feedback and reporting, simplifying the monitoring and troubleshooting of automation tasks.

These advantages establish Ansible as a popular choice for CM, automation, and orchestration across a wide spectrum of IT environments and industries.

Ansible has a simple architecture. It has a **control node** that takes care of managing multiple **managed nodes**. All you need is a control node server to install Ansible and the nodes to manage using the control node (also known as managed nodes). The managed nodes should allow an SSH connection from the Ansible control node—something like the following diagram:

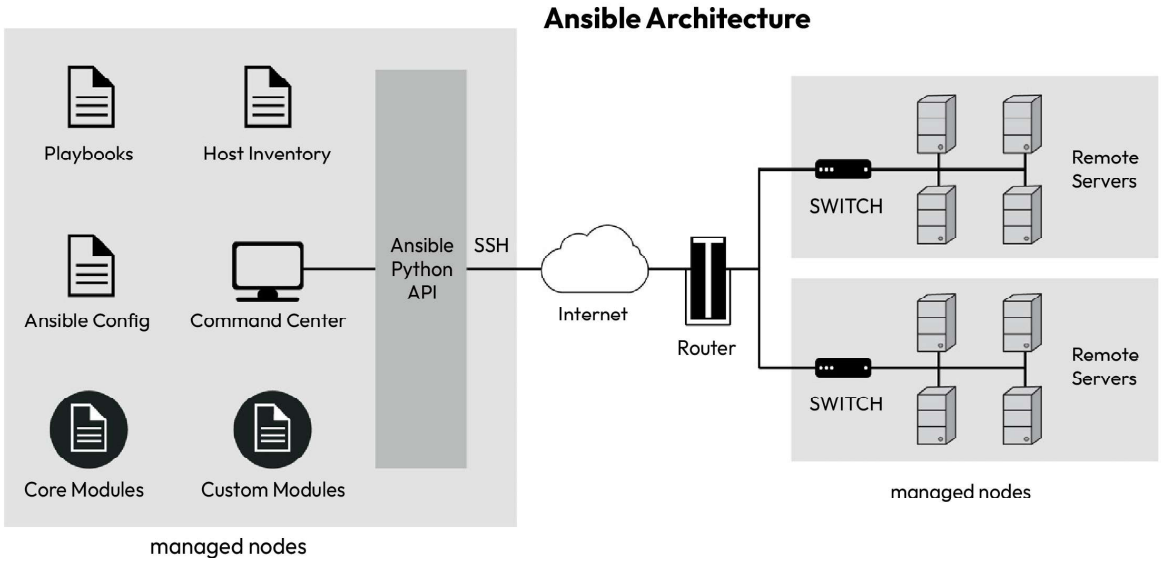

Figure 9.1 – Ansible architecture

Now, let's go ahead and see how we can install and set up the required configuration using Ansible. Let's look at how to install Ansible in the next section.

Setting up Ansible

We need to set up and install Ansible in the control node, but before we do that, we will have to spin three servers to start the activity—an Ansible control node and two managed nodes.

Setting up inventory

The idea is to set up a two-tier architecture with **Apache** and **MySQL**. So, let's use Terraform to spin up the three servers.

Let's first `cd` into the directory where the Terraform templates are located and then edit the `terraform.tfvars` file to fill in the required details. (Please refer to *Chapter 8*, *Infrastructure as Code (IaC) with Terraform*, for more details about how to get the attributes):

```
$ cd ~/modern-devops/ch9/setup-ansible-terraform
$ vim terraform.tfvars
```

Then, use the following commands to spin up the servers using Terraform:

```
$ terraform init
$ terraform plan -out ansible.tfplan
$ terraform apply ansible.tfplan
```

Once the `terraform apply` command is completed successfully, we will see three servers—`ansible-control-node`, `web`, and `db`, and the associated resources created within the `ansible-exercise` resource group.

The `terraform apply` output also provides the public IP addresses of the Ansible control node and the `web` VM. You should see the public IP address we got in the output.

> **Note**
>
> It might take a while for Azure to report the output, and if you did not get the IP addresses during `terraform apply`, you could subsequently run `terraform output` to get the details.

Ansible requires the control node to connect with managed nodes via SSH. Now, let's move on and look at how we can communicate with our managed nodes (also known as inventory servers).

Connecting the Ansible control node with inventory servers

We've already set up **passwordless SSH** between the control node and managed nodes when we provisioned the infrastructure using Terraform. Let's look at how we did that to understand it better.

We created an **Azure Virtual Network** (**VNet**), a **subnetwork** (**subnet**), and three **Azure VMs** called `control-node`, `web`, and `db` within that subnet. If we look at the VM resource configuration, we also have a `custom_data` field that can be used to pass an initialization script to the VM, as follows:

```
resource "azurerm_virtual_machine" "control_node" {
  name                  = "ansible-control-node"
  ...
```

```
  os_profile {
   ...
   custom_data    = base64encode(data.template_file.control_node_init.rendered)
  }
}
resource "azurerm_virtual_machine" "web" {
  name                  = "web"
  ...
  os_profile {
   ...
   custom_data    = base64encode(data.template_file.managed_nodes_init.rendered)
  }
}
resource "azurerm_virtual_machine" "db" {
  name                  = "db"
  ...
  os_profile {
   ...
   custom_data    = base64encode(data.template_file.managed_nodes_init.rendered)
  }
}
```

As we can see, the `control_node` VM refers to a `data.template_file.control_node_init` resource, and the `web` and `db` nodes refer to a `data.template_file.managed_nodes_init` resource. These are `template_file` resources that can be used for template files. Let's look at the resources as follows:

```
data "template_file" "managed_nodes_init" {
  template = file("managed-nodes-user-data.sh")
  vars = {
    admin_password = var.admin_password
  }
}
data "template_file" "control_node_init" {
  template = file("control-node-user-data.sh")
  vars = {
    admin_password = var.admin_password
  }
}
```

As we can see, the `managed_nodes_init` resource points to the `managed-nodes-user-data.sh` file and passes an `admin_password` variable to that file. Similarly, the `control_node_init` resource points to the `control-node-user-data.sh` file. Let's look at the `managed-nodes-user-data.sh` file first:

```
#!/bin/sh
sudo useradd -m ansible
echo 'ansible ALL=(ALL) NOPASSWD:ALL' | sudo tee -a /etc/sudoers
sudo su - ansible << EOF
ssh-keygen -t rsa -N '' -f ~/.ssh/id_rsa
```

```
printf "${admin_password}\n${admin_password}" | sudo passwd ansible
EOF
```

As we can see, it is a shell script that does the following:

1. Creates an `ansible` user.
2. Adds the user to the `sudoers` list.
3. Generates an `ssh` key pair for passwordless authentication.
4. Sets the password for the `ansible` user.

As we've generated the `ssh` key pair, we would need to do the same within the control node with some additional configuration. Let's look at the `control-node-user-data.sh` script, as follows:

```
#!/bin/sh
sudo useradd -m ansible
echo 'ansible ALL=(ALL) NOPASSWD:ALL' | sudo tee -a /etc/sudoers
sudo su - ansible << EOF
ssh-keygen -t rsa -N '' -f ~/.ssh/id_rsa
sleep 120
ssh-keyscan -H web >> ~/.ssh/known_hosts
ssh-keyscan -H db >> ~/.ssh/known_hosts
sudo apt update -y && sudo apt install -y sshpass
echo "${admin_password}" | sshpass ssh-copy-id ansible@web
echo "${admin_password}" | sshpass ssh-copy-id ansible@db
EOF
```

The script does the following:

1. Creates an `ansible` user
2. Adds the user to the `sudoers` list
3. Generates an `ssh` key pair for passwordless authentication
4. Adds the `web` and `db` VMs to the `known_hosts` file to ensure we trust both hosts
5. Installs the `sshpass` utility to allow for sending the `ssh` public key to the `web` and `db` VMs
6. Copies the `ssh` public key to the `web` and `db` VMs for passwordless connectivity

These files get executed automatically when the VMs are created; therefore, passwordless SSH should already be working. So, let's use an **SSH client** to log in to `ansible-control-node` using the IP address we got in the last step. We will use the username and password we configured in the `terraform.tfvars` file:

```
$ ssh ssh_admin@104.46.61.213
```

Once you are in the control node server, switch the user to `ansible` and try doing an SSH to the `web` server using the following commands:

```
$ sudo su - ansible
$ ssh web
```

And if you land on the `web` server, passwordless authentication is working correctly.

Repeat the same steps to check whether you can connect with the `db` server.

Exit the prompts until you are in the control node.

Now, as we're in the control node, let's install Ansible.

Installing Ansible in the control node

Ansible requires a Linux/Unix machine (preferably), and you should have Python `2.x` or `3.x` installed.

As the Ansible control node runs on Ubuntu, Ansible provides a **personal package archive** (**PPA**) repository that we can configure to download and install Ansible using `apt` commands.

Use the following commands to install Ansible on the server:

```
$ sudo apt update
$ sudo apt install software-properties-common -y
$ sudo apt-add-repository --yes --update ppa:ansible/ansible
$ sudo apt install ansible -y
```

To check whether Ansible has been installed successfully, run the following command:

```
$ ansible --version
ansible 2.9.27
```

And, as we see, `ansible 2.9.27` is successfully installed on your control node.

Ansible uses an inventory file to manage nodes. Therefore, the next step is to set up an inventory file.

Setting up an inventory file

An inventory file within Ansible is a file that allows you to group your managed nodes according to roles. For example, you can define roles such as `webserver` and `dbserver` and group related servers together. You can use IP addresses, hostnames, or aliases for that.

> **Tip**
> Always use aliases because they provide room for IP address and hostname changes.

You can run Ansible commands on hosts or a group of hosts using the role tagged to them. There is no limit to servers that can have a particular role. If your server uses a non-standard SSH port, you can also use that port within the inventory file.

The default location of the Ansible inventory file is `/etc/ansible/hosts`. If you look at the `/etc/ansible` directory ownership, it is owned by the `root` user. We want to use the `ansible` user that we created for security purposes. Therefore, we must change the `/etc/ansible` directory ownership and its subdirectories and files to `ansible`. Use the following command to do so:

```
$ sudo chown -R ansible:ansible /etc/ansible
```

We can then switch the user to `ansible` and clone the Git repository that contains the required files into the control server using the following commands:

```
$ sudo su - ansible
$ git clone https://github.com/PacktPublishing/Modern-DevOps-Practices-2e.git \
modern-devops
$ cd ~/modern-devops/ch9/ansible-exercise
```

In our scenario, we have a web server called `web` and a database server called `db`. Therefore, if you check the host file called `hosts` within the repository, you will see the following:

```
[webservers]
web ansible_host=web
[dbservers]
db ansible_host=db
[all:vars]
ansible_python_interpreter=/usr/bin/python3
```

The `[all:vars]` section contains variables applicable to all groups. Here, we're explicitly defining `ansible_python_interpreter` to `python3` so that Ansible uses `python3` instead of `python2`. As we're using Ubuntu, `python3` comes installed as default, and `python2` is deprecated.

We also see that instead of using `web` directly, we've specified an `ansible_host` section. That defines `web` as an alias, pointing to a host with the hostname `web`. You can also use the IP address instead of the hostname if required.

> **Tip**
>
> Always group the inventory according to the function performed. That helps us to apply a similar configuration to a large number of machines with a similar role.

As we want to keep the configuration with code, we would wish to stay within the Git repository itself. So, we must tell Ansible that the inventory file is in a non-standard location. To do so, we will create an Ansible configuration file.

Setting up the Ansible configuration file

The Ansible configuration file defines global properties that are specific to our setup. The following are ways in which you can specify the Ansible configuration file, and the first method overrides the next – the settings are not merged, so keep that in mind:

- By setting an environment variable, `ANSIBLE_CONFIG`, pointing to the Ansible configuration file
- By creating an `ansible.cfg` file in the current directory
- By creating an `ansible.cfg` file in the home directory of the current user
- By creating an `ansible.cfg` file in the `/etc/ansible` directory

> **Tip**
> If you manage multiple applications, with each application in its Git repositories, having a local `ansible.cfg` file in every repository will help keep the applications decentralized. It will also enable GitOps and make Git the **single source of truth**.

So, if you check the `ansible.cfg` file in the current directory, you will see the following:

```
[defaults]
inventory = ./hosts
host_key_checking = False
```

Now, to check whether our inventory file is correct, let's list our inventory by using the following command:

```
$ ansible-inventory --list -y
all:
  children:
    dbservers:
      hosts:
        db:
          ansible_host: db
          ansible_python_interpreter: /usr/bin/python3
    ungrouped: {}
    webservers:
      hosts:
        web:
          ansible_host: web
          ansible_python_interpreter: /usr/bin/python3
```

We see that there are two groups—`dbservers` containing `db` and `webservers` containing `web`, each using `python3` as the `ansible_python_interpreter`.

If we want to see all the hosts, we can use the following command:

```
$ ansible --list-hosts all
  hosts (2):
```

```
    web
    db
```

If we want to list all hosts that have the `webservers` role, we can use the following command:

```
$ ansible --list-hosts webservers
  hosts (1):
    web
```

Now, let's check whether Ansible can connect to these servers by using the following command:

```
$ ansible all -m ping
web | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
db | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

And, as we can observe, we get a successful response for both servers. So, we're all set up and can start defining the configuration. Ansible offers **tasks** and **modules** to provide CM. Let's look at these in the next section.

Ansible tasks and modules

Ansible tasks form the basic building block of running Ansible commands. Ansible tasks are structured in the following format:

```
$ ansible <options> <inventory>
```

Ansible modules are reusable code for a particular function, such as running a `shell` command or creating and managing users. You can use Ansible modules with Ansible tasks to manage configuration within managed nodes. For example, the following command will run the `uname` command on each managed server:

```
$ ansible -m shell -a "uname" all
db | CHANGED | rc=0 >>
Linux
web | CHANGED | rc=0 >>
Linux
```

So, we get a reply from the `db` server and the `web` server, each providing a return code, `0`, and an output, `Linux`. If you look at the command, you will see that we have provided the following flags:

- `-m`: The name of the module (`shell` module here)
- `-a`: The parameters to the module (`uname` in this case)

The command finally ends with where we want to run this task. Since we've specified `all`, it runs the task on all servers. We can run this on a single server, a set of servers, a role, or multiple roles, or use a wildcard to select the combination we want.

The tasks have three possible statuses—`SUCCESS`, `CHANGED`, and `FAILURE`. The `SUCCESS` status denotes that the task was successful, and Ansible took no action. The `CHANGED` status denotes that Ansible had to change the existing configuration to apply the expected configuration, and `FAILURE` denotes an error while executing the task.

Ansible modules are reusable scripts that we can use to define configuration within servers. Each module targets a particular aspect of CM. Modules are used in both Ansible tasks and playbooks. There are many modules available for consumption, and they are available at `https://docs.ansible.com/ansible/latest/collections/index_module.html`. You can pick and choose modules according to your requirements and use cases.

> **Tip**
>
> As Ansible is idempotent, always use modules specific to your task and avoid using `command` and `shell` modules. For example, use the `apt` module to install a package instead of the `command` module to run `apt install <package> -y`. If your playbook starts to look like code, then you're doing something fundamentally wrong.

Tasks do not make sense when we have a series of steps to follow while setting up a server. Therefore, Ansible provides *playbooks* for this activity. Let's have a look at this in the next section.

Introduction to Ansible playbooks

Imagine you're a conductor leading an orchestra. In this scenario, Ansible playbooks are akin to your musical score, guiding every musician to create a harmonious symphony of automation in the tech world.

In the realm of tech and automation, Ansible playbooks provide the following:

- **Musical score for automation**: Just as a conductor uses a musical score with notations to guide each instrument, an Ansible playbook contains a set of instructions and actions for orchestrating specific IT tasks and configurations, spanning from software deployments to system configurations.
- **Harmonious guidance**: Ansible playbooks take a similar approach. You declare the desired IT state, and Ansible plays the role of the conductor, ensuring that all the necessary steps are followed, much like specifying, "*I want a flawless musical performance*," and Ansible orchestrates the entire process.
- **Tasks and reusability**: Ansible playbooks are organized into tasks and roles, as with musical sheets and instruments. These tasks can be reused across various playbooks, promoting consistency and saving time.

- **Instrument selection and direction**: Just as a conductor selects which instruments play at which times, playbooks specify which servers or machines (the inventory) should execute tasks. You can direct specific server groups or individual machines.
- **Harmonious execution**: Ansible can skilfully coordinate tasks on multiple machines simultaneously, much as a conductor harmonizes the efforts of different musicians to create a beautiful composition.
- **Fine-tuned performance**: If unexpected challenges arise during the performance, a conductor adjusts and guides the musicians to ensure a flawless outcome. Similarly, Ansible playbooks incorporate error-handling strategies to handle unexpected issues during automation.

Ansible playbooks are a collection of tasks that produce the desired configuration within the managed nodes. They have the following features:

- They help in managing configuration within multiple remote servers using declarative steps
- They use a sequential list of idempotent steps, and steps that match the expected configuration are not applied again
- Tasks within the playbook can be synchronous and asynchronous
- They enable GitOps by allowing the steps to be stored using a simple YAML file to keep in source control, providing CaC

Ansible playbooks consist of multiple **plays**, and each play is mapped to a group of **hosts** using a **role** and consists of a series of **tasks** required to achieve them—something like the following diagram:

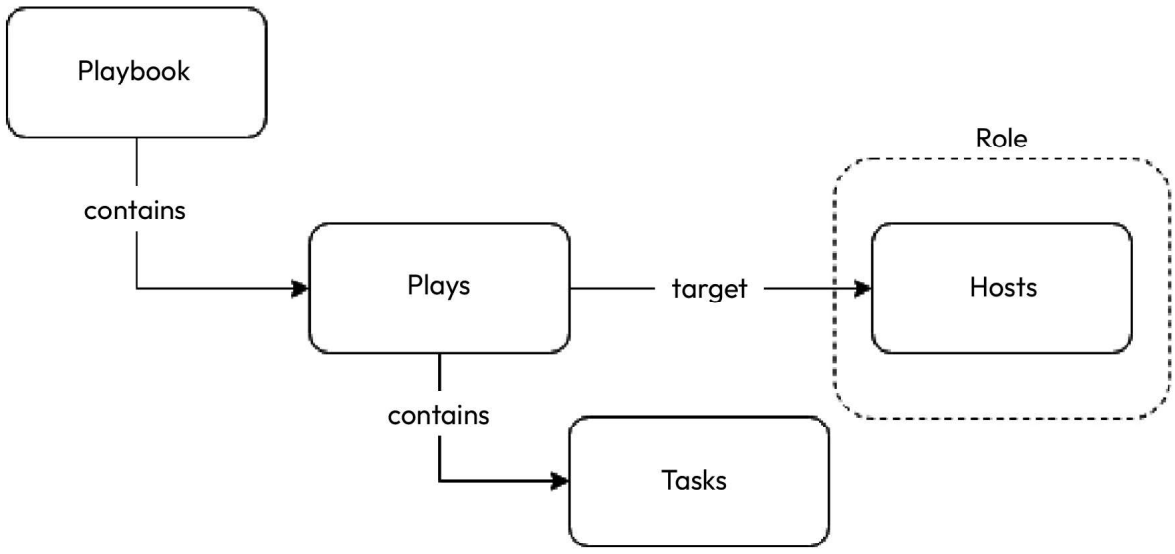

Figure 9.2 – Playbooks

The following `ping.yaml` file is an example of a simple playbook that pings all servers:

```
---
  - hosts: all
    tasks:
      - name: Ping all servers
        action: ping
```

The YAML file contains a list of plays, as the list directive shows. Each play consists of a `hosts` attribute that defines the role to which we want to apply the play. The `tasks` section consists of a list of tasks, each with `name` and `action` attributes. In the preceding example, we have a single play with a single task that pings all servers.

Checking playbook syntax

It is a best practice to check playbook syntax before applying it to your inventory. To check your playbook's syntax, run the following command:

```
$ ansible-playbook ping.yaml --syntax-check
playbook: ping.yaml
```

The syntax is correct, as we get a response with the playbook name. Now, let's go ahead and apply the playbook.

Applying the first playbook

To apply the playbook, run the following command:

```
$ ansible-playbook ping.yaml
PLAY [all] *********************************************
TASK [Gathering Facts] *********************************
ok: [db]
ok: [web]
TASK [Ping all servers] ********************************
ok: [db]
ok: [web]
PLAY RECAP *********************************************
db : ok=2 changed=0 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
web : ok=2 changed=0 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
```

There are three elements of play execution:

- **Gathering facts**: Ansible checks for all hosts that are part of the role, logs in to each instance, and gathers information from each host it uses while executing the tasks from the plays.
- **Run tasks**: Then, it runs the tasks of each play, as defined in the playbook.
- **Play recap**: Ansible then provides a recap of the tasks it executed and the hosts it ran them on. This includes a list of all successful and failed responses.

As we've investigated an elementary example of playbooks, we must understand how to use Ansible playbooks effectively. In the next section, let's look at Ansible playbooks in action with a better example.

Ansible playbooks in action

Let's set up an Apache server for a custom website that connects with a MySQL backend—in short, a **Linux**, **Apache**, **MySQL**, and **PHP** (**LAMP**) stack using Ansible.

The following directory contains all resources for the exercises in this section:

```
$ cd ~/modern-devops/ch9/lamp-stack
```

We have created the following custom `index.php` page that tests the connection to the MySQL database and displays whether it can connect or not:

```
...
<?php
mysqli_connect('db', 'testuser', 'Password@1') or die('Could not connect the database :
Username or password incorrect');
echo 'Database Connected successfully';
?>
...
```

We create several Ansible playbooks according to the logical steps we follow with CM.

It is an excellent practice to update the packages and repositories at the start of every configuration. Therefore, we need to start our playbook with this step.

Updating packages and repositories

As we're using Ubuntu, we can use the `apt` module to update the packages. We must update packages and repositories to ensure the latest package index is available with all our `apt` repositories and avoid any untoward issues while installing packages. The following playbook, `apt-update.yaml`, performs the update:

```
---
- hosts: webservers:dbservers
  become: true
  tasks:
    - name: Update apt packages
      apt: update_cache=yes cache_valid_time=3600
```

The YAML file begins with a list of plays and contains a single play in this case. The `hosts` attribute defines a colon-separated list of `roles/hosts` inventory to apply the playbook. In this case, we've specified `webservers` and `dbservers`. The `become` attribute specifies whether we want to execute the play as a `root` user. So, as we've set `become` to `true`, Ansible will perform all play tasks with `sudo` privileges. The play contains a single task—`Update apt packages`. The task uses the `apt`

module and consists of `update_cache=yes`. It will run an `apt update` operation on all nodes with the `webservers` and `dbservers` roles. The next step is to install packages and services.

Installing application packages and services

We will use the `apt` module to install the packages on Ubuntu, and the `service` module to start and enable the service.

Let's start by installing Apache on the web servers using the following `install-webserver.yaml` playbook:

```
---
- hosts: webservers
  become: true
  tasks:
    - name: Install packages
      apt:
        name:
        - apache2
        - php
        - libapache2-mod-php
        - php-mysql
        update_cache: yes
        cache_valid_time: 3600
        state: present
    - name: Start and Enable Apache service
      service: name=apache2 state=started enabled=yes
```

As this configuration is for `webservers`, we've specified that within the `hosts` attribute. The `tasks` section defines two tasks—`Install packages` and `Start and Enable Apache service`. The `Install packages` task uses the `apt` module to install `apache2`, `php`, `libapache2-mod-php`, and `php-mysql`. The `Start and Enable Apache service` task will start and enable the `apache2` service.

Similarly, we will install and set up the MySQL service using the following `install-dbserver.yaml` playbook:

```
---
- hosts: dbservers
  become: true
  tasks:
  - name: Install packages
    apt:
      name:
      - python-pymysql
      - mysql-server
      update_cache: yes
      cache_valid_time: 3600
      state: present
```

```
  - name: Start and enable MySQL service
    service:
      name: mysql
      state: started
      enabled: true
```

This playbook will run two tasks—`Install packages` and `Start and enable MySQL service`. The `Install packages` task will install the `python-mysql` and `mysql-server` packages using the `apt` module. The `Start and enable MySQL service` task will start and enable the MySQL service.

Configuring applications

The next step in the chain is to configure the applications. There are two playbooks for this. The first will configure Apache on `webservers`, and the second will configure MySQL on `dbservers`.

The following `setup-webservers.yaml` playbook will configure `webservers`:

```
---
- hosts: webservers
  become: true
  tasks:
  - name: Delete index.html file
    file:
      path: /var/www/html/index.html
      state: absent
  - name: Upload application file
    copy:
      src: index.php
      dest: /var/www/html
      mode: 0755
    notify:
      - Restart Apache
  handlers:
  - name: Restart Apache
    service: name=apache2 state=restarted
```

This playbook runs on all nodes with the `webservers` role, and there are three tasks in this playbook. The `Delete index.html file` task uses the `file` module to delete the `/var/www/html/index.html` file from the web server. That is because we are using `index.php` as the index page and not `index.html`. The `Upload application file` task then uses the `copy` module to copy the `index.php` file from the Ansible control node to the web server at the `/var/www/html` destination, with a mode of `0755`. The `Upload application file` task also has a `notify` action that will call the `Restart Apache` handler if this task has a status of `CHANGED`. A `handlers` section within the playbook defines handlers that listen to notify events. In this scenario, if there is a change in the `Upload application file` task, the `Restart Apache` handler will be triggered and will restart the `apache2` service.

We will use the following `setup-dbservers.yaml` playbook to configure MySQL on `dbservers`:

```
---
- hosts: dbservers
  become: true
  vars:
    mysql_root_password: "Password@1"
  tasks:
  - name: Set the root password
    copy:
      src: client.my.cnf
      dest: "/root/.my.cnf"
      mode: 0600
    notify:
      - Restart MySQL
  - name: Create a test user
    mysql_user:
      name: testuser
      password: "Password@1"
      login_user: root
      login_password: "{{ mysql_root_password }}"
      state: present
      priv: '*.*:ALL,GRANT'
      host: '%'
  - name: Remove all anonymous user accounts
    mysql_user:
      name: ''
      host_all: yes
      state: absent
      login_user: root
      login_password: "{{ mysql_root_password }}"
    notify:
    - Restart MySQL
  - name: Remove the MySQL test database
    mysql_db:
      name: test
      state: absent
      login_user: root
      login_password: "{{ mysql_root_password }}"
    notify:
    - Restart MySQL
  - name: Change bind address
    lineinfile:
      path:  /etc/mysql/mysql.conf.d/mysqld.cnf
      regexp: ^bind-address
      line: 'bind-address            = 0.0.0.0'
    notify:
    - Restart MySQL
  handlers:
  - name: Restart MySQL
    service: name=mysql state=restarted
```

This playbook is a bit more complicated, but let's break it down into parts to facilitate our understanding.

There is a `vars` section in this playbook that defines a `mysql_root_password` variable. We need this password while executing MySQL tasks. The first task is to set up the root password. The best way to set that up is by defining a `/root/.my.cnf` file within MySQL that contains the root credentials. We are copying the following `client.my.cnf` file to `/root/.my.cnf` using the `copy` module:

```
[client]
user=root
password=Password@1
```

Then, the `Create a test user` task uses the `mysql_user` module to create a user called `testuser`. It requires values for the `login_user` and `login_password` attributes, and we are supplying `root` and `{{ mysql_root_password }}`, respectively. It then goes ahead and removes all anonymous users and also removes the test database. It then changes the bind address to `0.0.0.0` using the `lineinfile` module. The `lineinfile` module is a powerful module that helps manipulate files by first grepping a file using a regex and then replacing those lines with the line attribute's value. All these tasks notify the `Restart MySQL` handler that restarts the MySQL database service.

Combining playbooks

As we've written multiple playbooks, we need to execute them in order. We cannot configure the services before installing packages and services, and there is no point in running an `apt` update after installing the packages. Therefore, we can create a playbook of playbooks.

To do so, we've created a YAML file, `playbook.yaml`, that has the following content:

```
---
- import_playbook: apt-update.yaml
- import_playbook: install-webserver.yaml
- import_playbook: install-dbserver.yaml
- import_playbook: setup-webservers.yaml
- import_playbook: setup-dbservers.yaml
```

This YAML file contains a list of plays, and every play contains an `import_playbook` statement. The plays are executed in order as specified in the file. Now, let's go ahead and execute the playbook.

Executing playbooks

Executing the playbook is simple. We will use the `ansible-playbook` command followed by the playbook YAML file. As we've combined playbooks in a `playbook.yaml` file, the following command will run the playbook:

```
$ ansible-playbook playbook.yaml
PLAY [webservers:dbservers] *************************
```

```
TASK [Gathering Facts] ******************************
ok: [web]
ok: [db]
TASK [Update apt packages] **************************
ok: [web]
ok: [db]
PLAY [webservers] ***********************************
TASK [Gathering Facts] ******************************
ok: [web]
TASK [Install packages] *****************************
changed: [web]
TASK [Start and Enable Apache service] ***************
ok: [web]
PLAY [dbservers] ************************************
TASK [Gathering Facts] ******************************
ok: [db]
TASK [Install packages] *****************************
changed: [db]
TASK [Start and enable MySQL service] ****************
ok: [db]
PLAY [webservers] ***********************************
TASK [Gathering Facts] ******************************
ok: [web]
TASK [Delete index.html file] ************************
changed: [web]
TASK [Upload application file] ***********************
changed: [web]
RUNNING HANDLER [Restart Apache] ********************
changed: [web]
PLAY [dbservers] ************************************
TASK [Gathering Facts] ******************************
ok: [db]
TASK [Set the root password] *************************
changed: [db]
TASK [Update the cnf file] ***************************
changed: [db]
TASK [Create a test user] ****************************
changed: [db]
TASK [Remove all anonymous user accounts] ************
ok: [db]
TASK [Remove the MySQL test database] ***************
ok: [db]
TASK [Change bind address] ***************************
changed: [db]
RUNNING HANDLER [Restart MySQL] **********************
changed: [db]
PLAY RECAP ******************************************
db: ok=13   changed=6    unreachable=0    failed=0
skipped=0    rescued=0    ignored=0
web: ok=9    changed=4    unreachable=0    failed=0
skipped=0    rescued=0    ignored=0
```

As we can see, the configuration is applied on both `webservers` and `dbservers`, so let's run a `curl` command to the web server to see what we get:

```
$ curl web
<html>
<head>
<title>PHP to MQSQL</title>
</head>
<body>Database Connected successfully</body>
</html>
```

As we can see, the database is connected successfully! That proves that the setup was successful.

There are several reasons why the way we approached the problem was not the best. First, there are several sections within the playbook where we've hardcoded values. While we have used variables in a few playbooks, we've also assigned values to variables within them. That does not make the playbooks a candidate for reuse. The best way to design software is to keep reusability in mind. Therefore, there are many ways in which we can redesign the playbooks to foster reusability.

Designing for reusability

Ansible provides variables for turning Ansible playbooks into reusable templates. You can substitute variables in the right places using **Jinja2** markup, which we've already used in the last playbook. Let's now look at Ansible variables, their types, and how to use them.

Ansible variables

Ansible variables, as with any other variables, are used to manage differences between managed nodes. You can use a similar playbook for multiple servers, but sometimes, there are some differences in configuration. Ansible variables help you template your playbooks so that you can reuse them for a variety of similar systems. There are multiple places where you can define your variables:

- Within the Ansible playbook within the `vars` section
- In your inventory
- In reusable files or roles
- Passing variables through the command line
- Registering variables by assigning the return values of a task

Ansible variable names can include *letters*, *numbers*, and *underscores*. You cannot have a Python *keyword* as a variable, as Ansible uses Python in the background. Also, a variable name cannot begin with a number but can start with an underscore.

You can define variables using a simple key-value pair within the YAML files and following the standard YAML syntax.

Variables can broadly be of three types—*simple variables*, *list variables*, and *dictionary variables*.

Simple variables

Simple variables are variables that hold a single value. They can have *string*, *integer*, *double*, or *boolean* values. To refer to simple Ansible variables within the playbook, use them within Jinja expressions, such as {{ `var_name` }}. You should always quote Jinja expressions, as the YAML files will fail to parse without that.

The following is an example of a simple variable declaration:

```
mysql_root_password: bar
```

And this is how you should reference it:

```
  - name: Remove the MySQL test database
    mysql_db:
      name: test
      state: absent
      login_user: root
      login_password: "{{ mysql_root_password }}"
```

Now, let's look at list variables.

List variables

List variables hold a list of values you can reference using an index. You can also use list variables within loops. To define a list variable, you can use the standard YAML syntax for a list, as in the following example:

```
region:
  - europe-west1
  - europe-west2
  - europe-west3
```

To access the variable, we can use the index format, as in this example:

```
region: " {{ region[0] }} "
```

Ansible also supports more complex dictionary variables. Let's have a look.

Dictionary variables

Dictionary variables hold a complex combination of *key-value pairs*, the same as a Python dictionary. You can define dictionary variables using the standard YAML syntax, as in the following example:

```
foo:
  bar: one
  baz: two
```

There are two ways in which to refer to these variables' values. For example, in dot notation, we can write the following:

```
bar: {{ foo.bar }}
```

And in bracket notation, we can depict the same thing using the following expression:

```
bar: {{ foo[bar] }}
```

We can use either dot or bracket notation in the same way as in Python.

> **Tip**
> While both dot and bracket notation signify the same thing, bracket notification is better. With dot notation, some keys can collide with the methods and attributes of Python dictionaries.

Now, let's look at ways of sourcing variable values.

Sourcing variable values

While you can manually define variables and provide their values, sometimes we need dynamically generated values; for example, if we need to know the server's hostname where Ansible is executing the playbook or want to use a specific value returned from a task within a variable. Ansible provides a list of variables and system metadata during the gathering facts phase for the former requirement. That helps determine which variables are available and how to use them. Let's understand how we can gather that information.

Finding metadata using Ansible facts

Ansible facts are metadata information associated with the managed nodes. Ansible gets the facts during the *gathering facts* stage, and we can use the `facts` variable directly within the playbook. We can use the `setup` module as an Ansible task to determine the facts. For example, you can run the following command to get the Ansible facts for all nodes with the `webservers` role:

```
$ ansible -m setup webservers
web | SUCCESS => {
    "ansible_facts": {
        "ansible_all_ipv4_addresses": [
```

```
            "10.0.2.5"
        ],
        ...
        "ansible_hostname": "web",
```

So, as we can see, we get `ansible_facts` with multiple variables associated with the inventory item. As we have a single server here, we get `web` server details. Within the piece, we have an `ansible_hostname` attribute called `web`. We can use that `ansible_hostname` attribute within our playbook if we need to.

Sometimes, we want to source a task's output to a particular variable to use the variable in any subsequent tasks of the playbook. Let's look at how we can do that.

Registering variables

If a task within your playbook, for example, needs a value from the result of a preceding task, we can use the `register` attribute.

The following directory contains all the resources for exercises in this section:

```
$ cd ~/modern-devops/ch9/vars-exercise
```

Let's look at the following example `register.yaml` file:

```
- hosts: webservers
  tasks:
    - name: Get free space
      command: free -m
      register: free_space
      ignore_errors: true
    - name: Print the free space from the previous task
      debug:
        msg: "{{ free_space }}"
```

The playbook contains two tasks. The first task uses the `command` module to execute a command, `free -m`, and registers the result in the `free_space` variable. The subsequent task uses the previous task's output using the `debug` module to print `free_space` as a message to the console.

Let's run the playbook to see for ourselves:

```
$ ansible-playbook register.yaml
PLAY [webservers] ***********************************
TASK [Gathering Facts] ******************************
ok: [web]
TASK [Get free space] *******************************
changed: [web]
TASK [Print the free space from the previous task] ***
ok: [web] => {
    "msg": {
```

```
        "stdout": "              total        used
free      shared  buff/cache   available\nMem:           3.3G
170M        2.6G        2.2M        642M        3.0G\nSwap:
 0B          0B          0B",
    }
PLAY RECAP *************************************
web: ok=3    changed=1    unreachable=0    failed=0    skipped=0
    rescued=0    ignored=0
```

Now that we've understood variables, let's look at other aspects that will help us improve the last playbook.

Jinja2 templates

Ansible allows for templating files using dynamic Jinja2 templates. You can use the Python syntax within the file, starting with { { and ending with } }. That will allow you to substitute variables during runtime and run complex computations on variables.

To understand this further, let's modify the `index.php` file to supply the MySQL username and password dynamically during execution:

```
...
<?php
mysqli_connect('db', '{{ mysql_user }}', '{{ mysql_password }}')
or die('Could not connect the database : Username or password
incorrect');
echo 'Database Connected successfully';
?>
...
```

As we can see, instead of hardcoding the username and password, we can use templates to substitute the variable values during runtime. That will make the file more reusable and will fit multiple environments. Ansible provides another important aspect of coding reusability within your playbooks—Ansible **roles**. Let's have a look at this in the next section.

Ansible roles

Well, the last playbook looks a bit cluttered. You have a lot of files within it, and none of them are reusable. The code we've written can only set up the configuration in a particular way. This may work fine for smaller teams with limited configurations to manage, but it is not as simple as it looks for most enterprises.

Ansible roles help to standardize an Ansible setup and promote reusability. With roles, you can automatically load **var files**, **handlers**, **tasks**, and other Ansible artifacts using a standard directory structure relative to your playbooks. The directory structure is as follows:

```
<playbook>.yaml
roles/
    <role>/
        tasks/
```