```
        handlers/
        library/
        files/
        templates/
        vars/
        defaults/
        meta/
```

The `roles` directory contains multiple subdirectories for each role. Each role directory contains multiple standard directories:

- `tasks`: This directory contains a list of tasks' YAML files. It should contain a file called `main.yaml` (or `main.yml` or `main`), containing an entire list of tasks or importing tasks from other files within the directory.
- `handlers`: This directory contains a list of handlers associated with the role within a file called `main.yaml`.
- `library`: This directory contains Python modules that can be used with the role.
- `files`: This directory contains all files that we require for our configuration.
- `templates`: This directory contains the Jinja2 templates that the role deploys.
- `vars`: This directory contains a `main.yaml` file with a list of variables associated with the role.
- `defaults`: This directory contains a `main.yaml` file containing the default variables associated with the role that can be easily overridden by any other variable that includes inventory variables.
- `meta`: This directory contains the metadata and dependencies associated with the role within a `main.yaml` file.

Some best practices revolve around managing your Ansible configuration through the folder structure. Let's look at some of these next.

> **Tip**
> While choosing between the `vars` and `defaults` directories, the rule of thumb is to put variables that will not change within the `vars` directory. Put variables that are likely to change within the `defaults` directory.

So, we'll go and use the `defaults` directory as much as we can. There are some best practices regarding roles that we should follow as well. Let's look at some of them.

> **Tip**
> Think about the full life cycle of a specific service while designing roles rather than building the entire stack—in other words, instead of using `lamp` as a role, use `apache` and `mysql` roles instead.

We will create three roles for our use—`common`, `apache`, and `mysql`.

> **Tip**
> Use specific roles, such as `apache` or `mysql`, instead of using `webserver` or `dbserver`. Typical enterprises have a mix and match of multiple web servers and database technologies. Therefore, giving a generic name to a role will confuse things.

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

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

The following is the directory structure we will follow for our scenario:

```
├── ansible.cfg
├── hosts
├── output.log
├── playbook.yaml
```

There are three roles that we'll create—`apache`, `mysql`, and `common`. Let's look at the directory structure of the `apache` role first:

```
└── roles
    ├── apache
    │   ├── defaults
    │   │   └── main.yaml
    │   ├── handlers
    │   │   └── main.yaml
    │   ├── tasks
    │   │   ├── install-apache.yaml
    │   │   ├── main.yaml
    │   │   └── setup-apache.yaml
    │   └── templates
    │       └── index.php.j2
```

There is also a `common` role that will apply to all scenarios. The following directory structure defines that:

```
    ├── common
    │   └── tasks
    │       └── main.yaml
```

Finally, let's define the `mysql` role through the following directory structure:

```
    └── mysql
        ├── defaults
        │   └── main.yaml
        ├── files
        ├── handlers
        │   └── main.yaml
        ├── tasks
```

```
│   ├── install-mysql.yaml
│   ├── main.yaml
│   └── setup-mysql.yaml
└── templates
    └── client.my.cnf.j2
```

The `apache` directory consists of the following:

- We've used the same `index.php` file we created in the last exercise, converted it to a Jinja2 template called `index.php.j2`, and copied it to `roles/apache/templates`.
- The `handlers` directory contains a `main.yaml` file that contains the `Restart Apache` handler.
- The `tasks` directory contains an `install-apache.yaml` file that includes all tasks required to install Apache. The `setup-apache.yaml` file consists of a list of tasks that will set up Apache, similar to what we did in the previous exercise. The `main.yaml` file contains tasks from both files, using `include` directives such as the following:

```
---
- include: install-apache.yaml
- include: setup-apache.yaml
```

- The `defaults` directory contains the `main.yaml` file, which contains the `mysql_username` and `mysql_password` variables and their default values.

> **Tip**
>
> Use as few variables as possible and try to default them. Use defaults for variables in such a way that minimal custom configuration is needed.

The `mysql` directory consists of the following:

- We've modified `client.my.cnf` and converted that to a `j2` file. The `j2` file is a Jinja2 template file we will use in the role through the `template` module in the `Set the root password` task. The file exists within the `templates` directory:

```
[client]
user=root
password={{ mysql_root_password }}
```

  As we can see, we're providing the password through a Jinja2 expression. When we run the `mysql` role through the playbook, the value of `mysql_root_password` will be substituted in the `password` section.

- The `handlers` directory contains the `Restart MySQL` handler.

- The `tasks` directory consists of three files. The `install-mysql.yaml` file contains tasks that install `mysql`, and the `setup-mysql.yaml` file contains tasks that set up `mysql`. The `main.yaml` file combines both these files using `include` task directives, as follows:

```
---
- include: install-mysql.yaml
- include: setup-mysql.yaml
```

- The `defaults` directory contains a `main.yaml` file with a list of variables we will use within the role. In this case, it just contains the value of `mysql_root_password`.

The `common` directory contains a single directory called `tasks` that includes a `main.yaml` file with a single task to run an `apt update` operation.

The main directory contains `ansible.cfg`, `hosts`, and `playbook.yaml` files. While the `hosts` and `ansible.cfg` files are the same as the last exercise, the `playbook.yaml` file looks like the following:

```
---
- hosts: webservers
  become: true
  roles:
    - common
    - apache
- hosts: dbservers
  become: true
  roles:
    - common
    - mysql
```

The playbook is now a concise one with a lot of reusable elements. It consists of two plays. The first play will run on all web servers with the `root` user and apply `common` and `apache` roles to them. The second play will run on all nodes with the `dbservers` role with the `root` user and use `common` and `mysql` roles.

> **Tip**
>
> Always keep roles loosely coupled. In the preceding example, the `apache` role has no dependency on `mysql` and vice versa. This will allow us to reuse configuration with ease.

Now, let's go ahead and execute the playbook:

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

```
db: ok=10 changed=0 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
web:  ok=7 changed=0 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
```

And, as we can see, there are no changes to the configuration. We've applied the same configuration but in a better way. If we want to share our configuration with people within the team, we can share the `roles` directory, and they can apply the role within their playbook.

There may be instances where we want to use a different value for the variable defined in the `roles` section. You can override variables within the playbook by supplying the variable values with the `extra-vars` flag, as follows:

```
$ ansible-playbook playbook.yaml --extra-vars "mysql_user=foo mysql_password=bar@123"
```

When we apply the playbook using the preceding command, we'll see that the user now changes to `foo` and that the password changes to `bar@123` in both the Apache and MySQL configurations:

```
...
PLAY RECAP
db: ok=9 changed=1 unreachable=0 failed= skipped=0 rescued=0 ignored=0
web: ok=7 changed=2 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
```

So, if we run the `curl` command to the web host, we will get the same response as before:

```
...
<body>Database Connected successfully</body>
...
```

Our setup is working correctly with roles. We've set up the Ansible playbook by following all the best practices and using reusable roles and templates. That is the way to go forward in designing powerful Ansible playbooks.

Summary

In this chapter, we've discussed Ansible and its core functionalities from a hands-on perspective. We began by understanding CaC, looked at Ansible and Ansible architecture, installed Ansible, understood Ansible modules, tasks, and playbooks, and then applied our first Ansible configuration. We then looked at fostering reusability with Ansible variables, Jinja2 templates, and roles and reorganized our configuration with reusability in mind. We also looked at several best practices along the way.

In the next chapter, we will combine Terraform with Ansible to spin up something useful and look at HashiCorp's Packer to create immutable infrastructure.

Questions

1. It is a best practice to avoid using `command` and `shell` modules as much as possible. (True/False)
2. Aliases help in keeping your inventory generic. (True/False)
3. What does the `ansible-playbook` command do?

   A. It runs an ad hoc task on the inventory.

   B. It runs a series of tasks on the inventory.

   C. It applies the plays and tasks configured with the playbook.

   D. It destroys the configuration from managed nodes.
4. Which of the following techniques helps in building reusability within your Ansible configuration? (Choose three)

   A. Use variables.

   B. Use Jinja2 templates.

   C. Use roles.

   D. Use tasks.
5. While naming roles, what should we consider? (Choose two)

   A. Name roles as precisely as possible.

   B. While thinking of roles, think of the service instead of the full stack.

   C. Use generic names for roles.
6. In which directory should you define variables within roles if the variable's value is likely to change?

   A. `defaults`

   B. `vars`
7. Handlers are triggered when the output of the task associated with the handler is …?

   A. `SUCCESS`

   B. `CHANGED`

   C. `FAILED`
8. Does a `SUCCESS` status denote that the task did not detect any changed configuration? (True/False)
9. What are the best practices for inventory management? (Choose three)

   A. Use a separate inventory for each environment.

   B. Group the inventory by functions.

C. Use aliases.

D. Keep the inventory file in a central location.

Answers

1. True
2. True
3. C
4. A, B, C
5. A, B
6. A
7. B
8. True
9. A, B, and C

10
Immutable Infrastructure with Packer

In the previous chapter, we looked at configuration management with Ansible and the tool's core concepts. We also discussed Terraform and IaC in *Chapter 8*, *Infrastructure as Code (IaC) with Terraform*. In this chapter, we will look at another way of provisioning your infrastructure and configuration using both tools, as well as another one, called **Packer**. With all three tools, let's boot up a scalable **Linux**, **Apache**, **MySQL**, **and PHP** (**LAMP**) stack on Azure.

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

- Immutable infrastructure with HashiCorp's Packer
- Creating the Apache and MySQL playbook
- Building the Apache and MySQL images using Packer and Ansible provisioners
- Creating the required infrastructure with Terraform

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; 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 `ch10` directory to access the required resources:

```
$ git clone https://github.com/PacktPublishing/Modern-DevOps-Practices-2e.git \
   modern-devops
$ cd modern-devops/ch10
```

You also need to install **Terraform** and **Ansible** on your system. Refer to *Chapter 8*, *Infrastructure as Code (IaC) with Terraform*, and *Chapter 9*, *Configuration Management with Ansible*, for more details on installing and setting up Terraform and Ansible.

Immutable infrastructure with HashiCorp's Packer

Imagine you are the author of a book and you need to make changes to an existing edition. When you want to make changes, such as improving the content or fixing the issues and ensuring the book is up to date, you don't edit the existing book. Instead, you create a new edition with the desired updates while keeping the existing editions intact, like the new edition of this book. This concept aligns with **immutable infrastructure**.

In IT and systems management, immutable infrastructure is a strategy where, instead of making changes to existing servers or **Virtual Machines** (**VMs**), you generate entirely new instances with the desired configuration. These new instances replace the old ones instead of modifying them, like creating a new book edition when you want to incorporate changes.

Here's how it works:

- **Building from scratch**: When you need to update a part of your infrastructure, you avoid making direct changes to the existing servers or machines. Instead, you create new ones from a pre-established template (an image) that includes the updated configuration.
- **No in-place modifications**: Like not editing an existing book, you avoid making in-place modifications to current servers. This practice reduces the risk of unforeseen changes or configuration inconsistencies.
- **Consistency**: Immutable infrastructure ensures that every server or instance is identical because they all originate from the same template. This uniformity is valuable for ensuring reliability and predictability.
- **Rolling updates**: When it's time to implement an update, you systematically replace the old instances with the new ones in a controlled manner. This minimizes downtime and potential risks.
- **Scalability**: Scaling your infrastructure becomes effortless by generating new instances as needed. This is akin to publishing new book editions when there's a surge in demand, or things become outdated.
- **Rollback and recovery**: If issues arise from an update, you can swiftly revert to the previous version by re-creating instances from a known good template.

So, consider immutable infrastructure as a means of maintaining your infrastructure by creating new, improved instances rather than attempting to revise or modify existing ones. This approach elevates consistency, reliability, and predictability within your IT environment.

To understand this further, let's consider the traditional method of setting up applications via Terraform and Ansible. We would use Terraform to spin up the infrastructure and then use Ansible on top to apply the relevant configuration to the infrastructure. That is what we did in the last chapter. While that is a viable approach, and many enterprises use it, there is a better way to do it with modern DevOps approaches and immutable infrastructure.

Immutable infrastructure is a ground-breaking concept that emerged due to the problems with **mutable infrastructure**. In a mutable infrastructure approach, we generally update servers in place. So, we follow a mutable process when we install Apache in a VM using Ansible and customize it further. We may want to update the servers, patch them, update our Apache to a newer version, and update our application code from time to time.

The issue with this approach is that while we can manage it well with Ansible (or related tools, such as **Puppet**, **Chef**, and **SaltStack**), the problem always remains that we are making live changes in a production environment that might go wrong for various reasons. Worse, it might update something we did not anticipate or test in the first place. We also might end up in a partial upgrade state that might be difficult to roll back.

With the scalable infrastructure that the cloud provides, you can have a dynamic horizontal scaling model where VMs scale with traffic. Therefore, you can have the best possible utilization of your infrastructure – the best bang for your buck! The problem with the traditional approach is that even if we use Ansible to apply the configuration to new machines, it is slower to get ready. Therefore, the scaling is not optimal, especially for bursty traffic.

Immutable infrastructure helps you manage these problems by taking the same approach we took for containers – *baking configuration directly into the OS image using modern DevOps tools and practices*. Immutable infrastructure helps you deploy the tested configuration to production by replacing the existing VM without doing any updates in place. It is faster to start and easy to roll back. You can also version infrastructure changes with this approach.

HashiCorp has an excellent suite of DevOps products related to infrastructure and configuration management. HashiCorp provides **Packer** to help you create immutable infrastructure by baking configurations directly in your VM image, rather than the slow process of creating a VM with a generic OS image and then customizing it later. It works on a similar principle as Docker uses to bake container images; that is, you define a template (configuration file) that specifies the source image, the desired configuration, and any provisioning steps needed to set up the software on the image. Packer then builds the image by creating a temporary instance with the base image, applying the defined configuration, and capturing the machine image for reuse.

Packer provides some of the following key features:

- **Multi-platform support**: Packer works on the plugin architecture and, therefore, can be used to create VM images for a lot of different cloud and on-premises platforms, such as VMware, Oracle VirtualBox, Amazon EC2, Azure's ARM, Google Cloud Compute, and container images for Docker or other container runtimes.
- **Automation**: Packer automates image creation and eliminates manual effort to build images. It also helps you with your multi-cloud strategy, as you can use a single configuration to build images for various platforms.
- **Fosters GitOps**: Packer configurations are machine-readable and written in HCL or JSON, so they can easily sit with your code. This, therefore, fosters GitOps.
- **Integration with other tools**: Packer integrates well with other HashiCorp tools, such as Terraform and Vagrant.

Packer uses a staging VM to customize the image. The following is the process that Packer follows while building the custom image:

1. You start with Packer configuration HCL files to define the base image you want to start from and where to build the image. You also define the provisioner for building the custom image, such as Ansible, and specify what playbooks to use.
2. When you run a Packer build, Packer uses the details in the configuration files to create a build VM from the base image, run the provisioner to customize it, turn off the build VM, take a snapshot, and save that as a disk image. It finally saves the image in an image repository.
3. You can then build the VM from the custom image using Terraform or other tools.

The following figure explains the process in detail:

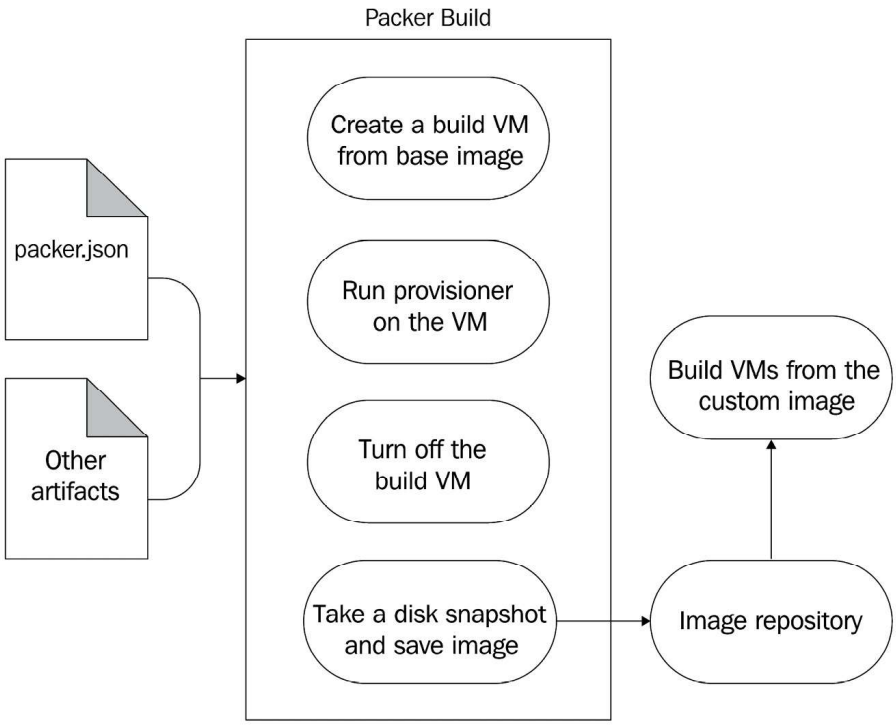

Figure 10.1 – Packer build process

The result is that your application is quick to start up and scales very well. For any changes within your configuration, create a new disk image with Packer and Ansible and then use Terraform to apply the changes to your resources. Terraform will then spin down the old VMs and spin up new ones with the new configuration. If you can relate it to the container deployment workflow, you can make real sense of it. It's akin to using the container workflow within the VM world! But is immutable infrastructure for everyone? Let's understand where it fits best.

When to use immutable infrastructure

Deciding to switch to immutable infrastructure is difficult, especially when your Ops team treats servers as pets. Most people get paranoid about the idea of deleting an existing server and creating a new one for every update. Well, you need to do a lot of convincing when you first come up with the idea. However, it does not mean that you must use immutable infrastructure to do proper DevOps. It all depends on your use case.

Let's look at each approach's pros and cons to understand them better.

Pros of mutable infrastructure

Let's begin with the pros of mutable infrastructure:

- If adequately managed, mutable infrastructure is faster to upgrade and change. It makes security patches quicker.
- It is simpler to manage, as we don't have to worry about building the entire VM image and redeploying it for every update.

Cons of mutable infrastructure

Next, let's see the cons of mutable infrastructure:

- It eventually results in configuration drift. When people start making changes manually in the server and do not use a config management tool, it becomes difficult to know what's in the server after a particular point. Then, you will have to start relying on snapshots.
- Versioning is impossible with mutable infrastructure, and rolling back changes is troublesome.
- There is a possibility of partial updates because of technical issues such as a patchy network, unresponsive **apt** repositories, and so on.
- There is a risk because changes are applied directly to the production environment. There is also a chance that you will end up in an unanticipated state that is difficult to troubleshoot.
- Because of configuration drift, it is impossible to say that the current configuration is the same as being tracked in version control. Therefore, building a new server from scratch may require manual intervention and comprehensive testing.

Similarly, let's look at the pros and cons of immutable infrastructure.

Pros of immutable infrastructure

The pros of immutable infrastructure are as follows:

- It eliminates configuration drift as the infrastructure cannot change once deployed, and any changes should come via the CI/CD process.
- It is DevOps-friendly as every build and deployment process inherently follows modern DevOps practices.
- It makes discrete versioning possible as every image generated from an image build can be versioned and kept within an image repository. That makes rollouts and rollbacks much more straightforward and promotes modern DevOps practices such as **canary** and **blue-green** deployments with A/B testing.
- The image is pre-built and tested, so we always get a predictable state from immutable infrastructure. We, therefore, reduce a lot of risk from production implementations.

- It helps with horizontal scaling on the cloud because you can now create servers from pre-built images, making new VMs faster to start up and get ready.

Cons of immutable infrastructure

The cons of immutable infrastructure are as follows:

- Building and deploying immutable infrastructure is a bit complex, and it is slow to add updates and manage urgent hotfixes
- There are storage and network overheads in generating and managing VM images

So, as we've looked at the pros and cons of both approaches, it ultimately depends on how you currently do infrastructure management and your end goal. Immutable infrastructure has a huge benefit, and therefore, it is something that every modern DevOps engineer should understand and implement if possible. However, technical and process constraints prevent people from doing it – while some constraints are related to the technology stack, most are simply related to processes and red tape. Immutable infrastructure is best when you need consistently reproducible and exceptionally reliable deployments. This approach minimizes the risk of configuration drift and streamlines updates by reconstructing entire environments instead of tweaking existing elements. It proves especially advantageous in scenarios such as microservices architectures, container orchestration, and situations where rapid scaling and the ability to roll back changes are paramount.

We all know that DevOps is not all about tools but it is a cultural change that should originate from the very top. If it is not possible to use immutable infrastructure, you can always use a **config management** tool such as Ansible on top of live servers. That makes things manageable to a certain extent.

Now, moving on to Packer, let's look at how to install it.

Installing Packer

You can install Packer on a variety of platforms in a variety of ways. Please refer to `https://developer.hashicorp.com/packer/downloads`. As Packer is available as an **apt** package, use the following commands to install Packer on Ubuntu Linux:

```
$ wget -O- https://apt.releases.hashicorp.com/gpg | sudo \
gpg --dearmor -o /usr/share/keyrings/hashicorp-archive-keyring.gpg
$ echo "deb [signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg] \
https://apt.releases.hashicorp.com $(lsb_release -cs) main" | \
sudo tee /etc/apt/sources.list.d/hashicorp.list
$ sudo apt update && sudo apt install -y packer
```

To verify the installation, run the following command:

```
$ packer --version
1.9.2
```

As we see, Packer is installed successfully. We can proceed with the next activity in our goal – *creating playbooks*.

Creating the Apache and MySQL playbooks

As our goal is to spin up a scalable **LAMP stack** in this chapter, we must start by defining Ansible playbooks that would run on the build VM. We've already created some roles for Apache and MySQL in *Chapter 9*, *Configuration Management with Ansible*. We will use the same roles within this setup as well.

Therefore, we will have the following directory structure within the `ch10` directory:

```
├── ansible
│   ├── dbserver-playbook.yaml
│   ├── roles
│   │   ├── apache
│   │   ├── common
│   │   └── mysql
│   └── webserver-playbook.yaml
├── packer
│   ├── dbserver.pkr.hcl
│   ├── plugins.pkr.hcl
│   ├── variables.pkr.hcl
│   ├── variables.pkrvars.hcl
│   └── webserver.pkr.hcl
└── terraform
    ├── main.tf
    ├── outputs.tf
    ├── terraform.tfvars
    └── vars.tf
```

We have two playbooks within the `ansible` directory – `webserver-playbook.yaml` and `dbserver-playbook.yaml`. Let's look at each to understand how we write our playbooks for Ansible.

`webserver-playbook.yaml` looks like the following:

```
---
- hosts: default
  become: true
  roles:
    - common
    - apache
```

`dbserver-playbook.yaml` looks like the following:

```
---
- hosts: default
  become: true
  roles:
```

```
    - common
    - mysql
```

As we can see, both playbooks have `hosts` set to `default`. That is because we will not define the inventory for this playbook. Instead, Packer will use the build VM to build the image and dynamically generate the inventory.

> **Note**
>
> Packer will also ignore any `remote_user` attributes within the task and use the user present in the Ansible provisioner's config.

As we've already tested this configuration in the previous chapter, all we need to do now is define the Packer configuration, so let's go ahead and do that in the next section.

Building the Apache and MySQL images using Packer and Ansible provisioners

We will now use Packer to create the Apache and MySQL images. Before defining the Packer configuration, we have a few prerequisites to allow Packer to build custom images.

Prerequisites

We must create an **Azure service principal** for Packer to interact with Azure and build the image.

First, log in to your Azure account using the Azure CLI with the following command:

```
$ az login
```

Now, set the subscription to the subscription ID we got in response to the `az login` command to an environment variable using the following:

```
$ export SUBSCRIPTION_ID=<SUBSCRIPTION_ID>
```

Next, let's set the subscription ID using the following command:

```
$ az account set --subscription="${SUBSCRIPTION_ID}"
```

Then, create the service principal with contributor access using the following command:

```
$ az ad sp create-for-rbac --role="Contributor" \
--scopes="/subscriptions/${SUBSCRIPTION_ID}"
{"appId": "00000000-0000-0000-0000-00000", "name": "http://azure-
cli-2021-01-07-05-59-24",  "password": "xxxxxxxxxxxxxxxxxxxxxxxx", "tenant": "00000000-
0000-0000-0000-0000000000000"}
```

We've successfully created the service principal. The response JSON consists of `appId`, `password`, and `tenant` values that we will use in the subsequent sections.

> **Note**
>
> You can also reuse the service principal we created in *Chapter 8*, *Infrastructure as Code (IaC) with Terraform*, instead.

Now, let's go ahead and set the values of these variables in the `packer/variables.pkrvars.hcl` file with the details:

```
client_id = "<VALUE_OF_APP_ID>"
client_secret = "<VALUE_OF_PASSWORD>"
tenant_id = "<VALUE_OF_TENANT>"
subscription_id = "<SUBSCRIPTION_ID>"
```

We will use the variable file in our Packer build. We also need a resource group for storing the built images.

To create the resource group, run the following command:

```
$ az group create -n packer-rg -l eastus
```

Now, let's go ahead and define the Packer configuration.

Defining the Packer configuration

Packer allows us to define configuration in JSON as well as HCL files. As JSON is now deprecated and HCL is preferred, let's define the Packer configuration using HCL.

To access resources for this section, switch to the following directory:

```
$ cd ~/modern-devops/ch10/packer
```

We will create the following files in the `packer` directory:

- `variables.pkr.hcl`: Contains a list of variables we would use while applying the configuration
- `plugins.pkr.hcl`: Contains the Packer plugin configuration
- `webserver.pkr.hcl`: Contains the Packer configuration for building the web server image
- `dbserver.pkr.hcl`: Contains the Packer configuration for building the `dbserver` image
- `variables.pkrvars.hcl`: Contains the values of the Packer variables defined in the `variables.pkr.hcl` file

The `variables.pkr.hcl` file contains the following:

```
variable "client_id" {
  type    = string
}
variable "client_secret" {
  type    = string
}
variable "subscription_id" {
  type    = string
}
variable "tenant_id" {
  type    = string
}
```

The `variables.pkr.hcl` file defines a list of user variables that we can use within the `source` and `build` blocks of the Packer configuration. We've defined four string variables – `client_id`, `client_secret`, `tenant_id`, and `subscription_id`. We can pass the values of these variables by using the `variables.pkrvars.hcl` variable file we defined in the last section.

> **Tip**
>
> Always provide sensitive data from external variables, such as a variable file, environment variables, or a secret manager, such as HashiCorp's Vault. You should never commit sensitive information with code.

The `plugins.pkr.hcl` file contains the following block:

`packer`: This section defines the common configuration for Packer. In this case, we've defined the plugins required to build the image. There are two plugins defined here – `ansible` and `azure`. Plugins contain a `source` and `version` attribute. They contain everything you would need to interact with the technology component:

```
packer {
  required_plugins {
    ansible = {
      source  = "github.com/hashicorp/ansible"
      version = "=1.1.0"
    }
    azure = {
      source  = "github.com/hashicorp/azure"
      version = "=1.4.5"
    }
  }
}
```

The `webserver.pkr.hcl` file contains the following sections:

- `source`: The `source` block contains the configuration we would use to build the VM. As we build an `azure-arm` image, we define the source as follows:

```
source "azure-arm" "webserver" {
  client_id                         = var.client_id
  client_secret                     = var.client_secret
  image_offer                       = "UbuntuServer"
  image_publisher                   = "Canonical"
  image_sku                         = "18.04-LTS"
  location                          = "East US"
  managed_image_name                = "apache-webserver"
  managed_image_resource_group_name = "packer-rg"
  os_type                           = "Linux"
  subscription_id                   = var.subscription_id
  tenant_id                         = var.tenant_id
  vm_size                           = "Standard_DS2_v2"
}
```

  Different types of sources have different attributes that help us connect and authenticate with the cloud provider that the source is associated with. Other attributes define the build VM's specification and the base image that the build VM will use. It also describes the properties of the custom image we're trying to create. Since we're using Azure in this case, its source type is `azure-arm` and consists of `client_id`, `client_secret`, `tenant_id`, and `subscription_id`, which helps Packer authenticate with the Azure API server. These attributes' values are sourced from the `variables.pkr.hcl` file.

> **Tip**
>
> The managed image name can also contain a version. That will help you build a new image for every new version you want to deploy.

- `build`: The `build` block consists of `sources` and `provisioner` attributes. It contains all the sources we want to use, and the `provisioner` attribute allows us to configure the build VM to achieve the desired configuration. We've defined the following `build` block:

```
build {
  sources = ["source.azure-arm.webserver"]
  provisioner "ansible" {
    playbook_file = "../ansible/webserver-playbook.yaml"
  }
}
```

We've defined an **Ansible provisioner** to customize our VM. There are a lot of provisioners that Packer provides. Luckily, Packer provides the Ansible provisioner out of the box. The Ansible provisioner requires the path to the playbook file; therefore, in this case, we've provided `../ansible/webserver-playbook.yaml`.

> **Tip**
>
> You can specify multiple sources in the `build` block, each with the same or different types. Similarly, we can have numerous provisioners, each executed in parallel. So, if you want to build the same configuration for multiple cloud providers, you can specify multiple sources for each cloud provider.

Similarly, we've defined the following `dbserver.pkr.hcl` file:

```
source "azure-arm" "dbserver" {
  ...
  managed_image_name                = "mysql-dbserver"
  ...
}
build {
  sources = ["source.azure-arm.dbserver"]
  provisioner "ansible" {
    playbook_file = "../ansible/dbserver-playbook.yaml"
  }
}
```

The `source` block has the same configuration as the web server apart from `managed_image_name`. The `build` block is also like the web server, but instead, it uses the `../ansible/dbserver-playbook.yaml` playbook.

Now, let's look at the Packer workflow and how to use it to build the image.

The Packer workflow for building images

The Packer workflow comprises two steps – `init` and `build`.

As we already know, Packer uses plugins to interact with the cloud providers; therefore, we need to install them. To do so, Packer provides the `init` command.

Let's initialize and install the required plugins using the following command:

```
$ packer init .
Installed plugin github.com/hashicorp/ansible v1.1.0 in "~/.config/packer/plugins/github.
com/hashicorp/ansible/packer-plugin-ansible_v1.1.0_x5.0_linux_amd64"
Installed plugin github.com/hashicorp/azure v1.4.5 in "~/.config/packer/plugins/github.
com/hashicorp/azure/packer-plugin-azure_v1.4.5_x5.0_linux_amd64"
```

As we can see, the plugin is now installed. Let's now go ahead and build the image.

We use the `build` command to create an image using Packer. As we would need to pass values to variables, we will specify the variable values using a command-line argument, as in the following command:

```
$ packer build -var-file="variables.pkrvars.hcl" .
```

Packer would build parallel stacks using both the `webserver` and `dbserver` configs.

Packer first creates temporary resource groups to spin up staging VMs:

```
==> azure-arm.webserver: Creating resource group ...
==> azure-arm.webserver:  -> ResourceGroupName : 'pkr-Resource-Group-7dfj1c2iej'
==> azure-arm.webserver:  -> Location          : 'East US'
==> azure-arm.dbserver: Creating resource group ...
==> azure-arm.dbserver:  -> ResourceGroupName : 'pkr-Resource-Group-11xqpuxsm3'
==> azure-arm.dbserver:  -> Location          : 'East US'
```

Packer then validates and deploys the deployment templates and gets the IP addresses of the staging VMs:

```
==> azure-arm.webserver: Validating deployment template ...
==> azure-arm.webserver: Deploying deployment template ...
==> azure-arm.webserver:  -> DeploymentName : 'pkrdp7dfj1c2iej'
==> azure-arm.webserver: Getting the VM's IP address ...
==> azure-arm.webserver:  -> IP Address : '104.41.158.85'
==> azure-arm.dbserver: Validating deployment template ...
==> azure-arm.dbserver: Deploying deployment template ...
==> azure-arm.dbserver:  -> DeploymentName : 'pkrdp11xqpuxsm3'
==> azure-arm.dbserver: Getting the VM's IP address ...
==> azure-arm.dbserver:  -> IP Address : '40.114.7.11'
```

Then, Packer uses SSH to connect with the staging VMs and provisions them with Ansible:

```
==> azure-arm.webserver: Waiting for SSH to become available...
==> azure-arm.dbserver: Waiting for SSH to become available...
==> azure-arm.webserver: Connected to SSH!
==> azure-arm.dbserver: Connected to SSH!
==> azure-arm.webserver: Provisioning with Ansible...
==> azure-arm.dbserver: Provisioning with Ansible...
==> azure-arm.webserver: Executing Ansible: ansible-playbook -e packer_build_
name="webserver" -e packer_builder_type=azure-arm --ssh-extra-args '-o IdentitiesOnly=yes'
-e ansible_ssh_private_key_file=/tmp/ansible-key328774773 -i /tmp/packer-provisioner-
ansible747322992 ~/ansible/webserver-playbook.yaml
==> azure-arm.dbserver: Executing Ansible: ansible-playbook -e packer_build_
name="dbserver" -e packer_builder_type=azure-arm --ssh-extra-args '-o IdentitiesOnly=yes'
-e ansible_ssh_private_key_file=/tmp/ansible-key906086565 -i /tmp/packer-provisioner-
ansible3847259155 ~/ansible/dbserver-playbook.yaml
azure-arm.webserver: PLAY RECAP ************************************************************
**
azure-arm.webserver: default: ok=7 changed=5 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
azure-arm.dbserver: PLAY RECAP *************************************************************
azure-arm.dbserver: default: ok=11 changed=7 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
```

Once the Ansible run is complete, Packer gets the disk details, captures the images, and creates the machine images in the resource groups we specified in the Packer configuration:

```
==> azure-arm.webserver: Querying the machine's properties
==> azure-arm.dbserver: Querying the machine's properties
==> azure-arm.webserver: Querying the machine's additional disks properties ...
==> azure-arm.dbserver: Querying the machine's additional disks properties ...
==> azure-arm.webserver: Powering off machine ...
```

```
==> azure-arm.dbserver: Powering off machine ...
==> azure-arm.webserver: Generalizing machine ...
==> azure-arm.dbserver: Generalizing machine ...
==> azure-arm.webserver: Capturing image ...
==> azure-arm.dbserver: Capturing image ...
==> azure-arm.webserver: -> Image ResourceGroupName: 'packer-rg'
==> azure-arm.dbserver: -> Image ResourceGroupName: 'packer-rg'
==> azure-arm.webserver: -> Image Name: 'apache-webserver'
==> azure-arm.webserver:  -> Image Location: 'East US'
==> azure-arm.dbserver: -> Image Name: 'mysql-dbserver'
==> azure-arm.dbserver:  -> Image Location: 'East US'
```

Finally, it removes the deployment object and the temporary resource group it created:

```
==> azure-arm.webserver: Deleting Virtual Machine deployment and its attached resources...
==> azure-arm.dbserver: Deleting Virtual Machine deployment and its attached resources...
==> azure-arm.webserver: Cleanup requested, deleting resource group ...
==> azure-arm.dbserver: Cleanup requested, deleting resource group ...
==> azure-arm.webserver: Resource group has been deleted.
==> azure-arm.dbserver: Resource group has been deleted.
```

It then provides the list of artifacts it has generated:

```
==> Builds finished. The artifacts of successful builds are:
--> azure-arm: Azure.ResourceManagement.VMImage:
OSType: Linux
ManagedImageResourceGroupName: packer-rg
ManagedImageName: apache-webserver
ManagedImageId: /subscriptions/Id/resourceGroups/packer-rg/providers/Microsoft.Compute/
images/apache-webserver
ManagedImageLocation: West Europe
OSType: Linux
ManagedImageResourceGroupName: packer-rg
ManagedImageName: mysql-dbserver
ManagedImageId: /subscriptions/Id/resourceGroups/packer-rg/providers/Microsoft.Compute/
images/mysql-dbserver
```

If we look at the `packer-rg` resource group, we will find that there are two VM images within it:

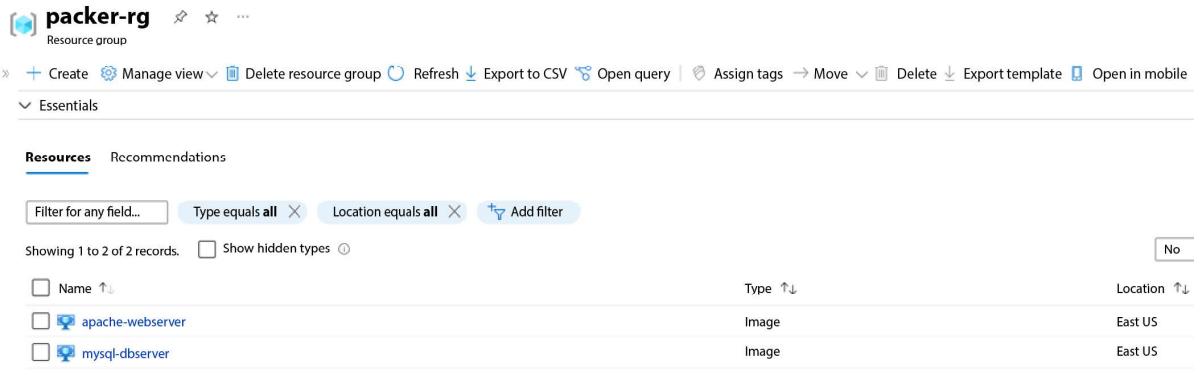

Figure 10.2 – Packer custom images

We've successfully built custom images with Packer!

> **Tip**
> It isn't possible to rerun Packer with the same managed image name once the image is created in the resource group. That is because we don't want to override an existing image accidentally. While you can override it by using the `-force` flag with `packer build`, you should include a version within the image name to allow multiple versions of the image to exist in the resource group. For example, instead of using `apache-webserver`, you can use `apache-webserver-0.0.1`.

It's time to use these images and create our infrastructure with them now.

Creating the required infrastructure with Terraform

Our goal was to build a scalable LAMP stack, so we will define a **VM scale set** using the `apache-webserver` image we created and a single VM with the `mysql-dbserver` image. A VM scale set is an autoscaling group of VMs that will scale out and scale back horizontally based on traffic, similar to how we did with containers on Kubernetes.

We will create the following resources:

- A new resource group called `lamp-rg`
- A virtual network within the resource group called `lampvnet`
- A subnet within `lampvnet` called `lampsub`
- Within the subnet, we create a **Network Interface Card** (**NIC**) for the database called `db-nic` that contains the following:
  - A network security group called `db-nsg`
  - A VM called `db` that uses the custom `mysql-dbserver` image
- We then create a VM scale set that includes the following:
  - A network profile called `webnp`
  - A backend address pool
  - A load balancer called `web-lb`
  - A public IP address attached to `web-lb`
  - An HTTP probe that checks the health of port `80`

The following figure explains the topology graphically:

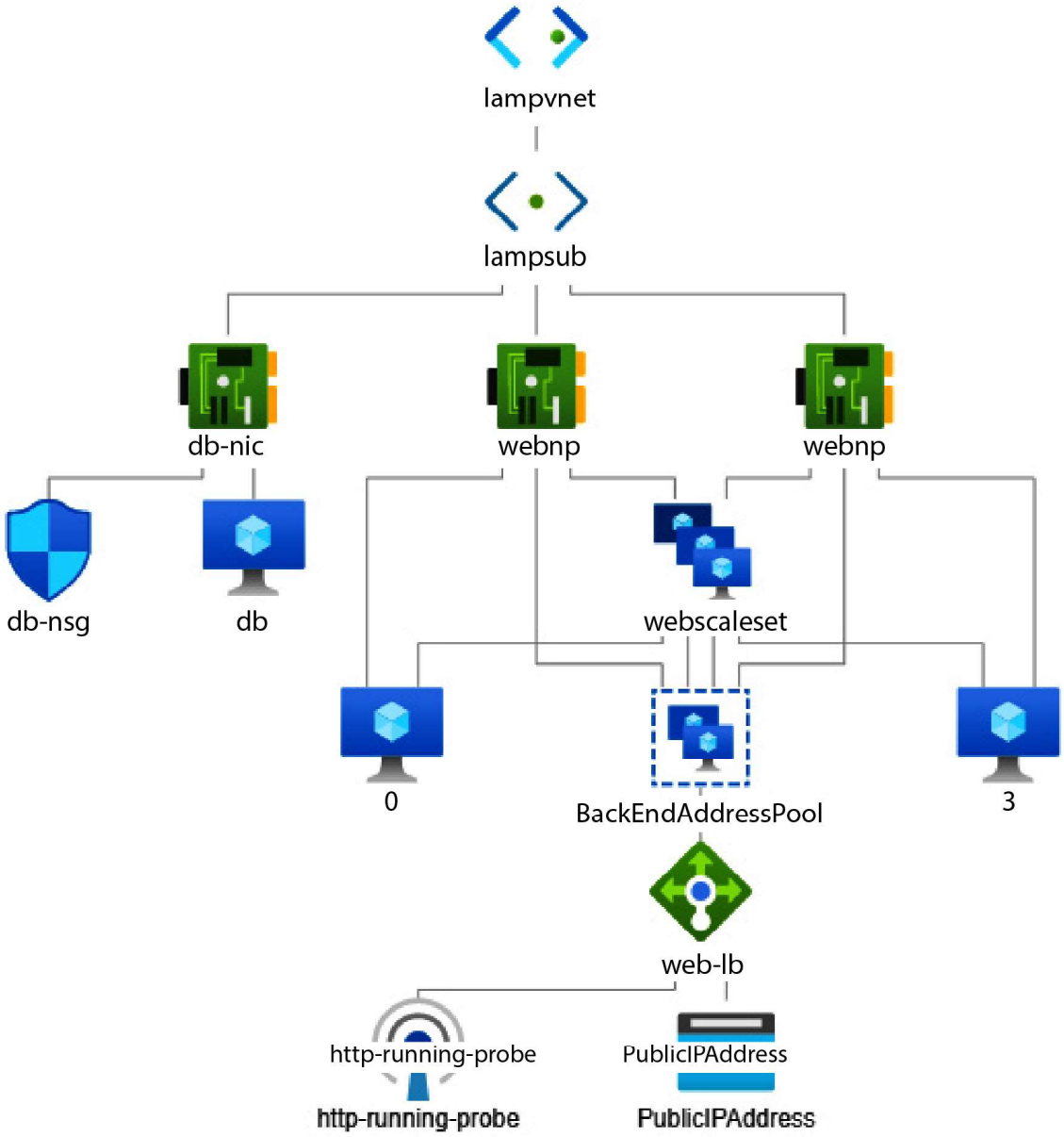

Figure 10.3 – Scalable LAMP stack topology diagram

To access resources for this section, switch to the following directory:

```
$ cd ~/modern-devops/ch10/terraform
```

We use the following Terraform template, `main.tf`, to define the configuration.

We first define the Terraform providers:

```
terraform {
  required_providers {
    azurerm = {
      source  = "azurerm"
    }
  }
}
provider "azurerm" {
  subscription_id = var.subscription_id
  client_id       = var.client_id
  client_secret   = var.client_secret
  tenant_id       = var.tenant_id
}
```

We then define the custom image data sources so that we can use them within our configuration:

```
data "azurerm_image" "websig" {
  name             = "apache-webserver"
  resource_group_name = "packer-rg"
}
data "azurerm_image" "dbsig" {
  name             = "mysql-dbserver"
  resource_group_name = "packer-rg"
}
```

We then define the resource group, virtual network, and subnet:

```
resource "azurerm_resource_group" "main" {
  name     = var.rg_name
  location = var.location
}
resource "azurerm_virtual_network" "main" {
  name                = "lampvnet"
  address_space       = ["10.0.0.0/16"]
  location            = var.location
  resource_group_name = azurerm_resource_group.main.name
}
resource "azurerm_subnet" "main" {
  name                 = "lampsub"
  resource_group_name  = azurerm_resource_group.main.name
  virtual_network_name = azurerm_virtual_network.main.name
  address_prefixes       = ["10.0.2.0/24"]
}
```

As the Apache web servers will remain behind a network load balancer, we will define the load balancer and the public IP address that we will attach to it:

```
resource "azurerm_public_ip" "main" {
  name                = "webip"
  location            = var.location
  resource_group_name = azurerm_resource_group.main.name
  allocation_method   = "Static"
  domain_name_label   = azurerm_resource_group.main.name
}
resource "azurerm_lb" "main" {
  name                = "web-lb"
  location            = var.location
  resource_group_name = azurerm_resource_group.main.name
  frontend_ip_configuration {
    name                 = "PublicIPAddress"
    public_ip_address_id = azurerm_public_ip.main.id
  }
  tags = {}
}
```

We will then define a backend address pool to the load balancer so that we can use this within the Apache VM scale set:

```
resource "azurerm_lb_backend_address_pool" "bpepool" {
  loadbalancer_id     = azurerm_lb.main.id
  name                = "BackEndAddressPool"
}
```

We will define an HTTP probe on port `80` for a health check and attach it to the load balancer:

```
resource "azurerm_lb_probe" "main" {
  loadbalancer_id     = azurerm_lb.main.id
  name                = "http-running-probe"
  port                = 80
}
```

We need a **NAT rule** to map the load balancer ports to the backend pool port, and therefore, we will define a load balancer rule that will map port `80` on the load balancer with port `80` of the backend pool VMs. We will also attach the HTTP health check probe in this config:

```
resource "azurerm_lb_rule" "lbnatrule" {
  resource_group_name            = azurerm_resource_group.main.name
  loadbalancer_id                = azurerm_lb.main.id
  name                           = "http"
  protocol                       = "Tcp"
  frontend_port                  = 80
  backend_port                   = 80
  backend_address_pool_ids       = [ azurerm_lb_backend_address_pool.bpepool.id ]
  frontend_ip_configuration_name = "PublicIPAddress"
```

```
  probe_id                      = azurerm_lb_probe.main.id
}
```

Now, we will define the VM scale set within the resource group using the custom image and the load balancer we defined before:

```
resource "azurerm_virtual_machine_scale_set" "main" {
  name                = "webscaleset"
  location            = var.location
  resource_group_name = azurerm_resource_group.main.name
  upgrade_policy_mode = "Manual"
  sku {
    name     = "Standard_DS1_v2"
    tier     = "Standard"
    capacity = 2
  }
  storage_profile_image_reference {
    id=data.azurerm_image.websig.id
  }
```

We then go ahead and define the OS disk and the data disk:

```
  storage_profile_os_disk {
    name              = ""
    caching           = "ReadWrite"
    create_option     = "FromImage"
    managed_disk_type = "Standard_LRS"
  }
  storage_profile_data_disk {
    lun          = 0
    caching        = "ReadWrite"
    create_option  = "Empty"
    disk_size_gb   = 10
  }
```

The OS profile defines how we log in to the VM:

```
  os_profile {
    computer_name_prefix = "web"
    admin_username       = var.admin_username
    admin_password       = var.admin_password
  }
  os_profile_linux_config {
    disable_password_authentication = false
  }
```

We then define a network profile that will associate the scale set with the load balancer we defined before:

```
  network_profile {
    name    = "webnp"
    primary = true
```

```
    ip_configuration {
      name      = "IPConfiguration"
      subnet_id = azurerm_subnet.main.id
      load_balancer_backend_address_pool_ids = [azurerm_lb_backend_address_pool.bpepool.
id]
      primary = true
    }
  }
  tags = {}
}
```

Now, moving on to the database configuration, we will start by defining a network security group for the database servers to allow ports `22` and `3306` from internal servers within the virtual network:

```
resource "azurerm_network_security_group" "db_nsg" {
    name                = "db-nsg"
    location            = var.location
    resource_group_name = azurerm_resource_group.main.name
    security_rule {
        name                       = "SSH"
        priority                   = 1001
        direction                  = "Inbound"
        access                     = "Allow"
        protocol                   = "Tcp"
        source_port_range          = "*"
        destination_port_range     = "22"
        source_address_prefix      = "*"
        destination_address_prefix = "*"
    }
    security_rule {
        name                       = "SQL"
        priority                   = 1002
        direction                  = "Inbound"
        access                     = "Allow"
        protocol                   = "Tcp"
        source_port_range          = "*"
        destination_port_range     = "3306"
        source_address_prefix      = "*"
        destination_address_prefix = "*"
    }
    tags = {}
}
```

We then define a NIC to provide an internal IP to the VM:

```
resource "azurerm_network_interface" "db" {
  name                = "db-nic"
  location            = var.location
  resource_group_name = azurerm_resource_group.main.name
  ip_configuration {
    name                          = "db-ipconfiguration"
    subnet_id                     = azurerm_subnet.main.id
```

```
    private_ip_address_allocation = "Dynamic"
  }
}
```

We will then associate the network security group to the network interface:

```
resource "azurerm_network_interface_security_group_association" "db" {
    network_interface_id      = azurerm_network_interface.db.id
    network_security_group_id = azurerm_network_security_group.db_nsg.id
}
```

Finally, we'll define the database VM using the custom image:

```
resource "azurerm_virtual_machine" "db" {
  name                  = "db"
  location              = var.location
  resource_group_name   = azurerm_resource_group.main.name
  network_interface_ids = [azurerm_network_interface.db.id]
  vm_size               = var.vm_size
  delete_os_disk_on_termination = true
  storage_image_reference {
    id   = data.azurerm_image.dbsig.id
  }
  storage_os_disk {
    name              = "db-osdisk"
    caching           = "ReadWrite"
    create_option     = "FromImage"
    managed_disk_type = "Standard_LRS"
  }
  os_profile {
    computer_name  = "db"
    admin_username = var.admin_username
    admin_password = var.admin_password
  }
  os_profile_linux_config {
    disable_password_authentication = false
  }
  tags = {}
}
```

Now, as we've defined everything we needed, fill the `terraform.tfvars` file with the required information, and go ahead and initialize our Terraform workspace by using the following command:

```
$ terraform init
```

As Terraform has initialized successfully, use the following command to apply the Terraform configuration:

```
$ terraform apply
Apply complete! Resources: 13 added, 0 changed, 0 destroyed.
Outputs:
web_ip_addr = "40.115.61.69"
```

As Terraform has applied the configuration and provided the load balancer IP address as an output, let's use that to navigate to the web server:

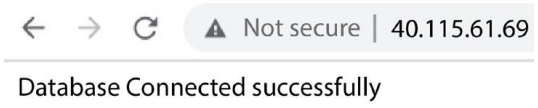

Figure 10.4 – LAMP stack working correctly

As we get the `Database Connected successfully` message, we see that the configuration is successful! We've successfully created a scalable LAMP stack using Packer, Ansible, and Terraform. It combines *IaC*, *configuration as code*, *immutable infrastructure*, and modern DevOps practices to create a seamless environment without manual intervention.

Summary

In this chapter, we have covered immutable infrastructure with Packer. We used Packer with the Ansible provisioner to build custom images for Apache and MySQL. We used the custom images to create a scalable LAMP stack using Terraform. The chapter introduced you to the era of modern DevOps, where everything is automated. We follow the same principles for building and deploying all kinds of infrastructure, be it containers or VMs. In the next chapter, we will discuss one of the most important topics of DevOps – **continuous integration**.

Questions

1. Immutable infrastructure helps avoid configuration drift. (True/False)
2. It is a best practice to source sensitive data from external variables such as environment variables or a secret management tool such as HashiCorp's Vault. (True/False)
3. What modifications must we make to our existing playbooks to allow Packer to use them?

   A. Remove any existing `ansible.cfg` files from the current working directory.

   B. Remove any host files from the current working directory.

   C. Update the `hosts` attribute to default within the playbook.

   D. None of the above.
4. Which of the following are the limitations of using the Ansible provisioner with Packer? (Choose two)

   A. You cannot pass Jinja2 macros as is to your Ansible playbooks.

   B. You cannot define `remote_user` within your Ansible playbooks.

C. You cannot use Jinja2 templates within your Ansible playbooks.

D. You cannot use roles and variables within your Ansible playbooks.

5. While naming managed images, what should we consider? (Choose two)

   A. Name images as specifically as possible.

   B. Use the version as part of the image.

   C. Don't use the version as part of the image name. Instead, always use the `-force` flag within the Packer build.

6. When using multiple provisioners, how are configurations applied to the build VM?

   A. One after the other based on occurrence in the HCL file

   B. Parallelly

7. We can use a single set of Packer files to build images with the same configuration in multiple cloud environments. (True/False)

8. What features does a VM scale set provide? (Choose two)

   A. It helps you horizontally scale VM instances with traffic.

   B. It helps you auto-heal faulty VMs.

   C. It helps you do canary deployments.

   D. None of the above.

Answers

1. True
2. True
3. C
4. A, B
5. A, B
6. B
7. True
8. A, B, C

11

Continuous Integration with GitHub Actions and Jenkins

In the previous chapters, we looked at individual tools that will help us implement several aspects of modern DevOps. Now, it's time to look at how we can combine all the tools and concepts we've learned about and use them to create a **continuous integration** (**CI**) pipeline. First, we will introduce a sample microservices-based blogging application, **Blog App**, and then look at some popular open source and SaaS-based tools that can get us started quickly with CI. We will begin with **GitHub Actions** and then move on to **Jenkins** with **Kaniko**. For every tool, we will implement CI for Blog App. We will try to keep the implementations cloud-agnostic. Since we've used the **GitOps** approach from the beginning, we will also use the same here. Finally, we will cover some best practices related to build performance.

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

- The importance of automation
- Introduction to the sample microservices-based blogging application – Blog App
- Building a CI pipeline with GitHub Actions
- Scalable Jenkins on **Kubernetes** with Kaniko
- Automating a build with triggers
- Build performance best practices

Technical requirements

For this chapter, you will 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 `ch11` directory to access the required resources:

```
$ git clone https://github.com/PacktPublishing/Modern-DevOps-Practices-2e.git \
modern-devops
$ cd modern-devops/ch11
```

So, let's get started!

The importance of automation

Automation is akin to having an efficient team of robots at your disposal, tirelessly handling repetitive, time-consuming, and error-prone tasks. Let's simplify the significance of automation:

- **Efficiency**: Think of it as having a magical helper who completes tasks in a fraction of the time you would take. Automation accelerates repetitive tasks, executing actions, processing data, and running commands far more swiftly than humans.
- **Consistency**: Humans can tire or become distracted, leading to inconsistencies in task execution. Automation guarantees that tasks are consistently carried out according to predefined rules, every single time.
- **Accuracy**: Automation operates without the fatigue or lapses that humans may experience. It adheres to instructions with precision, minimizing the likelihood of errors that could result in costly repercussions.
- **Scale**: Whether managing one system or a thousand, automation effortlessly scales operations without additional human resources.
- **Cost savings**: By reducing the reliance on manual labor, automation yields significant cost savings in terms of time and human resources.
- **Risk reduction**: Certain tasks, such as making data backups and performing security checks, are crucial but can be overlooked or skipped by humans. Automation ensures these tasks are consistently performed, mitigating risks.
- **Faster response**: Automation detects and responds to issues in real time. For instance, it can automatically restart a crashed server or adjust resource allocation during high traffic, ensuring uninterrupted user experiences.
- **Resource allocation**: Automating routine tasks liberates human resources to concentrate on more strategic and creative endeavors that require critical thinking and decision-making.

- **Compliance**: Automation enforces and monitors compliance with policies and regulations, reducing the potential for legal and regulatory complications.
- **Data analysis**: Automation processes and analyzes vast data volumes rapidly, enabling data-driven decision-making and insights.
- **24/7 operations**: Automation operates tirelessly, 24/7, guaranteeing continuous operations and availability.
- **Adaptability**: Automation can be reprogrammed to adapt to evolving requirements and environments, making it versatile and future-proof.

In the tech realm, automation is the bedrock of modern IT operations, spanning from automating software deployments to managing cloud resources and configuring network devices. It empowers organizations to streamline processes, enhance reliability, and remain competitive in the fast-paced digital landscape.

In essence, automation resembles an exceedingly efficient, error-free, round-the-clock workforce that empowers individuals and organizations to accomplish more with less effort.

To benefit from automation, the project management function is quickly diluting, and software development teams are transitioning to Agile teams that deliver in Sprints iteratively. Therefore, if there is a new requirement, we don't wait for the entire thing to be signed off before we start doing design, development, QA, and so on. Instead, we break software into workable features and deliver them in smaller chunks to get value and customer feedback quickly. That means rapid software development with less risk of failure.

Well, the teams are agile, and they develop software faster. Still, many things in the **software development life cycle** (**SDLC**) process are conducted manually, such as the fact that some teams generate Code Builds only after completing the entire development for that cycle and later find numerous bugs. It becomes difficult to trace what caused that problem in the first place.

What if you could know the cause of a broken Build as soon as you check the code into source control? What if you understand that the software fails some tests as soon as the builds are executed? Well, that's CI for you in a nutshell.

CI is a process through which developers frequently check code into a source code repository, perhaps several times a day. Automated tooling behind the scenes can detect these commits and then build, run some tests, and tell you upfront whether the commit has caused any issues. This means that your developers, testers, product owners, operations team, and everyone comes to know what has caused the problem, and the developer can fix it quickly. This creates a feedback loop in software development. We always had a manual feedback loop within software development, which was slow. So, either you wait a long time before doing your next task or do the wrong thing until you realize it is too late to undo all of that. This adds to the rework effort of everything you have done hitherto.

As we all know, fixing a bug earlier in the SDLC cycle is cheaper than fixing it later. Therefore, CI aims to provide continuous feedback on the code quality early in the SDLC. This saves your developers and the organization a lot of time and money on fixing bugs they detect when most of your code is tested. Therefore, CI helps software development teams develop better software faster.

Since we've mentioned Agile, let's briefly discuss how it compares with DevOps. Agile is a way of working and is silent on the tools, techniques, and automation required to achieve it. DevOps is an extension of the Agile mindset and helps you implement it effectively. DevOps focuses heavily on automation and looks at avoiding manual work wherever possible. It also encourages software delivery automation and seeks to amplify or replace traditional tools and frameworks. With the advent of modern DevOps, specific tools, techniques, and best practices simplify the life of a developer, QA, and operator. Modern public cloud platforms and DevOps provide teams with ready-to-use dynamic infrastructure that helps businesses reduce the time to market and build scalable, elastic, high-performing infrastructure to keep enterprises live with minimal downtime.

When introducing modern DevOps in the first chapter, we discussed that it usually applies to modern cloud-native applications. I've built an example microservices-based Blog App to demonstrate this. We will use this application in this and future chapters of this book to ensure seamless development and delivery of this application using modern DevOps tools and practices. We'll look at the sample application in the next section.

Introduction to the sample microservices-based blogging application – Blog App

Blog App is a sample modern microservices-based blogging web application that allows users to create, manage, and interact with blog posts. It caters to both authors and readers. Users can sign up to this platform using their email addresses and start writing blog posts. Readers can publicly view all blog posts created by several authors, and logged-in users can also provide reviews and ratings.

The application is written in a popular Python-based web framework called **Flask** and uses **MongoDB** as the database. The application is split into several microservices for user, post, review, and rating management. There is a separate frontend microservice that allows for user interaction. Let's look at each microservice:

- **User Management**: The User Management microservice provides endpoints to create a user account, update the profile (name and password), and delete a user account.
- **Posts Management**: The Posts Management microservice provides endpoints to create, list, get, update, and delete posts.
- **Reviews Management**: The Reviews Management microservice allows users to add reviews on posts and update and delete them. Internally, it interacts with the Ratings Management microservice to manage the ratings provided, along with the reviews.

- **Ratings Management**: The Ratings Management microservice manages ratings for posts associated with a particular review. This microservice is called from the Reviews Management microservice internally and is not exposed to the Frontend microservice.
- **Frontend**: The Frontend microservice is a Python Flask user interface application built using **Bootstrap**, which provides users with a rich and interactive user interface. It allows users to sign up, log in, view, and navigate between posts, edit their posts, add and update reviews, and manage their profiles. The microservice interacts with the backend microservices seamlessly using HTTP requests.

The **users**, **posts**, **reviews**, and **ratings** microservices interact with **MongoDB** as the database.

The following service diagram shows the interactions graphically:

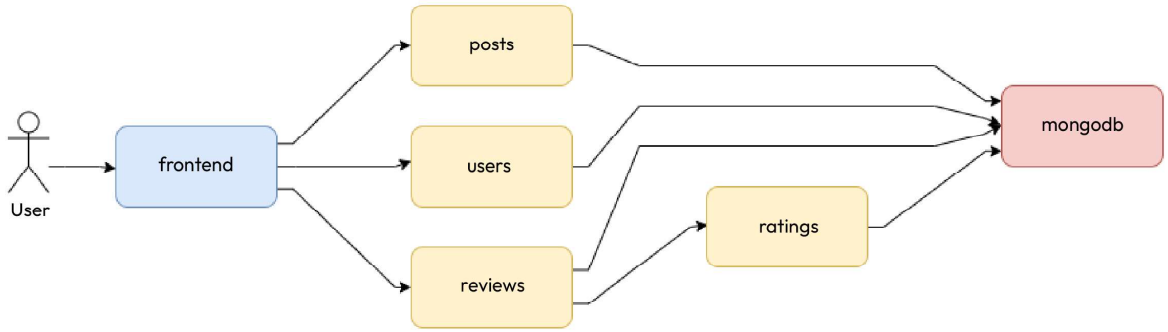

Figure 11.1 – Blog App services and interactions

As we can see, the individual microservices are fairly decoupled from each other and, therefore, can independently scale. It is also robust because the other parts of the application will work if a particular microservice is not working. The individual microservices can be independently developed and deployed as separate components, adding to the application's flexibility and maintainability. This application is an excellent example of leveraging microservices to build a modern, feature-rich web application.

Now, let's implement CI for this application. To implement CI, we will need a CI tool. We'll look at some of the popular tools and the options you have in the next section.

Building a CI pipeline with GitHub Actions

GitHub Actions is a SaaS-based tool that comes with **GitHub**. So, when you create your GitHub repository, you get access to this service out of the box. Therefore, GitHub Actions is one of the best tools for people new to CI/CD and who want to get started quickly. GitHub Actions helps you automate tasks, build, test, and deploy your code, and even streamline your workflow, making your life as a developer much easier.

Here's what GitHub Actions can do for you:

- **CI**: GitHub Actions can automatically build and test your code whenever you push changes to your repository. This ensures that your code remains error-free and ready for deployment.
- **CD**: You can use GitHub Actions to deploy your application to various hosting platforms, such as AWS, Azure, and GCP. This allows you to deliver updates to your users quickly and efficiently.
- **Workflow automation**: You can create custom workflows using GitHub Actions to automate repetitive tasks in your development process. For example, you can automatically label and assign issues, trigger builds on specific events, or send notifications to your team.
- **Custom scripts**: GitHub Actions allows you to run custom scripts and commands, giving you full control over your automation tasks. Whether you need to compile code, run tests, or execute deployment scripts, GitHub Actions can handle it.
- **Community actions**: GitHub Actions has a marketplace where you can find pre-built actions created by the community. These actions cover a wide range of tasks, from publishing to `npm` to deploying to popular cloud providers. You can easily incorporate these actions into your workflow.
- **Scheduled jobs**: You can schedule actions to run at specific times or intervals. This is handy for tasks such as generating reports, sending reminders, or performing maintenance during non-peak hours.
- **Multi-platform support**: GitHub Actions supports various programming languages, operating systems, and cloud environments, which means you can build and deploy applications for different platforms with ease.
- **Integration**: GitHub Actions seamlessly integrates with your GitHub repositories, making it a natural extension of your development environment. You can define workflows by using YAML files directly in your repository.

GitHub Actions revolutionizes the way developers work by automating routine tasks, ensuring code quality, and streamlining the SDLC. It's a valuable tool for teams and individual developers looking to enhance productivity and maintain high-quality code.

Now, let's create a CI pipeline for our sample Blog App. Blog App consists of multiple microservices, and each microservice runs on an individual **Docker** container. We also have unit tests written for each microservice, which we can run to verify the code changes. If the tests pass, the build will pass; otherwise, it will fail.

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

```
$ cd ~/modern-devops/blog-app
```

This directory contains multiple microservices and is structured as follows:

```
.
├── frontend
│   ├── Dockerfile
│   ├── app.py
│   ├── app.test.py
│   ├── requirements.txt
│   ├── static
│   └── templates
├── posts
│   ├── Dockerfile
│   ├── app.py
│   ├── app.test.py
│   └── requirements.txt
├── ratings ...
├── reviews ...
└── users ...
```

The `frontend` directory contains files for the **frontend** microservice, and notably, it includes `app.py` (the Flask application code), `app.test.py` (the unit tests for the Flask application), `requirements.txt` (which contains all Python modules required by the app), and `Dockerfile`. It also includes a few other directories catering to the user interface elements of this app.

The **posts**, **reviews**, **ratings**, and **users** microservices have the same structure and contain `app.py`, `app.test.py`, `requirements.txt`, and `Dockerfile` files.

So, let's start by switching to the `posts` directory:

```
$ cd posts
```

As we know that Docker is inherently CI-compliant, we can run the tests using `Dockerfile` itself. Let's investigate the Dockerfile of the posts service:

```
FROM python:3.7-alpine
ENV FLASK_APP=app.py
ENV FLASK_RUN_HOST=0.0.0.0
RUN apk add --no-cache gcc musl-dev linux-headers
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
EXPOSE 5000
COPY . .
RUN python app.test.py
CMD ["flask", "run"]
```

This `Dockerfile` starts with the `python:3.7-alpine` base image, installs the requirements, and copies the code into the working directory. It runs the `app.test.py` unit test to check whether the code would work if we deploy it. Finally, the `CMD` command defines a `flask run` command to run when we launch the container.

Let's build our `Dockerfile` and see what we get:

```
$ docker build --progress=plain -t posts .
#4 [1/6] FROM docker.io/library/python:3.7-alpine
#5 [internal] load build context
#6 [2/6] RUN apk add --no-cache gcc musl-dev linux-headers
#7 [3/6] COPY requirements.txt requirements.txt
#8 [4/6] RUN pip install -r requirements.txt
#9 [5/6] COPY . .
#10 [6/6] RUN python app.test.py
#10 0.676 ----------------------------------------------------
#10 0.676 Ran 8 tests in 0.026s
#11 exporting to image
#11 naming to docker.io/library/posts done
```

As we can see, it built the container, executed a test on it, and responded with `Ran 8 tests in 0.026s` and an `OK` message. Therefore, we could use `Dockerfile` to build and test this app. We used the `--progress=plain` argument with the `docker build` command. This is because we wanted to see the stepwise output of the logs rather than Docker merging progress into a single message (this is now a default behavior).

Now, let's look at GitHub Actions and how we can automate this step.

Creating a GitHub repository

Before we can use GitHub Actions, we need to create a GitHub repository. As we know that each microservice can be independently developed, we will place all of them in separate Git repositories. For this exercise, we will focus only on the **posts** microservice and leave the rest to you as an exercise.

To do so, go to `https://github.com/new` and create a new repository. Give it an appropriate name. For this exercise, I am going to use `mdo-posts`.

Once you've created it, clone the repository by using the following command:

```
$ git clone https://github.com/<GitHub_Username>/mdo-posts.git
```

Then, change the directory into the repository directory and copy the `app.py`, `app.test.py`, `requirements.txt`, and `Dockerfile` files into the repository's directory using the following commands:

```
$ cd mdo-posts
$ cp ~/modern-devops/blog-app/posts/* .
```

Now, we need to create a GitHub Actions workflow file. We'll do this in the next section.

Creating a GitHub Actions workflow

A GitHub Actions workflow is a simple YAML file that contains the build steps. We must create this workflow in the `.github/workflows` directory within the repository. We can do this using the following command:

```
$ mkdir -p .github/workflows
```

We will use the following GitHub Actions workflow file, `build.yaml`, for this exercise:

```
name: Build and Test App
on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2
    - name: Login to Docker Hub
      id: login
      run: docker login -u ${{ secrets.DOCKER_USER  }} -p ${{ secrets.DOCKER_PASSWORD }}
    - name: Build the Docker image
      id: build
      run: docker build . --file Dockerfile --tag ${{ secrets.DOCKER_USER  }}/
mdo-posts:$(git rev-parse --short "$GITHUB_SHA")
    - name: Push the Docker image
      id: push
      run: docker push ${{ secrets.DOCKER_USER  }}/mdo-posts:$(git rev-parse --short
"$GITHUB_SHA")
```

This file comprises the following:

- `name`: The workflow's name – `Build and Test App` in this case.
- `on`: This describes when this workflow will run. In this case, it will run if a `push` or `pull` request is sent on the `main` branch.
- `jobs`: A GitHub Actions workflow contains one or more jobs that run in parallel by default. This attribute includes all jobs.
- `jobs.build`: This is a job that does the container build.
- `jobs.build.runs-on`: This describes where the build job will run. We've specified `ubuntu-latest` here. This means that this job will run on an Ubuntu VM.
- `jobs.build.steps`: This consists of the steps that run sequentially within the job. The build job consists of four build steps: `checkout`, which will check out the code from your repository; `login`, which will log in to Docker Hub; `build`, which will run a Docker build on your code; and `push`, which will push your Docker image to **Docker Hub**. Note that we tag the image with the Git commit SHA. This relates the build with the commit, making Git the single source of truth.
- `jobs.build.steps.uses`: This is the first step and describes an action you will run as a part of your job. Actions are reusable pieces of code that you can execute in your pipeline. In this case, it runs the `checkout` action. It checks out the code from the current branch where the action is triggered.

> **Tip**
> Always use a version with your actions. This will prevent your build from breaking if a later version is incompatible with your pipeline.

- `jobs.build.steps.name`: This is the name of your build step.
- `jobs.build.steps.id`: This is the unique identifier of your build step.
- `jobs.build.steps.run`: This is the command it executes as part of the build step.

The workflow also contains variables within `${{ }}`. We can define multiple variables within the workflow and use them in the subsequent steps. In this case, we've used two variables – `${{ secrets.DOCKER_USER }}` and `${{ secrets.DOCKER_PASSWORD }}`. These variables are sourced from **GitHub secrets**.

> **Tip**
> It is best practice to use GitHub secrets to store sensitive information. Never store these details directly in the repository with code.

You must define two secrets within your repository using the following URL: `https://github.com/<your_user>/mdo-posts/settings/secrets/actions`.

Define two secrets within the repository:

```
DOCKER_USER=<Your Docker Hub username>
DOCKER_PASSWORD=<Your Docker Hub password>
```

Now, let's move this `build.yml` file to the `workflows` directory by using the following command:

```
$ mv build.yml .github/workflows/
```

Now, we're ready to push this code to GitHub. Run the following commands to commit and push the changes to your GitHub repository:

```
$ git add --all
$ git commit -m 'Initial commit'
$ git push
```

Now, go to the **Workflows** tab of your GitHub repository by visiting `https://github.com/<your_user>/mdo-posts/actions`. You should see something similar to the following:

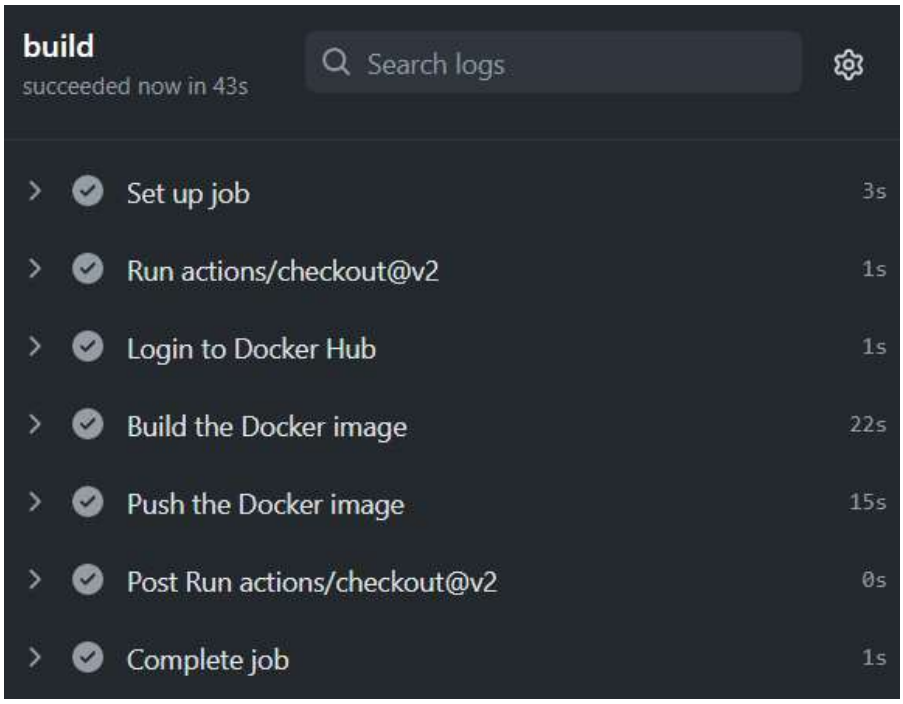

Figure 11.2 – GitHub Actions

As we can see, GitHub has run a build using our workflow file, and it has built the code and pushed the image to **Docker Hub**. Upon visiting your Docker Hub account, you should see your image present in your account:

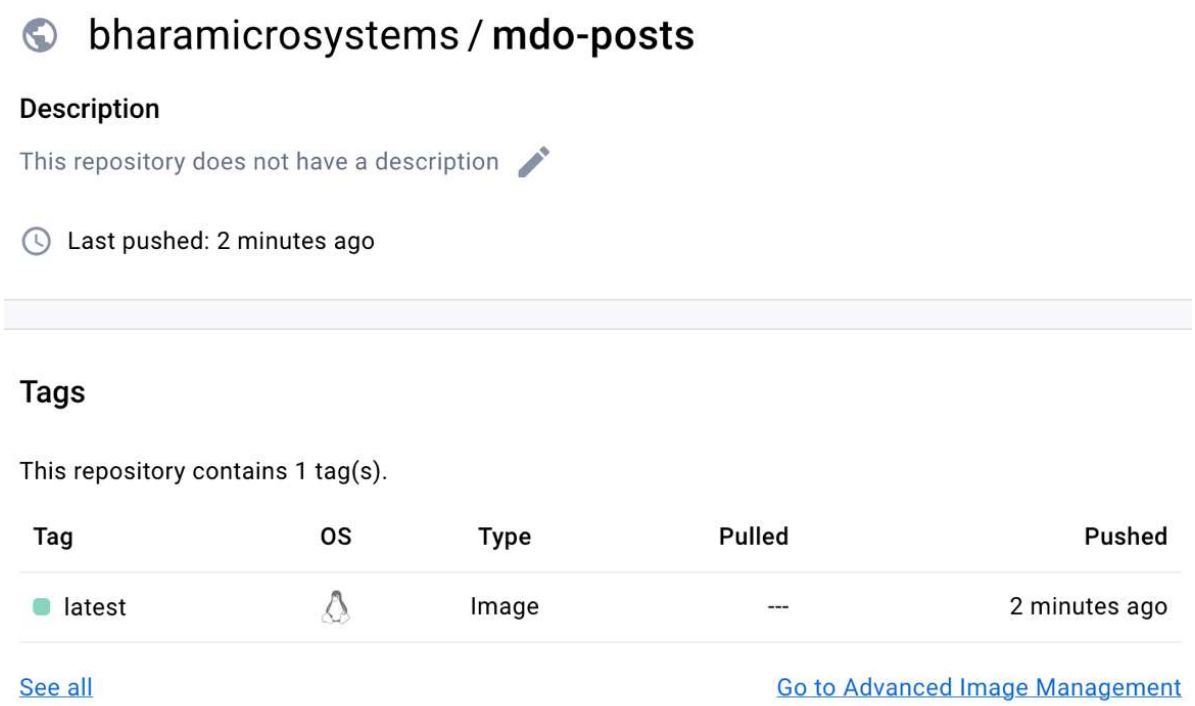

Figure 11.3 – Docker Hub image

Now, let's try to break our code somehow. Let's suppose that someone from your team changed the `app.py` code, and instead of returning `post` in the `create_post` response, it started returning `pos`. Let's see what would happen in that scenario.

Make the following changes to the `create_post` function in the `app.py` file:

```
@app.route('/posts', methods=['POST'])
def create_post():
    ...
    return jsonify({'pos': str(inserted_post.inserted_id)}), 201
```

Now, commit and push the code to GitHub using the following commands:

```
$ git add --all
$ git commit -m 'Updated create_post'
$ git push
```